查看所有版本

Vitis 统一软件平台文档

应用加速开发

UG1393 (v2021.1) 2021 年 7 月 19 日

XILINX

目录

第一部分

Vitis 入门

按设计进程浏览内容

赛灵思文档按一组标准设计进程进行组织，以便帮助您查找当前开发任务相关的内容。所有 Versal™ ACAP 设计进程的对应设计中心均可在 Xilinx.com 网站上找到。本文档涵盖了以下设计进程：

- 主机软件开发：应用代码开发以及加速器开发，包括库、XRT 和 Graph API 的用法。

本部分包含以下章节：

XILINX

第 1 章

Vitis 软件平台版本说明

本节包含有关此版本中的 Vitis™ 软件平台功能特性和更新的信息。其中还包含有关用于 Versal™ AI 引擎开发的 Vitis 软件平台功能特性和更新的信息。

新增功能

如需了解有关该版本的 Vitis™ 统一软件开发平台中的新增功能的信息，请参阅 Vitis 新增功能页面。

受支持的平台

数据中心加速器卡

Alveo™ 数据中心加速器卡的最新 Vitis 目标平台可通过以下网址来访问：china.xilinx.com/products/boards-and-kits/alveo.html。

要了解每个加速器卡以及可用目标平台的规格，请参阅《Alveo 数据中心加速器卡平台用户指南》(UG1120)。每个加速器卡的入门部分都包含有关在该卡上部署应用的信息。

如需了解有关设置 XRT 和平台的更多信息，请参阅安装 Xilinx Runtime 和平台。

嵌入式平台

如需了解有关 Vitis 核开发套件可用的嵌入式平台的信息，请访问嵌入式平台下载页面。嵌入式处理器平台（如 Versal VCK190 平台、Zynq UltraScale+ MPSoC ZCU102/ZCU104 基本平台以及 Zynq-7000 基本平台）均为同时适用于 Vitis 应用加速开发流程和 Vitis 嵌入式软件开发流程的可选平台。但在大多数情况下，您可以使用 Vitis IDE 创建自己的平台。

用于 AI 引擎开发的 Versal 平台

VCK190 平台可用于 Vitis 应用加速开发流程，如《Versal ACAP AI 引擎编程环境用户指南》(UG1076) 中所述。此平台支持开发下列设计：

- AI 引擎 Graph 和内核
- 可编程逻辑内核
- 以 Versal 器件中的 Arm 处理器上运行的 Linux 或裸机操作系统为目标的主机应用。

行为更改

下表指定了该版本与前版本之间影响移植行为或流程的差异。

表 1：**行为更改汇总**

区域	行为
Vitis HLS[1]	“Git Repository”原先可从左下象限内访问。现已迁移至“Console”区域内。
	“Analysis”透视图已不再存在。其中报告和视图现在可从“Synthesis”布局来访问。
	编译指示 HLS SHARED 原先为独立编译指示。现已将其指定为包含在 `pragma HLS STREAM type=` 选项内。 • `pragma HLS SHARED` 现已改为 `pragma HLS STREAM type=shared`。 • `pragma HLS SHARED` 和 `pragma HLS STABLE` 现已合并为 `pragma HLS STREAM type=unsync`（共享且未同步）。
	Vivado IP 流程的 `config_interface -m_axi_offset` 的默认设置现已改为 slave。这意味着将 `m_axi` 接口添加到 Vivado IP 时，会同时添加 `s_axilite` 接口，并通过该接口来管理偏移。
	浮点累加器和 MAC 可提供全新的精度，以增强通过 `config_op` 命令来进行控制的能力。要在 2021.1 中复现 2020.2 结果，请使用以下命令： `config_op facc -impl auto -precision low`
Vitis 剖析	在 `xrt.ini` 文件中，`profile=true` 已更改为 `opencl_summary=true` 和 `opencl_device_counter=true` 以捕获内核侧数据。这些选项可以结合在一起指定，也可以各自单独指定。
Vitis 时间线	所有追踪结果（`opencl_trace=true`、`data_transfer_trace=true`、`stall_trace=all` 等）都已被添加到 Vitis 分析器中的“Application Timeline”内。您可在查看报告时指定要添加到“Application Timeline”的元素。
	`timeline_trace` 已改为 `opencl_trace`。
Vitis 调试	不再支持在硬件仿真期间进行 GDB 内核调试。
Vitis AI 引擎	默认最优化级别已从 2020.2 中的 `xlopt=0` 改为 2021.1 中的 `xlopt=1`。
	通过使用 `launch_hw_emu.sh` 的 `-aie-sim-options`，即可通过文本文件来启用 `AIE_PROFILE` 以便剖析 AI 引擎。
	对 `x86simulator` 进行的更改：已添加包切换构造支持、GDB 调试和 `printf()` 宏。
	已添加 XRT 本机 C++ API 用于控制 Graph (`xrt::graph`)。
	现已为访问 GMIO 的设计提供了硬件仿真支持。
	在 ADF Graph 中已弃用针对 PL 内核的支持。

注释：

1. 欲知详情，请参阅《Vitis 高层次综合用户指南》(UG1399)。

已知问题

如需了解 Vitis 软件平台的已知问题，请参阅赛灵思答复记录 76498。

XILINX

第 2 章

安装

安装要求

Vitis™ 软件平台由适用于交互式工程开发的集成设计环境 (IDE) 和适用于脚本化或手动应用开发的命令行工具组成。Vitis 软件平台还包含 Vivado® Design Suite 用于在目标器件上实现内核，并用于开发定制硬件平台。

此处列出的部分要求仅适用于软件加速功能，不适用于嵌入式软件开发功能。赛灵思建议安装所有必要的程序包，以获取最佳的 Vitis 软件平台体验。

要在计算机上安装并运行 Vitis，您的系统必须满足以下最低要求。

注释：从 2021.2 起，应用加速开发流程将终止针对下列操作系统的支持：

- RHEL/CentOS 7.6, 7.7
- Ubuntu 16.04.5 LTS 和 16.04.6 LTS
- Ubuntu 18.04.1 LTS、18.04.2 LTS 和 18.04.3 LTS

注释：Windows 操作系统不支持应用加速开发流程。

表 2：**应用加速开发流程最低系统要求**

组件	要求	
	开发 （构建机器操作系统）	**部署（主机操作系统）** 通过 XRT 启用
操作系统	64 位 Linux： • CentOS/RHEL 7.6、7.7、7.8、8.1 和 8.2 • RHEL 8.3 • Ubuntu 16.04.5 LTS、16.04.6 LTS、18.04.1 LTS、18.04.2 LTS、18.04.3 LTS、18.04.4 LTS、18.04.5 LTS、20.04 LTS 和 20.04.1 LTS • Amazon Linux 2 AL2 LTS	对于本地加速（Alveo 数据中心加速器卡）： • CentOS/RHEL 7.6、7.7、7.8、8.1 和 8.2 • RHEL 8.3 • Ubuntu 16.04.5 LTS、16.04.6 LTS、18.04.1 LTS、18.04.2 LTS、18.04.3 LTS、18.04.4 LTS、18.04.5 LTS、20.04 LTS 和 20.04.1 LTS • Amazon Linux 2 AL2 LTS
		对于边缘加速（嵌入式平台）： • PetaLinux 2021.1
系统存储空间	对于 Alveo 卡：64 GB（建议 80 GB） 对于嵌入式：32 GB	
互联网连接	下载驱动程序与实用工具的前提条件。	

表 2：**应用加速开发流程最低系统要求** (续)

组件	要求	
	开发 （构建机器操作系统）	**部署（主机操作系统）** 通过 XRT 启用
硬盘空间	100 GB	

表 3：**AI 引擎开发流程最低系统要求**

组件	要求
操作系统	64 位 Linux： · CentOS/RHEL 7.6、7.7、7.8、8.1 和 8.2 · RHEL 8.3 · Ubuntu 16.04.5 LTS、16.04.6 LTS、18.04.1 LTS、18.04.2 LTS、18.04.3 LTS、18.04.4 LTS、18.04.5 LTS、20.04 LTS 和 20.04.1 LTS
系统存储空间	64 GB（建议 80 GB）
互联网连接	下载驱动程序与实用工具的前提条件。
硬盘空间	100 GB

OpenCL 可安装客户端驱动程序加载器

Vitis™ 环境支持 OpenCL 可安装客户端驱动程序 (ICD) 扩展 (`cl_khr_icd`)。此扩展支持在同一系统上共存多个 OpenCL 实现。ICD 加载器充当所有已安装的平台的监管程序，并为所有 API 调用提供标准处理程序。

应用可从已安装的平台列表中选择 OpenCL 平台。基于应用指定的平台 ID，ICD 即可将 OpenCL 主机调用分派至正确的运行时。

提示：如果您的系统具有或者使用多个版本的 OpenCL 库，那么可安装此可选程序包。

赛灵思不提供 OpenCL ICD 库，因此应使用如下步骤在系统上安装该库。

Ubuntu

在 Ubuntu 上，ICD 库随分发版打包在一起。安装以下程序包：

```
sudo apt-get install ocl-icd-libopencl1
sudo apt-get install opencl-headers
sudo apt-get install ocl-icd-opencl-dev
```

RHEL/CentOS

对于 RHEL/CentOS，请使用 EPEL 安装以下程序包：

```
sudo yum install ocl-icd
sudo yum install ocl-icd-devel
sudo yum install opencl-headers
```

注释：如需了解有关安装 EPEL 的更多信息，请参阅 https://fedoraproject.org/wiki/EPEL。

Vitis 软件平台安装

安装 Vitis 软件平台

请确保您的系统满足 安装要求 中描述的所有要求。

提示：为了缩短安装时间，请禁用杀毒软件并关闭所有已打开的非必要程序。

1. 访问赛灵思下载网站。
2. 下载对应您的操作系统的安装程序。
3. 运行安装程序，这样会打开 Xilinx Unified 2021.1 Installer 的“Welcome”页面。
4. 单击“Next”打开此安装程序的“Select Install Type”页面。
5. 输入您的赛灵思用户帐户凭证，然后选择“Download and Install Now”。
6. 单击“Next”打开此安装程序的“Accept License Agreements”页面。
7. 单击每个“I Agree”复选框接受条款。
8. 单击“Next”打开此安装程序的“Select Product to Install”页面。
9. 选择“Vitis”，然后单击“Next”以打开安装程序的“Vitis Unified Software Platform”页面。
10. 选择设计工具和器件以自定义安装（可选）。

    重要提示！请勿取消选中以下选项。该选项是安装的必需选项。

    - “Devices” → “Install devices for Alveo and Xilinx Edge acceleration platforms”

    注释：Vitis 统一软件平台会一并安装 Vitis 工具和 Vivado Design Suite。您无需单独安装 Vivado 工具。如果需要，您还可以安装 System Generator 和 Model Composer。
11. 单击“Next”打开此安装程序的“Select Destination Directory”页面
12. 指定安装目录、复查位置摘要信息、复查所需的磁盘空间以确保有足够空间可用，然后单击“Next”以打开安装程序的“Installation Summary”页面。
13. 单击“Install”开始安装软件。

重要提示！请勿取消选中以下选项。该选项是安装的必需选项。

- “Devices” → “Install devices for Alveo and Xilinx Edge acceleration platforms”

注释：Vitis 统一软件平台会一并安装 Vitis 工具和 Vivado Design Suite。您无需单独安装 Vivado 工具。如果需要，您还可以安装 System Generator 和 Model Composer。

成功安装 Vitis 软件后，会显示一条确认消息，并提示您运行 `installLibs.sh` 脚本。

1. 在以下位置找到此脚本：`<install_dir>/Vitis/<release>/scripts/installLibs.sh`，其中 `<install_dir>` 为安装位置，`<release>` 是安装版本。
2. 使用 `sudo` 权限运行此脚本，如下所示：

```
sudo installLibs.sh
```

此命令会根据操作系统来为 Vitis 工具安装必要数量的软件包。

重要提示！请留意此脚本返回的任何消息。您可能需要手动安装任何缺失的软件包。

安装 Xilinx Runtime 和平台

赛灵思的 Xilinx Runtime (XRT) 是以用户空间与内核驱动程序组件组合的形式来实现的。XRT 支持 Alveo PCIe 卡以及 Versal 和 Zynq UltraScale+ MPSoC 嵌入式系统平台，并可提供连接至赛灵思可编程逻辑器件的软件接口。

您必须安装 XRT 后才能在 Vitis 应用加速开发流程中使用。您无需为选择下载的每个额外平台重新安装 XRT。

注释： 如果目标为 Arm® 嵌入式平台则无需安装 XRT：Vitis 编译器自带 `xclbinutil` 副本用于硬件生成和软件编译，您可使用来自 `sysroot` 的 XRT。在下载页面上查找“Common images for Embedded Vitis platforms”。

重要提示！XRT 安装使用标准 Linux RPM 和 Linux DEB 分发文件，所有软件安装和固件安装都需要 root 访问权限。

`<rpm-dir>` 或 `<deb-dir>` 是下载的安装包的安装目录。

要下载并安装适用于您的操作系统的 XRT 程序包，请执行以下操作。

1. 转至 https://china.xilinx.com/xrt。
2. 在 “Getting Started” 页面上，您可以选择下载适用于特定 Alveo 数据中心加速器卡或适用于嵌入式平台的 XRT 程序包。选择平台后，您将转至网站，其中提供了有关下载 XRT 和所选平台的必要文件的指示信息。
3. 遵循指示信息安装 XRT 及您所选的平台。

提示：在平台下载页面上提供了有关安装 Alveo 数据中心加速器卡的指示信息。在后续章节中可找到有关嵌入式平台的指示信息。

安装嵌入式平台

嵌入式平台可从 Vitis 嵌入式平台下载页面下载，以供在 Vitis 统一软件平台中使用。对于 Vitis 嵌入式软件开发流程，您可将嵌入式平台与 Linux、单机/裸机或 RTOS 域搭配使用。为支持 Vitis 应用加速开发流程，嵌入式平台必须运行 Linux，并将 XRT 集成到 `rootfs` 中。在下载页面上可找到受支持的平台的完整列表。

要安装平台，请下载 zip 文件，并将其解压到 `/opt/xilinx/platforms`，或者将其解压到其它位置，并将该位置添加到 `PLATFORM_REPO_PATHS` 环境变量中。

嵌入式平台需要 `sysroot` 才能为 Vitis 应用加速流程执行主机应用交叉编译。在下载页面上查找“Common images for Embedded Vitis platforms”块，下载并解压适用于您的平台架构的常用镜像。

运行 `sdk.sh` 即可解压并安装 `sysroot`。`-d` 选项允许您选择 `sysroot` 的安装位置。此程序包还提供了预编译的内核镜像和 `rootfs`。

您可将 `sysroot` 添加到 Makefile 以供在您的命令行工程中使用，或者 Vitis IDE 将提示您将其添加到自己的应用工程中。例如，在您的 Makefile 中，将 `<SYSROOT>` 指向 `/<install_path>/aarch64-xilinx-linux`，此文件是运行 `sdk.sh` 生成的。

设置用于运行 Vitis 软件平台的环境

要配置用于运行 Vitis 软件平台的环境，请在命令 shell 中运行以下脚本以设置要在该 shell 中运行的工具：

```
#set up XILINX_VITIS and XILINX_VIVADO variables
source <Vitis_install_path>/Vitis/2021.1/settings64.sh
#set up XILINX_XRT for data center platforms (not required for embedded
platforms)
source /opt/xilinx/xrt/setup.sh
```

提示：其中也提供了 `.csh` 脚本。

这样即可为 Vitis 应用加速开发流程、Vitis 嵌入式软件开发流程和 AI 引擎工具设置工具，用于在 Versal AI 引擎器件上进行开发。

要使用按 安装 Xilinx Runtime 和平台 中所述下载的任何平台，请将以下环境变量设置为指向这些平台的所在位置：

```
export PLATFORM_REPO_PATHS=<path to platforms>
```

这样即可为工具识别平台文件的位置，并使其可供您的设计工程访问。

第 3 章

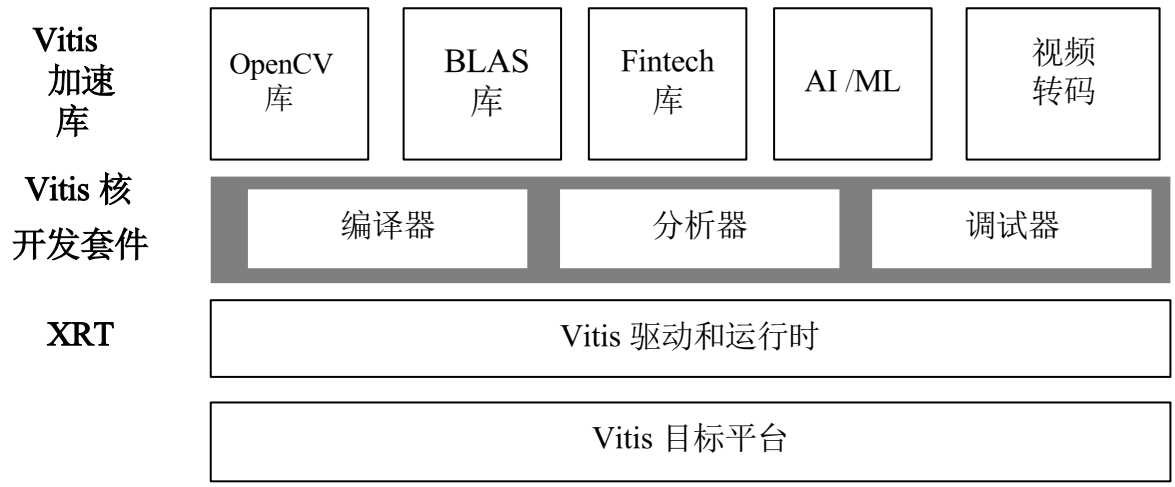
Vitis 加速环境简介

使用 Vitis 软件平台进行加速流程应用开发

Vitis™ 统一软件平台工具将赛灵思软件开发的方方面面全部组合到统一的环境内。Vitis 软件平台支持 Vitis 嵌入式软件开发流程以满足希望迁移至下一代技术的赛灵思软件开发套件 (SDK) 用户的使用需求，也支持 Vitis 应用加速开发流程，以满足希望使用基于赛灵思 FPGA 的最新软件加速功能的软件开发者的需求。此处内容主要与应用加速流程以及 Vitis 核开发套件和赛灵思的 Xilinx Runtime (XRT) 的使用有关。

Vitis 应用加速开发流程提供了相应的框架，可通过使用标准编程语言来为软件和硬件组件开发和交付 FPGA 加速应用。软件组件或主机程序是使用 C/C++ 语言开发的，可在 x86 或嵌入式处理器上运行，借助 OpenCL™ API 调用来管理与加速器的运行时交互。硬件组件或内核则可使用 C/C++、OpenCL C 或 RTL 来开发。Vitis 软件平台有助于促进对异构应用的硬件和软件元素进行并发开发和测试。

图 1：**Vitis 统一软件平台**



X23292-082921

如上图所示，Vitis 统一软件平台由以下功能特性和元素组成：

- 以加速硬件平台为目标的 Vitis 技术（例如，Alveo™ 数据中心加速器卡）和基于 Versal 或 Zynq® UltraScale+™ MPSoC 的嵌入式处理器平台。
- XRT 可提供 API 和驱动程序，以供您的主机程序用于连接到目标平台，并处理您的主机程序与加速内核之间的传输事务。

- Vitis 核开发套件可以提供软件开发工具堆栈（例如，编译器和交叉编译器）、分析器以及调试器，开发工具可用于构建主机程序和内核代码，分析器供您用于对应用性能进行剖析和分析，调试器则可帮助您定位和修复应用中的任何问题。
- Vitis 加速库可提供性能优化的 FPGA 加速，仅需最低限度的代码更改，且无需重新实现算法即可充分发挥赛灵思自适应计算的所有优势。Vitis 加速库可用于常用数学、统计数据、线性代数和 DSP 的常用函数，并且可用于特定领域的应用，例如，视觉和图像处理、计量金融、数据库、数据分析以及数据压缩等。如需了解有关 Vitis 加速库的更多信息，请访问 https://xilinx.github.io/Vitis_Libraries/。

FPGA 加速

相比传统 CPU/GPU 加速，赛灵思 FPGA 可提供诸多优势，包括能够实现可在处理器上运行的任意功能的定制架构，可以更低的功耗损耗实现更高的性能。相比于处理器架构，赛灵思器件中由可编程逻辑 (PL) 互连结构所构成的结构在应用执行中可支持实现高度并行性。

为了在赛灵思器件上实现软件加速的优势，您应想方设法将加速目标设定为硬件中的应用程序内存在密集计算的各部分。在定制硬件上实现这些功能即可帮助您在性能与功耗之间达到理想平衡。

如需了解有关如何设计硬件架构以实现最优性能的更多信息，以及有关其它设计技巧建议的信息，请回顾 使用 Vitis 软件平台加速应用的方法论。

执行模型

在 Vitis 核开发套件中，应用程序被分为主机应用部分和硬件加速内核部分，并在这两部分之间建立通信通道。以 C/C++ 语言编写并使用 API 抽象（如 OpenCL）的主机程序在主机处理器（例如，x86 服务器或适用于嵌入式平台的 Arm 处理器）上运行，而硬件加速内核则在赛灵思器件的可编程逻辑 (PL) 区域内运行。

由 XRT 管理的 API 调用用于处理主机程序与硬件加速器之间的传输事务。主机与内核之间的通信（包括控制和数据传输）都会跨 PCIe® 总线或 AXI 总线（针对嵌入式平台）发生。在硬件中的特定存储器位置之间传输控制信息时，使用全局存储器在主机程序与内核之间进行数据传输。主机处理器和硬件加速器均可访问全局存储器，但仅限主机应用才能访问主机存储器。

例如，在典型应用中，主机首先将内核要操作的数据从主机存储器传输到全局存储器中。随后，内核对此数据进行操作，并将结果存储回全局存储器。内核完成操作后，主机会将结果传回主机存储器。主机与全局存储器之间的数据传输会引发时延，这可能牺牲总体应用性能。为了在真实系统中实现加速，硬件加速内核带来的优势必须大于数据传输所添加的时延。

目标平台包含 FPGA 加速内核、全局存储器和直接存储器访问 (DMA) 用于执行存储器传输。内核可以拥有一个或多个全局存储器接口，并且可编程。Vitis 核开发套件执行模型可分解为以下步骤：

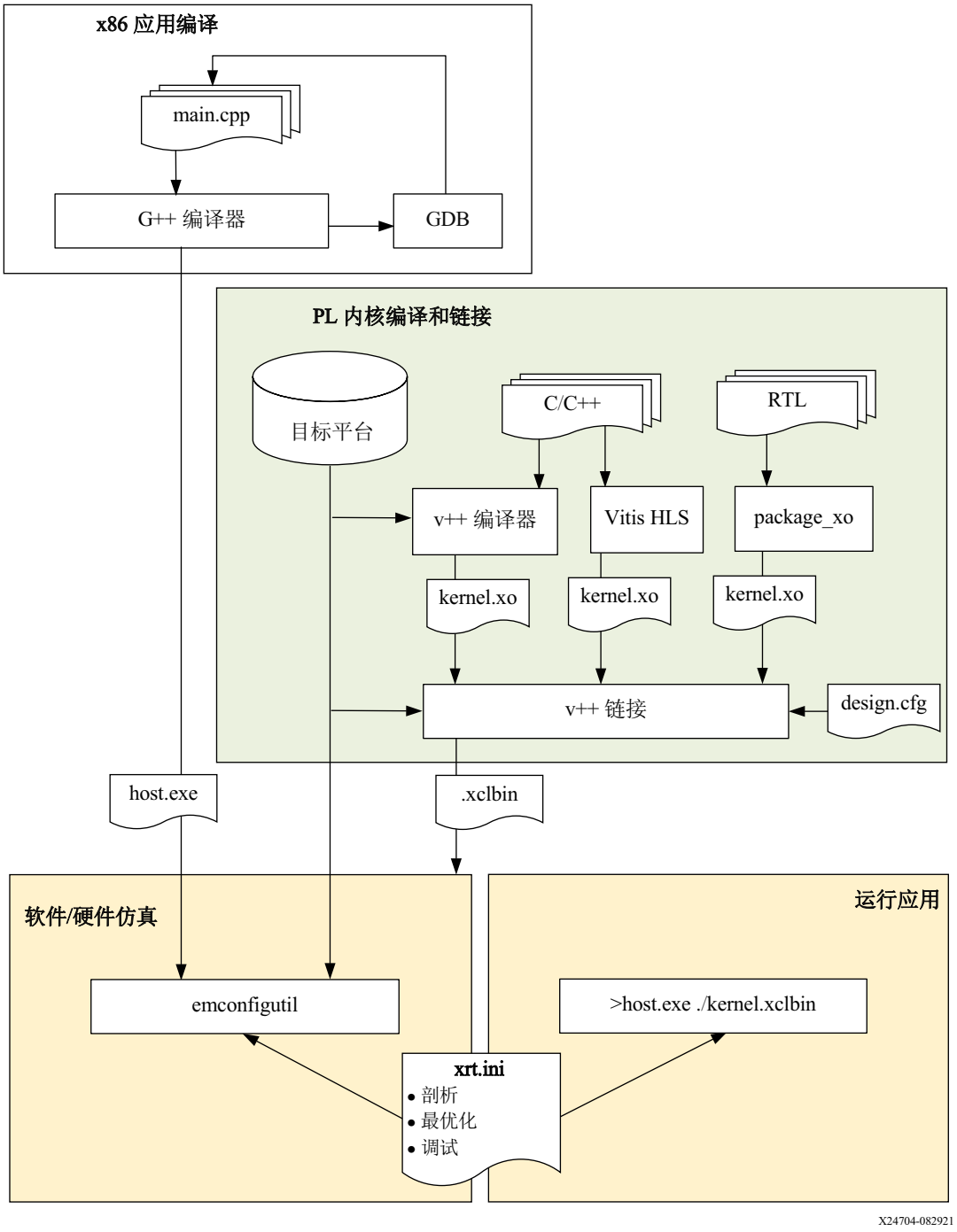
1. 主机程序将内核所需的数据写入已连接的器件的全局存储器，此器件可通过 Alveo 数据中心加速器卡上的 PCIe 接口或者嵌入式平台上的 AXI 总线来连接。
2. 主机程序使用其输入参数来设置内核。
3. 主机程序在 FPGA 上触发内核函数的执行。
4. 必要时，内核在从全局存储器中读取数据时执行所需的计算。
5. 内核将数据写回全局存储器并通知主机它已完成任务。
6. 主机程序将数据从全局存储器读回主机存储器，并根据需要继续处理。

FPGA 可以在加速器上容纳多个内核实例，包括不同类型的多个内核和相同内核的多个实例。XRT 以用户不可知方式统筹安排加速器内主机程序与内核之间的交互。如需获取 XRT 架构文档，请访问 https://xilinx.github.io/XRT/。

数据中心应用加速开发流程

下图描述了构建和运行应用以供在 Alveo 数据中心加速器卡上使用所需的步骤。这些步骤总结如下，在这整篇文档中均可找到有关每个步骤的详细信息。

图 2：**适用于数据中心加速器卡的应用开发流程。**



- x86 应用编译：编译主机应用以供在 x86 处理器上运行，该处理器使用 G++ 编译器来创建主机可执行文件。该主机程序可与 PL 区域中的内核进行交互。如需了解有关写入主机应用的更多信息，请参阅 第二部分：开发应用。如需了解有关编译主机应用的更多信息，请参阅 构建主机程序。
- PL 内核编译和链接：

  PL 内核经编译后即可在目标平台的 PL 区域内实现。PL 内核可编译到赛灵思对象格式 (XO) 文件中，编译可使用 Vitis 编译器 (`v++`)、Vitis HLS（适用于 C/C++ 内核）或 `package_xo` 命令（适用于 RTL 内核）来执行。如需了解有关内核编码的信息，请参阅 C/C++ 内核 或 RTL 内核。

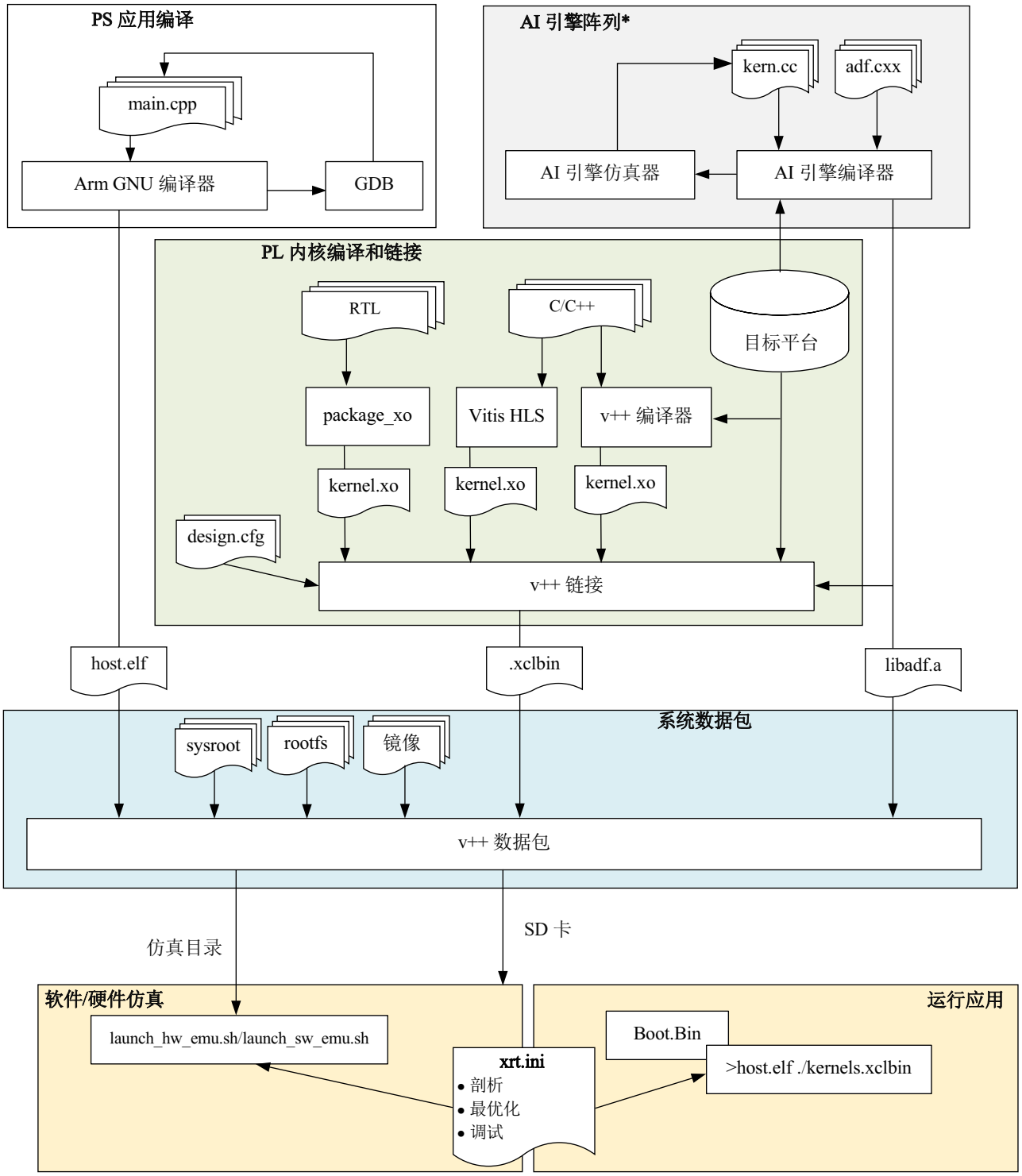
  Vitis 编译器还会将内核 XO 文件与硬件平台相链接，从而为应用创建器件可执行文件 (`.xclbin`)。如需了解更多信息，请参阅 构建器件二进制文件。

  赛灵思对象 (XO) 文件通过 `v++ --link` 命令与目标硬件平台相链接，以创建器件二进制文件 (`.xclbin`)，此文件将被加载到目标平台上的赛灵思器件中。
- 运行应用：对于 Alveo 数据中心加速器卡，`.xclbin` 文件是运行系统所必需的构建对象。运行应用时，可以运行软件仿真、硬件仿真或者在实际物理加速器平台上运行。如需了解更多信息，请参阅 运行应用硬件构建。
  - 当构建目标为软件或硬件仿真时，`emconfigutil` 命令可构建目标平台的仿真模型。Vitis 编译器会在器件二进制文件中生成内核的仿真 (simulation) 模型，运行此应用则会运行系统的此模型。如 构建目标 中所述，仿真 (emulation) 目标允许您在相对较快的周期内对设计进行构建、运行和迭代，并进行应用调试和性能评估。
  - 当构建目标为硬件系统时，目标平台是物理器件。Vitis 编译器会使用 Vivado Design Suite 生成 `.xclbin` 来运行综合与实现，并解决时序。运行应用会在硬件上运行系统。构建进程将自动执行以生成高质量的结果。但熟悉硬件的开发者可以在其设计进程中充分利用 Vivado 工具。

嵌入式处理器应用加速开发流程

下图描述了使用 Arm® 处理器和内核构建和运行应用所需的步骤，这些处理器和内核在 Versal ACAP、Zynq UltraScale + MPSoC 和 Zynq-7000 SoC 器件的可编程逻辑区域内运行。这些步骤总结如下，在这整篇文档中均可找到有关每个步骤的详细信息。

图 3：**Versal ACAP 和 Zynq UltraScale+ MPSoC 器件的应用开发流程**



*可选，仅适用于 Versal AI 引擎核系列。

X24705-082921

- PS 应用编译：编译主机应用以在 Cortex®-A72 或 Cortex-A53 核处理器上运行，此类处理器使用 GNU Arm 交叉编译器来创建 ELF 文件。主机程序会与器件的 PL 和 AI 引擎区域中的内核进行交互。如需了解有关写入主机应用的更多信息，请参阅 第二部分：开发应用。如需了解有关编译主机应用的更多信息，请参阅 构建主机程序。
- AI 引擎阵列（可选，仅供 Versal AI 引擎 Core 系列使用）：部分 Versal ACAP 器件将超长指令字 (VLIW) 处理器的 AI 引擎阵列与单指令流多数据流 (SIMD) 矢量单元整合在一起，后者专为计算密集型应用（例如，5G 无线和人工智能 (AI) 应用）而经过高度优化。AI 引擎 Graph 和内核是使用 Vitis 工具（例如，`aiecompiler` 和 `aiesimulator`）构建的，可集成到嵌入式加速器应用加速流程中，如《Versal ACAP AI 引擎编程环境用户指南》(UG1076) 中所述。
- PL 内核编译和链接：

  PL 内核经编译后即可在目标平台的 PL 区域内实现。PL 内核可使用 Vitis 编译器 (`v++`)、Vitis HLS（针对 C/C++ 内核）或 `package_xo` 命令（针对 RTL 内核）编译为赛灵思对象格式 (XO) 文件。如需了解有关内核编码的更多信息，请参阅 C/C++ 内核 或 RTL 内核。

  Vitis 编译器还会将内核 XO 文件与硬件平台相链接，从而为应用创建器件可执行文件 (`.xclbin`)。如需了解更多信息，请参阅 构建器件二进制文件。

  赛灵思对象 (XO) 文件通过 `v++ --link` 命令与目标硬件平台相链接，以创建器件二进制文件 (`.xclbin`)，此文件将被加载到目标平台上的赛灵思器件中。
- 系统封装：`v++ --package` 命令可用于收集配置和启动系统以及加载和运行应用（包括主机应用和 PL 内核二进制文件）所需的文件。此步骤会构建必要的封装以运行软件或硬件仿真和调试，或者创建 SD 卡以在硬件上运行您的应用。如需了解更多信息，请参阅 封装系统。
- 运行应用：运行应用时，可以运行软件仿真、硬件仿真或者在实际物理加速器平台上运行。在嵌入式处理器平台上运行应用与在数据中心加速器卡上运行应用不同。如需了解更多信息，请参阅 运行应用硬件构建。
  - 当构建目标为软件或硬件仿真时，QEMU 环境会对硬件器件进行建模。Vitis 编译器会在器件二进制文件中生成内核的仿真 (simulation) 模型，运行应用则会在系统的 QEMU 模型中运行。如 构建目标 中所述，仿真 (emulation) 目标允许您在相对较快的周期内对设计进行构建、运行和迭代，并进行应用调试和性能评估。
  - 当构建目标为硬件系统时，目标平台是物理器件。Vitis 编译器会使用 Vivado Design Suite 生成 `.xclbin` 来运行综合与实现，并解决时序。运行应用会在硬件上运行系统。构建进程将自动执行以生成高质量的结果，但熟悉硬件的开发者可以在其设计进程中充分利用 Vivado 工具。

构建目标

Vitis 编译器构建进程会生成主机程序可执行文件和 FPGA 二进制文件 (`.xclbin`)。FPGA 二进制文件的性质是由构建目标确定的。

- 当构建目标为软件或硬件仿真 (emulation) 时，Vitis 编译器会在 FPGA 二进制文件中生成内核的仿真 (simulation) 模型。这些仿真 (emulation) 目标允许您在相对较快的周期内构建、运行和迭代设计、调试应用并评估性能。
- 当构建目标为硬件系统时，Vitis 编译器会为硬件加速器生成 `.xclbin`，并使用 Vivado Design Suite 来运行综合与实现。它使用这些工具和预定义、经验证的设置来提供理想的结果质量。使用 Vitis 核开发套件不需要具备这些工具的知识；但是，熟悉硬件的开发者可以充分利用这些工具并使用所有可用功能特性来实现内核。

Vitis 编译器提供了 3 个不同的构建目标，其中 2 个仿真目标用于调试和确认目的，而默认硬件目标则用于生成实际的 FPGA 二进制文件：

- 软件仿真 (`sw_emu`)：主机应用代码和内核代码均编译为在主机处理器上运行。这样即可通过快速构建并运行 (fast build-and-run) 循环来实现迭代算法优化。此目标可用于确定语法错误、执行与应用程序一起运行的内核代码的源代码级调试，以及验证系统行为。

- 硬件仿真 (hw_emu)：内核代码编译到硬件模型 (RTL) 中，此模型在专用仿真器内运行。此构建和运行循环需要更长时间，但是提供详细的、周期精确的内核活动视图。此目标用于测试逻辑的功能，该逻辑将引入 FPGA 用于获取初始性能估算。
- 硬件 (hw)：内核代码编译到硬件模型 (RTL) 中，然后在 FPGA 上实现，从而生成将在实际 FPGA 上运行的二进制文件。

教程与示例

为帮助您快速开始使用 Vitis 核开发套件，您可在 https://github.com/xilinx/Vitis-Tutorials 上的以下仓库中查找教程、应用示例以及硬件内核。

- Vitis 应用加速开发流程教程：提供多种教程，通过这些教程可以教授有关工具流程和应用开发的具体概念。

  入门路径教程最适合新用户入门。

- Vitis 示例：包含许多示例，用于演示良好的设计实践、编码指南、常用应用的设计模式以及（最重要的）最优化技巧，从而最大程度提升应用性能。板载示例分为几种主要类别。每种类别都具有多种关键概念，采用适用的 OpenCL™ C 和 C/C++ 框架通过各不同示例来展示。所有示例都包含 Makefile 以支持软硬件仿真并在硬件上运行，还包含 README.md 文件，其中具有示例的详细解释。

现在您已掌握了 Vitis 核开发套件的各要素，并且已经了解了如何为加速编写和构建应用，下一步，让我们来回顾下解决您的设计问题的最佳方法。

XILINX

第 4 章

使用 Vitis 软件平台加速应用的方法论

引言

此内容侧重于数据中心应用和基于 PCIe® 的加速卡，但此处确立的概念一般同样适用于嵌入式应用。

加速：工业类比法

CPU、GPU 和可编程器件之间存在明显差异。了解这些差异是有效开发各类器件的应用和实现最优加速的关键。

CPU 和 GPU 都具有预定义的架构，具有固定数量的核、固定指令集和严格的存储器架构。GPU 通过核数量和通过采用 SIMD/SIMT 并行化来扩展性能。相比之下，可编程器件则是完全可定自定义的架构。开发者创建针对应用程序需求进行最优化的计算单元。性能是通过创建深度流水打拍的数据路径而不是使计算单元数量倍增来实现的。

将 CPU 视为一系列车间，每个车间都雇佣一名技术熟练的工人。这些工人可以使用各种通用工具，让他们可以构建几乎任何东西。每个工人一次制作一件产品，先后使用不同的工具将原料转化为成品。根据任务性质，此顺序变换过程可能需要许多步骤。车间是独立的，工人可以在不分心也无需考虑协调问题的情况下完成不同的任务。

GPU 也有车间和工人，且数量更多，而工人也更专业。他们只能使用专用工具，能做的工作更少，但能非常高效地完成这些工作。GPU 工人在重复执行少量相同任务时，以及当所有人同时执行相同工作时，效率最高。毕竟，有这么多不同的工人，给所有人相同的订单效率更高。

可编程器件将这种车间类比法引入工业时代。如果把 CPU 和 GPU 看作是由大量不同工人组成的多个小组，这些小组按顺序步骤将输入变换为输出，那么可编程器件就是具有组装线和传送带的工厂。经由各组工人按组装线分配任务，将原材料逐步变换为成品。每个工人重复执行相同的任务，并且半成品通过输送带从一个工人转移到另一个工人。这样即可实现更高的产量。

不同于 CPU 和 GPU 中的车间和工人，可编程器件的另一个主要区别在于工厂和组装线当前并不存在。为了让类比更形象，我们把可编程器件视为待开发的空地。这意味着器件开发者可以建造工厂、组装线和工作站，然后按所需任务对其进行自定义，而不是使用通用工具。就像空地大小一样，器件空间不是无限的，这对于器件中可建造的工厂数量和大小都造成了限制。因此，正确设计这些工厂的架构和配置是器件编程工艺的关键。

传统的软件开发是在预定义的架构上进行功能编程的。可编程器件开发重点在于对架构进行编程以实现期望的功能。

方法论概述

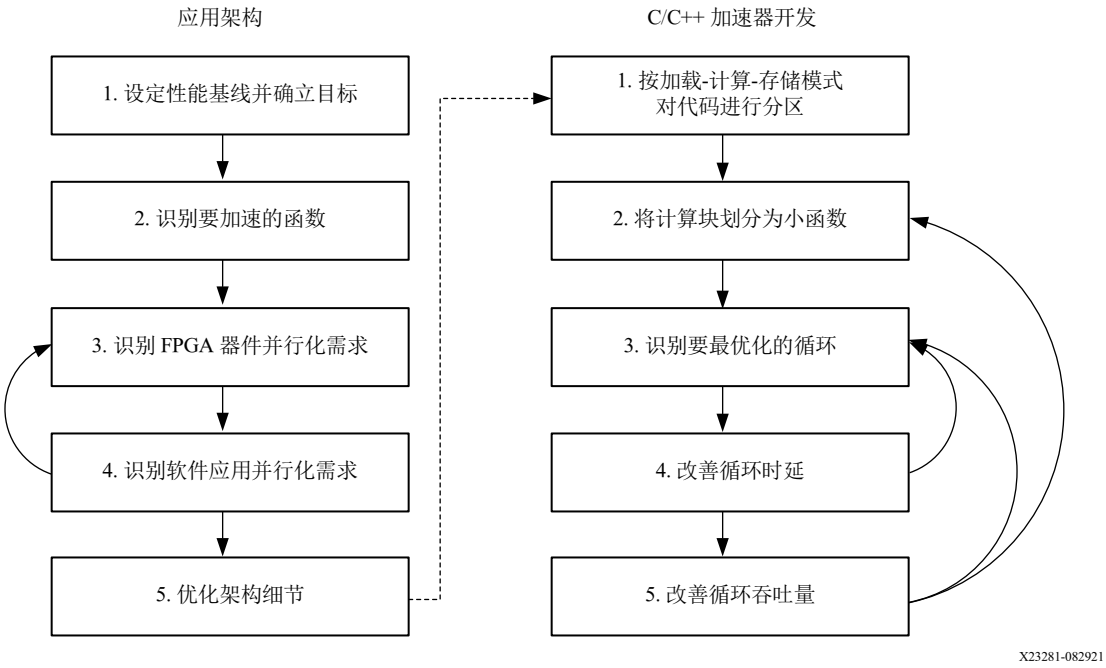
方法论包含 2 个主要阶段：

1. 构建应用
2. 开发 C/C++ 内核

在第一阶段，开发者制定出有关应用架构的关键决策，包括决定应映射到器件内核的软件函数、所需的并行度及其交付方式。

在第二阶段，开发者实现内核。这主要涉及构建源代码并应用所需的编译器编译指示来创建所需的内核架构并满足性能目标。

图 4：**方法论概述**



性能最优化是一个迭代过程。加速应用程序的初始版本可能无法产生最佳结果。本指南中描述的方法论是一个涉及持续性能分析和重复更改实现的方方面面的过程。

建议

熟练掌握 Vitis™ 统一软件平台编程和执行模型对于着手实施采用此方法论的工程至关重要。以下资源提供了使用 Vitis 软件平台提高工作效率所需的知识：

- 第二部分：开发应用
- GitHub 上的 Vitis 应用加速开发流程教程。

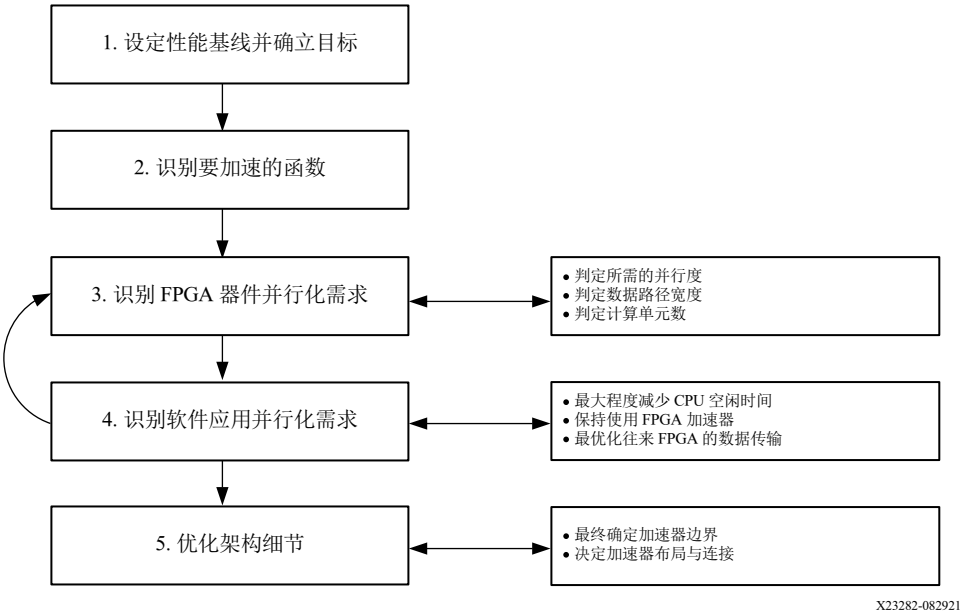
除了熟练掌握 Vitis 软件平台的关键知识外，熟悉以下主题同样有助于利用此方法论实现最佳结果：

- 应用域
- 软件加速原理
- 器件的概念、特性和架构
- 目标器件加速器卡和对应目标平台的功能特性
- 硬件实现中的并行度 (http://kastner.ucsd.edu/hlsbook/)

器件加速型应用的架构方法论

开始开发加速应用前，正确架构此应用至关重要。在此阶段，开发者需制定出有关应用架构的关键决策，并确定诸如应映射到器件内核的软件函数、所需的并行度以及交付方式等要素。

图 5：**应用架构方法论**



本节将介绍此过程中涉及的各个步骤。采用迭代方法完成此过程有助于优化分析，从而制定出更好的设计决策。

步骤 1：确立基线应用性能并确立目标

首先从测量运行时和吞吐量性能开始，以识别现有平台上当前正在运行的应用的瓶颈。应为整个应用程序（端到端）以及应用程序中的每个主要函数生成这些性能数据。最有效的方法是使用剖析工具（如 `valgrind`、`callgrind` 和 GNU `gprof`）来运行应用。这些工具生成的剖析数据可显示调用图形，包括所有函数的调用次数及其执行时间。这些数据能够为大部分后续分析进程确立基线。耗用执行时间最多的函数适合卸载到 FPGA 上并在其中进行加速。

测量运行时间

测量运行时间是软件开发的标准做法。这可以使用常见的软件分析工具（如 gprof）来完成，也可以使用定时器和性能计数器检测代码来完成。

下图显示了使用 gprof 生成的剖析报告示例。这些报告能够方便地显示函数调用次数以及耗用的时间量（运行时）。

图 6：**Gprof 输出示例**

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
70.45     25.54    25.54      256    99.76    99.76  F4(int*, int*, int*)
12.44     30.05     4.51      256    17.61    17.61  F2(int*, int*)
 9.91     33.64     3.59      256    14.03    14.03  F1(int*, int*, int*)
 7.83     36.48     2.84      256    11.08    11.08  F3(int*, int*)
 0.00     36.48     0.00      256     0.00   142.48  F(int*, int*)
```

测量吞吐量

吞吐量是处理数据的速率。要计算给定函数的吞吐量，请将函数处理的数据量除以函数的运行时间。

$$T_{SW} = \max(V_{INPUT}, V_{OUTPUT}) / \text{Running Time}$$

某些函数处理的数据量是预先确定的。在此情况下，可以使用简单的代码检查来确定此数据量。在某些其它情况下，数据量是可变的。在此情况下，使用计数器检测应用程序代码以动态测量数据量非常有用。

测量吞吐量与测量运行时间一样重要。虽然器件内核可以缩短整体运行时间，但它们对于应用吞吐量的影响更大。因此，将吞吐量视为主要最优化目标非常重要。

确定可达到的最大吞吐量

在大多数器件加速系统中，可达到的最大吞吐量受到 PCIe® 总线的限制。PCIe 性能则会受到诸多不同方面的影响，例如，主板、驱动程序、目标平台和传输大小等。预先运行 DMA 测试以测量 PCIe 传输的有效吞吐量，从而确定加速潜力的上限，例如 xbutil dma 测试。

图 7：**Alveo U200 数据中心加速器卡上 dmatest 结果样本**

```
$ xbutil dmatest
INFO: Found total 1 card(s), 1 are usable
Total DDR size: 65536 MB
Reporting from mem_topology:
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 11381.7 MB/s
Host <- PCIe <- FPGA read bandwidth  =  8358.9 MB/s
Data Validity & DMA Test on bank1
Host -> PCIe -> FPGA write bandwidth = 11235.3 MB/s
Host <- PCIe <- FPGA read bandwidth  =  7485.3 MB/s
INFO: xbutil dmatest succeeded.
```

由于系统受 I/O 制约，因此无法满足超出此上限吞吐量的加速目标。同样，在定义内核性能和 I/O 要求时，请记住这个上限。

确立总体加速目标

在开发早期确定加速目标是必要的，因为加速目标和基线性能之间的比率将推动分析和决策过程。

加速目标可以是硬目标或软目标。例如，实时视频应用程序可能有每秒处理 60 帧的硬性要求。数据科学应用程序则可能有比其它实现快 10 倍的软目标。

无论哪种方式，领域专业知识对于设置可实现且有意义的加速目标都是很重要的。

步骤 2：确定要加速的函数

确立性能基线后，下一步是确定应在器件中加速哪些函数。

提示：此时，请最大程度减少对现有代码的更改，以便能够在 FPGA 上快速生成有效的设计，并获取基线性能和资源数值。

选择要在硬件中加速的函数时，有两方面需要考量：

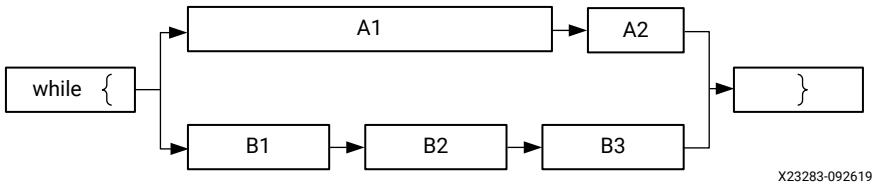
- 性能瓶颈：应用程序最常调用哪些函数？
- 加速潜力：这些函数是否具有加速潜力？

识别性能瓶颈

在纯顺序应用中，通过查看剖析报告即可轻松识别性能瓶颈。但是，大多数真实应用均为多线程应用，在寻找性能瓶颈时考虑并行度的影响非常重要。

下图显示了含两条并行路径的应用的性能剖析。每个矩形的宽度与每个函数的性能成比例。

图 8：**具有两条并行路径的应用**



并行化情境下的上述性能可视化表明，仅对两条路径中的一条进行加速并不能提升应用的总体性能。因为路径 A 和 B 重新收敛，所以它们相互依赖彼此才能完成。同样，即使把 A2 加速 100 倍，也不会对上方路径的性能产生重大影响。因此，该示例中的性能瓶颈是函数 A1、B1、B2 和 B3。

在寻找加速对象时，请考虑整个应用程序的性能，而不仅仅考虑单个函数。

确定加速潜力

属于软件应用瓶颈的函数并不一定能在器件中运行得更快。通常需要详细分析来准确判定给定函数的实际加速潜力。但是，可以根据一些简单的准则来评估函数是否具有硬件加速的潜力：

- 该函数的计算复杂度是多少？

  计算复杂度表示执行函数所需的基本计算的运算数量。在可编程器件中，加速是通过创建高度并行且深度流水打拍的数据路径来实现的。这就像是先前类比中的组装线。组装线越长，包含的工作站越多，则与车间内工人采用顺序步骤相比，就更高效。

所谓适合加速的函数，表示在此类函数中需要对每个输入样本按顺序执行一连串深度运算才能生成输出样本。

- 该函数的计算密集度是多少？

  函数的计算密集度是运算总数占输入和输出数据总量的比例。计算密集度较高的函数更适合用于加速，因为将数据移植到加速器的开销相对较低。

- 什么是函数的数据访问局部性剖析？

  数据复用、空间局部性和时间局部性的概念对于评估将数据移植到加速器的开销的可优化程度很有用。空间局部性反映了多个连续存储器访问操作之间的平均距离。时间局部性反映了程序执行期间任一地址的访问操作的平均数量。这些测量值越低越好，因为这样数据更便于缓存在加速器中，从而降低了对全局存储器进行代价不菲且可能冗余的访问的需求。

- 函数吞吐量与器件中可达到的最大吞吐量相比如何？

  器件加速的应用属于分布式多进程系统。总体应用的吞吐量不会超过其最慢的函数的吞吐量。这种瓶颈的本质因应用而异，可能源于系统的任何方面：I/O、计算或数据迁移。开发者可以通过将最慢的函数的吞吐量除以选定函数的吞吐量来确定最大加速潜力。

  $$\text{最大加速潜力} = T_{Min} / T_{SW}$$

  在 Alveo 数据中心加速器卡上，PCIe 总线会对数据传输施加吞吐量限制。虽然此限制可能并非应用的真正瓶颈，但可能确立上限，从而用于早期估算。例如，假设 PCIe 吞吐量为 10 GB/s 且软件吞吐量为 50 MB/s，那么此函数的最大加速因数为 200x。

这 4 项标准并不意味着保证能实现加速，但确实可作为可靠的工具，用于识别可在器件上加速的正确函数。

步骤 3：确定器件并行化需求

识别要加速的函数并明确总体加速目标之后，下一步是确定满足目标所需的并行化级别。

这里同样适用工厂类比来理解内核内部可行的并行化。

如前文所述，组装线允许逐步处理输入和同步处理输入。在硬件中，这种并行化操作称为流水打拍。组装线上的工作站数量对应于硬件流水线中的阶数。

内核内部的另一个并行维度是同时处理多个样本的能力。这就像在输送带上同时放置不只一个样本，而是多个样本。为了适应这种需求，需定制组装线工作站以并行处理多个样本。这样即可有效定义内核中的数据路径的宽度。

通过增加组装线的数量可以进一步扩展性能。这可以通过在工厂中布置多条组装线来实现，也可以通过构建多个相同的工厂并在每个工厂中布置一条或多条组装线来实现。

开发者需要确定哪种并行化技术组合在满足加速目标方面最有效。

估算无并行化情况下的硬件吞吐量

没有任何并行化情况下的内核吞吐量的估算方法如下：

$$T_{HW} = \text{Frequency} / \text{Computational Intensity} = \text{Frequency} * \max(V_{INPUT}, V_{OUTPUT}) / V_{OPS}$$

Frequency 表示内核的时钟频率。该值由目标加速平台或目标平台来确定。例如，Alveo U200 数据中心加速器卡上的最大内核时钟频率为 300 MHz。

如前文所述，函数的计算密集度是运算总数占输入和输出数据总量的比例。以上公式清晰表明了含大量运算和少量数据的功能更适合用于加速。

确定所需的并行度

根据以上公式完成计算后，即可估算初始硬件/软件性能比。

$$\text{Speed-up} = T_{HW}/T_{SW} = F_{max} * \text{Running Time} / V_{ops}$$

在没有任何并行化的情况下，初始速度很有可能小于 1。

下一步，计算满足性能目标所需的并行度：

$$\text{Parallelism Needed} = T_{Goal} / T_{HW} = T_{Goal} * V_{ops} / (F_{max} * \max(V_{INPUT}, V_{OUTPUT}))$$

这种并行度可以通过各种方式实现：拓宽数据路径、使用多个引擎，以及使用多个内核实例。随后，开发者应根据自身需求及其应用特征来判定最佳组合。

确定数据路径应并行处理的样本数量

一种可能性是通过创建更宽的数据路径和并行处理更多样本来加速计算。某些算法很适合这种方法，而其它算法则不然。重要的是要了解算法的性质以确定此方法是否有效，如果可行，则应确定为满足性能目标而需要并行处理的样本数量。

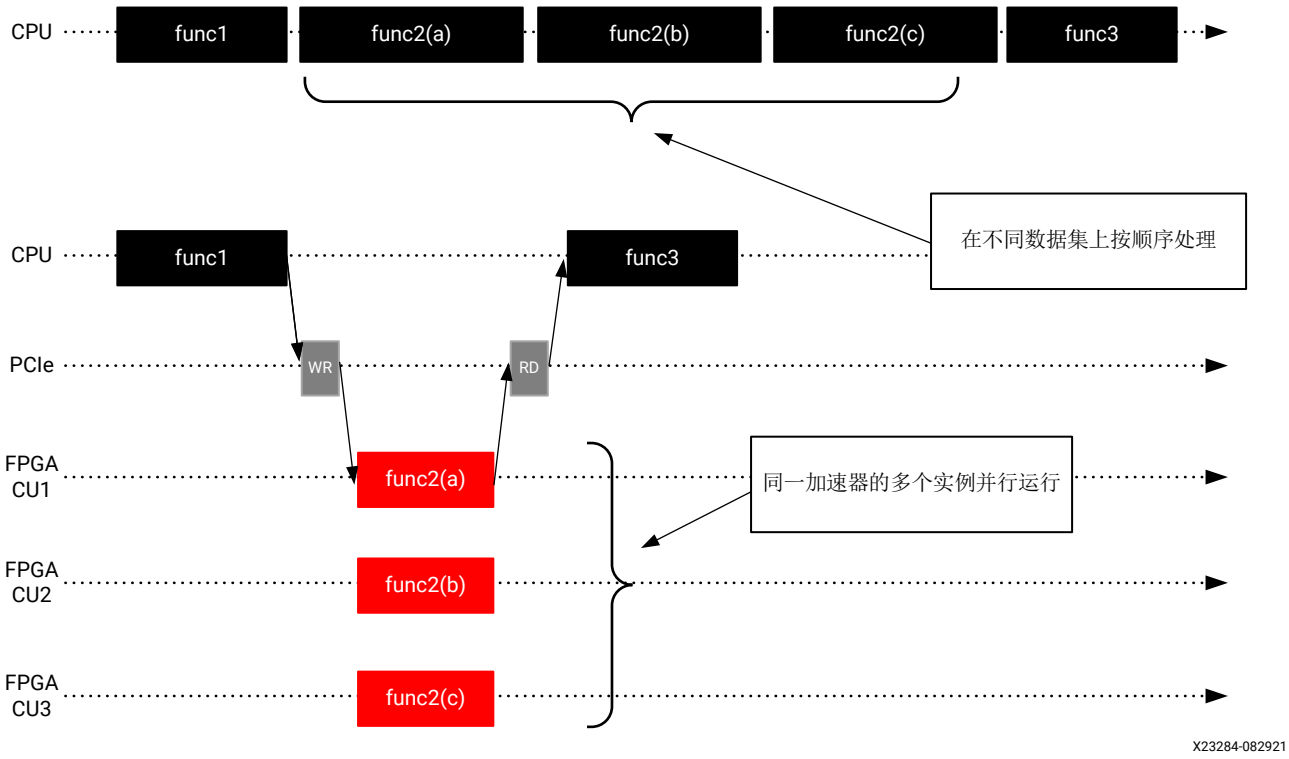
使用更宽的数据路径并行处理更多样本可减少加速函数的时延（运行时间），从而提高性能。

确定器件中可以例化和应该例化的内核数量

如果数据路径无法并行化（或无法充分并行化），请考虑添加更多内核实例，如 创建内核的多个实例 中所述。这通常被称为使用多个计算单元 (CU)。

添加更多内核实例可允许对目标函数并行执行更多次调用，从而提升应用性能，如下所示。多个数据集将由不同的实例同时处理。应用性能与实例数量呈线性关系，前提是主机应用可以使内核保持忙碌。

图 9：**使用多个计算单元提高性能**



如使用多个计算单元教程中所示，Vitis 技术通过添加更多实例来简化性能的缩放调整。

至此，开发者应该能够更准确地理解硬件中满足性能目标所需的并行度，并且能够通过数据路径宽度与内核实例的组合来理解实现此并行度的方式。

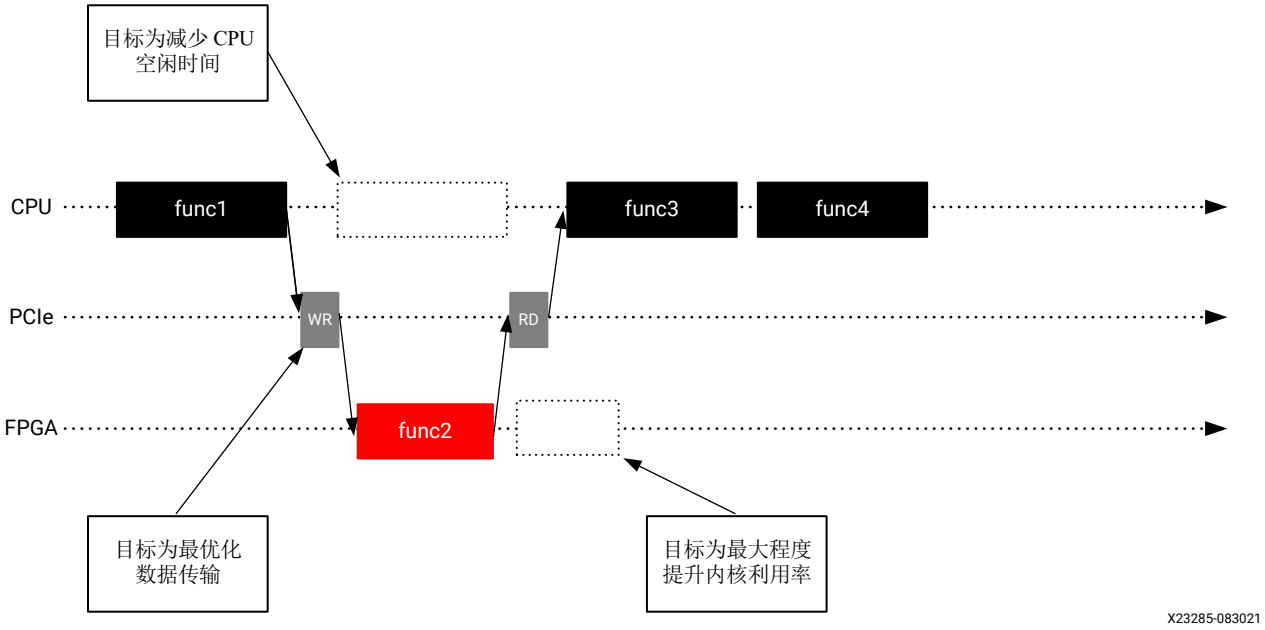
步骤 4：确定软件应用程序并行化需求

虽然硬件器件及其内核旨在提供潜在的并行化，但必须合理设计软件应用以利用这种潜在的并行化。

软件应用中的并行化表示主机应用程序执行下列操作的能力：

- 在器件内核运行时，最大程度减少空闲时间并执行其它任务。
- 使器件内核保持处于活动状态，以便尽早并尽量频繁执行新计算。
- 最优化往来器件的数据传输。

图 10：**软件最优化目标**



以工厂和组装线为例，主机应用就像是企业总部，始终忙于规划下一代产品，而工厂则负责制造当代产品。

同样，总部必须协调货物往返工厂的运输并将其需求发送给工厂。如果物流部门不给工厂发送原材料或用于创建产品的蓝图，那么建造这么多工厂有什么意义呢？

在运行器件内核时最大程度减少 CPU 空闲时间

所谓器件加速，即将某些计算从主处理器卸载到器件中的内核。在纯顺序模型中，应用将保持空闲，并等待结果就绪，然后才恢复处理，如上图所示。

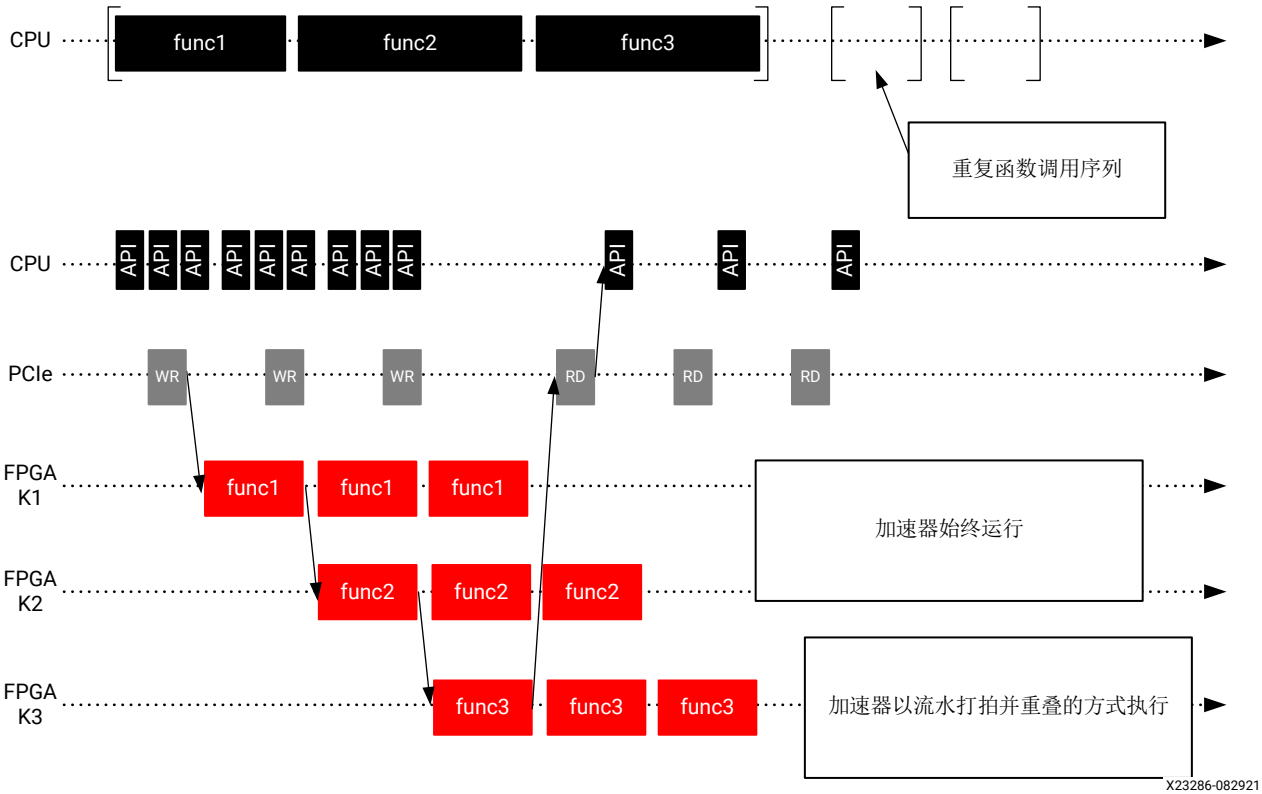
设计软件应用程序时应避免这种空闲周期。首先确定不依赖于内核结果的应用程序部分。然后构造应用程序，以便这些函数可以在主机上与器件中运行的内核并行执行。

使器件内核始终保持在使用中状态

内核可能存在于器件中，但仅在应用请求时，内核才会运行。为了尽可能提高性能，请将应用设计为使内核保持繁忙。

从概念上讲，这是通过在当前请求完成之前发出后续请求来实现的。这将导致流水打拍和重叠执行，从而以最优方式使用内核，如下图所示。

图 11：**加速器的流水打拍执行**



在此示例中，原始应用重复调用 func1、func2 和 func3。对于这 3 个函数，已创建了对应的内核（K1、K2 和 K3）。简单的实现会按顺序运行这 3 个内核，就像原始软件应用一样。但这意味着每个内核只有三分之一的时间处于活动状态。更好的方法是构建软件应用程序，使其能够向内核发出流水打拍请求。这样在 K1 开始处理新数据集的同时，K2 即可开始处理 K1 的第一项输出。通过这种方法，即可以最大利用率持续运行这 3 个内核。

如需了解有关软件流水打拍的更多信息，请参阅 Vitis 应用加速开发流程教程。

对往来器件的数据传输进行最优化

在加速应用中，数据必须从主机传输到器件，在基于 PCIe 的应用案例中尤其如此。这样会产生时延，时延对于应用的总体性能来说可能代价高昂。

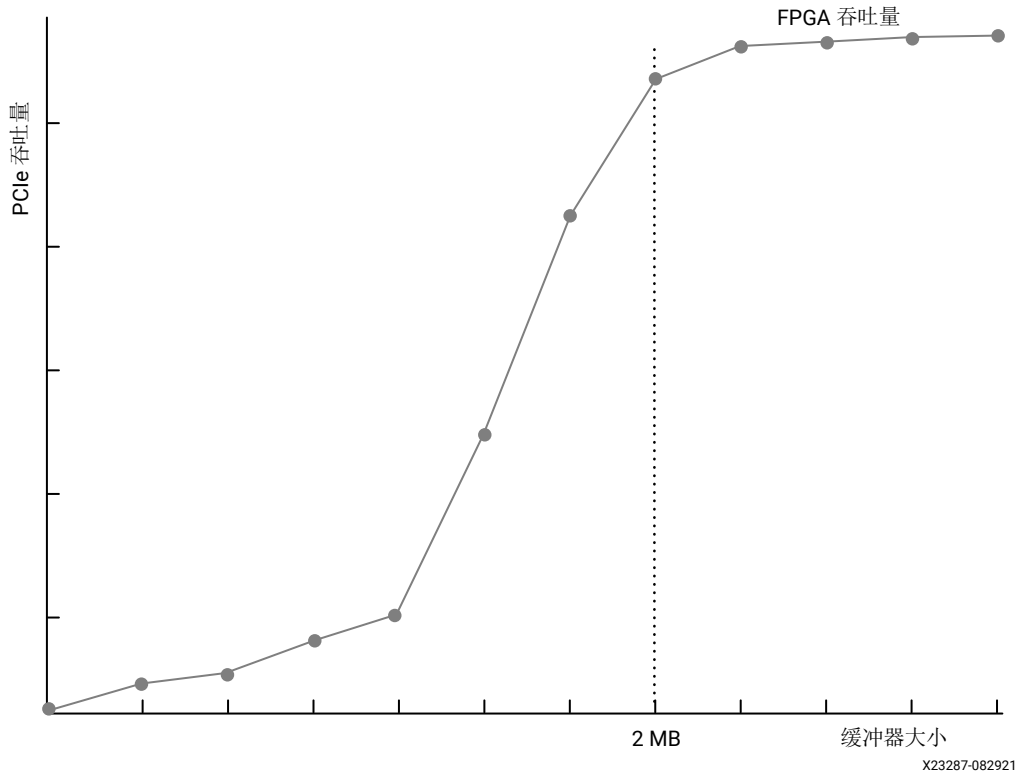
数据需在正确的时间进行传输，否则如果内核必须等待数据变为可用，就会对应用性能造成负面影响。因此，在内核需要之前传输数据非常重要。这是通过将数据传输与内核执行重叠来实现的，如 使器件内核始终保持在使用中状态 中所述。如前图中的顺序所示，此方法支持隐藏数据传输的时延开销，并避免出现内核不得不等待数据就绪的状况。

最优化数据传输的另一种方法是传输最佳大小的缓冲器。如下图所示，根据传输的缓冲器大小，PCIe 的有效吞吐量差异巨大。缓冲器越大，吞吐量越大，因此就更能确保加速器始终具有操作数据并且不浪费周期。通常，传输的数据最好不少于 1 MB。提前运行 DMA 测试对于找到最佳缓冲器大小非常有用。并且，在确定最佳缓冲器大小时，请考虑缓冲器过大对于资源利用率和传输时延的影响。

最优化数据传输的另一种方法是传输最佳大小的缓冲器。根据传输的缓冲器大小，数据传输的有效吞吐量差异巨大。缓冲器越大，吞吐量越大，因此就更能确保加速器始终具有操作数据并且不浪费周期。

如上图所示，在基于 PCIe 的系统上，通常，传输的数据最好不少于 1 MB。提前使用 `xbutil` 实用工具运行 DMA 测试对于找到最佳缓冲器大小非常有用。如需了解更多信息，请参阅 dmatest。

图 12：**PCIe 传输的性能是缓冲器大小的函数**



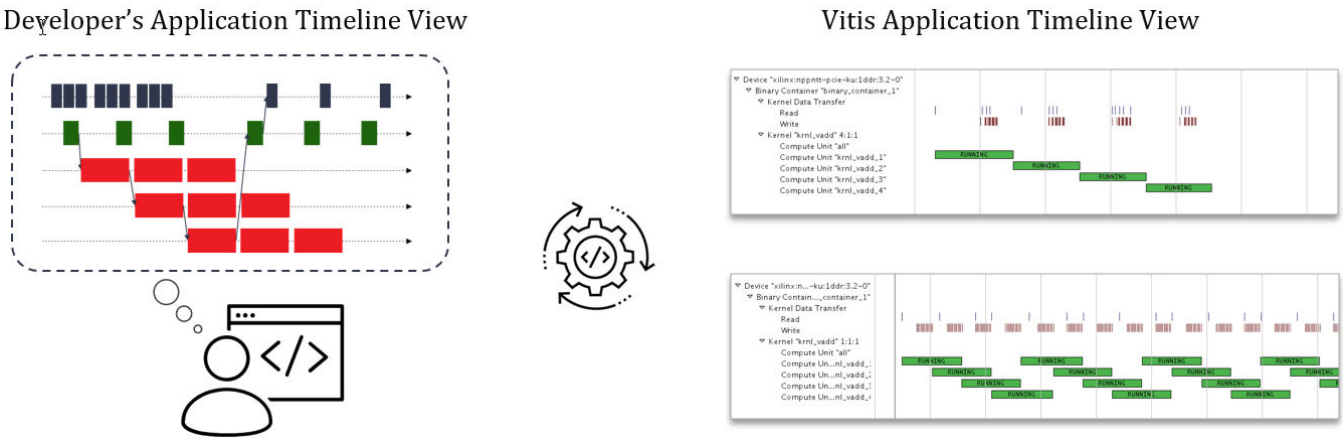
赛灵思建议，在公共缓冲器中集合多组数据，以实现尽可能高的吞吐量。

将目标应用时间线概念化

现在，对于需要加速的函数、为满足性能目标所需的并行化措施以及应用交付方式，开发者应该已经有了较为深刻的理解。

在此情况下，以期望的应用时间线的形式来总结这些信息是很有用的。应用时间线序列（如 使器件内核始终保持在使用中状态 中所示）是表示应用运行时实际性能和并行度的有效方式。这些序列可以用来演示应用如何调动构建到架构中的潜在并行度。

图 13：**应用时间线**



Vitis 软件平台可基于实际应用运行来生成时间线视图。如果开发者有自己预测的目标时间线，可将其与实际结果进行比较，识别潜在问题，并基于最优结果进行迭代和收敛，如上图所示。

步骤 5：优化架构细节

在继续开发应用程序及其内核之前，最后要做的是根据到目前为止做出的顶层决策来优化和推导二级架构细节。

完成内核边界

如前所述，可以通过创建多个内核实例（计算单元）来提高性能。但是，添加 CU 存在 I/O 端口、带宽和资源方面的成本。

在 Vitis 软件平台流程中，内核端口最大位宽为 512 位（64 个字节），并存在器件资源方面的固定成本。最重要的是，目标平台给可供使用的端口最大数量设置了限制。请谨记这些限制，并以最佳方式使用这些端口及其带宽。

利用多个计算单元进行扩展的另一种方法是通过在内核内部添加多个引擎来进行扩展。这种方法能够以添加更多 CU 相同的方式来提升性能：即，由内核内部的不同引擎来并发处理多个数据集。

将多个引擎置于相同内核内部能够充分利用内核的 I/O 端口的带宽。如果数据路径引擎不需要端口的完整带宽，那么在内核引擎内部添加更多引擎比创建多个含单一引擎的 CU 更有效。

将多个引擎置于单个内核中还能够减少连接到全局存储器的需仲裁的端口数量和传输事务数量，从而提升有效带宽。

另一方面，这种变换需要对内核内部的显式 I/O 多路复用行为进行编码。这是开发者需要做出的妥协。

确定内核布局和连接

最终确定内核边界后，开发者即可明确要例化的内核数量，以及需要连接到全局存储器资源的端口数量。

此时，了解目标平台的功能以及可用的全局存储器资源就显得至关重要。例如，Alveo™ U200 数据中心加速器卡具有 4 x 16 GB 的 DDR4 存储体和 3 x 128 KB 的 PLRAM 存储体，分布在 3 个超级逻辑区域 (SLR) 内。如需了解更多信息，请参阅 Vitis 软件平台版本说明。

如果把内核比作工厂，那么全局存储体就是货物进出工厂的仓库。SLR 类似于独立的工业区，其中已存在仓库，并且可建造工厂。虽然可以将货物从一个区域的仓库转移到另一个区域的工厂，但这可能会增加延迟和复杂性。

使用多个 DDR 有助于平衡数据传输负载并提高性能。但是，这附带有成本，因为每个 DDR 控制器都会占用器件资源。在决定如何将内核端口连接到存储体时，请权衡考虑这些注意事项。如 将内核端口映射到存储器 中所述，这些连接是通过简单的编译器开关来建立的，因此可以很方便地根据需要更改配置。

完善架构细节后，开发者应已掌握了开始实现内核以及最终组装整个应用所需的所有信息。

C/C++ 内核的开发方法论

Vitis 软件平台支持以 C/C++ 或 RTL（Verilog、VHDL 或 System Verilog）建模的内核。此方法指南适用于 C/C++ 内核。如需了解有关开发 RTL 内核的详细信息，请参阅 RTL 内核。

在架构定义阶段，应该已经确定了以下关于最佳应用程序性能的关键内核要求：

- 吞吐量目标
- 时延目标
- 数据路径宽度
- 引擎数
- 接口带宽

这些要求推动了内核开发和最优化过程。因为整体应用程序性能取决于满足指定吞吐量的每个内核，所以主要目标是实现内核吞吐量目标。

因此，内核开发方法遵循吞吐量驱动的方法，并从外向内开展。这种方法有两个阶段，如下图所示：

1. 定义和实现内核的宏架构
2. 编码和最优化内核的微架构

开始内核开发进程之前，必须了解功能、算法和架构间的差异及其与内核开发进程的关系。

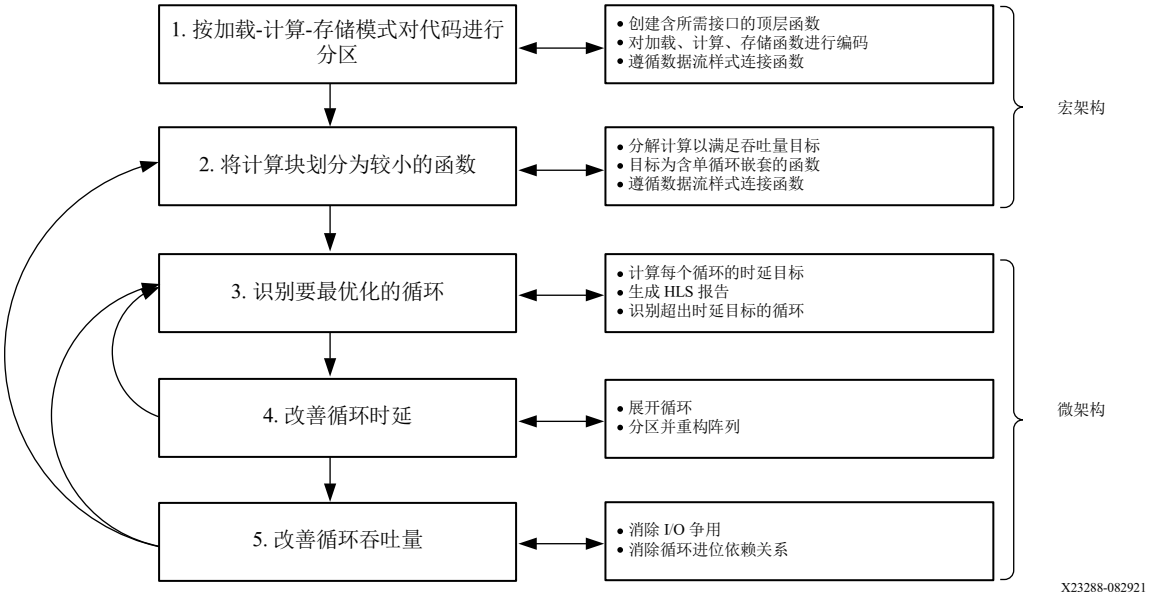
- 功能是输入参数与输出结果之间的数学关系。
- 算法是执行特定功能的一系列步骤。任一给定功能可使用各种不同算法来执行。例如，排序功能可使用“快速排序 (quick sort)”算法或“冒泡排序 (bubble sort)”算法来实现。
- 在此上下文中，架构表示算法底层硬件实现的特性。例如，特定排序算法实现时，并行执行的比较器可多可少、可包含 RAM 或基于寄存器的存储器等。

您必须理解，Vitis 编译器会根据以 C/C++ 语言编写的算法来生成最优化的硬件架构。但它并不会将特定算法变换为其它算法。即使软件程序可自动转换（或综合）为硬件，但要实现可接受的结果质量 (QoR)，仍需要额外工作（例如，重写软件）以帮助 HLS 工具实现期望的性能目标。

因此，由于算法直接影响数据访问局部性以及计算并行性的可能性，您所选择的算法就会对可实现的性能产生重大影响，比编译器的功能或者用户指定的编译指示产生的影响更大。为此，您需要了解正确编写软件的最佳实践，以确保在 FPGA 器件上正常执行软件。在接下来的几个章节内，将探讨如何首先识别部分宏观级别架构最优化以明确程序结构，然后聚焦更细化的微观级别架构最优化来实现性能目标。请参阅《Vitis 高层次综合用户指南》(UG1399) 中的《软件方法论》以获取更多信息。

以下方法论假定您已明确了适用于要加速的功能的算法。

图 14：**内核开发方法论**



关于高层次综合编译器

在开始内核开发进程之前，开发者应熟悉高层次综合 (HLS) 概念。HLS 编译器将 C/C++ 语言代码转换为 RTL 设计，随后映射到器件互连结构。

HLS 编译器比标准软件编译器的限制更多。例如，存在不受支持的结构，包括：系统函数调用、动态存储器分配和递归函数。如需了解更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399)。

更重要的是，请始终牢记 C/C++ 源代码的结构对生成的硬件实现的性能有很大影响。此方法指南将帮助您构建代码以满足应用程序吞吐量目标。如需了解有关内核编程的具体信息，请参阅 C/C++ 内核。

验证考虑因素

本指南中描述的这种方法本质上是迭代，涉及连续的代码修改。赛灵思建议在每次修改后验证代码。可使用标准软件验证方法或者使用 Vitis 集成设计环境 (IDE) 软件或硬件仿真流程来执行代码验证。在任何一种情况下，请确保您的测试提供足够的覆盖率和验证质量。

步骤 1：将代码分区为加载 - 计算 - 存储模式

内核本质上就是定制数据路径（专为期望的功能而优化）和关联的数据存储与移动网络。此数据存储与移动网络也称为内核的存储器架构或存储器层级，负责以尽可能高效的方式将数据移入和移出内核，以及通过定制数据路径来进行移动。

了解内核访问全局存储器代价不菲且宽带有限后，审慎规划内核的这方面操作就显得尤为重要。

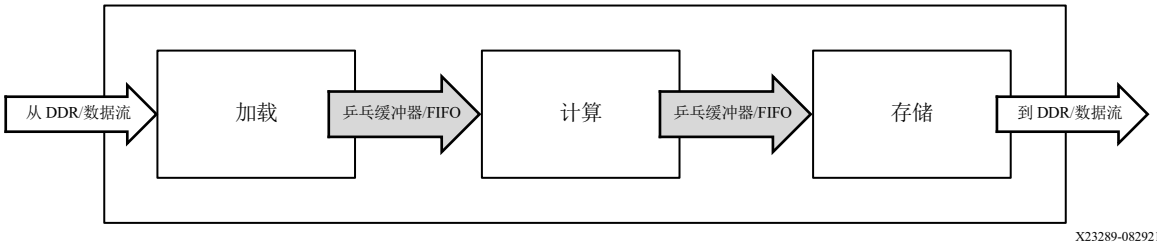
为此，内核开发方法论的第一步就是要求按“加载 - 计算 - 存储”模式来构造内核代码。

这意味着创建带有下列元素的顶层函数：

- 与所需内核接口匹配的接口参数。

- 三个子函数：加载、计算和存储。
- 本地阵列或 `hls::stream` 变量，用于在这些函数之间传递数据。

图 15： **“加载 - 计算 - 存储”模式**



以这种方式构造的内核代码可支持任务级流水打拍（也称为 HLS 数据流）。此编译器最优化可使设计中的每个函数同时运行，从而创建任务并发运行的流水线。这是我们工厂组装线的前提条件，这种结构是实现和维持所需吞吐量的关键。如需了解有关 HLS 数据流的更多信息，请参阅 数据流最优化。

加载函数负责将内核外部的数据（即，全局存储器）移动到内核内部的计算函数。此函数不执行任何数据处理，而是侧重于有效的数据传输，包括必要时的缓冲 (buffering) 和缓存 (caching)。

顾名思义，计算函数是完成所有处理的地方。在开发流程的这个阶段，计算函数的内部结构并不重要。

存储函数是加载函数的镜像操作。它负责将数据移出内核，获取计算函数的结果并将它们传输到内核外的全局存储器。

创建满足性能目标的加载 - 计算 - 存储结构首先要在内核中设计数据流。需要考虑的因素包括：

- 数据如何从内核外部流入内核?
- 内核处理这些数据的速度有多快?
- 如何将处理后的数据写入内核的输出?

理解和可视化数据移动模块框图将有助于分区和构建内核中的不同函数。

在 GitHub 仓库 Vitis 示例中可以找到有关加载 - 计算 - 存储模式的工作示例。

使用所需的接口创建顶层函数

Vitis 技术根据顶层函数的参数来推断内核接口。因此，首先请编写一个内核顶层函数，其参数与所需的接口相匹配。

输入参数应作为标量传递。输入和输出数据块应作为指针传递。应使用编译器编译指示来完成接口定义。欲知详情，请参阅 接口。

对加载函数和存储函数进行编码

内核与全局存储器之间的数据传输对整体系统性能有很大影响。如果无法正常进行数据传输，就会限制内核运行速度。因此，最优化加载函数和存储函数就显得尤为重要，因为这样才能以有效的方式将数据移入和移出内核并以最佳方式馈送给计算函数。

全局存储器中的数据布局与软件应用程序中的数据布局相匹配。在编写加载函数和存储函数时，此布局必须已知。反之，如果某个数据布局更有利于将数据移入和移出内核，则可以在软件应用程序中调整缓冲器布局。无论哪种方式，内核开发者和应用开发者都需要就缓冲器和全局存储器中数据的组织方式达成一致。

以下是提高进出内核的数据传输效率的指南。

将端口宽度与数据路径宽度匹配

在 Vitis 软件平台中，内核端口位宽最高为 512 位，这表示内核在每个时钟周期内每个端口最多可以读取或写入 64 个字节。

赛灵思建议将内核端口宽度与计算函数中的数据路径宽度相匹配。例如，如果数据路径需要并行处理 16 个字节以满足目标吞吐量，那么端口位宽应设为 128 位以允许并行读取和写入 16 个字节。

在某些情况下，最好访问接口的各位元的完整宽度，即使数据路径无此需求也是如此。当许多内核尝试访问同一个全局存储体时，这有助于减少争用。但是，这通常会导致需要增加内核中的额外缓冲和内部存储器资源。

使用突发传输

对全局存储器的第一次读取或写入请求代价高昂，但后续的连续操作则不然。以突发方式传输数据可隐藏存储器访问时延，并提高存储器控制器的带宽使用率和效率。

除非绝对需要，否则应始终避免对全局存储器执行原子访问。加载和存储函数应编码为始终推断突发传输事务。这可通过使用 `memcpy` 操作来实现（如 GitHub 示例的 `vadd.cpp` 文件中所示），也可以通过创建紧凑的 `for` 循环并按顺序访问所有必需的值来实现（如 第二部分：开发应用 的 接口 中所述）。

最大程度减少全局存储器的数据传输次数

由于访问全局存储器可能会给应用程序增加大量时延，因此请仅执行必要的传输。

准则是仅读写必要的值，并且仅执行一次读写。如果计算函数必须多次使用相同的值，那么请在本地缓冲该值，而不是重复从全局存储器读取该值。完成适当的缓冲和缓存结构编码可能是实现吞吐量目标的关键。

编码计算函数

计算函数是完成所有实际处理的地方。该方法的第一步专注于使顶层结构正确并最优化数据移动。优先考虑的是具有正确接口的函数并确保功能正确。以下部分重点介绍计算函数的内部结构。

连接加载、计算和存储函数

使用标准 C/C++ 变量和阵列来连接顶层接口以及加载、计算和存储函数。这对于使用 `hls::stream` 类来对数据流传输行为进行建模也很有用。

数据流传输是一种数据传输形式，其中数据样本从第一个样本开始按顺序发送。数据流传输不需要地址管理，可以使用 FIFO 实现。如需了解有关 `hls::stream` 类的更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399) 中的使用 HLS 数据流传输。

连接函数时，请使用 HLS 编译器所需的规范形式。如需了解更多信息，请参阅 数据流最优化。这有助于编译器使用数据流最优化来构建高吞吐量的任务集。主要建议包括：

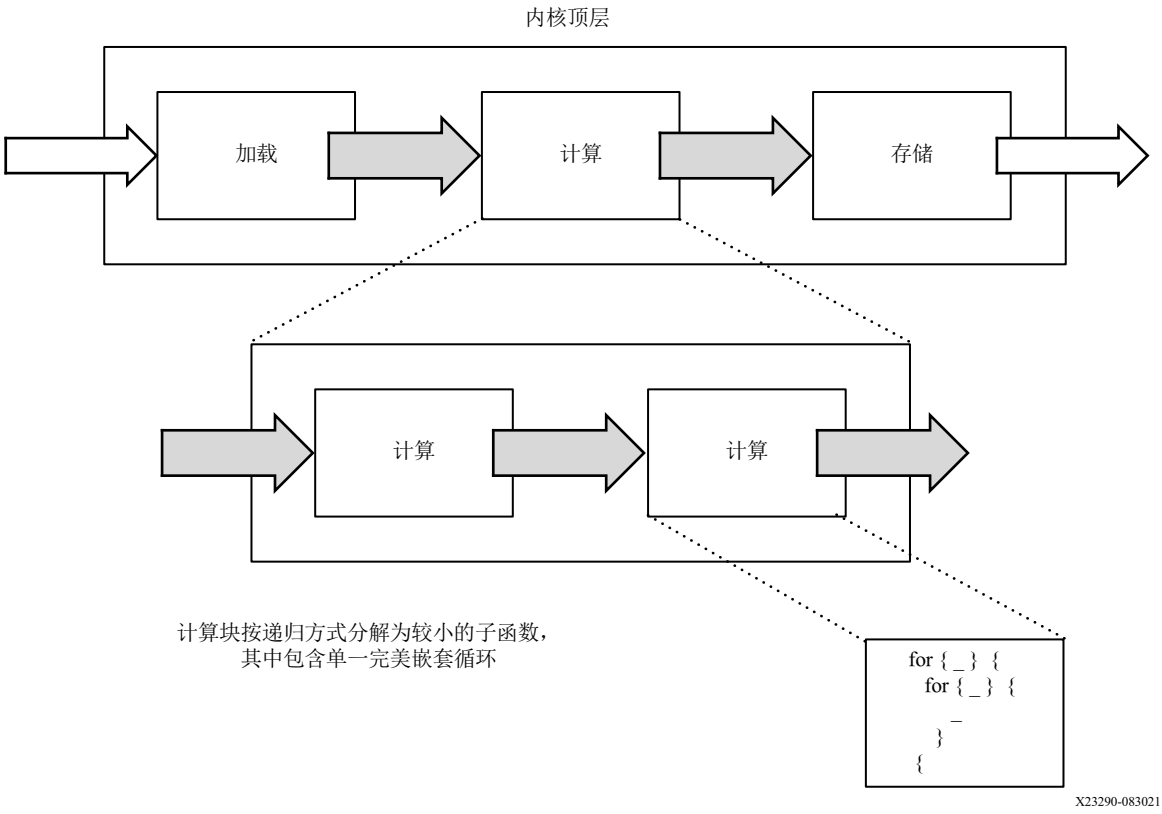
- 应仅正向传输数据，尽可能避免反馈。
- 每个连接应该有一个生产者和一个使用者。
- 只有加载和存储函数才能访问内核的主接口。

至此，开发者已创建内核的顶层函数，并已对接口和加载/存储函数进行了编码，下一个目标是按所需吞吐量在内核中移动数据。

步骤 2：将计算块分区为较小的函数

下一步是优化主计算函数，将其分解为一系列较小的子函数，如下图所示。

图 16：**计算块子函数**



分解以识别吞吐量目标

在以此方法所创建的数据流系统中，最慢的任务将成为瓶颈。

$$\text{Throughput(Kernel)} = \min(\text{Throughput}(\text{Task}_1), \text{Throughput}(\text{Task}_2), \ldots, \text{Throughput}(\text{Task}_N))$$

因此，在分解进程中，请始终牢记内核吞吐量目标，并评估每个子函数是否能够满足此吞吐量目标。

在此方法论的如下步骤中，开发者将通过运行 Vitis HLS 编译器获得实际吞吐量数据。如果无法改进这些结果，开发者将不得不迭代并进一步分解计算阶段。

以具有单循环嵌套的函数为目标

一般而言，如果函数中包含顺序循环，则这些循环在 HLS 编译器生成的硬件实现中按顺序执行。这通常是不可取的，因为顺序执行会妨碍吞吐量。

但是，如果将这些顺序循环推送到顺序函数中，则 HLS 编译器可以应用数据流最优化并生成允许对每个任务进行流水打拍和重叠执行的实现。如需了解有关数据流最优化的更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399) 中的利用任务级别并行化：数据流最优化。

在此分区和优化过程中，将顺序循环放入单个函数中。理想情况下，最低级别的计算块应该只包含一个完美嵌套的循环。如需了解有关循环的更多信息，请参阅 循环。

使用数据流“规范形式”连接计算函数

在分解计算函数时，适用与顶层函数内的连接相同的规则。以建立前馈连接为目标，并为每个连接变量设置单个生产者和使用者。如果变量必须由多个函数使用，则应明确复制它。

将数据块从一个计算块移动到另一个计算块时，开发者可以选择使用阵列或 hls::stream 对象。

使用阵列需要更少的代码更改，这通常是在分解过程中取得进展的最快方法。但是，使用 hls::stream 对象可能会导致设计使用更少的存储器资源并缩短时延。它还有助于开发者了解数据如何在内核中移动，在最优化吞吐量时这始终是需要掌握的重要信息。

使用 hls::stream 对象通常很有用，但由开发者决定将阵列转换为数据流的最佳时机。某些开发者会早早进行转换，某些开发者则在最后进行转换，将其作为最终的最优化步骤。此操作也可以使用 pragma HLS dataflow 来完成。

在此阶段，维护内核架构的图形表示对于推理数据相依性、数据移动、控制流和并发度非常有用。

步骤 3：确定需要最优化的循环

此时，开发者已经创建了一个数据流架构，其中包含数据运动和处理函数，旨在维持内核的吞吐量目标。下一步是确保每个处理函数的实现方式均能提供期望的吞吐量。

如前所述，函数吞吐量的测量方式是将处理的数量除以函数的时延或运行时间。

$$T = \max(V_{INPUT}, V_{OUTPUT}) / \text{时延}$$

在本方法论中所述的“由外到内”分解过程的该阶段，函数所耗用和生成的数据量和目标吞吐量都应已知。这样，开发者即可轻松得到每个函数的时延目标。

Vitis HLS 编译器生成有关函数和循环的吞吐量和时延的详细报告。确定目标时延后，HLS 报告即可用于确定哪些函数和循环不满足其时延目标且需要注意，如 HLS 报告 中所述。

循环的时延可按如下方式计算：

$$Latency_{Loop} = (Steps + II \times (TripCount - 1)) \times ClockPeriod$$

其中：

- Steps：单循环迭代的持续时间，以时钟周期数来度量
- TripCount：循环中的迭代次数。
- II：启动时间间隔，两次连续迭代开始之间的时钟周期数。当循环未流水打拍时，其 II 等于步数 (Steps)。

假设时钟周期已给定，有三种方法可以减少循环的时延，从而提高函数的吞吐量：

- 减少循环中的步数（缩短每次迭代的执行时间）。
- 减少循环次数，以便减少循环执行的迭代数量。
- 缩短启动时间间隔，以便提升循环迭代启动频率。

假设循环次数远大于步数，将 II 或循环次数减半可足以使循环的吞吐量加倍。

了解此信息是最优化循环并使时延超过其目标的关键。默认情况下，Vitis HLS 编译器将尝试以尽可能低的 II 生成循环实现。首先看看如何通过减少循环次数或步数来改善时延。如需了解更多信息，请参阅 循环。

步骤 4：改善循环时延

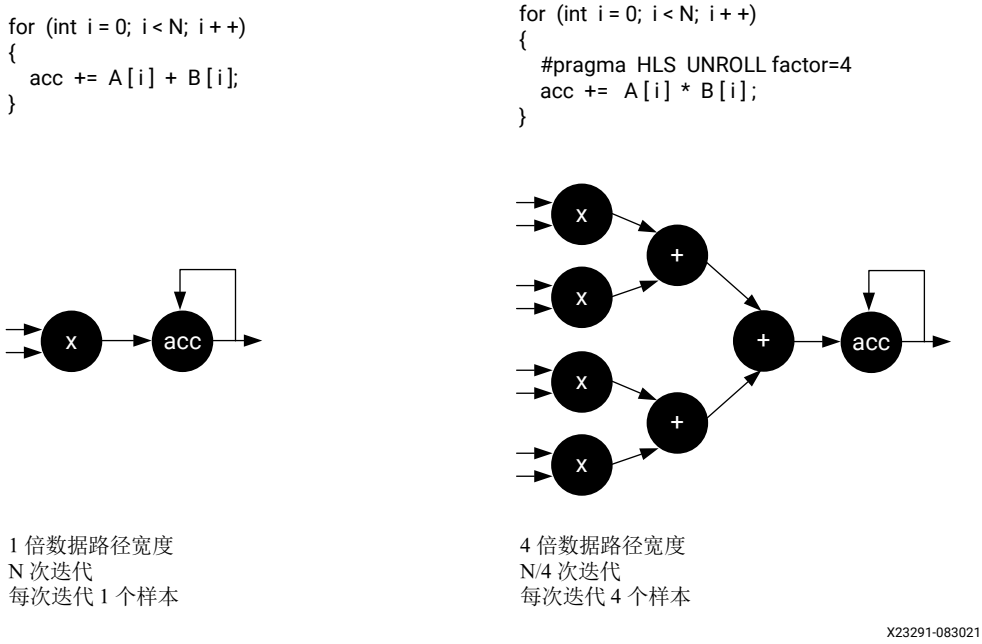
在识别出超过其目标的循环时延之后，要考虑的第一个最优化是循环展开。

应用循环展开

循环展开可解开循环，允许循环的多次迭代一起执行，从而减少循环的总循环次数。

以工业生产做个类比，工厂是内核，组装线是数据流水线，而工作站是计算函数。展开创建的工作站可以处理输送带上同时到达的多个对象，从而提高性能。

图 17：**循环展开**



循环展开能够按相应比例拓宽生成的数据路径。随着并行处理的样本数量增加，带宽要求通常也会增加。这有两个含义：

- 函数 I/O 的宽度必须与数据路径的宽度匹配，反之亦然。
- 当 I/O 要求超出内核端口的最大大小（512 位 / 64 个字节）后，循环展开和数据路径拓宽将无法再带来更多益处。

以下准则有助于最优化循环展开的使用：

- 从循环嵌套中的最内层循环开始。
- 评估哪个展开因子将消除所有循环附有的依赖关系。
- 为了获得更有效的结果，请展开具有固定循环次数的循环。
- 如果在展开的循环中存在函数调用，那么内联这些函数可以改善结果，原因是资源共享方式更有效，但代价是综合时间更长。另请注意，互连可能会变得越来越复杂，后续会导致布线问题。

- 不要盲目地展开循环。在展开循环时，脑中应始终牢记明确的目标。

应用阵列分区

展开循环会更改函数的 I/O 要求和数据访问模式。如果循环进行阵列访问（几乎总是如此），请确保生成的数据路径可以并行访问所需的所有数据。

如果展开循环不能实现期望的性能提升效果，最常见的原因是存在存储器访问瓶颈。

默认情况下，Vitis HLS 编译器将大型阵列映射到存储器资源，其字宽等于一个阵列元素的大小。在大多数情况下，应用循环展开时需要更改此默认映射。

如 阵列配置 中所述，HLS 编译器支持各种编译指示来分区和重塑阵列。在循环展开时，请考虑使用这些编译指示来创建支持所期望的并行访问级别的存储器结构。

展开和分区阵列足以满足目标循环的时延和吞吐量目标。如果满足目标，即可转移到下一个感兴趣的循环。否则，请考虑采用其它最优化方法来提高吞吐量。

步骤 5：提高循环吞吐量

如果通过减少循环次数来改善循环时延仍不足够，请尝试缩短启动时间间隔 (II) 的方法。

循环 II 是两次循环迭代开始之间的时钟周期计数。Vitis HLS 编译器将始终尝试对循环进行流水打拍、最大程度降低 II，并尽早开始循环迭代，理想情况是每个时钟周期（II = 1）启动新的迭代。

有两个主要因素会导致 II 受限：

- I/O 争用
- 循环进位依赖关系

HLS Schedule Viewer 会自动高亮限制 II 的循环依赖关系。在改善循环 II 时，它是一个非常有用的可视化工具。

消除 I/O 争用

当在每次循环迭代期间必须多次访问内部存储器资源的给定 I/O 端口时，就会出现 I/O 争用。如果启动时间间隔 (II) 低于每次循环期间访问任一 I/O 资源的次数，则该循环无法进行流水打拍。如果在任一循环迭代期间，端口 A 必须访问 4 次，那么在单端口 RAM 中可能的最低 II 将为 4。

开发者需要评估这些 I/O 访问是否必要或是否可以避免。减少 I/O 争用的最常用方法是：

- 创建内部高速缓存结构

  如果某些有问题的 I/O 访问涉及访问先前循环迭代中已经访问过的数据，那么可以修改代码以便在本地复制早期迭代中访问的值。保留本地数据高速缓存有助于减少对外部 I/O 访问的需求，从而改善循环的潜在 II。

  Vitis 加速示例 GitHub 仓库中提供的此示例演示了如何在本地使用移位寄存器、如何缓存先前读取的值以及如何改善滤波器的吞吐量。

- 重新配置 I/O 和存储器

  正如前面关于改善时延的部分所述，HLS 编译器将阵列映射到存储器，默认存储器配置无法为所需的吞吐量提供足够的带宽。在此情况下，也可使用阵列分区和重塑编译指示来创建具有更高带宽的存储器结构，从而改善循环的潜在 II。

消除循环进位依赖关系

最常见的循环进位依赖关系是当循环迭代依赖于前一次迭代中计算所得值时。根据与阵列还是标量变量存在依赖关系，处理方式存在差异。如需了解更多信息，请参阅 《Vitis HLS 流程》中的展开循环以改善流水打拍。

- 消除与阵列之间的依赖关系

  HLS 编译器执行索引分析以确定是否存在阵列依赖关系（先写入后读取、先读取后写入、先写入后写入）。该工具可能无法始终静态解决潜在的依赖关系，并且在这种情况下将报告错误的依赖关系。

  特殊的编译器编译指示可以覆盖这些依赖关系并改进设计的 II。在这种情况下，请小心不要覆盖有效的依赖关系。

- 消除对标量的依赖关系

  与标量存在依赖关系时，通常存在反馈路径，并且其中计算调度为在多个时钟周期内执行。通常在这些反馈路径上能找到复杂的算术运算，例如乘法、除法或取模。反馈路径中的周期数直接限制了潜在的 II，应该减少周期数以改善 II 和吞吐量。为此，请分析反馈路径以确定是否可以和如何才能缩短该路径。这可以使用 HLS 调度约束或代码修改（例如减小位宽）来完成。

高级技巧

如果通常最好的情况是 II 为 1，那么它极有可能不是唯一足够的情况。目的是满足时延和吞吐量目标。在这方面，II 和展开因子的各种组合通常就足够了。

本指南中介绍的最优化方法和技巧应有助于实现大多数目标。HLS 编译器还支持许多其它最优化选项，这些选项在特定情况下非常有用。如需获取这些最优化选项的完整参考资料，请参阅 HLS 编译指示。

第二部分

开发应用

本部分包含以下章节：

- 模型编程
- 主机编程
- C/C++ 内核
- RTL 内核
- 利用 Vitis 加速的最佳实践

XILINX

第 5 章

模型编程

Vitis™ 核开发套件支持使用赛灵思提供的编程接口或者根据业界标准的 OpenCL™ 框架 (https://www.khronos.org/opencl/) 来执行异构计算。主机程序在处理器（x86 或 Arm®）上执行，并通过赛灵思的 Xilinx Runtime (XRT) 将计算密集型任务卸载到赛灵思器件的可编程逻辑 (PL) 上运行的硬件内核中执行。

器件拓扑结构

在 Vitis 核开发套件中，目标器件可包含赛灵思 MPSoC 或 UltraScale+™ FPGA，这些 MPSoC 或 FPGA 通过 PCIe 总线之类的 x86 主机连接到处理器，或者通过 AXI4 接口连接到 Arm 处理器。FPGA 包含可编程区域，用于实现和执行硬件内核。

FPGA 平台包含一个或多个全局存储体。从 CPU 到内核以及从内核到 CPU 的数据传输正是通过这些全局存储体来完成的。FPGA 中运行的内核可包含一个或多个存储器接口 (`m_axi`)。用户可配置从全局存储体到这些存储器接口的连接，并通过 Vitis 链接选项来定义这些连接，如 链接内核 中所述。内核还能使用数据流传输接口 (`axis`) 在不同内核之间直接进行数据流传输。数据流传输连接还可通过 `v++` 链接选项来进行管理。

在赛灵思器件 PL 中可以实现多个内核，从而实现显著的应用加速。单一内核也可以多次进行例化。内核实例数量属于可编程要素，用户可在构建 FPGA 二进制文件时，通过指定链接选项来确定数量。如需了解有关指定这些选项的更多信息，请参阅 链接内核。

内核属性

在 Vitis 应用加速开发流程中，内核会处理赛灵思器件的 PL 区域内正在执行的各元素。Vitis 软件平台支持以 C/C++、RTL 或 OpenCL C/C++ 编写的内核。无论采用任何源语言，所有内核都具有相同的属性并且必须遵循相同的要求。

内核可定义为软件可控的内核或非软件可控的内核。这意味着内核通过诸如主机应用等软件来控制或者不受软件管理，改由数据驱动。

软件可控内核

软件可控内核会公开一个可编程的寄存器接口，以允许主机软件应用通过寄存器读取和写入来与内核进行交互。这些都属于最常见且适用最广泛的内核类型。有 2 种类型的软件可控内核：用户管理的内核和 XRT 管理的内核。

注释： XRT 管理的内核是用户管理的内核的其中一种专用格式。

用户管理的内核与 XRT 管理的内核的主要区别在于内核执行模式。由于 XRT 依赖于 Vitis HLS 所生成的 `ap_ctrl_chain` 和 `ap_ctrl_hs` 执行协议，XRT 管理的内核更适合 C++ 开发者，如 C/C++ 内核 和 使用 Vitis HLS 编译内核 中所述。另一方面，用户管理的内核可支持许多不同的用户定义的执行协议（常见于 Vivado RTL IP 中），因此更适合使用 RTL 内核 的 RTL 设计师。

Vitis 应用加速开发流程支持使用 XRT 本机 C/C++ API 编写的主机程序，此 API 同时支持用户管理的内核和 XRT 管理的内核以及某些高级设计（例如，永续内核）。它还支持使用 OpenCL API 来处理 XRT 管理的内核的主机应用。后续章节将简要描述编程 API 以及 XRT 管理的内核或用户管理的内核所需的不同硬件接口。

表 4：**使用 XRT API 进行软件控制**

XRT 管理的内核	用户管理的内核
· XRT 管理的内核的对象类为 `xrt::kernel` · 软件应用使用更高层次的命令来与 XRT 管理的内核进行通信，例如，`set_arg`、`run` 和 `wait` · 用户无需了解可编程寄存器和内核执行协议的低层次细节。 · 控制寄存器和状态寄存器可以为 XRT 提供已知接口，用于与内核进行交互，从而使这些高层次命令可供使用 · 如果需要，还可以将 XRT 管理的内核作为用户管理的内核（使用原子寄存器读取和写入）来进行控制 · OpenCL API 还可搭配 XRT 管理的内核 (`cl::kernel`) 一起使用	· 用户管理的内核的对象类为 `xrt::ip` · 软件应用通过 AXI4-Lite 接口使用原子寄存器读取和写入来与用户管理的内核进行通信。 · 应用开发者负责确认内核中每个寄存器的地址偏移和用途，并正确使用这些寄存器 · 没有可供使用的检查、高层次控制或剖析功能。用户负责运行仿真以进行性能分析/调试。

设计语言

软件可控内核可使用 RTL 或 C/C++ 来开发：

- RTL：对于 RTL 开发者而言，用户管理的内核是最自然且推荐的内核类型。这些内核可以提供更大的灵活性、提供广泛的控制可能性，并且要求比 XRT 管理的内核更少。如需了解更多信息，请参阅 RTL 内核。
- C++：对于 C/C++ 开发者而言，XRT 管理的内核是默认推荐的内核类型，如 C/C++ 内核 中所述。Vitis 编译器使用 Vitis HLS 自动生成与高层次 XRT API 兼容的接口，从而显著减少开发者需要担忧的细节问题。

硬件接口

内核接口用于与主机应用、其它内核或器件 I/O 进行数据交换。用户管理的内核与 XRT 管理的内核都有相同的接口要求，以下列出了这些要求。

- 可编程接口：AXI4-Lite 从接口。内核只能有单一 AXI4-Lite 接口。
- 数据接口：任意数量和组合的 AXI4 存储器映射接口和 AXI4-Stream 接口。
- 时钟与复位：如 时钟和复位要求 中所述。

提示：XRT 管理的内核对于 AXI4-Lite 接口内的控制寄存器（包括启动位和停止位）具有具体的要求，如 XRT 管理的内核的控制要求 中所述。用户管理的内核则可以实现用户指定的任意控制结构。

下表基于您的应用中的数据特性，细化了所需的接口类型。

表 5：**内核接口类型**

寄存器接口 (AXI4-Lite)	存储器映射接口 (M_AXI)	数据流传输接口 (AXI4-Stream)
· 寄存器接口必须使用单一 AXI4-Lite 接口来实现。 · 此类接口是专为在主机应用与内核之间传输标量而设计的。 · 寄存器读写由主机应用发起。 · 内核则充当从接口。	· 存储器映射接口必须使用一个或多个 AXI4 主接口来实现。 · 此类接口是专为内核与全局存储器（DDR、PLRAM 和 HBM）之间的双向数据传输设计的。 · 它引入了额外的时延，用于存储器传输。 · 内核充当主接口，对存储到全局存储器中的数据进行访问。 · 主机应用会根据数据集大小来分配缓冲器。 · 缓冲器的基址是由主机应用通过其 AXI4-Lite 接口向内核提供的。	· 数据流传输接口必须使用一个或多个 AXI4-Stream 接口来实现。 · 它是专为内核之间的双向数据传输而设计的。 · 访问模式为顺序访问。 · 不使用全局存储器。 · 数据集大小无限。 · 可使用边带信号来指示数据流传输中的最后一个值。

执行模式

用户管理的内核没有预定义的执行模式。由内核设计师判断如何实现控制协议和执行机制。应用开发者则负责管理内核操作，根据用户定义的内核控制协议，按相应顺序在主机应用上执行寄存器读取和写入。

XRT 管理的内核则提供已定义的内核执行模式，以支持内核的重叠执行或顺序执行，如 XRT 文档的受支持的内核执行模型中所述。

- 内核由主机应用使用 API 调用来启动。当内核准备好处理新数据时，它会通过控制寄存器中相应的位来通知主机应用。
- 默认控制协议 `ap_ctrl_chain` 允许将同一内核的多次执行加以重叠，以流水打拍方式来运行，从而提升总体应用吞吐量。
- 如果需要，可使用 `ap_ctrl_hs` 控制协议来禁用重叠执行，此控制协议会强制内核按顺序运行，即等待至上一轮运行完成后再开始下一轮运行。

非软件控制的内核

这些内核存在于器件中，但对于软件应用不可见或者不可供软件应用直接访问。这些内核没有可编程寄存器接口。这些内核必须具有至少一个 AXI4-Stream 接口。内核通过这些数据流传输接口与系统其余部分进行同步。

非软件控制的内核被视为一项高级功能，仅当无法使用软件可控的内核时，才应使用。由于此类内核没有可编程寄存器接口，因此需要通过内核的数据接口来传递控制相关的信息。

非软件控制的内核无需软件 API，因为主机应用不与内核直接交互。内核可作为 RTL 内核 来开发，或者也可作为 C/C++ 内核 来开发（如 在用户管理的永续内核中进行数据流传输 中所述）。

硬件接口

内核接口用于与主机应用、其它内核或器件 I/O 进行数据交换。非软件控制的内核应具有下列接口要求：

- 可编程接口：无 AXI4-Lite 接口。
- 数据接口：至少 1 个 AXI4-Stream 接口。
- 时钟与复位：如 时钟和复位要求 中所述。

时钟和复位要求

这些时钟和复位要求适用于软件可控制内核与软件不可控制内核。

表 6：**要求**

C/C++/OpenCL C 内核	RTL 内核
· C 内核的时钟端口和复位端口上无需来自用户的任何输入。HLS 工具生成的 RTL 始终包含时钟端口 `ap_clk` 和复位端口 `ap_rst_n`。 · HLS 内核只能有 1 个时钟/复位。	· RTL 内核需要至少 1 个时钟端口，但每个内核都能包含多个时钟。RTL 可包含的时钟数量主要取决于平台支持的时钟数量。大部分数据中心平台都仅支持 2 个时钟，但大部分嵌入式平台都可包含多个时钟。 · （可选）低电平有效复位端口可通过时钟上的 ASSOCIATED_RESET 参数来与时钟相关联。

主机编程

在 Vitis™ 环境中，可使用赛灵思的 Xilinx Runtime (XRT) 本机 C++ API 或业界标准的 OpenCL™ API 以本机 C++ 来编写主机应用。此处对 XRT 本机 API 进行了简要描述，在 XRT 文档网站上的 XRT 本机 API 下提供了更多详细信息。请参阅 OpenCL 编程 以获取有关使用 OpenCL API 编写主机应用的探讨内容。

提示：如需获取使用 XRT 本机 API 进行主机编程的示例，请参阅 Vitis_Accel_Examples 中的 host_xrt。

总之，主机应用的结构可分为以下步骤：

1. 指定加速器器件 ID 并加载 `.xclbin`
2. 设置内核与内核实参
3. 在主机与内核之间传输数据
4. 运行内核并返回结果

要使用 XRT 本机 API，主机应用必须与 `xrt_coreutil` 库相链接。使用 XRT 本机 C++ API 编译主机代码需要 C++ 标准和 `-std=c++14` 或更高版本。例如：

```
g++ -g -std=c++14 -I$XILINX_XRT/include -L$XILINX_XRT/lib -lxrt_coreutil -
pthread
```

重要提示！要对主机程序启用多线程，从 Vitis 核开发套件应用调用 `fork()` 系统调用时请谨慎处理。`fork()` 不会复制所有运行时线程。因此，在 Vitis 核开发套件中，子进程无法作为完整应用来运行。建议使用 `posix_spawn()` 系统调用从 Vitis 软件平台应用启动另一个进程。

指定器件 ID 并加载 XCLBIN

要正确使用赛灵思的 Xilinx Runtime (XRT) 环境，主机应用需要识别用于运行内核的加速器卡和器件 ID，并将器件二进制文件 (`.xclbin`) 加载到器件中。

XRT API 包含用于在加速器卡上指定器件 ID 的器件类 (`xrt::device`) 以及用于为运行时定义程序的 XCLBIN 类 (`xrt::xclbin`)。您必须在自己的源代码中使用以下 include 语句来加载这些类：

```
#include <xrt/xrt_kernel.h>
```

以下代码片段通过指定来自目标平台的器件 ID 创建器件对象，然后将 .xclbin 加载到器件中，并为程序返回 UUID。

```
//Setup the Environment
unsigned int device_index = 0;
std::string binaryFile = parser.value("kernel.xclbin");
std::cout << "Open the device" << device_index << std::endl;
auto device = xrt::device(device_index);
std::cout << "Load the xclbin " << binaryFile << std::endl;
auto uuid = device.load_xclbin(binaryFile);
```

提示：可使用 xbutil 命令获取特定加速器卡的器件 ID。

设置 XRT 管理的内核与内核实参

识别器件并加载程序后，主机应用应识别在器件上执行的内核，并设置内核实参。在已加载的 .xclbin 文件中定义了与主机应用交互的所有内核，因此应根据其中定义来识别这些内核。

对于 XRT 管理的内核，XRT API 提供了内核类 (xrt::kernel)，用于访问 .xclbin 文件中包含的内核。内核对象会识别 XRT 管理的内核，这些内核已加载到赛灵思器件内的 .xclbin 文件中并且可供主机应用运行。

提示：如 设置用户管理的内核和实参缓冲器 中所述，您应使用 IP 类 (xrt::ip) 来识别 .xclbin 文件中的用户管理的内核。

使用内核和缓冲器对象需要在源代码中添加以下 include 语句：

```
#include <xrt/xrt_kernel.h>
#include <xrt/xrt_bo.h>
```

以下代码示例用于识别已加载到 device 上的程序 (uuid) 中定义的内核 ("vadd")：

```
auto krnl = xrt::kernel(device, uuid, "vadd");
```

提示：您也可以使用 xclbinutil 命令来检验现有 .xclbin 文件的内容，并确定其中包含的内核。

识别内核或者要运行的内核后，您需要定义缓冲器对象以与内核实参关联，并启用从主机应用到内核实例或计算单元 (CU) 的数据传输：

```
std::cout << "Allocate Buffer in Global Memory\n";
auto bo0 = xrt::bo(device, vector_size_bytes, krnl.group_id(0));
auto bo1 = xrt::bo(device, vector_size_bytes, krnl.group_id(1));
auto bo_out = xrt::bo(device, vector_size_bytes, krnl.group_id(2));
```

内核对象 (xrt::kernel) 包含了返回与每个内核实参关联的存储器的方法，即 kernel.group_id()。由于针对标量实参并未创建缓冲器，因此您将向每个内核缓冲器实参分配一个缓冲器对象。

创建多个计算单元

构建 `.xclbin` 文件时，可使用 `--connectivity.nk` 选项来指定要实现到硬件中的内核实例或计算单元 (CU) 的数量，如 创建内核的多个实例 中所述。构建 `.xclbin` 后，即可从主机应用访问这些 CU。

用户可以使用单一内核对象 (`xrt::kernel`) 来执行多个 CU，前提是这些 CU 具有相同的接口连接方式，即这些 CU 具有相同的存储器连接 (`krnl.group_id`)。如果并非所有 CU 都有相同的内核连接，那么您可以为内核的每项唯一配置创建独立的内核对象，如以下示例所示。

```
krnl1 = xrt::kernel(device, xclbin_uuid, "vadd:{vadd_1,vadd_2}");
krnl2 = xrt::kernel(device, xclbin_uuid, "vadd:{vadd_3}");
```

在以上示例中，`krnl1` 可用于启动 CU `vadd_1` 和 `vadd_2`，这 2 个 CU 具有相匹配的连接，而 `krnl2` 则可用于启动 `vadd_3`，该 CU 的连接与前 2 个不同。

提示：如果您为多个无匹配连接的 CU 创建单个内核对象，那么在执行内核时，XRT 会将一个或多个具有匹配连接的 CU 分配到该内核对象，并忽略硬件中的其它内核。

在主机与内核之间传输数据

在加速器卡或器件中的存储器上传入和传出数据使用的是缓冲器对象 (`xrt::bo`)，这些缓冲器对象是 设置 XRT 管理的内核与内核实参 期间创建的。

类构造函数通常会分配常规 4K 对齐的缓冲器对象。以下代码会创建常规缓冲器对象，其中包含由堆存储器中的用户空间分配的主机反向指针，此外还会创建器件侧缓冲器，此缓冲器在与内核实参 (`krnl.group_id`) 关联的存储体内进行分配。`xrt::bo` 构造函数中的可选标记允许您创建非标准类型的缓冲器，以供在特殊环境下使用，如创建特殊缓冲器 中所述。

```
std::cout << "Allocate Buffer in Global Memory\n";
auto bo0 = xrt::bo(device, vector_size_bytes, krnl.group_id(0));
auto bo1 = xrt::bo(device, vector_size_bytes, krnl.group_id(1));
auto bo_out = xrt::bo(device, vector_size_bytes, krnl.group_id(2));
```

重要提示！单一缓冲器大小不能超过 4 GB，但为了最大限度提升从主机到全局存储器的吞吐量，赛灵思还建议尽可能保留大小至少为 2 MB 的缓冲器。

建立缓冲器并在其中填充数据后，有多种方法可以启用在主机与内核之间进行数据传输，如下所述：

- 使用 `xrt::bo::sync()`：使用 `xrt::bo::sync` 通过 `XCL_BO_SYNC_TO_DEVICE` 标记将数据从主机同步到器件，或者通过 `XCL_BO_SYNC_FROM_DEVICE` 标记使用 `xrt::bo::write` 或 `xrt::bo::read` 将数据从器件同步到主机，以便从主机应用写入缓冲器或者从器件读取缓冲器。

  ```
  bo0.write(buff_data);
  bo0.sync(XCL_BO_SYNC_BO_TO_DEVICE);
  bo1.write(buff_data);
  bo1.sync(XCL_BO_SYNC_BO_TO_DEVICE);
  ...
  bo_out.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
  bo_out.read(buff_data);
  ```

  注释：如果按从用户指针创建缓冲器中所述使用用户指针创建了缓冲器，那么执行 `xrt::bo::sync` 调用足矣，无需 `xrt::bo::write` 或 `xrt::bo::read` 命令。

- 使用 `xrt::bo::map()`：此方法可将主机侧缓冲器反向指针映射到用户指针。

```
// Map the contents of the buffer object into host memory
auto bo0_map = bo0.map<int*>();
auto bo1_map = bo1.map<int*>();
auto bo_out_map = bo_out.map<int*>();
```

后续，主机代码即可为数据读写实践用户指针。但在写入映射的指针之后（或者从映射的指针读取之前），`xrt::bo::sync()` 命令应搭配所需方向标记用于执行 DMA 操作。

```
for (int i = 0; i < DATA_SIZE; ++i) {
   bo0_map[i] = i;
   bo1_map[i] = i;
}

// Synchronize buffer content with device side
bo0.sync(XCL_BO_SYNC_BO_TO_DEVICE);
bo1.sync(XCL_BO_SYNC_BO_TO_DEVICE);
```

XRT 本机 API 还支持其它缓冲器类型和传输场景，如其它缓冲器中所述。

在器件上执行内核

内核的执行与名为 `xrt::run` 的类相关联，该类用于实现启动和等待内核执行的方法。大部分与内核对象的交互都是通过 `xrt::run` 对象来完成的，这些对象是从内核创建的，用于表示内核的执行。

运行对象可根据内核对象来显式构造，或者可通过启动内核执行来隐式构造，如下所示。

```
std::cout << "Execution of the kernel\n";
auto run = krnl(bo0, bo1, bo_out, DATA_SIZE);
run.wait();
```

以上代码示例演示的是使用 `xrt::kernel()` 运算符搭配返回 `xrt::run` 对象的内核实参列表来启动内核执行。这是异步运算符，在启动运行后返回。`xrt::run::wait()` 成员函数用于阻塞当前线程直至运行完成为止。

提示：完成内核执行后，`xrt::run` 对象即可用于按需重新启动相同的内核函数。

以下代码显示了运行内核的另一种方法：

```
auto run = xrt::run(krnl);
run.set_arg(0,bo0); // Arguments are specified starting from 0
run.set_arg(0,bo1);
run.set_arg(0,bo_out);
run.start();
run.wait();
```

在此示例中，运行对象是根据内核对象来显式构造的，内核实参是使用 `run.set_args()` 指定的，运行执行则是由 `run.start()` 命令启动的。最后，当前线程正在等待内核完成，因此处于阻塞状态。

当内核完成执行后，您可使用如下示例所示的代码将内核结果同步回主机应用：

```
// Get the output;
bo_out.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

// Validate our results
if (std::memcmp(bo_out_map, bufReference, DATA_SIZE))
   throw std::runtime_error("Value read back does not match reference");
```

设置用户管理的内核和实参缓冲器

用户管理的内核需要为主机应用使用 XRT 本机 API，这些内核被指定为 `xrt::ip` 类的 IP 对象。以下高层次综述解释了如何构造您的主机应用，以从 `.xclbin` 文件访问用户管理的内核。

1. 添加以下头文件以包含 XRT 本机 API：

   ```
   #include "experimental/xrt_ip.h"
   #include "xrt/xrt_bo.h"
   ```

   - `experimental/xrt_ip.h`：将 IP 定义为 `xrt::ip` 对象。
   - `xrt/xrt_bo.h`：在 XRT 本机 API 中创建缓冲器对象。

2. 按 指定器件 ID 并加载 XCLBIN 中所述，设置应用环境。

3. IP 对象 (`xrt::ip`) 是根据 `xrt::device` 对象、`.xclbin` 的 `uuid` 以及用户管理的内核的 `name` 来构造的。`xrt::ip` 不同于标准 `xrt::kernel`，前者表示 XRT 不管理 IP，但提供对寄存器的访问权：

   ```
   //User Managed Kernel = IP
   auto ip = xrt::ip(device, uuid, "Vadd_A_B");
   ```

4. 为 IP 实参创建缓冲器：

   ```
   auto <buf_name> = xrt::bo(<device>,<DATA_SIZE>,<flag>,<bank_id>);
   ```

   其中缓冲器对象构造函数使用以下字段：

   - `<device>`：加速器卡的 `xrt::device` 对象。
   - `<DATA_SIZE>`：缓冲器大小，由数据宽度和数量来定义。
   - `<flag>`：用于创建缓冲器对象的标记。
   - `<bank_id>`：在器件上定义存储体，应在此存储体中为 IP 访问分配缓冲器。指定的存储体必须与 `.xclbin` 文件内对应的 IP 端口连接相匹配。否则，运行应用时，将出现 `bad_alloc`。您可使用 `--connectivity.sp` 命令指定内核实参的赋值，如 将内核端口映射到存储器 中所述。

   例如：

   ```
   auto buf_in_a = xrt::bo(device,DATA_SIZE,xrt::bo::flags::normal,0);
   auto buf_in_b = xrt::bo(device,DATA_SIZE,xrt::bo::flags::normal,0);
   ```

   提示：请验证 IP 连接以确定特定的存储体，否则在 Vitis 生成的 `.xclbin.info` 文件中可能会出现此信息。

例如，.xclbin 为用户管理的内核生成如下信息可以为主机代码中缓冲器对象的构造过程提供指导信息。

```
Instance:        Vadd_A_B_1
   Base Address: 0x1c00000

   Argument:          scalar00
   Register Offset:   0x10
   Port:              s_axi_control
   Memory:            <not applicable>

   Argument:          A
   Register Offset:   0x18
   Port:              m00_axi
   Memory:            bank0 (MEM_DDR4)

   Argument:          B
   Register Offset:   0x24
   Port:              m01_axi
   Memory:            bank0 (MEM_DDR4)
```

5. 在主机与器件之间传输数据：

```
    auto a_data = buf_in_a.map<int*>();
    auto b_data = buf_in_b.map<int*>();

    // Sync Buffers
    buf_in_a.sync(XCL_BO_SYNC_BO_TO_DEVICE);
    buf_in_b.sync(XCL_BO_SYNC_BO_TO_DEVICE);
```

xrt::bo::map() 允许将主机侧缓冲器反向指针映射到用户指针。但在读取映射的指针之前或者写入映射的指针之后，应使用 xrt::bo::sync()（含方向标记）来执行 DMA 操作。

6. 准备好缓冲器（如上所示缓冲器创建、同步操作）之后，您即可通过定向寄存器写入操作将所有必要信息都传递到 IP。例如，以下所示代码显示了有关通过 xrt::ip::write_register() 命令传递缓冲器基址的信息。

   随后，通过写入寄存器来将数据从主机应用移至内核。

```
    ip.write_register(REG_OFFSET_A,a_addr);
    ip.write_register(REG_OFFSET_A+4,a_addr>>32);

    ip.write_register(REG_OFFSET_B,b_addr);
    ip.write_register(REG_OFFSET_B+4,b_addr>>32);
```

7. 启动 IP 执行。由于此 IP 为用户管理的 IP，您可采用任意数量的寄存器写入/读取操作来控制 IP 的启动、检查状态或重新启动，以便触发 IP 执行。以下示例使用 s_axilite 接口来访问控制寄存器中的控制信号。

```
    uint32_t axi_ctrl = 0;
    std::cout << "INFO:IP Start" << std::endl;
    axi_ctrl = IP_START;
    ip.write_register(CSR_OFFSET, axi_ctrl);

    // Wait until the IP is DONE
    axi_ctrl =0;
    while((axi_ctrl & IP_IDLE) != IP_IDLE) {
        axi_ctrl = ip.read_register(CSR_OFFSET);
    }
```

8. 完成 IP 执行后，您可通过 xrt::bo::sync 命令将数据传回，只需在此命令中包含相应的标记以指明缓冲器传输方向即可。

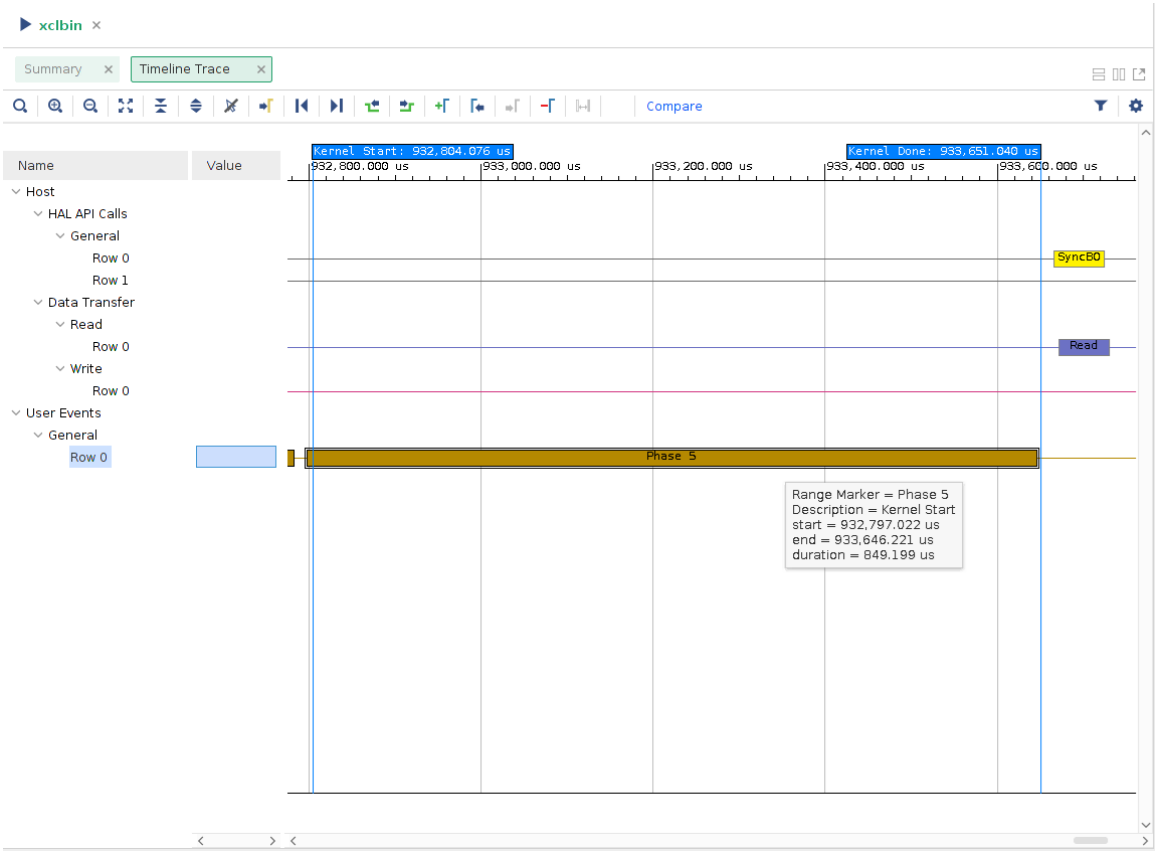
```
    buf_in_b.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
```

9. （可选）对应用进行剖析。

   由于 XRT 不负责启动或停止内核，您无法像处理 XRT 管理的内核那样对 `user_managed` 内核的操作进行直接剖析。但您可以使用 `user_range` 和 `user_event` 对象（如 主机应用的定制剖析 中所述）来对主机应用的元素进行剖析。例如，以下代码可捕获从主机应用写入寄存器所耗费的时间。

```
// Write Registers
range.start("Phase 4a", "Write A Register");
ip.write_register(REG_OFFSET_A,a_addr);
ip.write_register(REG_OFFSET_A+4,a_addr>>32);
range.end();
range.start("Phase 4b", "Write B Register");
ip.write_register(REG_OFFSET_B,b_addr);
ip.write_register(REG_OFFSET_B+4,b_addr>>32);
range.end()
```

   您可在 Vitis 分析器中对应用和内核操作的某些方面进行观测，如下图所示。



为用户管理的内核启用自动重新启动

某些用户管理的内核会实现来自 Vitis HLS 的 `ap_ctrl_chain` 协议，在此类内核中，您可将 `s_axilite` 控制寄存器中的 `auto_restart` 置位，这样此内核将自动重新启动。使用 `auto_restart` 位的用户管理的内核称为永续内核，如 在用户管理的永续内核中进行数据流传输 中所述。

对永续内核进行编程要求主机应用在 `s_axilite` 控制寄存器内的 `0x00` 地址处设置 `auto_restart` 信号，否则内核将仅在单次执行模式下运行，随后等待主机应用将其重新启动。要对内核控制寄存器进行编程，请使用以下流程：

1. 对于访问用户管理的内核的主机应用，请将其设置为 `xrt::ip` 类的 IP 对象，如前文所述。

2. 将值 129（二进制值 10000001）写入控制寄存器、将 ap_start 和 auto_restart 置位，这样即可支持内核在永续模式下运行。s_axilite 控制寄存器位于 0x00 下，就像其它 ap_ctrl_chain 内核一样。

   重要提示！请勿向控制寄存器空间写入任何其它内容，否则可能导致无法确定的行为。

以下代码示例用于设置 IP 对象，并按上述方式写入控制寄存器以对其进行设置。ap_start 置位后，永续内核即可开始执行，而 auto_restart 置位后，此用户将永不停止运行。

```
auto ip = xrt::ip(device_2, xclbinId, "krnl_stream_vdatamover");
auto ip = xrt::ip(device, uuid, "krnl_stream_vdatamover");
        int startNow =129;
        size_t control_offset = 0;
        ip.write_register(control_offset,startNow);
```

总结

如先前各主题中所述，Vitis 核开发套件中主机程序的建议的编码样式包括以下几个要点：

1. 在 Vitis 核开发套件中，通过对一个或多个内核进行单独编译/链接来构建 .xclbin 文件。device.load_xclbin(binaryFile) 命令用于加载内核二进制文件。
2. 从加载的器件二进制文件创建 xrt::kernel 对象，并将缓冲器对象 (xrt::bo) 与分配给内核实参的存储体加以关联。
3. 使用 xrt::bo::sync 命令以及缓冲器读写命令在主机应用与内核之间往返传输数据。
4. 内核执行方式为使用 xrt::run 对象启动内核并等待内核执行。
5. 此外，您可在完成 XRT API 调用后根据需要添加纠错用于调试目的。

XILINX

第 7 章

C/C++ 内核

在 Vitis™ 核开发套件中，内核代码通常是算法中计算密集的部分，应在 FPGA 上加速。Vitis 核开发套件支持以 C/C++、OpenCL™ 以及 RTL 编写的内核代码。本章侧重于基于 C/C++ 的代码的编码样式。

通常，现成软件无法有效转换为 FPGA 上的加速硬件。即使软件程序可自动转换（或综合）为硬件，但要实现可接受的结果质量 (QoR)，仍需要额外工作（例如，重写软件元素）以帮助 Vitis HLS 实现期望的性能目标。为此，您需要了解正确编写软件的最佳实践，以确保在 FPGA 上正常执行软件，如《Vitis 高层次综合用户指南》(UG1399) 中的面向软件程序员的设计原则所述。

重要提示！内核函数声明必须在报头文件中使用 `extern "C"` 链接加以封装，或者必须封装内核源代码中的整个函数。

```
extern "C" {
           void kernel_function(int *in, int *out, int size);
        }
```

进程执行模式

如 内核属性 中所述，XRT 管理的内核有 2 种类型的执行模式。这些模式是由块协议确定的，这些块协议是由 Vitis HLS 在内核编译期间分配给内核的。此块协议可使用 `#pragma HLS INTERFACE` 来指定。以下列出了这些模式以及用于启用这些模式的块协议：

- Pipeline：由默认块协议 `ap_ctrl_chain` 启用，允许多个内核在执行中重叠，且其中单一内核即可在完成执行一项任务的同时开始执行下一项任务
- Sequential：由 `ap_ctrl_hs` 启用的连续访问模式，要求内核完成执行一项任务后才能开始执行下一项任务

如需了解有关 XRT 如何支持这些执行模式的更多信息，请参阅受支持的内核执行模式。

流水线执行

如果内核在操作来自先前传输事务的数据的同时仍能接受更多数据，那么 XRT 即可发送下一批数据，如 暂时性数据并行化：主机到内核数据流 中所述。流水线 (Pipeline) 模式允许内核重叠多个内核运行，从而改善整体吞吐量。

为支持流水线模式，内核必须使用 `ap_ctrl_chain` 协议，这是 Vitis HLS 使用的默认协议。此协议还可通过向函数返回赋值 `#pragma HLS INTERFACE` 来启用，如以下示例所示。

```
void kernel_name( int *inputs,
                  ...          )// Other input or Output ports
{
#pragma HLS INTERFACE ap_ctrl_chain port=return bundle=control
```

要成功完成流水线执行，内核应为内核队列提供较长的时延，否则可能没有足够时间可供内核处理每一批数据，这样也就无法发挥流水线的优势。如果流水打拍的内核无法以流水打拍方式来处理数据，那么它会还原为顺序 (sequential) 执行。

重要提示！为了充分利用主机到内核数据流，内核还必须分阶段写入处理数据，例如，在循环级别进行流水打拍（如 循环流水打拍 中所述）或者在任务级别进行流水打拍（如 数据流最优化 中所述）。

因固有原因，XRT 管理的内核还支持纯顺序模式，可使用 `ap_ctrl_hs` 块协议来为 `#pragma HLS INTERFACE` 中的函数返回配置此模式。

永续模式

默认情况下，Vitis HLS 生成的内核包含由主机应用控制的同步。该主机可监控内核的启动和终止。但在某些情况下，内核无需受主机控制，例如，在持续性数据流传输中。这些内核可使用 `ap_ctrl_chain` 块协议的 `auto_restart` 符号，如 在用户管理的永续内核中进行数据流传输 中所述。此类内核被视为用户管理的内核，因为用户设置 `ap_start` 位和 `auto_restart` 位以启动内核执行，但除了初始启动外，它大体上属于非软件控制的内核。

数据类型

由于使用原生 C 语言数据类型（如 `int`、`float` 或 `double`）来写入和验证代码更快，因此一般首次编码时都会使用这些数据类型。但代码是在硬件中实现的，硬件中使用的所有运算符大小都取决于加速器代码中使用的数据类型。默认原生 C/C++ 语言数据类型可能导致硬件资源更大且更慢，从而限制内核性能。因此，改为考虑使用高位精度的数据类型可以确保将代码最优化以供在硬件内实现。使用高位精度或任意精度数据类型能够使硬件运算符更小且更快。这样即可将更多逻辑布局到可编程逻辑中，并允许逻辑以更高的时钟频率来执行，同时降低功耗。

请考虑在代码中使用高位精度的数据类型代替使用原生 C/C++ 语言数据类型。

建议：请考虑在代码中使用高位精度的数据类型代替使用原生 C/C++ 语言数据类型。

在以下章节中，着重探讨了 Vitis 编译器支持的 2 个最常用的任意精度数据类型（任意精度整数类型和任意精度定点类型）。

注释：这些数据类型应仅用于 C/C++ 内核，不得用于 OpenCL 内核（或在主机代码内使用）。

任意精度整数类型

在头文件 `ap_int.h` 内，分别通过 `ap_int` 或 `ap_uint` 来为有符号整数和无符号整数定义任意精度整数数据类型。要使用任意精度整数数据类型，请执行以下操作：

- 将头文件 `ap_int.h` 添加到源代码。
- 将位类型更改为 `ap_int<N>` 或 `ap_uint<N>`，其中 N 是介于 1 到 1024 之间的位大小。

以下示例显示了如何添加头文件并实现 2 个变量来使用 9 位整数和 10 位无符号的整数。

```
#include "ap_int.h"
ap_int<9> var1 // 9 bit signed integer
ap_uint<10> var2 // 10 bit unsigned integer
```

任意精度定点数据类型

某些现有应用使用浮点数据类型，因为这些数据类型是为其它硬件架构编写的。但定点数据类型非常适合替代需要多个时钟周期才能完成的浮点类型。在考量为应用和加速器选择实现浮点运算还是定点运算时，请谨慎评估两者在功耗、成本、效率和精度方面的利弊。

如《由浮点转为定点助力降低功耗与成本》(WP491) 中所述，为应用使用定点运算替代浮点运算可以提升能效，并降低所需的总功耗。除非需要浮点类型的完整范围，否则通常使用定点类型即可实现相同精度，并且硬件更小也更快。

定点数据类型将数据作为整数位和小数位形式进行建模。定点数据类型需要 `ap_fixed` 报头，并且支持有符号格式和无符号格式，如下所示：

- 报头文件：`ap_fixed.h`
- 有符号定点：`ap_fixed<W,I,Q,O,N>`
- 无符号定点：`ap_ufixed<W,I,Q,O,N>`
- W = 总宽度 < 1024 位
- I = 整数位宽度。I 的值必须小于或等于宽度 (W)。W 减去 I 即表示小数部分的位数。只能使用常量整数表达式来指定整数宽度。
- Q = 量化模式。只能使用预定义的枚举值来指定 Q。可接受的值包括：
  - `AP_RND`：舍入到正无穷。
  - `AP_RND_ZERO`：舍入到 0。
  - `AP_RND_MIN_INF`：舍入到负无穷。
  - `AP_RND_INF`：舍入到无穷。
  - `AP_RND_CONV`：收敛（无偏）舍入
  - `AP_TRN`：截断。不指定 Q 时，这是默认值。
  - `AP_TRN_ZERO`：截断到 0。
- O = 上溢模式。只能使用预定义的枚举值来指定 O。可接受的值包括：
  - `AP_SAT`：饱和。
  - `AP_SAT_ZERO`：饱和到 0。
  - `AP_SAT_SYM`：对称饱和。
  - `AP_WRAP`：卷绕。不指定 O 时，这是默认值。
  - `AP_WRAP_SM`：符号量值卷绕。
- N = 上溢卷绕 (WRAP) 模式中的饱和位数。只能使用常量整数表达式作为参数值。默认值为零 (0)。

在以下代码示例中，使用 `ap_fixed` 类型来定义有符号 18 位变量，其中 6 个位表示二进制小数点前的整数值，暗示 12 个位表示二进制小数点后的小数值。量化模式设为舍入到正无穷 (`AP_RND`)。由于未指定上溢模式和饱和位数，因此使用默认值 `AP_WRAP` 和 0。

```
#include <ap_fixed.h>
...
  ap_fixed<18,6,AP_RND> my_type;
...
```

执行计算时，如果其中所含变量具有不同位数 (W) 或不同精度 (I)，那么二进制小数点将自动对齐。如需了解有关使用定点数据类型的更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399) 中的 C++ 任意精度定点类型。

接口

在 FPGA 上的主机与内核之间会发生两种类型的数据传输。在主机 CPU 与加速器之间通过全局存储体来传输数据指针。标量数据从主机直接传输至内核。

往来内核传输数据所耗费的时间也会对应用架构的吞吐量目标产生影响。由于数据传输的高开销，考虑如何将计算与应用中存在的通信（数据移动）加以重叠就显得尤为重要。请参阅《Vitis 高层次综合用户指南》(UG1399) 中的设计高效的内核。

Vitis 核开发套件中包含的 Vitis HLS 工具会为 C/C++ 内核函数的参数自动分配接口端口。这些端口分配是在 `v++` 编译进程中完成的。以下章节提供了有关这些接口端口的更多详细信息，包括是否允许手动分配端口或者使用 INTERFACE 编译指示来覆盖默认分配。如果代码中不含用户定义的 INTERFACE 编译指示，那么 Vitis 工具将分配以下接口协议：

- AXI4 主接口 (`m_axi`) 分配到 C/C++ 函数的指针实参。
- AXI4-Lite 接口 (`s_axilite`) 分配到标量实参、阵列的控制信号、全局变量以及软件仿真的返回值。
- Vitis HLS 会自动推断突发传输事务以聚集存储器访问，从而最大程度提升吞吐量带宽和/或最大程序减小时延。如需了解有关突发传输的更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399) 中的最优化突发传输事务。
- 使用 `hls::stream` 来定义参数类型时，Vitis HLS 工具会推断 `axis` 数据流传输接口。

存储器映射接口

存储器映射接口是根据指针参数推断得到的。这些接口允许内核在全局存储器中读取和写入数据，全局存储器即内核与主机应用之间共享的存储器。因此，存储器映射接口是跨加速应用的多个元素分享数据的便利方式，但仅限顺序内核执行模型和流水打拍内核执行模型才支持这些接口，如 内核属性 中所述。

要自定义 Vitis 工具在编译期间分配的默认接口，可使用 INTERFACE 编译指示。为了实现最优性能，赛灵思建议执行突发传输，如果可能，最好按 AXI 协议上限每次传输 4 KB 数据。

内核接口

```
void cnn( int *pixel, // Input pixel
  int *weights, // Input Weight Matrix
  int *out, // Output pixel
  ... // Other input or Output ports
```

在以上示例中，内核函数具有 3 个指针参数：`pixel`、`weights` 和 `out`。默认情况下，Vitis 编译器将把这 3 个参数映射到同一个 AXI4 接口 (`m_axi`)。

编译器推断的默认接口映射等同于以下 INTERFACE 编译指示：

```
#pragma HLS INTERFACE m_axi port=pixel   offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out     offset=slave bundle=gmem
```

提示：该工具不会将推断的编译指示添加到代码中，此处显示此编译指示用于演示分配给接口端口的默认设置。

INTERFACE 编译指示上的 bundle 关键字用于定义端口的名称。系统编译器将为每个独立的捆绑包 (bundle) 名称创建一个端口，从而生成已编译的内核对象 (XO) 文件，此文件中包含单一 AXI 接口 m_axi_gmem。当针对不同接口使用相同 bundle 名称时，会导致将这些接口映射到同一个端口。

提示： gmem 名称表示全局存储器，但它并非关键字，仅用于保持一致性。您可以为捆绑包分配自己的名称。

共享端口有助于通过消除 AXI 接口来节省 FPGA 资源，但它可能限制内核性能，因为所有存储器传输都必须经过同一个端口。m_axi 端口具有独立的 READ 通道和 WRITE 通道，因此通过单一 m_axi 端口即可同时执行读写操作。但通过创建多个端口、使用不同捆绑包名称连接到多个存储体，可能导致内核带宽和吞吐量增加。有许多选项可用于配置 INTERFACE，如 pragma HLS interface 中所述。在代码中手动定义 INTERFACE 编译指示的原因包括但不限于：

- 为 INTERFACE 编译指示指定捆绑包，用于将 AXI 信号拆分为多个独立的捆绑包。
- 指定接口宽度不同于默认 int = 64 字节（512 位）。
- 为突发传输事务指定 AXI 属性。

```
void cnn( int *pixel, // Input pixel
  int *weights, // Input Weight Matrix
  int *out, // Output pixel
  ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel   offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out     offset=slave bundle=gmem
```

在以上示例中，2 个 bundle 名称会创建 2 个不同端口：gmem 和 gmem1。内核将通过 gmem 端口访问 pixel 和 out 数据，而 weights 数据则将通过 gmem1 端口来访问。因此，内核将能够并行访问 pixel 和 weights，从而提升内核吞吐量。

重要提示！ 使用全小写字符来指定 bundle= 名称，以便您使用 connectivity.sp 选项将其分配给特定存储体。

在 v++ 编译期间使用 INTERFACE 编译指示可生成编译后的内核对象 (XO) 文件，其中包含 2 个独立的 AXI 接口：m_axi_gmem 和 m_axi_gmem1，这 2 个接口可按需连接到全局存储器。在系统编译器链接期间，可在配置文件中使用 connectivity.sp 选项将不同接口映射到不同的全局存储体，如 将内核端口映射到存储器 中所述。

存储器接口宽度约束

往来全局存储器与内核之间的最大数据宽度为 512 位。为了最大程度提升数据传输速率，建议您完整使用此数据宽度。默认情况下，在 Vitis 内核流程中，Vitis HLS 工具会将内核接口端口自动上调至 512 位以改善突发访问。如需获取更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399) 中的自动调整端口宽度。

提示： Vitis HLS 中的“综合汇总 (Synthesis Summary)”报告包含有关拓宽端口的信息。但是，要复查此报告，您需要启动该工具。

使用自动端口宽度调整功能有利有弊，您需在使用时审慎考量：

- 缩短来自存储器的读取时延，因为该工具读取的是大型载体而不是数据类型大小。
- 添加所需资源以缓冲大型载体，并将数据转移至数据路径大小。
- 自动端口宽度调整仅支持标准 C 数据类型，不支持非聚合类型，例如，ap_int、ap_uint、结构体或阵列。

提示： 您可以根据需要禁用自动端口拓宽，并手动调整内核端口大小。

突发访问全局存储器

从内核访问全局存储体会产生巨大的时延，因此应在突发内完成全局存储器传输。如需了解有关突发传输的更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399) 中的最优化突发传输事务。

提示：Vitis HLS 中的“综合汇总 (Synthesis Summary)”报告包含有关内核中的突发传输的详细信息。但是，要复查此报告，您需要启动该工具。

要推断突发，建议采用以下循环流水打拍编码样式。

```
hls::stream<datatype_t> str;

INPUT_READ: for(int i=0; i<INPUT_SIZE; i++) {
  #pragma HLS PIPELINE
  str.write(inp[i]); // Reading from Input interface
}
```

在代码示例中，经流水打拍的 `for` 循环用于从输入存储器接口中读取数据，并将数据写入内部 `hls::stream` 变量。以上编码样式旨在从突发中的全局存储体读取数据。

这种编码样式建议用于在独立函数内部实现以上示例中的 `for` 循环操作，并应用 `dataflow` 最优化，如 数据流最优化 中所述。以下示例代码显示了其效果，它允许编译器在读取、执行和写入函数间建立数据流。

```
top_function(datatype_t * m_in, // Memory data Input
  datatype_t * m_out, // Memory data Output
  int inp1,     // Other Input
  int inp2) {   // Other Input
#pragma HLS DATAFLOW

hls::stream<datatype_t> in_var1;   // Internal stream to transfer
hls::stream<datatype_t> out_var1;  // data through the dataflow region

read_function(m_in, inp1, in_var1); // Read function contains pipelined for
loop
                            // to infer burst

execute_function(in_var1, out_var1, inp1, inp2); // Core compute function

write_function(out_var1, m_out); // Write function contains pipelined for
loop
                                 // to infer burst
}
```

标量输入

标量输入通常为从主机直接加载的控制变量。可将其视作为编程数据或参数，并且在这些数据或参数下执行主内核计算。这些内核是来自主机侧的只写输入。在以下函数中，标量参数为 `width` 和 `height`。

```
void process_image(int *input, int *output, int width, int height) {
```

对于标量实参，将分配由工具推断的默认 INTERFACE 编译指示。

```
#pragma HLS INTERFACE s_axilite port=width bundle=control
#pragma HLS INTERFACE s_axilite port=height bundle=control
```

在此示例中，有 2 个标量输入，分别指定图像的宽度 (`width`) 和高度 (`height`)。这些数据输入从主机直接进入内核，不通过全局存储体。工具不将所示编译指示添加到代码中。

重要提示！当前，Vitis 核开发套件仅支持每个内核一个控制接口捆绑。因此，所有标量数据输入和 `return` 函数的 `bundle=` 名称都应相同。在前述示例中，`bundle=control` 用于所有标量输入。

数据流传输接口

如果按顺序访问数据，那么就可以使用数据流传输接口。此接口支持从主机到内核以及从内核到主机的直接数据流传输，无需通过全局存储器作为中间步骤来移动数据。数据流传输接口也可以在 2 个内核之间使用，其中 1 个内核作为生产者将数据流传输到另一个充当使用者的内核。此传输同样是直接发生的，不使用全局存储器。如需了解更多信息，请参阅 数据流传输。

循环

对于高性能加速器而言，循环是其中不可或缺的一方面。通常，需要对循环进行流水打拍或者展开，方可充分利用 FPGA 架构的高度分布化和并行化特性，从而在性能提升方面比在 CPU 上运行时更胜一筹。

默认情况下，循环既不执行流水打拍，也不展开。在硬件中，执行每次循环迭代都需要至少 1 个时钟周期。从硬件角度来看，循环主体隐含“等待一个时钟周期”的意义。仅当前一次循环迭代完成之后，才能开始后一次迭代。

循环流水打拍

默认情况下，循环的每次迭代仅在前一次迭代完成后才会开始。在以下循环示例中，单次循环迭代会添加 2 个变量，并将结果存储在第 3 个变量中。假定在硬件中，此循环需要 3 个周期才能完成 1 次迭代。同时假定循环变量 `len` 为 20，即，`vadd` 循环在内核内运行 20 次迭代。因此，需要总计 60 个时钟周期（20 次迭代 * 3 个周期），才能完成此循环的所有操作。

```
vadd: for(int i = 0; i < len; i++) {
  c[i] = a[i] + b[i];
}
```

提示：最好始终按以上代码示例所示方式来标记循环 (`vadd:...`)。这样有助于在处理 Vitis 核开发套件时进行调试。请注意，标记在编译期间会生成警告，用户可放心忽略此警告。

循环流水打拍会以流水打拍方式来执行后续迭代。这意味着后续循环迭代会重叠并发运行，在循环主体的不同段中执行。循环流水打拍可藉由 pragma HLS pipeline 来启用。请注意，此编译指示置于循环主体内部。

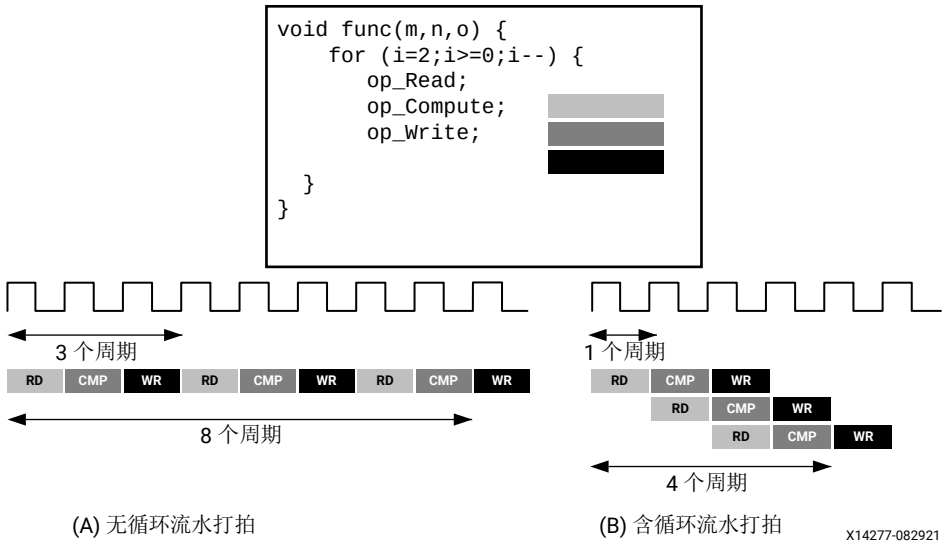
```
vadd: for(int i = 0; i < len; i++) {
  #pragma HLS PIPELINE
  c[i] = a[i] + b[i];
}
```

在以上示例中，假定循环的每次迭代需要 3 个周期：读取、添加和写入。如不使用流水打拍，则每过三个周期即可启动一次循环迭代。通过流水打拍，循环启动每个循环迭代的间隔将少于三个周期，例如，每过两个周期启动或者在每个周期内启动。

启动下一次循环迭代所需的周期数称为流水打拍循环的启动时间间隔 (II)。因此 II = 2 意味着每过两个周期启动一次循环迭代。II = 1 是理想情况，即每个周期内都会启动一次循环迭代。如果使用 `pragma HLS PIPELINE`，那么编译器始终会尝试实现 II = 1 的性能。

下图显示了流水打拍循环与非流水打拍循环的执行差异。在下图中，(A) 显示了默认顺序操作，每次输入读操作间存在 3 个时钟周期 (II = 3)，并且需要经过 8 个时钟周期后才会执行最后一次输出写操作。

图 18：**循环流水打拍**



在 (B) 所示的循环的流水打拍版本中，每个周期都会读取一次新输入样本 (II = 1)，仅需 4 个时钟周期后即可写入最终输出，在使用相同硬件资源的前提下显著改善 II 和时延。

重要提示！循环流水打拍会导致将流水打拍循环内部嵌套的任意循环展开。

如果循环内部存在任何数据依赖关系（如 循环依赖关系 中所述），则可能无法实现 II = 1，并且可能导致启动时间间隔增大。

循环展开

编译器还可将循环部分展开或完全展开，以便并行执行多次循环迭代。这是使用 pragma HLS unroll 来执行的。展开循环可显著提升并行度，从而实现超快设计。但由于循环迭代的所有操作都并行执行，因此需要大量可编程逻辑资源来实现硬件。由此导致编译器可能面临诸如处理大量资源等难题，并且可能遇到容量问题，从而导致内核编译进程减缓。最好仅对所含循环主体较小或者所含迭代数量较少的循环执行展开。

```
vadd: for(int i = 0; i < 20; i++) {
  #pragma HLS UNROLL
  c[i] = a[i] + b[i];
}
```

在前例中，您可以看到 `pragma HLS UNROLL` 已插入循环主体，以指示编译器完全展开循环。如果数据依赖关系允许，那么全部 20 项循环迭代都将并行执行。

提示：完全展开循环可能耗用大量器件资源，而部分展开循环则可提升性能同时减少所用硬件资源量。

循环已部分展开

要完全展开循环，此循环必须具有恒定界限（即上例中的 20）。但对于具有可变界限的循环，则可执行部分展开。部分展开的循环表示只能并行执行一定数量的循环迭代。

以下代码示例演示了部分展开的循环的工作方式：

```
array_sum:for(int i=0;i<4;i++){
  #pragma HLS UNROLL factor=2
  sum += arr[i];
}
```

在以上示例中，对于 UNROLL 编译指示赋值因数 2。这等同于手动复制循环主体，并发运行两个迭代，且运行数为迭代数的一半。为此编写的代码如下所示。此变换允许以上循环的两次迭代并行执行。

```
array_sum_unrolled:for(int i=0;i<4;i+=2){
  // Manual unroll by a factor 2
  sum += arr[i];
  sum += arr[i+1];
}
```

就像循环内部的数据依赖关系影响流水打拍式循环的启动时间间隔一样，展开的循环仅在数据依赖关系允许的前提下才会并行执行操作。如果某一循环迭代中的操作需要上一次循环的结果，则这两次迭代无法并行执行，但一旦来自任一迭代的数据可供另一迭代使用，即可立即执行。

建议：有效方法是首先对循环执行流水打拍 (PIPELINE)，然后对循环主体较小且迭代次数有限的循环执行展开 (UNROLL)，这样即可进一步提升性能。

循环依赖关系

循环中的数据依赖关系可能影响循环流水打拍或循环展开的结果。这些循环依赖关系可能存在于任一循环的单次迭代中，或者存在于任一循环的不同迭代之间。了解循环依赖关系的直接方法是检验极端示例。在以下代码示例中，循环结果用作为循环持续条件或循环退出条件。循环的每次迭代必须完成后才能开始下一次迭代。

```
Minim_Loop: while (a != b) {
  if (a > b)
    a -= b;
  else
    b -= a;
}
```

此循环无法流水打拍。前一次迭代结束后，下一次循环迭代才能开始。

处理与 Vitis 编译器之间的各种类型的依赖关系是一个牵涉很广的话题，需要详细了解编译器底层的高层次综合过程。如需了解更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399)。

嵌套循环

含嵌套循环的编码是很常见的。了解嵌套循环结构中循环的流水打拍方式是实现目标性能的关键。

如果将 HLS PIPELINE 编译指示应用于某个循环，并且此循环嵌套在另一个循环内，那么 v++ 编译器会尝试平铺此循环，以创建单一循环，并将 PIPELINE 编译指示应用于所构建的循环。平铺循环有助于改善内核性能。

编译器能够平铺以下类型的嵌套循环：

1. 完美嵌套循环：
   - 仅限内层循环才有循环主体。
   - 循环声明之间不指定任何逻辑或运算。
   - 所有循环边界均为常量。
2. 半完美嵌套循环：
   - 仅限内层循环才有循环主体。
   - 循环声明之间不指定任何逻辑或运算。
   - 内层循环边界必须为常量，但外层循环边界可为变量。

以下代码示例演示了完美嵌套循环的结构：

```
ROW_LOOP: for(int i=0; i< MAX_HEIGHT; i++) {
  COL_LOOP: For(int j=0; j< MAX_WIDTH; j++) {
    #pragma HLS PIPELINE
    // Main computation per pixel
  }
}
```

以上示例显示的嵌套循环结构包含两个循环，这两个循环对传入的像素数据执行部分计算。在大部分情况下，您希望在每个周期内对任一像素进行处理，因此将 PIPELINE 应用于嵌套循环主体结构。编译器能够平铺该示例中的嵌套循环结构，因为它是完美嵌套循环。

前述示例中的嵌套循环的两个循环声明之间不包含任何逻辑。在 `ROW_LOOP` 与 `COL_LOOP` 未布局任何逻辑；全部处理逻辑都位于 `COL_LOOP` 内。并且，这两个循环具有固定数量的迭代。这两项条件有助于 `v++` 编译器平铺循环并应用 PIPELINE 约束。

建议：如果外层循环具有变量边界，那么编译器仍可平铺此循环。您应始终尝试为内层循环设置常量边界。

顺序循环

如果设计中有多个循环，默认这些循环不会重叠，而是按顺序执行。本节介绍了顺序循环的数据流最优化概念。请考虑以下代码示例：

```
void adder(unsigned int *in, unsigned int *out, int inc, int size) {

  unsigned int in_internal[MAX_SIZE];
  unsigned int out_internal[MAX_SIZE];
  mem_rd: for (int i = 0 ; i < size ; i++){
    #pragma HLS PIPELINE
    // Reading from the input vector "in" and saving to internal variable
    in_internal[i] = in[i];
  }
  compute: for (int i=0; i<size; i++) {
  #pragma HLS PIPELINE
    out_internal[i] = in_internal[i] + inc;
  }

  mem_wr: for(int i=0; i<size; i++) {
  #pragma HLS PIPELINE
    out[i] = out_internal[i];
  }
}
```

在上述示例中，显示了 3 个顺序循环：mem_rd、compute 和 mem_wr。

- mem_rd 循环会从存储器接口读取输入矢量数据，并将其存储在内部存储空间中。
- 主 compute 循环会从内部存储空间读取，执行增量操作，并将结果保存到另一个内部存储空间中。
- mem_wr 循环会将数据从内部存储空间写回存储器中。

此代码示例使用 2 个不同循环来读取和写入存储器输入/输出接口，以推断突发读取/写入。

默认情况下，这些循环按顺序执行，无任何重叠。首先，mem_rd 循环完成读取所有输入数据，然后，compute 循环开始其操作。同样，compute 循环完成处理数据后，mem_wr 循环再开始写入数据。但这些循环可重叠执行，即当有足够数据可供 compute（或 mem_wr）循环处理时，允许前一个循环在 mem_rd（或 compute）循环完成处理其数据之前尽快开始操作。

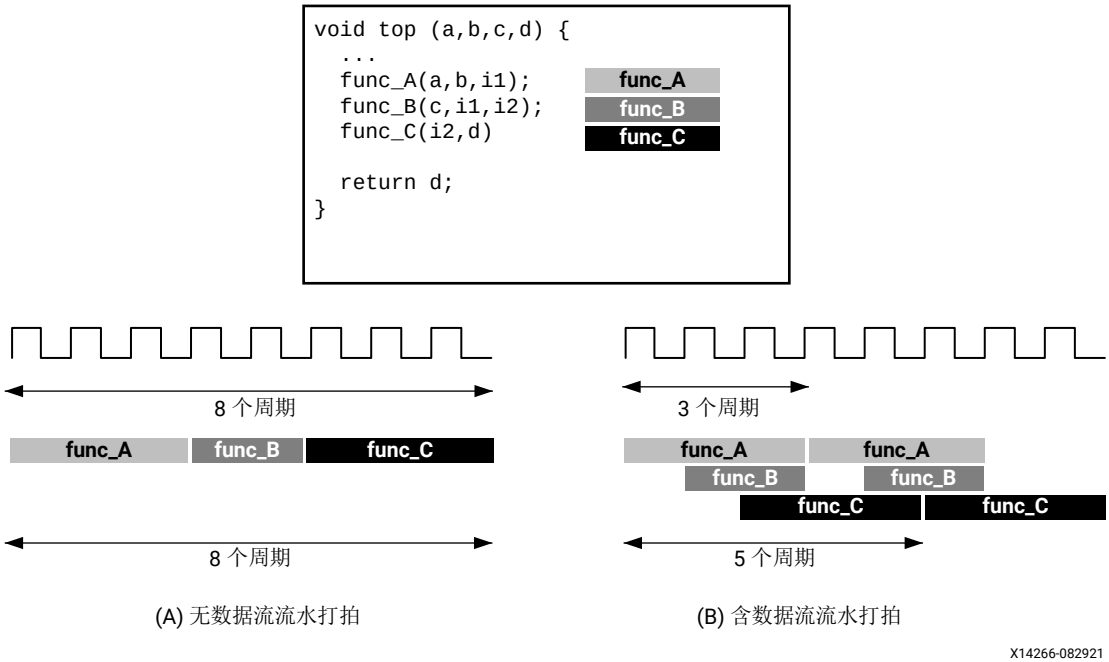
循环可使用数据流最优化来重叠执行，如 数据流最优化 中所述。

数据流最优化

数据流最优化是一种强大的技巧，它可通过支持内核内部的任务级流水打拍和并行化来提升内核性能。它允许 v++ 编译器通过并发调度内核的多个函数，来提升吞吐量并降低时延。这也称为任务级并行化。为此，您需要了解正确编写软件的最佳实践，以确保在 FPGA 上正常执行软件，如《Vitis 高层次综合用户指南》(UG1399) 中的吞吐量最优化中所述。

下图显示了数据流流水打拍的概念视图。默认行为是先执行并完成 func_A，然后执行并完成 func_B，最后执行并完成 func_C。启用 pragma HLS dataflow 后，编译器即可将每个函数调度为数据可用后立即执行。在此示例中，原始 top 函数的时延和时间间隔为 8 个时钟周期。借助数据流最优化，时间间隔缩短到仅 3 个时钟周期。

图 19：**数据流最优化**



数据流编码示例

在数据流编码示例中，您应留意：

1. 通过应用 pragma HLS dataflow 来指令编译器启用数据流最优化。它并非用于在 PS 与 PL 之间进行连接的数据移动器，而是用于解决数据流经加速器的方式。
2. `stream` 类用作为数据流区域内每个函数之间的数据传输通道。

> **提示：** `stream` 类可在可编程逻辑中用于推断先入先出 (FIFO) 存储器电路。此存储器电路在软件编程中充当队列，从而在函数之间提供数据级同步，并改善性能。

```
void compute_kernel(ap_int<256> *inx, ap_int<256> *outx, DTYPE alpha) {
  hls::stream<unsigned int>inFifo;
  #pragma HLS STREAM variable=inFifo depth=32
  hls::stream<unsigned int>outFifo;
  #pragma HLS STREAM variable=outFifo depth=32

  #pragma HLS DATAFLOW
  read_data(inx, inFifo);
  // Do computation with the acquired data
  compute(inFifo, outFifo, alpha);
  write_data(outx, outFifo);
  return;
}
```

数据流最优化的规范形式

赛灵思建议在数据流区域内使用规范形式来编写代码。函数和循环的数据流最优化都具有规范形式。

- 函数：函数内部的数据流的规范形式编码准则规定：
  1. 仅限在数据流区域内使用以下类型的变量：
     a. 局部非静态标量/阵列/指针变量。
     b. 局部静态 `hls::stream` 变量。
  2. 函数调用仅限正向传输数据。
  3. 阵列或 `hls::stream` 应仅含一个生产者函数和一个使用者函数。
  4. 函数实参（来自数据流区域外部的变量）应仅支持读取或写入，不得同时支持读取和写入。如果对同一函数实参同时执行读取和写入，那么应先读取后写入。
  5. 局部变量（按正向传输数据的变量）应先写入后读取。

  以下代码示例演示了函数中数据流的规范形式。请注意，第一个函数 (`func1`) 读取输入，最后一个函数 (`func3`) 写入输出。另请注意，一个函数创建的输出值将作为输入参数传递到下一个函数。

```
void dataflow(Input0, Input1, Output0, Output1) {
  UserDataType C0, C1, C2;
  #pragma HLS DATAFLOW
  func1(read Input0, read Input1, write C0, write C1);
  func2(read C0, read C1, write C2);
  func3(read C2, write Output0, write Output1);
}
```

- 循环：循环主体内部的数据流的规范形式编码准则包含适用于以上定义的函数的编码准则以及如下准则：
  1. 初始值为 0。

2. 循环条件是通过将循环变量与数值常量或循环主体内部不变的变量进行比较来形成的。
3. 按 1 递增。

以下代码示例演示了循环中数据流的规范形式。

```
void dataflow(Input0, Input1, Output0, Output1) {
             UserDataType C0, C1, C2;
             for (int i = 0; i < N; ++i) {
                  #pragma HLS DATAFLOW
                 func1(read Input0, read Input1, write C0, write C1);
                 func2(read C0, read C0, read C1, write C2);
                 func3(read C2, write Output0, write Output1);
               }
}
```

数据流故障排除

以下行为可能阻碍 Vitis 编译器执行数据流最优化：

1. 单一生产者使用者违例。
2. 绕过任务。
3. 任务间的反馈。
4. 任务的有条件执行。
5. 循环含多个退出条件或者循环内已定义条件。

如果在数据流区域内发生上述任何状况，您可能需要重构代码才能成功实现数据流最优化。

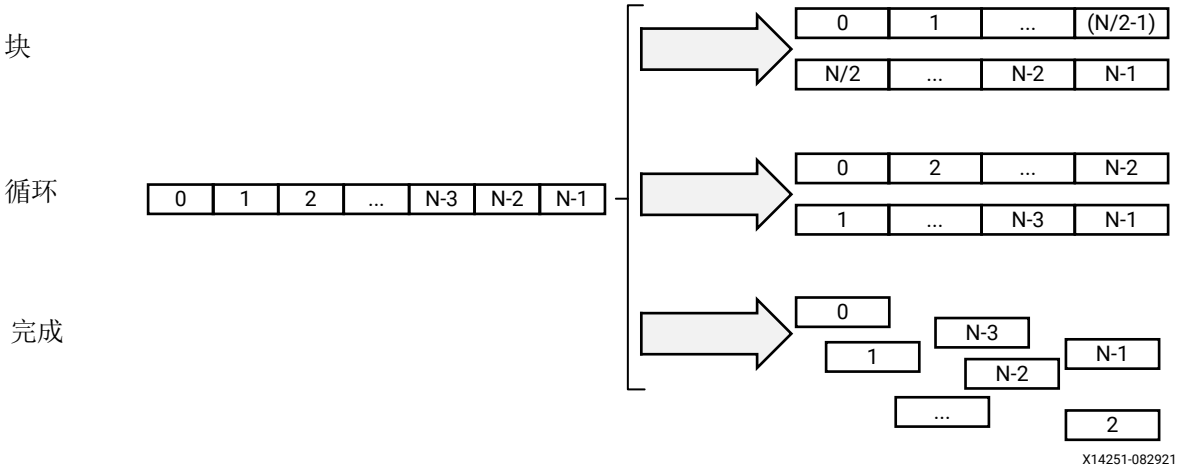
阵列配置

Vitis 编译器可将大型阵列映射到 PL 区域中的块 RAM 存储器。这些块 RAM 最多可包含 2 个访问点或端口。这可能限制应用性能，因为在硬件中实现时，无法并行访问阵列内的所有元素。

根据性能要求，您可能需要在同一个时钟周期内访问某一阵列的部分或全部元素。为此，pragma HLS array_partition 可用于指令编译器拆分阵列元素，并将其映射到较小的阵列或者映射到单个寄存器。编译器可提供 3 种类型的阵列分区，如下图所示。这 3 种分区类型分别是：

- `block`：原始阵列分割为原始阵列的连续元素块（大小相同）。
- `cyclic`：原始阵列分割多个大小相同的块，这些块交织成原始阵列的元素。
- `complete`：将阵列按其独立元素进行拆分。这对应于将存储器解析为独立寄存器。这是 ARRAY_PARTITION 编译指示的默认操作。

图 20：**阵列分区**



对于块分区和周期分区，`factor` 选项可指定要创建的阵列数量。在前图中，使用因子 2 将阵列分割为 2 个更小的阵列。如果阵列的元素数量并非该因子的整数倍，那么排列在后的阵列所含元素数量较少。

对多维阵列进行分区时，`dimension` 选项可用于指定对哪个维度进行分区。下图显示了使用 `dimension` 选项通过 3 种方式对以下代码示例进行分区的方式：

```
void foo (...) {
  // my_array[dim=1][dim=2][dim=3]
  // The following three pragma results are shown in the figure below
  // #pragma HLS ARRAY_PARTITION variable=my_array dim=3 <block|cyclic>
factor=2
  // #pragma HLS ARRAY_PARTITION variable=my_array dim=1 <block|cyclic>
factor=2
  // #pragma HLS ARRAY_PARTITION variable=my_array dim=0 complete
  int  my_array[10][6][4];
  ...
}
```

图 21：**阵列维度分区**

此图中的示例演示了如何通过对维度 3 进行分区来生成 4 个独立阵列，以及如何对维度 1 进行分区以生成 10 个独立分区。如果指定维度 0，则将对所有维度进行分区。

谨慎分区的重要性

完整的阵列分区会将所有阵列元素都映射到各独立寄存器。这样有助于提升内核性能，因为所有这些寄存器都可在同一周期内并发访问。

> **注意！**完整的大型阵列分区会耗用大量 PL 区域。甚至可能导致编译进程变慢，并出现容量问题。请仅在必要时执行阵列分区。请谨慎考虑选择对特定维度进行分区或者执行块分区或周期分区。

选择对特定维度进行分区

假设 A 和 B 是表示 2 个矩阵的二维阵列。请考量以下矩阵乘法算法：

```
int A[64][64];
int B[64][64];

ROW_WISE: for (int i = 0; i < 64; i++) {
  COL_WISE : for (int j = 0; j < 64; j++) {
    #pragma HLS PIPELINE
    int result = 0;
    COMPUTE_LOOP: for (int k = 0; k < 64; k++) {
      result += A[i ][ k] * B[k ][ j];
    }
    C[i][ j] = result;
  }
}
```

由于使用的是 PIPELINE 编译指示，ROW_WISE 和 COL_WISE 循环均已扁平化并结合在一起，COMPUTE_LOOP 已完全展开。要并发执行 COMPUTE_LOOP 的每次迭代，代码必须并行访问矩阵 A 的每个列和矩阵 B 的每个行。因此，矩阵 A 应在第二个维度内进行拆分，矩阵 B 应在第一个维度内进行拆分。

```
#pragma HLS ARRAY_PARTITION variable=A dim=2 complete
#pragma HLS ARRAY_PARTITION variable=B dim=1 complete
```

周期分区与块分区的选择

此处使用相同的矩阵乘法算法来演示如何通过了解底层算法的阵列访问模式，在周期分区与块分区之间进行选择，以及如何判定相应的因子。

```
int A[64 * 64];
int B[64 * 64];
#pragma HLS ARRAY_PARTITION variable=A dim=1 cyclic factor=64
#pragma HLS ARRAY_PARTITION variable=B dim=1 block factor=64

ROW_WISE: for (int i = 0; i < 64; i++) {
  COL_WISE : for (int j = 0; j < 64; j++) {
    #pragma HLS PIPELINE
    int result = 0;
    COMPUTE_LOOP: for (int k = 0; k < 64; k++) {
      result += A[i * 64 +  k] * B[k * 64 + j];
    }
    C[i* 64 + j] = result;
  }
}
```

在此版本的代码中，A 和 B 现在均为一维阵列。要并行访问矩阵 A 的每个列和矩阵 B 的每个行，可按上述示例中所示方式来使用周期分区和块分区。为并行访问矩阵 A 的每个列，此处应用 `cyclic` 分区，并将 `factor` 指定为行大小，此处即 64。同样，为并行访问矩阵 B 的每个行，此处应用 `block` 分区，并将 `factor` 指定为列大小，即 64。

通过缓存来最大程度减少阵列访问

由于阵列映射到含有限数量的访问端口的块 RAM，因此重复进行阵列访问可能限制加速器的性能。您应熟练掌握算法的阵列访问模式，并通过在本地缓存数据来限制阵列访问次数，从而提高内核性能。

以下代码示例显示了访问阵列导致最终实现性能受限的案例。在此示例中，对 `mem[N]` 阵列执行了 3 次访问以创建求和结果。

```
#include "array_mem_bottleneck.h"
dout_t array_mem_bottleneck(din_t mem[N]) {
  dout_t sum=0;
  int i;
  SUM_LOOP:for(i=2;i<N;++i)
    sum += mem[i] + mem[i-1] + mem[i-2];
  return sum;
}
```

前述示例中的代码可按以下示例中所示方式重写，以便按 II = 1 对代码进行流水打拍。通过执行预读取并手动对数据访问进行流水打拍，即可在循环每次迭代中仅指定 1 次阵列读取。这样可确保只需 1 个单端口块 RAM 即可实现性能目标。

```
#include "array_mem_perform.h"
dout_t array_mem_perform(din_t mem[N]) {
  din_t tmp0, tmp1, tmp2;
  dout_t sum=0;
  int i;
  tmp0 = mem[0];
  tmp1 = mem[1];
  SUM_LOOP:for (i = 2; i < N; i++) {
    tmp2 = mem[i];
    sum += tmp2 + tmp1 + tmp0;
    tmp0 = tmp1;
    tmp1 = tmp2;
  }
  return sum;
}
```

建议：请考虑通过缓存至本地寄存器来最大程度减少阵列访问，从而根据算法提升流水打拍性能。

函数内联

C 语言代码通常由多个函数组成。默认情况下，每个函数均由 Vitis 编译器单独进行编译和最优化。针对函数主体将生成一个唯一的硬件模块，并按需复用。

就性能而言，一般最好将函数内联或者消除函数层级。这样有助于 Vitis 编译器跨函数边界全局执行最优化。例如，如果在流水打拍的循环内部调用函数，那么内联函数有助于编译器进行更为激进的最优化，并提升循环的流水线性能（启动时间间隔 (II) 更短）。

以下 INLINE 编译指示置于函数主体内部，用于指示编译器将该函数内联。

```
foo_sub (p, q) {
  #pragma HLS INLINE
  ....
  ...
}
```

但如果函数主体过大并在主内核函数内部多次调用，那么内联函数可能因耗用过多资源而导致容量问题。如果您无法内联此类函数，那么请让 `v++` 编译器在其局部环境内单独最优化该函数。

在用户管理的永续内核中进行数据流传输

在典型的 XRT 管理的应用中，主机使用来自 `xrt::run` 类的 XRT 运行对象来管理内核的启动和停止，如 在器件上执行内核 中所述。在用户管理的内核中，则采用另一种启动和停止机制，但仍使用寄存器读取和写入调用由软件从主机应用来控制内核。这些 API 调用由 XRT 传输到低级调用中以启动和停止内核，这样会耗用少量时间用于在硬件上启动或停止内核并在每次启动内核时产生一些开销。当此数据作为内核输入（来自以太网或 SerDes）时，它以 1000 GB/s 的线速率运行，因此会产生巨大开销。

但数据驱动的内核数据流传输（例如此处所述内核）则可消除此类低级 API 调用，使内核响应来自以太网或 SerDes 的数据流时的数据率远高于上述线速率。此处所述的用户管理的内核只需启动一次并按需自动重启，因此被称为永续内核 (never-ending kernel)。这些内核在纯数据驱动的模式下执行，其数据流传输往来于 FPGA 的 I/O 管脚（以太网、SerDes）之间，或者在内核与其它内核之间执行往返数据流传输（内核至内核数据流传输）。

由于永续内核为数据驱动，内核操作取决于数据流，因此除初始启动之外，无需由用户通过主机应用对其进行管理，从而避免了来自主机程序的重复性 API 调用所产生的开销。这些内核需要由 Vitis HLS 指定的 `ap_ctrl_chain` 协议，并使用 `auto_restart` 位来使 `ap_start` 保持高位，并在内核启动后使其保持持续运行，或者直至 `auto_restart` 复位为止。主机应用必须手动启用 `auto_restart` 位，如 为用户管理的内核启用自动重新启动 中所述。下一节中详细讲解了内核要求。

内核编码指南

永续内核对于顶层函数实参具有下列编码要求：

- 内核通过实现 `ap_ctrl_chain` 块控制协议来启用 `auto_restart` 位。
- 内核支持 AXI4-Stream 接口 (`axis`)，且不含 AXI4 存储器映射 (`m_axi`) 接口，仅通过数据流传输来与其它内核进行交互。

在 C 语言中很难对使用数据流传输范例的设计进行建模。使用指针执行多次读取和/或写入访问的方法可能会引发问题，因为其中隐含了类型限定符。Vitis HLS 提供了 C++ 模板类 `hls::stream<ap_axis<N>>` 用于对流传输数据结构进行建模。在硬件上，`hls::stream` 是作为 `axis` 接口来实现的。

Vitis HLS 根据 `ap_axi_sdata.h` 中的定义，将 AXI4-Stream 接口作为含以下符号的结构体类型来实现：

```
template <typename T, size_t WUser, size_t WId, size_t WDest> struct axis
{ .. };
```

其中：

- `T` 是流传输数据类型
- `WUser` 是 TUSER 信号的宽度

- `WId` 是 TID 信号的宽度
- `WDest` 是 TDest 信号的宽度

当流传输数据类型 (`T`) 为简单的整数类型时，有 2 种预定义的 AXI4-Stream 实现类型可用：

1. AXI4-Stream 类的有符号实现：

```
hls::axis<ap_int<WData>, WUser, WId, WDest>
```

2. 无符号实现：

```
hls::axis<ap_uint<WData>, WUser, WId, WDest>
```

`TVALID`、`TREADY` 和 `TLAST` 均为 AXI4-Stream 协议的必需控制信号。旁路信号 `TKEEP`、`TSTRB`、`TUSER`、`TID` 和 `TDEST` 则属于特殊信号，不能用于传递其它簿记数据。为模板参数 `WUser`、`WId` 和 `WDest` 指定的值用于定义接口中旁路信号的使用方式，如《Vitis 高层次综合用户指南》(UG1399) 中的 AXI4-Stream 接口中所述。

以下示例显示了使用 AXI4-Stream 来处理数据驱动型永续内核的编程模型：

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32, 0, 0, 0> pkt;

extern "C" {
10 void krnl_stream_vdatamover(hls::stream<pkt> &in,
11     hls::stream<pkt> &out  // Internal Stream
12 ) {
13 #pragma HLS interface ap_ctrl_chain port=return
14 bool eos = false;
15 vdatamover:
16   do {
17     // Reading a and b streaming into packets
18     pkt t1 = in.read();
19
20     // Packet for output
21     pkt t_out;
22
23     // Reading data from input packet
24     ap_uint<DWIDTH> in1 = t1.data;
25
26     // Vadd operation
27     ap_uint<DWIDTH> tmpOut = in1;
28
29     // Setting data and configuration to output packet
30     t_out.data = tmpOut;
31     t_out.last = t1.last;
32     t_out.keep = -1; // Enabling all bytes
33
34     // Writing packet to output stream
35     out.write(t_out);
36
37     if (t1.last) {
38      eos = true;
39     }
40   } while (eos == false);
```

重要提示！ 永续内核展现出围绕内核使用 `while(1)` 循环的行为。因此，您不应在源代码中显式指定 `while(1)` 循环，以免行为出现不确定性。

总结

正如先前主题所述，使用 C/C++ 执行 FPGA 加速内核编码包含以下几个重要方面：

1. 考虑使用任意精度数据类型 `ap_int` 和 `ap_fixed`。
2. 了解用于确定标量和存储器接口的内核接口。如果链接阶段中将指定独立的 DDR 存储体，则使用含不同名称的 `bundle` 开关。
3. 对存储器接口使用突发读取和写入编码样式。
4. 在数据传输期间选择存储器数据输入和输出时，考虑利用 DDR 存储体的完整宽度。
5. 使用流水打拍和数据流来最大幅度提升性能。
6. 编写完美或半完美的嵌套循环，以便 `v++` 编译器可以有效应用流水线并将其扁平化。
7. 展开含少量迭代的循环，减少循环主体内部的操作计数。
8. 考虑了解阵列访问模式，并将 `complete` 分区应用于特定维度，或者应用 `block` 或 `cyclic` 分区，而不是应用整个阵列的 `complete` 分区。
9. 通过使用本地高速缓存来最大程度减少阵列访问，从而改善内核性能。
10. 考虑内联函数，尤其是在流水打拍区域内部。数据流内部函数不应内联。

XILINX

第 8 章

RTL 内核

在 Vitis 应用加速开发流程中，来自 Vivado® Design Suite 的 RTL IP 可封装为 XO 文件，此类文件可链接至 FPGA 可执行文件 (`.xclbin`) 中，前提是这些文件遵循 Vivado IP 封装准则以及 Vitis 编译器的系统链接相关要求。

如 内核属性 中所述，RTL 内核可以是用户管理的内核，这类内核不遵循 XRT 执行控制要求，而是改为实现由现有 RTL 设计所指定的任意数量的可行控制方案。或者，RTL 内核可遵循 XRT 管理的内核所需的 `ap_ctrl_chain` 或 `ap_ctrl_hs` 控制协议要求。

以下章节描述了有关 Vitis 编译器将内核链接到系统时需遵循的内核接口要求。这些要求适用于所有软件可控制的内核和非软件控制内核。其中还描述了 XRT 管理的内核的控制要求以及所有其它要求。最后还描述了相关开发流程，以帮助您在 Vivado® Design Suite 中将 RTL IP 作为 RTL 内核进行封装，以供在 Vitis 环境内使用。

RTL 内核的要求

要集成到 Vitis 工具流程中，RTL 模块必须至少满足 内核接口要求 中列举的要求。XRT 管理的内核与用户管理的内核都同样必须满足内核接口要求。

此外，XRT 管理的内核必须满足 XRT 管理的内核的控制要求 中所述的要求，才可供 XRT 执行和剖析。

用户管理的内核必须具有 Vitis 编译器所需的信号接口，才能允许编译器将内核链接到其它内核以及链接到目标平台，而无需遵守严格的 XRT 执行协议。通过这种方式，即可更快且更轻松地将现有 RTL IP 集成到 Vitis 环境中。

您可能需要修改自己的 RTL 模块才能满足以下部分中所概述的内核要求。

内核接口要求

为支持 Vitis 编译器将内核连接到目标平台，RTL 内核必须遵循 内核属性 中所述的要求。下表对各种接口要求进行了总结。

重要提示！ 在某些情况下，端口名称必须按表中所示进行定义。

表 7：**RTL 内核的接口和端口要求**

接口或端口	描述	注解
Clock	一个或多个时钟输入。	• 内核需要至少 1 个时钟。[1] • 可采用任意名称命名，但必须封装在总线接口内。 **重要提示！**封装 RTL 以供在 Vitis 环境内使用时，RTL IP 中的所有端口都必须与接口关联。否则，可能出现如下所示错误： `ERROR: UNDEF When packaging for Vitis, pins that are not part of an interface are not supported`
Reset	主低电平有效复位输入端口	• 可选端口。 • 可采用任意名称命名，但必须通过时钟上的 ASSOCIATED_RESET 属性与时钟信号关联。 • 该信号应经过内部流水打拍处理，以提升时序。 • 该信号由关联时钟域内的同步复位驱动。
interrupt	高电平有效中断。	• 可选端口。 • 使用此端口时，其名称必须与此处所示名称相同。
s_axi_control	唯一一个 AXI4-Lite 从控制接口	• 必需的端口。`s_axilite` 接口通常是必需的，但有时在使用 AXI4-Stream 接口时则存在例外情况。对于非软件控制的内核，也无需此端口。 • 使用此端口时，其名称必须与此处所示名称相同，且区分大小写。
AXI4_Memory Mapped Interface (m_axi)	AXI4 存储器映射接口，用于全局存储器访问	• 可选端口。 • 所有 AXI4 存储器映射接口都必须具有 64 位地址（在 Zynq-7000 器件上为 32 位）。 • RTL 内核开发者负责对全局存储器空间进行分区。全局存储器中的每一个分区都成为一个内核实参。每个分区的存储器偏移必须由软件应用通过 AXI4-Lite 接口中的寄存器提供给内核。 • AXI4 存储器映射接口不得使用“卷绕 (Wrap)”或“固定 (Fixed)”突发类型，并且不得使用窄（小型）突发。这意味着 AxSIZE 应与 AXI 数据总线宽度相匹配。 • 任何不符合上述要求的用户逻辑或 RTL 代码都必须加以卷绕或桥接，以满足这些要求。
AXI4_STREAM (axis)	AXI4-Stream 接口，适用于内核之间或者主机与内核之间的单向数据传输。	• 可选端口。 • 不得配合双向端口使用。 • 请使用 Vivado Design Suite 中的 STREAM 接口模板。 • 请参阅《Vitis 高层次综合用户指南》(UG1399) 中的 AXI4-Stream 接口，以获取有关接口要求的更多信息。

注释：

1. 此处列出的时钟要求适用于包含固定时钟的较新的平台 shell，如 管理时钟频率 中所述。可供在传统平台上使用的 RTL 内核支持两个名为 `ap_clk` 和 `ap_clk_2` 的时钟，还支持两个名为 `ap_rst_n` 和 `ap_rst_n_2` 的可选复位。

XRT 管理的内核的控制要求

重要提示！用户管理的内核无需下述控制寄存器和信号，但可在 `s_axilite` 接口中使用寄存器来实现控制结构，如 创建用户管理的 RTL 内核 中所述。如果您的 RTL 模块实现不同的控制结构，那么您可将其定义为 `user_managed` 内核，或者必须对其进行调整以使其符合 XRT 管理的要求。

下表概括了 XRT 管理的内核所需的寄存器映射，这些内核将在 Vitis 工具和 XRT 中使用。指定 `ap_ctrl_hs` 和 `ap_ctrl_chain` 控制协议的内核需要控制寄存器，如 执行模式 中所述。实现 `ap_ctrl_none` 和 `user_managed` 控制协议的内核则不需要下述控制寄存器。

提示：仅限实现中断的设计才需要中断相关的寄存器。

所有用户定义的寄存器必须从位置 `0x10` 开始；此位置以下的位置均为保留位置。这些寄存器包括用于内核实参的寄存器，例如传递给存储器映射接口的标量值和地址偏移。

表 8：**寄存器地址映射**

偏移	名称	描述
0x0	Control	控制并提供内核状态。
0x4	Global Interrupt Enable	用于启用对主机的中断。
0x8	IP Interrupt Enable	用于控制使用哪个 IP 生成的信号来生成中断。
0xC	IP Interrupt Status	提供中断状态。
0x10	Kernel arguments	包含标量和全局存储器实参等。

下表显示了通过控制寄存器 (`offset 0x0`) 访问的控制信号。控制寄存器及其信号由内核执行模式（`ap_ctrl_hs` 和 `ap_ctrl_chain`）来确定。

可用信号供不同控制协议使用，如 XRT 文档中的受支持的内核执行模型中所述。例如，对于顺序执行模式 `ap_ctrl_hs`，主机通常将 `0x00000001` 写入偏移 0 控制寄存器，这样即可设置位 0、清除位 1 和 2，并在读取 `ap_done` 信号时轮询，直至此信号为 1 为止。

表 9：**控制寄存器信号**

位	名称	描述
0	ap_start	当内核可以开始处理数据时，此信号断言有效。握手时清除此信号，同时 ap_done 断言有效。
1	ap_done	当内核已完成操作时，此信号断言有效。读取时清除此信号。
2	ap_idle	当内核处于空闲状态时，此信号断言有效。
3	ap_ready	当内核准备好接受新数据时，内核会将此信号断言有效。
4	ap_continue	此信号由 XRT 断言有效以允许内核继续保持运行
7	auto_restart	此信号用于启用自动内核重启，如 在用户管理的永续内核中进行数据流传输 中所述。
31:5	Reserved	保留

仅当内核有中断时，才需要以下与中断相关的寄存器。

表 10：**Global Interrupt Enable (0x4)**

位	名称	描述
0	Global Interrupt Enable	当此信号断言有效时，中断随 IP Interrupt Enable 位一起启用。
31:1	Reserved	保留

表 11：**IP Interrupt Enable (0x8)**

位	名称	描述
0	Interrupt Enable	当此信号断言有效时，中断随 Global Interrupt Enable 位一起启用。
31:1	Reserved	保留

表 12：**IP Interrupt Status (0xC)**

位	名称	描述
0	Interrupt Status	写入时切换。
31:1	Reserved	保留

中断

XRT 管理的 RTL 内核可以选择包含 `interrupt` 端口，其中包含单个中断。端口名称必须称为 `interrupt` 并且为高电平有效。当全局中断使能 (`GIE`) 位和中断使能寄存器 (`IER`) 位在“控制寄存器 (Control Register)”块中双双断言有效时，即可启用此端口。

默认情况下，IER 使用内部 `ap_done` 信号来触发中断。此外，仅当向 IP 中断状态寄存器 (IP Interrupt Status Register) 的位 0 写入 1 时，才会清除中断。

此逻辑应反映在 RTL 内核的 Verilog 代码中，并反映在关联的 `component.xml` 和 `kernel.xml` 文件中。`kernel.xml` 文件存储在 `kernel.xo` 文件内，并在使用 `package_xo` 命令或“RTL Kernel”Wizard 时自动生成。

创建用户管理的 RTL 内核

如果您的 RTL IP 不满足 Vitis 编译器的 AXI 接口要求（如 内核接口要求 中所述），那么您必须修改 IP 以实现所需接口。但如果您的 RTL IP 不满足 XRT 控制协议 `ap_ctrl_hs` 或 `ap_ctrl_chain`，那么您可将其定义为用户管理的内核，而无需重写 IP。

用户管理的内核无需满足 XRT 的控制要求，并且可实现各种执行机制。用户管理的内核允许您利用 Vitis 编译器的系统构建功能，同时允许您的内核实现您自己的控制方案。在此情况下并没有规定的内核启动、停止或控制方法。这些方法很大程度上取决于您的应用或系统的具体要求。可用的控制方案包括：

- 通过 `s_axilite` 控制接口访问寄存器，类似于 XRT 所使用的方法，但也可供您自己的实现使用。
- 通过软件驱动程序（例如，UIO 驱动程序）来访问主机应用中实现的硬件。
- 从独立组件提供的信号或者从其它内核触发您的内核的启动或停止响应。
- 提供数据驱动的方法，如 在用户管理的永续内核中进行数据流传输 中所述

重要提示！ 在 `s_axilite` 接口中为用户管理的内核实现控制寄存器的局限性之一在于此控制寄存器不得名为 CTRL。此名称为保留名称，专用于 XRT 管理的内核，在用户管理的内核（或 `ap_ctrl_none` 内核）上发现使用此名称时会返回“严重警告 (Critical Warning)”。

RTL 内核开发流程

本节解释了使用 Vivado Design Suite 内的“Package IP”功能特性来创建 RTL 内核的过程。“Package IP” 命令可提供“Package for Vitis”选项，该选项可以显著简化将现有 RTL IP 封装为赛灵思对象 (XO) 文件以供在 Vitis 环境内的过程。

封装后的 XO 文件作为一个容器，其中包含 Vivado IP 对象（包括源文件）以及关联的内核 XML 文件。通过使用 Vitis 编译器，可将 XO 文件与其它内核相结合，并与目标平台相链接，并通过构建 XO 文件来用于执行硬件流程或硬件仿真流程。

“Package for Vitis”功能特性可提供 DRC 用于检查封装 IP 的完整性，然后再生成 XO 文件，并且可自动执行 `package_xo` 命令以简化已封装的 RTL 内核的生成。

将 RTL 代码封装为 Vitis XO

重要提示！应首先按传统 RTL 验证方法对 RTL IP 进行完整验证，然后才能将其封装为内核。

如 内核接口要求 中所述，RTL 内核必须随以下必要接口一起封装：

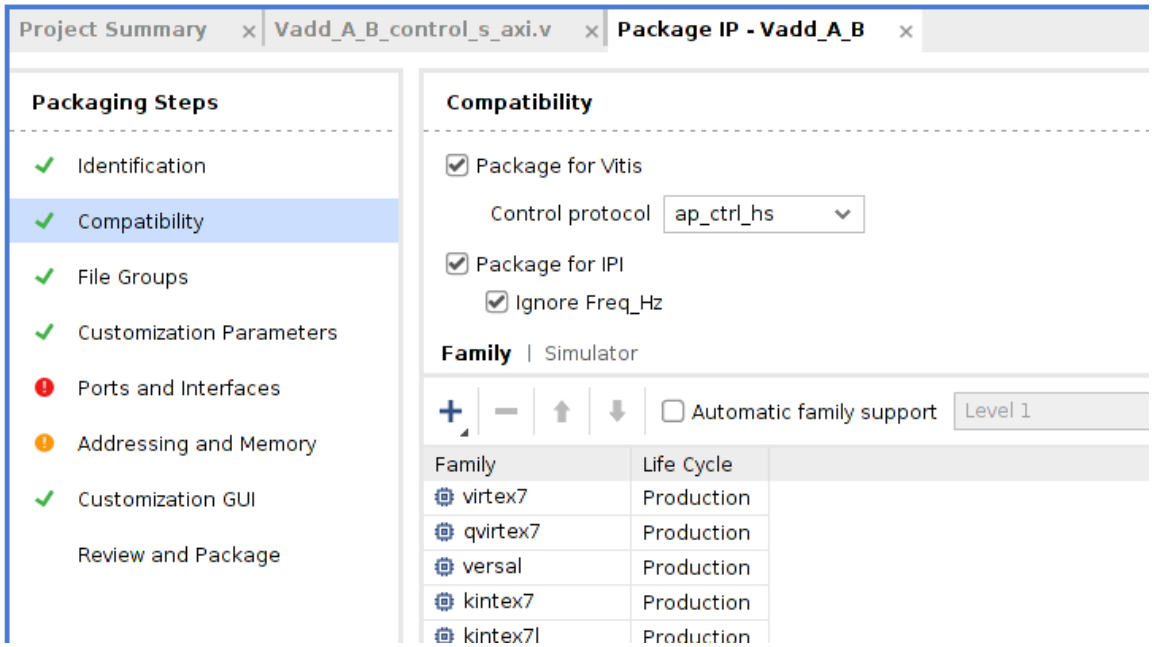
- AXI4-Lite 接口名称必须作为 `S_AXI_CONTROL` 封装，但是下面的 AXI 端口可以用不同的方式命名。
- 任意 AXI4 存储器映射接口都必须封装为 AXI4 主端点（支持 64 位地址）。

  建议：赛灵思强烈推荐 AXI4 接口与 AXI 元数据 `HAS_BURST=0` 和 `SUPPORTS_NARROW_BURST=0` 一起封装。这些属性可以在 IP 级 `bd.tcl` 文件中设置。这表明并未使用卷绕和固定突发类型，也未使用窄（小型）突发。
- 您还可实现 AXI4-Stream 接口。
- 内核需要至少 1 个时钟，但它可支持多个时钟。
  - 每个时钟都必须具有 1 个关联的总线接口用于将其识别为时钟。
  - 每个时钟都能有 1 个可选低电平有效复位，由时钟上的 ASSOCIATED_RESET 属性来指定。
  - 每个时钟都必须与内核上的每个 AXI4-Lite、AXI4 和 AXI4-Stream 接口关联。

要封装 IP，请使用以下步骤：

1. 创建和封装新 IP。
   a. 在 Vivado 工程中，添加 RTL 源文件，然后依次选中“Tools” → “Create and Package New IP”。
   b. 选择“Package your current project”，然后单击“Next”。
   c. 指定已封装的 IP 的位置。您可选择默认位置，也可以选择其它位置。
   d. 复查“Summary”页面，然后单击“Finish”以打开“Package IP”窗口。

   这样会打开“Package IP”窗口，以显示“Identification”页面。如需获取有关在 Vivado 工具中使用 IP 封装器的详细信息，请参阅《Vivado Design Suite 用户指南：创建和封装定制 IP》(UG1118)。
2. 选中“Packaging Steps”下的“Compatibility”。这样即可显示下图所示对话框。



a. 勾选“Package for Vitis”复选框以启用 RTL IP 封装进程，将其封装为 XO 以供在 Vitis 环境内使用。

b. 针对“RTL Kernel”选中“Control Protocol”。这样即可确定用于操作内核的控制机制。选项包括：

   - `user_managed`：定义软件可控制的内核，即由用户管理而非由 XRT 管理。这是首选的选项。如需了解更多信息，请参阅 创建用户管理的 RTL 内核。
   - `ap_ctrl_hs`：这是默认选项，为 XRT 管理的内核指定简单的顺序执行模型，如 软件可控内核 中所述。
   - `ap_ctrl_chain`：为 XRT 管理的内核指定流水打拍执行模型。
   - `ap_ctrl_none`：表示无控制协议，如 非软件控制的内核 中所述。

c. 检查确认“Package for IPI”和“Ignore Freq_Hz”均已启用。

启用这些复选框即可启用设计规则检查 (DRC)，`ipx::check_integrity` 命令会在封装 IP 和生成 XO 之前运行这些检查。这些 DRC 包括按 RTL 内核的要求 中所述方式检查所需的信号，并检查 XRT 管理的内核的控制协议和寄存器。如上图所示，遇到的任何问题都会向“Package IP”工具报告。

3. 将时钟关联到 AXI 接口。

   在“Package IP”窗口中选中“Ports and Interfaces”步骤即可将基准内核时钟与 AXI4 接口相关联，并按需复位信号。

   a. 右键单击 AXI4 接口，然后选中“Associate Clocks”。

      这样即可打开“Associate Clocks”对话框，其中会列出所有已识别的时钟信号。

   b. 选中相应的时钟，然后单击“OK”以将其与接口关联。

   c. 请务必对含每个 AXI 接口的时钟信号重复此步骤。

4. 单击“Addressing and Memory”步骤即可添加控制寄存器和偏移。

   使用 `ap_ctrl_hs` 或 `ap_ctrl_chain` 控制协议的 XRT 管理的内核需要控制寄存器，如 XRT 管理的内核的控制要求 中所述。下表显示了所需寄存器的列表。

提示：虽然 `ap_ctrl_none` 和 `user_managed` 控制协议无需控制寄存器，但如果在 RTL 设计中包含 `s_axilite` 接口，那么这两者仍可使用控制寄存器。在此情况下，具体寄存器可能不同于下表中所述，但 `names`、`offsets` 和 `widths` 的赋值进程是相同的。

表 13：**地址映射**

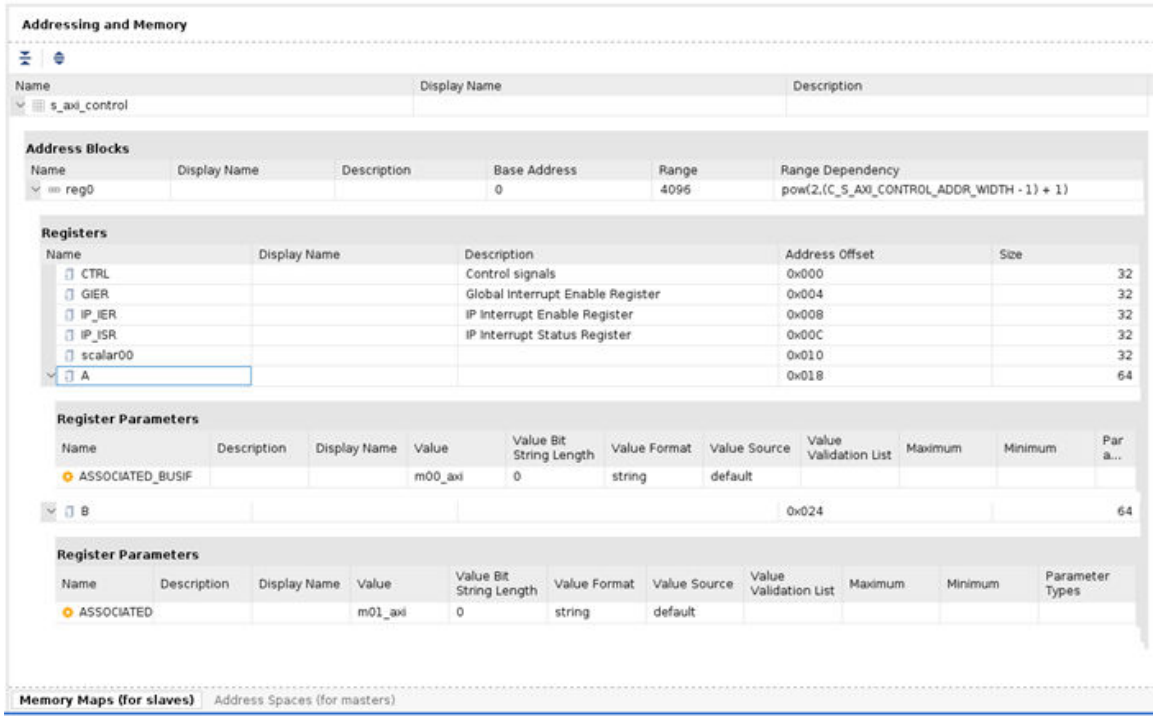
寄存器名称	描述	地址偏移	大小
CTRL	控制信号。 **重要提示！** CTRL 寄存器和 <kernel_args> 在所有内核上都是必需的。仅限含中断的设计才需要中断相关的寄存器。	0x000	32
GIER	全局中断使能寄存器。用于启用对主机的中断。	0x004	32
IP_IER	IP 中断使能寄存器。用于控制使用哪个 IP 生成的信号生成中断。	0x008	32
IP_ISR	IP 中断状态寄存器。提供中断状态。	0x00C	32
<kernel_args>	针对软件函数接口上所需的每个内核实参，都包含 1 个独立条目。所有用户定义的寄存器必须从位置 `0x10` 开始；此位置以下的位置均为保留位置。	0x010	32/64 标量实参位宽为 32 位。`m_axi` 和 `axis` 接口位宽则为 64 位。

a. 要创建该表中所述的地址映射，请在“Address Blocks”中右键单击并选择“Add Register”命令。

这样即可打开“Add Register”对话框，您可在其中输入上表中的某一寄存器名称。

b. 按需重复此步骤，以添加所需必需的寄存器。

这样即可创建“Registers”表格，如“Addressing and Memory”部分中所示。您可编辑该表以向每个寄存器添加“Description”、“Address Offset”和“Size”。这样，“Registers”表应如以下示例所示。



提示：此进程中每个步骤的对应 Tcl 命令都将写入 Tcl 控制台。您可据此执行整个进程，然后使用 Tcl 转录内容来创建脚本，以供为将来迭代自动执行整个进程。

c. 最后，选中表中每个指针实参的寄存器、右键单击并选择“Add Register Parameter”命令。在打开的对话框中输入 `ASSOCIATED_BUSIF` 名称，然后单击“OK”。

这样即可定义寄存器与 AXI4 接口之间的关联。在添加的参数的值字段中，输入分配给您正在定义的特定实参的 `m_axi` 接口的名称。在以上示例中，实参 `A` 使用的是 `m00_axi` 接口，而实参 `B` 使用的则是 `m01_axi` 接口。

5. 此时，您已准备就绪，可以进行 IP 封装了。
   a. 在“Package IP”窗口中选中“Review and Package”部分，复查“Summary and After Packaging”部分，然后按需执行更改。

      重要提示！您必须启用 IP 存档文件生成。如果“After Packaging”部分指出“An archive will not be generated.”，则您必须选择“Edit packaging settings”链接并启用“Create archive of IP”设置。

   b. 准备就绪后，即可单击“Package IP”。

      Vivado 工具会封装您的内核 IP、按需自动运行 `package_xo` 命令以生成 XO 文件，并打开对话框以通知您操作成功。

      为 RTL 内核生成的 XO 文件可供 Vitis 编译器在链接进程中用于连接至其它 HLS 或 RTL 内核，以及用于链接目标平台以完成整个系统。如需了解更多信息，请参阅 第三部分：构建和运行应用。

   c. 如果您的 RTL 内核包含某些定制功能，且对于自动运行的 `package_xo` 命令而言这些属于非标准功能，那么您可使用定制设置来手动运行此命令以重新生成 XO 文件和内核。请参阅 package_xo 命令 以获取有关此命令的详细信息。需手动运行 `package_xo` 命令的可能原因包括：
      - 指定不同的 IP 目录或 XO 路径
      - 输出 `kernel.xml` 文件副本以供稍后进行编辑和复用
      - 使用 `package_xo -kernel_files` 选项以包含 C 语言模型，以便支持对 RTL 内核进行软件仿真 (software emulation)

6. 可选：测试“Packaged IP”.

   要测试 RTL 内核是否已正确封装以供 IP integrator 使用，请尝试将封装的内核 IP 例化到 IP integrator 中的块设计中。如需了解有关此工具的信息，请参阅《Vivado Design Suite 用户指南：采用 IP integrator 设计 IP 子系统》(UG994)。

   此内核 IP 应显示上述各接口。请在画布视图中检验 IP。AXI 接口属性可以通过选择画布上的接口查看。然后，在“Block Interface Properties”窗口中，选中“Properties”选项卡，并展开“CONFIG”表格条目。如果接口为只读或只写，可以移除未使用的 AXI 通道，将 `READ_WRITE_MODE` 设置为只读或只写。

7. 可选：配置设计约束

   如果 RTL 内核具有设计约束 (`.xdc`)，并且此约束引用平台的静态区域的元素（例如，时钟），那么需将此约束文件标记为“late processing order”以确保正确应用 RTL 内核约束。

   有两种方法可标记约束以供后续处理：

   a. 如果在 `.ttcl` 文件中提供约束，请将 `<: setFileProcessingOrder "late" :>` 添加到此文件的 `.ttcl` 前导码部分，如下所示：

```
<: set ComponentName [getComponentNameString] :>
<: setOutputDirectory "./" :>
<: setFileName $ComponentName :>
<: setFileExtension ".xdc" :>
<: setFileProcessingOrder "late" :>
```

b. 如果在 .xdc 文件中定义约束，则请在 component.xml 中添加从 <spirit:define> 开始的 4 行内容。component.xml 中的这 4 行内容需位于调用 .xdc 文件的区域旁。在以下示例中，根据已定义的后续处理顺序来调用 my_ip_constraint.xdc 文件。

```
<spirit:file>
        <spirit:name>ttcl/my_ip_constraint.xdc</spirit:name>
        <spirit:userFileType>ttcl</spirit:userFileType>
        <spirit:userFileType>USED_IN_implementation</
spirit:userFileType>
        <spirit:userFileType>USED_IN_synthesis</spirit:userFileType>
        <spirit:define>
             <spirit:name>processing_order</spirit:name>
             <spirit:value>late</spirit:value>
        </spirit:define>
</spirit:file>
```

RTL 内核设计建议

虽然“RTL Kernel”Wizard 能有助于封装 RTL 设计以供在 Vitis 核开发套件内使用，但仍应遵循《UltraFast 设计方法指南（适用于赛灵思 FPGA 和 SoC）》(UG949) 提供的建议来设计底层的 RTL 内核。

除了遵循接口和封装要求外，设计内核时还应谨记下列性能目标：

- AXI4 接口的存储器性能最优化
- 结果质量考虑因素
- 调试和验证的考虑因素

AXI4 接口的存储器性能最优化

AXI4 接口通常连接到平台中的 DDR 存储器控制器。

建议：为了达成最优频率和资源利用率，建议每个存储器控制器使用一个接口。

为了发挥最佳的存储器控制器性能，AXI 接口行为建议如下：

- 使用的 AXI 数据宽度与本机存储器控制器 AXI 数据宽度（通常为 512 位）相匹配。
- 请勿使用 WRAP、FIXED 或大小过小的突发。
- 使用尽可能大的突发传输（根据 AXI4 协议限制，最多可用 4k 字节）。
- 避免使用已断言无效的写入选通。断言无效的写入选通可能导致 DDR 存储器控制器中的纠错码 (ECC) 逻辑执行读取 - 修改 - 写入操作。
- 使用 AXI 传输事务流水打拍。
- 如果 AXI 接口仅连接到一个 DDR 控制器，请避免使用线程。
- 如果内核无法提供完整写入传输事务（非阻塞写入请求），请避免生成写入地址命令。
- 如果内核不能在无反压（非阻塞读取请求）的情况下接受所有读取数据，则请避免生成读取地址命令。
- 如果需要只读或只写接口，则在将工程封装到内核之前，可以在顶层 RTL 文件中注释掉未使用通道的端口。
- 使用多个线程可能导致内核与存储器控制器之间的基础架构 IP 中的资源需求增加。

结果质量考虑因素

以下建议有助于提升时间和区域的结果：

- 对全部复位输入进行流水打拍并在内部分配复位，避免高扇出信号线。
- 仅复位基本控制逻辑触发器。
- 尽可能考虑寄存输入和输出信号。
- 了解内核相对于目标平台容量的大小，以确保适合，尤其是在例化多个内核的情况下。
- 识别使用堆叠硅片互联 (SSI) 技术的平台。这些器件包含多个裸片，跨这些裸片之间的任何逻辑都应该是触发器到触发器的路径。

调试和验证的考虑因素

- 应使用先进的验证技术（包括验证组件、随机化和协议检查器）在自己的测试激励文件中验证 RTL 内核。Vivado IP 目录提供 AXI Verification IP (VIP)，可帮助验证 AXI 接口。RTL 内核设计示例包含基于 AXI VIP 的测试激励文件，其中带有样本激励文件。
- 硬件仿真应该用于测试主机代码软件集成或查看多个内核之间的交互。

XILINX

第 9 章

利用 Vitis 加速的最佳实践

以下提供了一些在 Vitis™ 核开发套件内开发应用代码和硬件函数时需要牢记的要点。

- 请复查 使用 Vitis 软件平台加速应用的方法论 章节，以获取有关加速方法论的信息。
- 注意计算时间与输入和输出数据量比率较高的加速函数。使用 FPGA 内核可以大幅减少计算时间，但是数据量带来了传输时延。
- 对具有独立控制结构且不需要与主机定期同步的函数进行加速。
- 将大型数据块从主机传输到全局器件存储器。一次大规模传输要比多次小规模传输效率更高。运行带宽测试，寻找最佳传输大小。
- 仅在必要时才将数据复制回主机。由内核写入全局存储器的数据可供另一个内核直接读取。存储器资源包括 PLRAM（小尺寸，快速访问，时延最低）、HBM（中等尺寸，中等访问速度，存在一定时延）和 DDR （大尺寸，访问速度较慢，高时延）。
- 利用多个全局存储器资源优势，在各个内核之间均匀分配带宽。
- 通过执行 512 位宽突发，最大程度提升内核与全局存储器之间的带宽利用率。
- 将数据缓存在内核的本地存储器中。访问本地存储器比访问全局存储器快得多。
- 在主机应用中，使用事件和非阻塞传输事务，以并行和重叠方式启动多个请求。
- 在 FPGA 中，使用不同的内核来充分利用任务级并行度，使用多个 CU 来充分利用数据级并行度，从而并行执行多个任务并进一步提升性能。
- 在内核中，将任务级并行度用于数据流，将指令级并行度用于循环展开和循环流水打拍，从而最大程度提升吞吐量。
- 某些赛灵思 FPGA 包含多个被称为超级逻辑区域 (SLR) 的分区。将内核保存在与其访问的全局存储体相同的 SLR 中。
- 经常使用软件和硬件仿真来确认您的代码，确保其功能正确。
- 经常审查 Vitis 指南报告，因为它提供与工程缺陷有关的清晰且切实可行的反馈。

第三部分

构建和运行应用

主机程序与内核代码编写完成后，您即可构建应用，包括构建主机程序和平台文件 (`xclbin`)。主机程序与内核代码的构建进程遵循标准编译和链接进程，随后将对输出进行封装以供使用。但是，构建应用的第一步是识别构建目标，明确是为了应用测试或仿真执行构建还是为了目标硬件执行构建。构建完成主机程序和 FPGA 二进制文件后，也就意味着您已准备就绪，可以运行应用了。

本部分包含以下章节：

- 设置 Vitis 环境
- 构建目标
- 构建主机程序
- 构建器件二进制文件
- 封装系统
- v++ 命令的输出目录
- 运行仿真
- 运行应用硬件构建

XILINX

第 10 章

设置 Vitis 环境

Vitis™ 统一软件平台包含以下 3 大要素，必须正确安装并配置并使其能够正常协同工作：Vitis 核开发套件、赛灵思的 Xilinx Runtime (XRT) 以及加速器卡（如 Alveo™ 数据中心加速器卡）。如需了解安装和配置要求，请参阅 安装 中的描述。

如果您已安装 Vitis 软件平台的各要素，则需要通过运行以下脚本来设置环境以在特定命令 shell 内运行（.csh 脚本也已一并提供）。

```
#setup XILINX_VITIS and XILINX_VIVADO variables
source <Vitis_install_path>/settings64.sh
#setup XILINX_XRT
source /opt/xilinx/xrt/setup.sh
```

您也可以通过设置以下环境变量来指定可用平台的位置，以搭配 Vitis IDE 使用：

```
export PLATFORM_REPO_PATHS=<path to platforms>
```

提示：PLATFORM_REPO_PATHS 环境变量指向包含平台文件 (.xpfm) 的目录。

XILINX

第 11 章

构建目标

Vitis™ 工具的构建目标用于定义在编译和链接期间创建的 FPGA 二进制文件 (`.xclbin`) 的性质和内容。有 3 种不同的构建目标：软件仿真和硬件仿真这 2 个仿真目标用于确认和调试，默认系统硬件目标则用于生成 FPGA 二进制文件 (`.xclbin`) 以供加载到赛灵思器件中。

仿真目标的编译比实际硬件的编译要快得多。仿真运行是在仿真环境中执行的，可增强调试可视性，且无需使用真实的加速器卡。

表 14：**将仿真流程与硬件执行进行比较**

软件仿真	硬件仿真	硬件执行
主机应用使用内核的 C/C++ 或 OpenCL™ 模型来运行。	主机应用使用内核的仿真 RTL 模型来运行。	主机应用使用内核的实际硬件实现来运行。
用于确认系统的功能正确性。	测试主机/内核集成，获取性能估算结果。	确认系统正确运行且具备期望的性能。
构建时间越短，则设计迭代越快。	最佳调试功能、中等编译时间，内核可视性提升。	最终 FPGA 实现，构建时间长，且性能结果准确（真实）。

软件仿真

软件仿真 (`sw_emu`) 的主要目的是确保主机程序与内核的功能正确性。软件仿真可以提供纯功能执行，不对时序延迟或时延进行任何建模，它提供任何加速器性能指标。

内核代码始终在本机进行编译和运行。应用代码则满足以下任一条件：

- 在 x86 处理器上进行本机编译和运行（数据中心平台）
- 在 Arm® 处理器上执行交叉编译，并在仿真器中运行（嵌入式平台）

因此，软件仿真通常用于算法优化、调试功能问题以及支持开发者快速迭代代码以进行改进。快速编译和运行迭代的软件编程模型保留不变。

`v++` 编译器对内核代码执行最低限度的变换，以创建 FPGA 二进制文件，用于将主机程序与内核代码结合在一起来运行。软件仿真会提取基于 C 语言的内核代码，并使用 GCC 对其进行编译。它会将每个内核作为独立的 C 语言线程来运行。如果每个内核有多个计算单元，那么每个 CU 都作为独立线程来运行。因此，它模仿硬件的并行执行模型。虽然在硬件上运行时，内核内可能存在并行操作，但在每个内核内，会按顺序对执行进行建模。软件仿真驱动程序会实现 XRT API 并用于桥接运行 XRT 的用户应用与对硬件组件进行建模的器件进程。

提示：对于 RTL 内核，如果 C 模型与内核关联，则可以支持软件仿真。RTL 内核开发流程 提供了将 C 语言模型文件与 RTL 内核相关联的选项，以支持软件仿真流程。

以下描述了软件仿真的限制：

- 存在全局存储器限制 16 GB，对于仿真，不应超出此限制。

- 针对 AI 引擎内核，不支持软件仿真。
- 软件仿真不支持不含旁路的 AXI4-Stream 接口（请参阅《Vitis 高层次综合用户指南》(UG1399)）。

正如 Vitis 编译器命令 中所述，在 `v++` 命令中通过 `-t` 选项指定软件仿真目标：

```
v++ -t sw_emu ...
```

您可以将 GDB 调试器用于主机应用和内核代码、设置断点，或者使用 `printf()` 来打印信息和检查点。如需了解有关在软件仿真期间调试主机应用或内核的详细信息，请参阅 在软件仿真中调试。

硬件仿真

硬件仿真会对可编程逻辑设计运行 RTL 仿真，其中 PL 内核将与周期精确的硬件平台模型集成。

硬件仿真对于以下任务特别有用：

- 检查从 C、C++ 或 OpenCL 内核代码综合后的 RTL 代码的功能正确性。
- 测试不同内核或多个 CU 之间的交互
- 使用硬件波形来详细了解内核的内部活动
- 获取应用的初始性能估算

每个内核都编译为硬件模型 (RTL)。在硬件仿真期间，内核在 Vivado 逻辑仿真器内运行，通过波形查看器来检验内核设计。部分第三方仿真器也同样受支持，如 仿真器支持 中所述。此外，硬件仿真可为硬件实现提供性能和资源估算。

对于硬件平台内使用的关键 IP 提供了 SystemC 模型，如 Versal NoC/DDR 存储器、CIPS、PS 块、AI 引擎、UltraScale + MIG DDR 存储器和 AXI4 SmartConnect。这些 IP 模型在硬件仿真期间用于改善仿真性能和结果。

在硬件仿真中，编译和执行时间比软件仿真更长，但它可提供详细且周期精确的内核活动视图。赛灵思建议在硬件仿真期间使用小型数据集进行确认，以使运行时保持可管理性。

重要提示！硬件仿真中使用的 DDR 存储器模型和 Memory Interface Generator (MIG) 模型均为高级仿真模型。这些模型可提供良好的仿真性能，但是只能提供近似时延值并且不像内核那样具有周期精确性。因此，剖析汇总报告中显示的性能数值均为近似值，应仅用作为参考，用于比较不同内核实现之间的相对性能。

正如 Vitis 编译器命令 中所述，在 `v++` 命令中通过 `-t` 选项指定硬件仿真目标：

```
v++ -t hw_emu ...
```

系统硬件目标

如果构建目标为硬件，`v++` 会通过在设计上运行 Vivado 综合与实现，来为赛灵思器件构建 FPGA 二进制文件。一般情况下，此构建目标耗时比在 Vitis IDE 中生成软件或硬件仿真目标更长。但最终 FPGA 二进制文件可以加载到加速器卡的硬件中或嵌入式处理器平台中，并且应用可在其实际操作环境内运行。

正如 Vitis 编译器命令 中所述，在 `v++` 命令中通过 `-t` 选项指定系统硬件目标：

```
v++ -t hw ...
```

XILINX

第 12 章

构建主机程序

使用 XRT 本机 API 或 OpenCL™ API 调用以 C/C++ 编写的主机程序是使用基于 GNU 编译器集合 (GCC) 的 GNU C++ 编译器 (g++) 构建的。每个源文件都编译到一个对象文件 (.o) 内，并与赛灵思的 Xilinx Runtime (XRT) 共享库相连，以创建在主机 CPU 上运行的可执行文件。

提示： g++ 支持许多标准 GCC 选项，此处并未详细记录所有选项。如需了解相关信息，请参阅 GCC Option Summary。

为 x86 执行编译和链接

提示： 运行工具前，请按 设置 Vitis 环境 中所述方式设置命令 shell 或窗口。

使用 g++ 编译器将主机应用的每个源文件都编译到对象文件 (.o) 中。

```
g++ ... -c <source_file1> <source_file2> ... <source_fileN>
```

生成的对象文件 (.o) 与赛灵思的 Xilinx Runtime (XRT) 共享库相链接以创建可执行的主机程序。使用 -l 选项执行链接。

```
g++ ... -l <object_file1.o> ... <object_fileN.o>
```

为 x86 执行编译和链接遵循标准 g++ 流程。唯一要求是包含 XRT 头文件，并链接 XRT 共享库。

编译源代码时，需以下 g++ 选项：

- -I$XILINX_XRT/include/：XRT 的 include 目录。
- -I$XILINX_VIVADO/include：Vivado 工具的 include 目录。
- -std=c++11：定义 C++ 语言标准。

链接可执行文件时，需以下 g++ 选项：

- -L$XILINX_XRT/lib/：在 XRT 库中查找。
- -lOpenCL：在链接期间搜索指定的库。
- -lthread：在链接期间搜索指定的库。
- -lrt：在链接期间搜索指定的库。
- -lstdc++：在链接期间搜索指定的库。

注释：在 Vitis 示例中，您可以看到添加 `xcl2.cpp` 源文件的过程和 `-I../libs/xcl2` 包含语句。这些添加到主机程序的内容和 `g++` 命令可提供对代码示例所使用的帮助程序实用工具的访问权，但对于您自己的代码而言，通常并不需要访问这些工具。

构建 XRT 本机 API

XRT 为 C、C++ 和 Python 提供了一个 XRT 本机 API，如 XRT 网站 https://xilinx.github.io/XRT/2020.2/html/xrt_native_apis.html 上所述。要使用此 XRT 本机 API，主机应用必须与 `xrt_coreutil` 库相链接。命令行使用另一组不同设置（如下示例所示）来组合编译和链接进程：

```
g++ -g -std=c++14 -I$XILINX_XRT/include -L$XILINX_XRT/lib -lxrt_coreutil -
lthread \
-o host.exe host.cpp
```

重要提示！XRT API 需要使用 `-std=c++14`。

为 Arm 执行编译和链接

提示：运行工具前，请按 设置 Vitis 环境 中所述方式设置命令 shell 或窗口。

为 Arm 处理器执行主机程序 (`host.exe`) 的交叉编译和链接使用的是以下 2 步流程：

1. 使用 `g++` 的 GNU Arm 交叉编译器版本将 `host.cpp` 编译到对象文件 (`.o`) 中：

   注释：`aarch64` 适用于 Zynq® UltraScale+™ (A53) 和 Versal™ (A72) 器件。`aarch32` 适用于 Zynq-7000 SoC (A9)，工具链位于不同位置。

   ```
   $XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ \
   -D__USE_XOPEN2K8 -I$SYSROOT/usr/include/xrt -I$XILINX_VIVADO/include \
   -I$SYSROOT/usr/include -c -fmessage-length=0 -std=c++14 \
   --sysroot=$SYSROOT -o src/host.o ../src/host.cpp
   ```

2. 将对象文件与所需的库相链接，以构建可执行应用。

   ```
   $XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ \
   -o host.exe src/host.o -lxilinxopencl -lpthread -lrt -lstdc++ -lgmp -
   lxrt_core \
   -L$SYSROOT/usr/lib/ --sysroot=$SYSROOT
   ```

编译应用用于嵌入式进程时，必须为应用指定 `sysroot`。`sysroot` 是定义基本系统根文件结构的平台中的一部分，并按 安装嵌入式平台 中所述方式进行安装。

重要提示！以上示例依赖于使用 `$SYSROOT` 环境变量，必须使用该环境变量才能为嵌入式环境指定 `sysroot` 的位置。

以下是为边缘平台编译主机代码的关键要素：

- 编译：
  - 所需的交叉编译器为 `aarch64-linux-gnu-g++`，可在 Vitis 安装层级下找到。

- 所需的包含路径为：
  - `$SYSROOT/usr/include`
  - `$SYSROOT/usr/include/xrt`
  - `$XILINX_VIVADO/include`
- 链接：
  - `$SYSROOT/usr/lib`：库路径位置。
  - `xilinxopencl`：XRT 所需的库。
  - `pthread`：XRT 所需的库。
  - `rt`：XRT 所需的库。
  - `stdc++`：XRT 所需的库。
  - `gmp`：XRT 所需的库。
  - `xrt_core`：XRT 所需的库。

构建 XRT 本机 API

XRT 为 C、C++ 和 Python 提供了一个 XRT 本机 API，如 XRT 网站 https://xilinx.github.io/XRT/2020.2/html/xrt_native_apis.html 上所述。要使用此 XRT 本机 API，主机应用必须与 `xrt_coreutil` 库而不是 `xlinxopencl` 库相链接。命令行使用另一组不同设置（如下示例所示）来进行编译和链接：

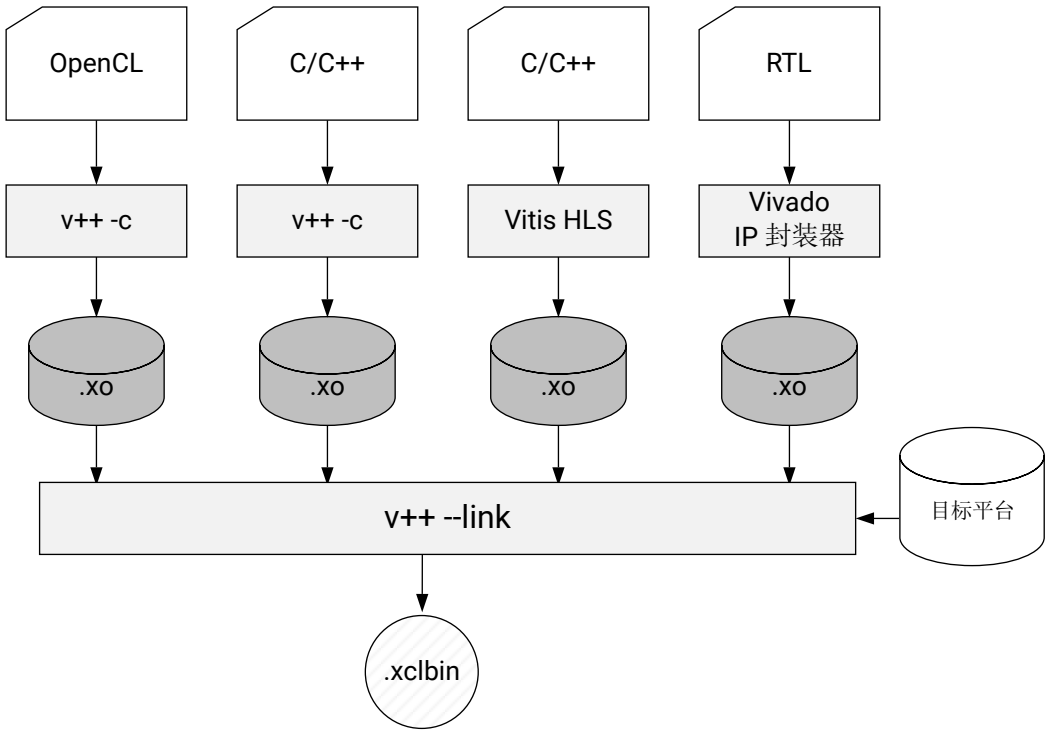
```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ -c \
-D__USE_XOPEN2K8 -I$SYSROOT/usr/include/xrt -I$XILINX_VIVADO/include \
-I$SYSROOT/usr/include -fmessage-length=0 -std=c++14 --sysroot=$SYSROOT \
-o src/host.o ../src/host.cpp
```

```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ -l \
-lxrt_coreutil -lpthread -lrt -lstdc++ -lgmp -lxrt_core -L$SYSROOT/usr/lib/
\
--sysroot=$SYSROOT -o host.exe src/host.o
```

构建器件二进制文件

内核代码是以 C、C++、OpenCL™ C 或 RTL 编写的，通过将内核代码编译到赛灵思对象 (XO) 文件中来进行构建，并将 XO 文件链接到赛灵思二进制文件 (`.xclbin`)，如下图所示。

图 22：**器件构建进程**



X21155-082921

该进程（如上所述）分 2 个步骤：

1. 从内核源代码构建赛灵思对象文件。
   - 对于 C、C++ 或 OpenCL 内核，`v++ -c` 命令可将源代码编译到赛灵思对象 (XO) 文件中。多个内核将编译到不同的 XO 文件中。
   - 对于 RTL 内核，Vivado IP 封装器命令会生成要用于链接的 XO 文件。如需了解更多信息，请参阅 RTL 内核。
   - 您也可以创建可在 Vitis™ HLS 工具中直接使用的内核对象 (XO) 文件。如需了解更多信息，请参阅 使用 Vitis HLS 编译内核。
2. 编译后，`v++ -l` 命令会将一个或多个内核对象 (XO) 与硬件平台 XSA 文件链接到一起，以生成赛灵思二进制文件 `.xclbin`。

提示： `v++` 命令可在命令行、脚本或构建系统（如 `make`）中使用，也可以通过 Vitis IDE 来使用，如 第七部分：使用 Vitis IDE 中所述。

使用 Vitis 编译器来编译内核

重要提示！ 运行工具前，请按 设置 Vitis 环境 中所述方式设置命令 shell 或窗口。

构建 `xclbin` 文件的第一阶段是使用赛灵思 Vitis 编译器来编译内核代码。为了正确编译内核，需要使用多个 `v++` 选项。以下是编译 `vadd` 内核的命令行示例：

```
v++ -t sw_emu --platform xilinx_u200_xdma_201830_2 -c -k vadd \
-I'./src' -o'vadd.sw_emu.xo' ./src/vadd.cpp
```

所用各实参如下所述。请注意，部分实参为必需。

- `-t <arg>`：指定构建目标，如 构建目标 中所述。软件仿真 (`sw_emu`) 用作为示例。可选。默认值为 hw。
- `--platform <arg>`：指定用于构建的加速器平台。这是必需的实参，因为运行时功能和目标平台均作为 FPGA 二进制文件的一部分链接在一起。要为嵌入式处理器应用编译内核，请指定嵌入式处理器平台：`--platform $PLATFORM_REPO_PATHS/zcu102_base/zcu102_base.xpfm`。
- `-c`：编译内核。必需。内核必须分 2 个独立步骤进行编译 (`-c`) 和链接 (`-l`)。
- `-k <arg>`：与源文件关联的内核名称。
- `-o'<output>.xo'`：指定编译器输出的共享对象文件。可选。
- `<source_file>`：为内核指定源文件。可指定多个源文件。必需。

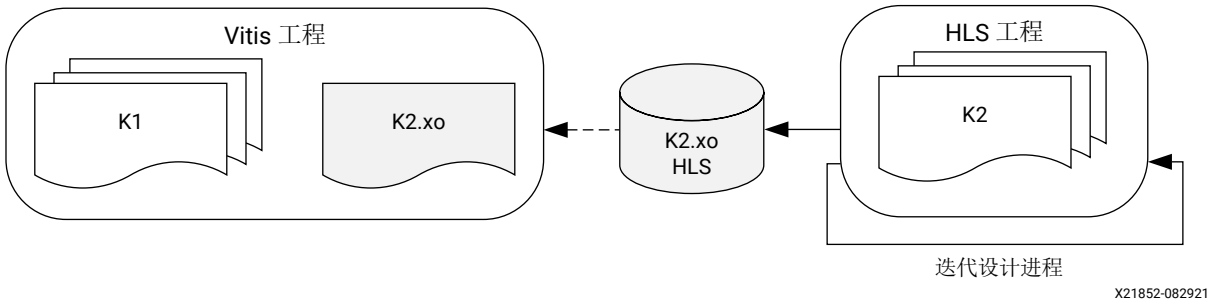
以上列表是可用的大量选项的一部分样本。请参阅 Vitis 编译器命令 以获取各命令行选项的详细信息。请参阅 v++ 命令的输出目录 以了解各输出文件的位置。

使用 Vitis HLS 编译内核

Vitis 核开发套件所描述的使用模型采用自上而下的方法，从 C/C++ 或 OpenCL 代码开始处理至已编译的内核。但您也可以直接开发内核以生成赛灵思对象 (XO) 文件，将两者搭配通过使用 `v++` 加以链接，从而生成 `.xclbin`。此方法可用于通过使用 Vitis HLS 工具来处理 C/C++ 内核（本节侧重点）或者通过使用 Vivado Design Suite 来处理 RTL 内核。如需了解更多信息，请参阅 RTL 内核。

以 RTL 或 C/C++ 直接开发内核来生成 XO 文件的方法有时被称为自下而上的流程。此方法支持您在 Vitis HLS 内确认内核性能并执行最优化，以及导出赛灵思对象文件以供在 Vitis 应用加速开发流程中使用。如需了解有关使用该工具的更多信息，请参阅 Vitis HLS 流程。

图 23：**Vitis HLS 自下而上流程**



Vitis HLS 自下而上流程的优势包括：

- 独立于主应用，对内核单独进行设计、确认和最优化。
- 支持采用团队协作方法开展设计，包括设计主机程序和内核开发。
- 特定内核最优化措施均保留在 XO 文件内。
- XO 文件集合可以像库一样来使用和复用。

在 Vitis HLS 中创建内核

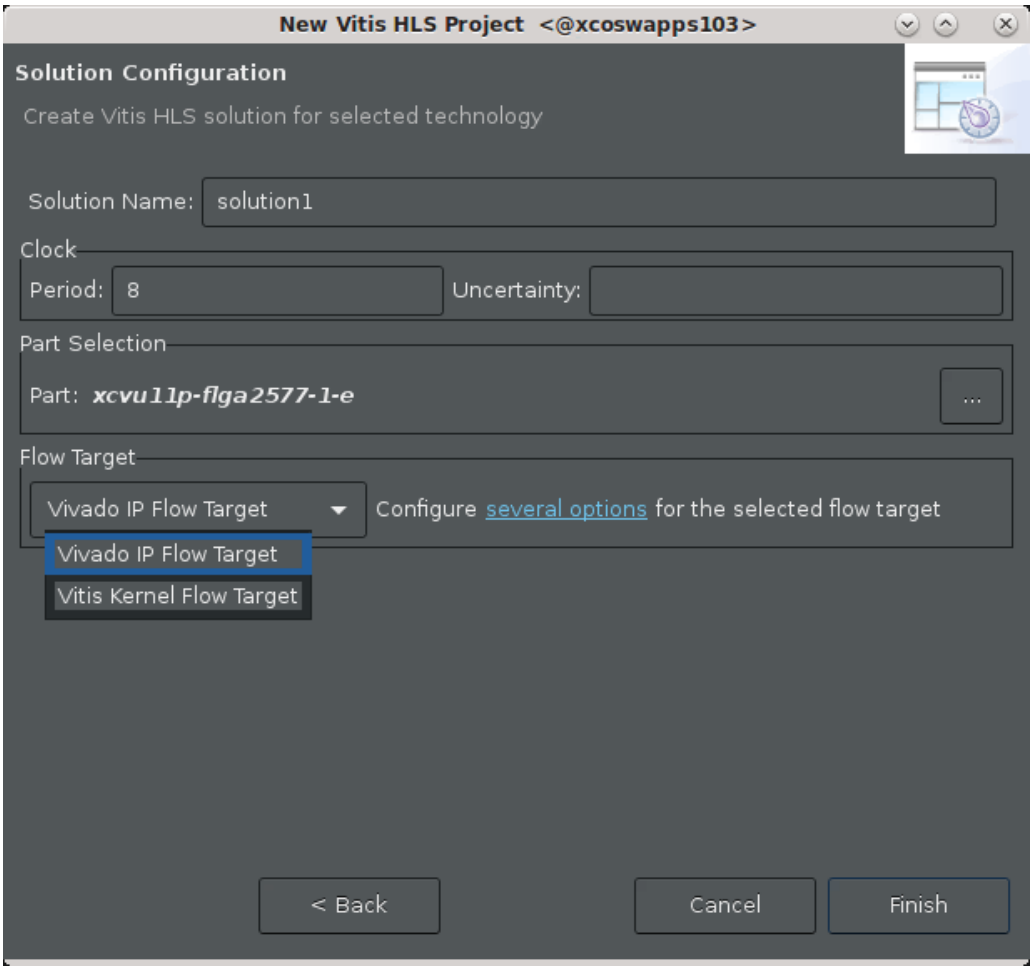
遵循标准 Vitis HLS 流程即可从 C/C++ 代码生成内核以供在 Vitis 核开发套件内使用。但由于内核需在 Vitis 软件平台内运行，因此必须满足标准内核要求（请参阅 内核属性）。最重要的是，必须将接口建模为 AXI 存储器接口，但标量参数除外，这些参数映射到 AXI4-Lite 接口。Vitis HLS 会使用此处所述的 Vitis 自下而上的流程来自动定义接口端口以满足标准内核要求。

以下概括了 HLS 内核创建和编译流程。请参阅 Vitis HLS 流程文档中的创建新的 Vitis HLS 工程以获取有关此流程的更完整的描述。

1. 启动 Vitis HLS 以打开集成设计环境 (IDE) 并指定“File” → “New Project”。
2. 在“New Vitis HLS Project”Wizard 中，使用“Project name”来指定工程名称、使用“Location”来为工程定义位置，然后单击“Next”。
3. 在“Add/Remove Files”页面中，单击“Add Files”以将内核源代码添加到工程中。单击“Browse”按钮选择“Top Function”以定义内核函数，完成后单击“Next”。
4. 如果您有基于 C 语言的仿真测试激励文件，可单击“Add Files”来指定此文件，或者可单击“Next”跳过此步骤。

   提示：如 Vitis HLS 文档中所述，强烈建议使用测试激励文件。

5. 在“Solution Configuration”页面中，您必须使用“Clock Period”来为内核指定时钟周期。



6. 单击“Part Selection”字段中的浏览按钮（“...”）打开“Device Selection”对话框以选择目标平台。选择“Boards”，并选中已编译的内核的目标平台，如下所示。单击“OK”以选中平台，然后返回至“Solution Configuration”页面。
7. 在“Solution Configuration”页面中，选中“Flow Target”下的“Vitis Kernel Flow Target”下拉菜单，然后单击“Finish”以完成此流程并创建您的 HLS 内核工程。

> **重要提示！**您必须选择“Vitis Kernel Flow Target”以从工程生成赛灵思对象 (XO) 文件。

创建 HLS 工程车后，您即可使用“Run C-Synthesis”运行 C 语言仿真来编译内核代码。请参阅 Vitis HLS 文档以获取 HLS 工具流程的完整描述。

完成综合后，内核可作为 XO 文件导出，以供在 Vitis 核开发套件内使用。您可通过主菜单中的“Solution” → “Export RTL”命令来获取导出命令。

指定文件位置，这样即可将内核作为赛灵思对象 (XO) 文件进行导出。

在 v++ 链接进程期间，此 (XO) 文件可用作为输入文件。如需了解更多信息，请参阅 链接内核。您还可将其添加到 Vitis IDE 中的应用工程内，如 创建 Vitis IDE 工程 中所述。

但请谨记，此处所述自下而上的流程中创建的 HLS 内核在 Vitis 应用加速开发流程中使用时存在某些限制。针对使用 HLS 内核的应用，不支持软件仿真，因为重复的头文件依赖关系可能造成问题。在硬件仿真流程中不支持针对 HLS 内核或 RTL 内核使用 GDB 调试。

用于创建内核的 Vitis HLS 脚本

如果您通过 Tcl 脚本运行 HLS 综合，则可通过编辑以下脚本来按前文所述创建 HLS 内核。

```
# Define variables for your HLS kernel:
set projName <proj_name>
set krnlName <kernel_name>
set krnlFile <kernel_source_code>
set krnlTB <kernel_test_bench>
set krnlPlatform <target_part>
set path <path_to_project>

#Script to create and output HLS kernel
open_project $projName
set_top $krnlName
add_files $krnlFile
add_files -tb $krnlTB
open_solution "solution1"
set_part $krnlPlatform
create_clock -period 10 -name default
config_flow -target vitis
csim_design
csynth_design
cosim_design
export_design -flow impl -format xo -output "./hlsKernel/hlsKernel.xo"
```

按 设置 Vitis 环境 中所述方式设置好您的环境后，使用以下命令运行 HLS 内核脚本。

```
vitis_hls -f <hls_kernel_script>.tcl
```

链接内核

提示：运行工具前，请按 设置 Vitis 环境 中所述方式设置命令 shell 或窗口。

内核可使用 C/C++、OpenCL C 或 RTL 语言来编写，无论使用何种语言，内核编译进程都会生成赛灵思对象 (XO) 文件。在链接阶段中，来自不同内核的 XO 文件都会与平台相链接，以创建 FPGA 二进制容器文件 (`.xclbin`) 供主机程序使用。

与编译类似，链接需要几个选项。以下是用于链接 `vadd` 内核二进制文件的命令行示例：

```
v++ -t sw_emu --platform xilinx_u200_xdma_201830_2 --link vadd.sw_emu.xo \
-o'vadd.sw_emu.xclbin' --config ./connectivity.cfg
```

此命令包含以下实参：

- `-t <arg>`：指定构建目标。软件仿真 (`sw_emu`) 用作为示例。链接时，所使用的 `-t` 和 `--platform` 实参必须与编译输入 (XO) 文件时所指定的实参相同。
- `--platform <arg>`：指定要与内核相链接的平台。要为嵌入式处理器应用链接内核，只需指定嵌入式处理器平台即可：`--platform $PLATFORM_REPO_PATHS/zcu102_base/zcu102_base.xpfm`
- `--link`：将内核与平台链接到 FPGA 二进制文件 (`xclbin`)。
- `<input>.xo`：输入对象文件。可指定将多个对象文件构建到 `.xclbin` 中。

- `-o'<output>.xclbin'`：指定输出文件名。链接阶段中的输出文件将是 `.xclbin` 文件。默认输出文件名为 `a.xclbin`
- `--config ./connectivity.cfg`：指定配置文件，用于提供不同用途的 `v++` 命令选项。如需了解有关 `--config` 选项的更多信息，请参阅 Vitis 编译器命令。

提示：请参阅 v++ 命令的输出目录 以了解各输出文件的位置。

除了链接赛灵思对象 (XO) 文外，链接进程还用于判定重要的架构详细信息。具体来说，在此进程中可指定要例化到硬件中的计算单元 (CU) 数量、分配从内核端口到全局存储器的连接以及要分配到 SLR 的 CU。以下章节探讨了其中部分构建选项。

创建内核的多个实例

默认情况下，连接器会从内核构建单个硬件实例。如果主机程序将多次执行相同内核，比如，为了满足数据处理的需求，那么它必须按顺序在硬件加速器上执行该内核。这可能会影响整体应用性能。但通过自定义内核链接阶段，您即可从单一内核例化多个硬件计算单元 (CU)。这样可以改善性能，因为主机程序现在可以执行多次重叠的内核调用，通过逐一分开运行各计算单元来并发执行内核。

链接期间，可通过在 `v++` 配置文件中使用 `connectivity.nk` 选项来创建单一内核的多个 CU。编辑配置文件以包含所需选项，并在 `v++` 命令行中通过 `--config` 选项来指定该配置文件，如 Vitis 编译器命令 中所述。

例如，对于 `vadd` 内核，可在配置文件中实现两个硬件实例，如下所示：

```
[connectivity]
#nk=<kernel name>:<number>:<cu_name>.<cu_name>...
nk=vadd:2
```

其中：

- `<kernel_name>`：指定要多次例化的内核名称。
- `<number>`：要在硬件中实现的内核实例或 CU 的数量。
- `<cu_name>.<cu_name>...`：为指定数量的实例指定实例名称。这是可选项，如果不指定 CU 名称，则默认设为 kernel_1。

随后，在 `v++` 命令行上指定配置文件：

```
v++ --config vadd_config.cfg ...
```

在以上 `vadd` 示例中，生成了 `vadd` 内核的两个实例：`vadd_1` 和 `vadd_2`。

提示：您可使用 `xclbinutil` 命令检验 `xclbin` 文件内容以检查结果。请参阅 xclbinutil 实用工具。

以下示例显示了 `xclbin` 二进制文件中的 `vadd` 内核的 3 个 CU，分别名为 `vadd_X`、`vadd_Y` 和 `vadd_Z`：

```
[connectivity]
nk=vadd:3:vadd_X.vadd_Y.vadd_Z
```

将内核端口映射到存储器

链接阶段为内核的存储器端口连接至存储器资源的时间，包括 DDR、HBM 和 PLRAM。默认情况下，在 v++ 链接进程中生成 xclbin 文件时，所有内核存储器接口均连接到相同的全局存储体（或 gmem）。因此导致每次只有 1 个内核接口可在存储体上进行数据传入和传出，使应用性能受到存储器访问的限制。

虽然 Vitis 编译器可以将 CU 自动连接到全局存储器资源，但您也可以手动指定每个内核实参（或接口）所连接到的全局存储体。正确配置内核到存储器的连接对于最大程度提升带宽、最优化数据传输和提升总体性能而言至关重要。即使器件中只有 1 个计算单元，将其输入和输出实参映射到不同全局存储体同样可以通过支持同时访问输入数据和输出数据来提升性能。

重要提示！最多可将 15 个内核接口连接到单个全局存储体。因此，如有超过 15 个存储器接口，则必须按上述方式显式执行存储器映射，并使用 --conectivity.sp 选项在不同存储体间分配连接。

以下示例是基于 内核接口 代码示例提供的。首先将内核实参分配到各独立捆绑包，以增加可用接口端口，然后将实参分配给各独立存储体。

1. 在 C/C++ 内核中，将实参分配给内核代码中的各独立捆绑包，然后再对其进行编译。

```
void cnn( int *pixel, // Input pixel
  int *weights, // Input Weight Matrix
  int *out, // Output pixel
  ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

   请注意，在以上示例中，为存储器接口输入 pixel 和 weights 分配不同的捆绑包名称，而 out 则与 pixel 捆绑在一起。这样即可创建 2 个独立的接口端口。

   重要提示！您必须使用全小写字符来指定 bundle= 名称，这样才能使用 --connectivity.sp 选项将其分配给特定存储体。

2. 编辑配置文件以包含 --connectivity.sp 选项，并在含 --config 选项的 v++ 命令行中指定此文件，如 Vitis 编译器命令 中所述。

   例如，对于以上所示 cnn 内核，配置文件中的 connectivity.sp 选项将如下所示：

```
[connectivity]
#sp=<compute_unit_name>.<argument>:<bank name>
sp=cnn_1.pixel:DDR[0]
sp=cnn_1.weights:DDR[1]
sp=cnn_1.out:DDR[2]
```

   其中：

   - <compute_unit_name> 是 CU 的实例名，由 connectivity.nk 选项确定（如 创建内核的多个实例 中所述），或者如果不指定多个 CU，则直接命名为 <kernel_name>_1。
   - <argument> 是内核实参的名称。或者，您可以按 C/C++ 内核的 HLS INTERFACE 编译指示中定义的方式来指定内核接口的名称，包括 m_axi_ 和 bundle 名称。在以上 cnn 内核中，端口为 m_axi_gmem 和 m_axi_gmem1。

     提示：对于 RTL 内核，接口由 kernel.xml 文件中定义的接口名称来指定。

   - 对于含 4 个 DDR 存储体的平台，<bank_name> 表示为 DDR[0]、DDR[1]、DDR[2] 和 DDR[3]。您还可将存储器指定为连续范围内的存储体，例如 DDR[0:2]，在此情况下，XRT 将在运行时分配该存储体。

某些平台还支持 PLRAM、HBM、HP 或 MIG 存储器，在此情况下，您可使用 PLRAM[0]、HBM[0]、HP[0] 或 MIG[0]。您可以使用 `platforminfo` 实用工具来获取有关指定平台上可用的全局存储体的信息。如需了解更多信息，请参阅 platforminfo 实用工具。

在同时包含 DDR 和 HBM 存储体的平台中，内核必须使用不同的 AXI 接口来访问不同的存储器。DDR 和 PLRAM 访问权限可通过单一端口共享。

重要提示！在某些情况下，还可能需要在主机代码中反映自定义的存储体分配方式，如 在主机代码中分配 DDR 存储体 中所述。

直接连接到主机存储器

在部分数据中心平台上提供了 PCIe® Slave-Bridge IP，用于支持内核直接访问主机存储器。要将器件二进制文件配置为连接至存储器，需对以下 `--connectivity.sp` 命令指定的链接进行更改。还需要更改加速器卡和主机应用，如 XRT 文档的主机存储器访问中所述。

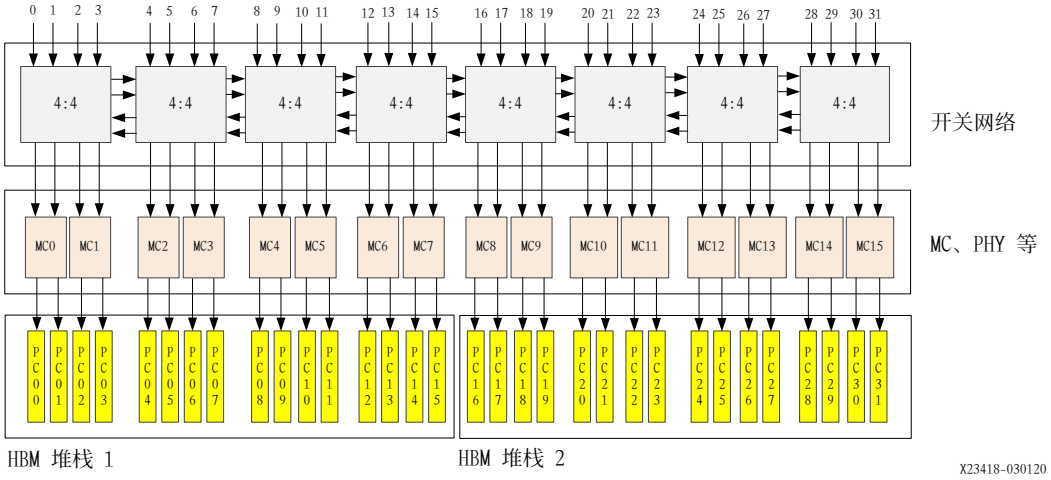
```
[connectivity]
## Syntax
##sp=<cu_name>.<interface_name>:HOST[0]
sp=cnn_1.m_axi_gmem:HOST[0]
```

在以上命令语法中，CU 名称与接口名称相同，但存储体名称已硬编码到 `HOST[0]` 中。

HBM 配置和使用

部分算法与存储器绑定，并受到基于 DDR 的 Alveo 卡上可用的 77GB/s 带宽的限制。对于此类应用，可使用基于高带宽存储器 (HBM) 的 Alveo 卡，这些卡可提供高达 460 GB/s 的存储器带宽。对于 Alveo 实现，在 FPGA 封装内整合了 2 个 16 层 HBM（HBM2 规范）堆栈，并通过中介层连接到 FPGA 互连结构。双 HBM 堆栈的高层次示意图如下所示。

图 24：**双 HBM 堆栈的高层次示意图。**



此实现可提供：

- 8 GB HBM 存储器
- 32 个 256 MB HBM 段，称为伪通道 (PC)
- 每条伪通道 1 个独立 AXI 通道，用于通过分段式交叉开关矩阵开关与 FPGA 进行通信

- 每 2 条 PC 提供 1 个双通道存储器控制器
- 每条 PC 最大 14.375 GB/s 理论带宽
- 针对 HBM 子系统最大 460 GB/S (32 * 14.375 GB/s) 理论带宽

虽然每条 PC 的最大理论性能为 14.375 GB/s，但它小于 DDR 通道的理论最大值 19.25 GB/s。为了超越 DDR 性能，设计必须有效利用多个 AXI 主接口连接到 HBM 子系统中。可编程逻辑具有 32 个 HBM AXI 接口，可以通过内置开关访问任一 HBM 堆栈上的任意 PC 中的任意存储器位置，此内置开关能够访问整个 8 GB 存储器空间。如需了解 HBM 的更多详细信息，请参阅《AXI High Bandwidth Controller LogiCORE IP 产品指南》(PG276)。

注释：由于内置开关的复杂性和灵活性，有大量组合都可能导致在特定存储器位置或开关本身中出现拥塞。受存储器控制器时序参数（总线周转）的影响，交织式读写传输事务会导致只读或只写操作相关性能下降。如果写入传输事务跨两个 HBM 堆栈，则也会产生性能降级，因此应避免。重要的是规划存储器存取，以尽可能限制内核执行的存储器存取，并将不同内核的存储器存取操作隔离到不同 HBM PC 中。

到 HBM 的连接由 HBM Memory Subsystem (HMSS) IP 管理，此 IP 启用所有 HBM PC，并将 XDMA 自动连接至 HBM 以便主机访问全局存储器。搭配 Vitis 编译器使用时，HMSS 会自动自定义为仅激活 `--connectivity.sp` 选项所指定的必要存储器控制器和端口，以将用户内核与 XDMA 连接到这些存储器控制器，从而实现最优化的带宽和时延。请参阅使用 HBM 教程以获取更多信息和示例。

在以下配置文件示例中，内核输入端口 `in1` 和 `in2` 分别连接到 HBM PC 0 和 1，并将输出缓冲器 `out` 写入 HBM PC 3-4。每个 HBM PC 为 256 Mb，为该内核提供总共 1 GB 的存储器访问。

```
[connectivity]
sp=krnl.in1:HBM[0]
sp=krnl.in2:HBM[1]
sp=krnl.out:HBM[3:4]
```

注释：在配置文件中，仅定义到 HBM 伪通道的映射，每个 AXI 接口都应仅访问可用的 32 个 HBM PC 中的少数几条连续 PC。HMSS 会选择相应的 HBM 端口以访问存储器，并尽可能提升带宽和降低时延。

HBM 端口位于器件的底部 SLR 中。对于 SSI 技术器件中跨超级逻辑区域 (SLR) 的 AXI 接口，HMSS 会自动处理这些接口的复杂布局和时序。默认情况下，如果在 `v++` 上不指定 `--connectivity.sp` 或 `--connectivity.slr` 选项，那么所有内核 AXI 接口访问 HBM[0] 和所有内核都分配到 SLR0。但您可以使用 `--connectivity.slr` 选项来指定内核的 SLR 分配。如需了解更多信息，请参阅 将计算单元分配给 SLR。

随机存取和 RAMA IP

HBM 在需要顺序数据存取的应用中表现良好。然而，对于需要随机数据存取的应用，根据应用要求（如读写操作比率、最小传输事务大小和正在寻址的存储器空间的大小），性能可能会有很大的不同。在此类情况下，当所需存储器超过单一 HBM PC 的 256 MB 限制时，为目标平台添加 Random Access Memory Attachment (RAMA) IP 可以显著提升随机存储器存取的效率。如需了解更多信息，请参阅 《RAMA LogiCORE IP 产品指南》(PG310)。

提示：为了在应用中有效使用 RAMA IP，内核应从多个 HBM PC 访问存储器，并且应在 AXI 传输事务 ID 端口 (AxID) 上使用静态单一 ID，或者缓慢更改（伪静态）AXI 传输事务 ID。如未满足这些条件，则在 RAMA IP 中用于提高性能的线程创建操作将难以产生效果，并将毫无意义地占用可编程逻辑资源。

在系统链接进程中使用以下 `v++` 命令选项指定用于定义感兴趣端口的 Tcl 脚本，从而为目标平台添加 RAMA IP。

```
v++ -l --advanced.param compiler.userPreSysLinkOverlayTcl=<path_to>/
user_tcl_file.tcl
```

在此用户指定的 Tcl 脚本内，提供了一个 API 以便您配置 HMSS 资源：

```
hbm_memory_subsystem::ra_master_interface <Endpoint AXI master interface>
[get_bd_cells hmss_0]
```

以下示例包含 2 个 AXI 主端口（M00_AXI 和 M01_AXI）用于进行随机存取。

```
hbm_memory_subsystem::ra_master_interface [get_bd_intf_pins dummy/M00_AXI]
[get_bd_cells hmss_0]
hbm_memory_subsystem::ra_master_interface [get_bd_intf_pins dummy/M01_AXI]
[get_bd_cells hmss_0]
validate_bd_design -force
```

重要的是，请使用 validate_bd_design 命令终止该 Tcl 脚本以便 HBM 子系统能够正确收集信息并更新块设计。

PLRAM 配置和使用

Alveo 加速器卡包含 HBM DRAM 和 DDR DRAM 存储器资源。在某些加速器卡中，另一个可供使用的存储器资源是内部 FPGA PLRAM（UltraRAM 和块 RAM）。支持平台通常在每个 SLR 内都包含 PLRAM 的实例。每个 PLRAM 的大小和类型均可先在目标平台上进行配置，然后再将内核或计算单元链接到系统。

您可以使用 Tcl 脚本来配置 PLRAM，然后再执行系统链接。在 v++ 命令行上可启用使用 Tcl 脚本，如下所示：

```
v++ -l --advanced.param compiler.userPreSysLinkOverlayTcl=<path_to>/
user_tcl_file.tcl
```

在此用户指定的 Tcl 脚本内，提供了一个 API 以便您配置 PLRAM 实例或存储器资源：

```
sdx_memory_subsystem::update_plram_specification <memory_subsystem_bdcell>
<plram_resource> <plram_specification>
```

<plram_specification> 是由以下条目组成的 Tcl 字典（以下条目为平台中每个实例的默认值）：

```
 {
    SIZE 128K # Up to 4M
    AXI_DATA_WIDTH 512 # Up to 512
    SLR_ASSIGNMENT SLR0 # SLR0 / SLR1 / SLR2
    READ_LATENCY 1 # To optimise timing path
    MEMORY_PRIMITIVE BRAM # BRAM or URAM
}
```

在以下示例中，PLRAM_MEM00 大小已更改为 2 MB，并由 UltraRAM 组成；PLRAM_MEM01 大小已更改为 4 MB，并由 UltraRAM 组成。PLRAM_MEM00 和 PLRAM_MEM01 对应于 --conectivity.sp 存储器资源 PLRAM[0] 和 PLRAM[1]。

```
# Setup PLRAM
sdx_memory_subsystem::update_plram_specification
[get_bd_cells /memory_subsystem] PLRAM_MEM00 { SIZE 2M AXI_DATA_WIDTH 512
SLR_ASSIGNMENT SLR0 READ_LATENCY 10 MEMORY_PRIMITIVE URAM}

sdx_memory_subsystem::update_plram_specification
[get_bd_cells /memory_subsystem] PLRAM_MEM01 { SIZE 4M AXI_DATA_WIDTH 512
SLR_ASSIGNMENT SLR0 READ_LATENCY 10 MEMORY_PRIMITIVE URAM}

validate_bd_design -force
save_bd_design
```

READ_LATENCY 是一个重要的属性，因为它用于设置深度级联存储器之间的流水线级数。这因设计而异，并影响平台的时序 QoR 和最终的内核时钟速率。在以上 PLRAM_MEM01 示例中：

- 共需要 4 MB 的存储器。

- 每个 UltraRAM 为 32 KB（64 位宽）。4 MB × 32 KB → 共计 128 个 UltraRAM。
- 每个 PLRAM 实例为 512 位宽 → 宽度中需要 8 个 UltraRAM。
- 总共 128 个 UltraRAM，宽度含 8 个 UltraRAM → 深度中含 16 个 UltraRAM。
- 有一条经验法则很有用，即选择深度/2 + 2 的读取时延，在此情况下 `READ_LATENCY` = 10。

这样即可每隔一个 UltraRAM 设置一条流水线，从而产生以下结果：

- UltraRAM 间能实现良好的时序性能。
- 布局灵活性；并非所有的 UltraRAM 都需要布局在同一个 UltraRAM 列中进行级联。

在计算单元之间指定数据流传输连接

Vitis 核开发套件支持两个内核之间的数据流传输，允许数据从一个内核直接移动到另一个内核，而无需通过全局存储器来向回发射数据。但此进程必须在内核代码本身中实现（如 在用户管理的永续内核中进行数据流传输 中所述），并且还必须在内核构建进程中指定。

内核的数据流传输端口可在 `v++` 链接期间使用 `--connectivity.sc` 选项来连接。该选项可在命令行上指定，或者也可在使用 `--config` 选项指定的 `config` 文件中指定，如 Vitis 编译器命令 中所述。

要将生产者内核的数据流传输输出端口连接到使用者内核的数据流传输输入端口，请在 `v++` 配置文件内使用 `connectivity.stream_connect` 选项来指定连接，如下所示：

```
[connectivity]
#stream_connect=<cu_name>.<output_port>:<cu_name>.<input_port>:
[<fifo_depth>]
stream_connect=vadd_1.stream_out:vadd_2.stream_in
```

其中：

- `<cu_name>` 是由 `connectivity.nk` 选项判定的 CU 的实例名称，如 创建内核的多个实例 中所述。
- `<output_port>` 或 `<input_port>` 是生产者内核或使用者内核中定义的数据流传输端口。
- `[:<fifo_depth>]` 会在 2 个数据流传输端口之间插入指定深度的 FIFO 以防止发生停滞。请指定整数值。

提示：如果输出端口与输入端口的端口宽度不匹配，那么在构建进程中，Vitis 编译器将在这两个端口之间自动插入数据宽度转换器。

将计算单元分配给 SLR

当前，数据中心加速器卡上的赛灵思器件使用由超级逻辑区域 (SLR) 组成的堆叠硅片来提供器件资源（包括全局存储器）。为了实现最佳性能，在向全局存储体分配端口时（如 将内核端口映射到存储器 中所述），最好将 CU 实例与相连的全局存储器分配到相同 SLR。在此情况下，您将把内核实例或 CU 手动分配到全局存储器所在 SLR 中，以确保最佳性能。

重要提示！如果您的内核过大，无法布局到单一 SLR 内，那么 Vitis 编译器会自动跨多个 SLR 进行逻辑布局。在此情况下，您不应分配 SLR，否则可能导致实现期间出错。

在配置文件中使用 connectivity.slr 选项即可将 CU 分配给 SLR。配置文件中的 connectivity.slr 选项语法如下：

```
[connectivity]
#slr=<compute_unit_name>:<slr_ID>
slr=vadd_1:SLR2
slr=vadd_2:SLR3
```

其中：

- <compute_unit_name> 是 CU 的实例名，由 connectivity.nk 选项确定（如 创建内核的多个实例 中所述），或者如果不指定多个 CU，则直接命名为 <kernel_name>_1。
- <slr_ID> 是 CU 分配到的 SLR 编号，格式为 SLR0、SLR1...，以此类推。

必须针对每个 CU 单独指定分配给 SLR 的 CU，但这并不是必需的。如果分配的 CU 连接到位于另一个 SLR 中的全局存储器，那么该工具将自动插入跨 SLR 寄存器以帮助达成时序收敛。如无 SLR 分配，v++ 连接器即可自由向任意 SLR 分配 CU。

编译配置文件以包含 SLR 分配后，您可在 v++ 链接进程中通过使用 --config 选项指定配置文件来使用此分配：

```
v++ -l --config config_slr.cfg ...
```

管理时钟频率

重要提示！仅限含有固定状态时钟的平台 shell 才支持此处描述的 --clock 选项，如 识别平台时钟 中所述。在包含固定时钟的旧平台 shell 上，内核以平台默认运行频率来运行。

通常在嵌入式处理器平台中以及在新的数据中心加速器卡中，器件二进制文件可将多个内核连接到含不同时钟频率的平台上。每个内核或者内核的每个独特实例都能够连接到指定的时钟频率或连接到多个时钟，并且不同内核可以使用平台所生成的不同时钟频率。

在编译进程 (v++ -c) 中，您可以使用 --hls.clock 命令来指定内核频率。这样您即可对以指定频率为目标的内核进行编译，并允许 Vitis HLS 工具按指定频率来执行内核逻辑确认。这只是编译的实现目标，但可提供最优化和反馈。

在链接进程中，当内核连接到平台以构建器件二进制文件时，您可以使用 v++ 命令的 --clock 选项 来为内核指定时钟频率。因此，时钟频率的管理流程如下：

1. 使用 Vitis 编译器按指定频率编译 HLS 代码：

   ```
   v++ -c -k <krnl_name> --hls.clock freqHz:<krnl_name>
   ```

   提示：freqHz 必须以 Hz 为单位（例如，250000000Hz 表示 250 MHz）。

2. 在链接期间，通过以下命令为内核中的每个时钟信号指定时钟频率或时钟 ID：

   ```
   v++ -l ... --clock.freqHz <freqHz>:kernelName.clk_name
   ```

您可使用来自平台 shell 脚本的时钟 ID 或者通过指定内核时钟频率来指定 --clock 选项。指定时钟 ID 时，内核频率由平台上该时钟 ID 的频率来定义。指定内核频率时，平台会尝试通过按比例缩放某一可用 fixed 平台时钟来创建指定频率。在某些情况下，只能实现近似时钟频率，您可指定 --clock.tolerance 或 --clock.default_tolerance 来标示可接受的范围。如果可用的固定时钟无法在可接受容限范围内进行缩放，则会发出一条警告，并且内核会连接到默认时钟。

识别平台时钟

加速器卡中时钟的处理方式已演变为支持多个平台时钟和多个时钟频率。时钟处理方式的改善取决于是否存在固定状态平台时钟可用于驱动额外的内核频率。您可使用 `platforminfo -v` 命令来判断目标平台 shell 是否有固定时钟。例如，以下命令可返回与 U200 平台的新 shell 相关的详细信息：

```
platforminfo -v -p xilinx_u200_gen3x16_xdma_1_202110_1 -o pfmClocks.txt
```

输出文件中报告的信息包括以下时钟详细信息：

```
==================
Clock Information
==================
...
...
  Clock Index:          2
    Frequency:          50.000000
    Name:               ii_level0_wire_ulp_m_aclk_ctrl_00
    Pretty Name:        PL 2
    Inst Ref:           ii_level0_wire
    Comp Ref:           ii_level0_wire
    Period:             20.000000
    Normalized Period:  .020000
    Status:             fixed
...
```

在以上示例中，您可以看到“Clock Index: 2”为处于 `fixed` 状态的时钟。实际上，虽然此处并未显示，但此平台 shell 可提供 3 个固定状态的时钟，这些时钟可用于驱动已链接的内核中的多个时钟频率。您可以按上文所述使用 `--clock.xxx` 选项来驱动内核时钟。

但是，在旧的平台 shell（如 `xilinx_u200_xdma_201830_2`）上，并没有固定平台时钟可用于驱动已链接的内核中的时钟频率。这可通过查看以下命令报告的“Clock Information”来判定：

```
platforminfo -v -p xilinx_u200_xdma_201830_2 -o pfmClocks.txt
```

在没有固定状态时钟的旧平台上，您可以使用 `v++ --kernel frequency` 选项来指定时钟频率，如 Vitis 编译器常规选项 中所述。

管理 Vivado 综合与实现结果

提示：本主题要求了解 Vivado Design Suite 工具和《UltraFast 设计方法指南（适用于赛灵思 FPGA 和 SoC）》(UG949) 中所述的设计方法论。

在大部分情况下，Vitis 环境将可编程逻辑区域的底层综合与实现进程完全抽象出来，因为 CU 与硬件平台相链接，并生成 FPGA 二进制文件 (`xclbin`)。这样即可使硬件开发者免于处理典型的硬件开发进程，以及诸如逻辑布局和布线延迟之类的约束的管理工作。Vitis 工具可以自动执行大部分 FPGA 实现进程。

但在某些情况下，您可能想要对 Vitis 编译器所部署的某些综合与实现进程稍作控制，尤其是在实现大型设计时。为此，Vitis 工具通过特定选项来提供部分控制，这些选项可在 `v++` 配置文件中指定，或者也可以从命令行来指定。以下是支持您与 Vivado 综合与实现结果进行交互并控制这些结果的部分方法。

- 使用 `--vivado` 选项来管理 Vivado 工具。

- 使用多种实现策略在困难设计上实现时序收敛。
- 使用 `-to_step` 和 `-from_step` 选项将编译或链接进程运行至某个特定步骤、对设计执行部分手动干预，然后从该步骤恢复执行。
- 以交互方式编辑 Vivado 工程并使用生成的 FPGA 二进制文件结果。

使用 -vivado 和 -advanced 选项

您可通过使用 `--vivado` 选项（如 --vivado 选项 中所述）和 `--advanced` 选项（如 --advanced 选项 中所述）来对标准 Vivado 综合与实现执行多种干预措施。

1. 传递含定制设计约束或脚本化操作的 Tcl 脚本。

   您可使用综合与实现步骤的 PRE 和 POST Tcl 脚本属性来创建 Tcl 脚本以将 XDC 设计约束分配给设计中的对象，并将这些 Tcl 脚本传递到 Vivado 工具。如需了解有关 Tcl 脚本编制的更多信息，请参阅《Vivado Design Suite 用户指南：使用 Tcl 脚本》(UG894)。虽然只有 1 个综合步骤，但有多个实现步骤，如《Vivado Design Suite 用户指南：实现》(UG904) 中所述。您可将 Vivado 工具的 Tcl 脚本指定为在 (PRE) 步骤之前运行或者在 (POST) 步骤之后运行。Tcl 脚本可分配到的具体步骤包括：`SYNTH_DESIGN`、`INIT_DESIGN`、`OPT_DESIGN`、`PLACE_DESIGN`、`ROUTE_DESIGN` 和 `WRITE_BITSTREAM`。

   提示：您还可使用 `--vivado.prop run.impl_1.steps.phys_opt_design.is_enabled=1` 选项启用多个可选步骤。启用后，这些步骤同样可包含 Tcl PRE 脚本和 POST 脚本。

   Tcl PRE 和 POST 脚本分配示例如下：

   ```
   --vivado.prop run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/.../xxx.tcl
   ```

   在前述示例中，脚本已分配为先于 PLACE_DESIGN 步骤运行。命令行的细分方式如下：

   - `--vivado` 是用于为 Vivado 工具指定指令的 `v++` 命令行选项。
   - `prop` 关键字用于表示您正在传递属性设置。
   - `run.` 关键字用于表示您正在传递运行 (run) 属性。
   - `impl_1.` 表示运行轮次名称。
   - `STEPS.PLACE_DESIGN.TCL.PRE` 表示当前指定的运行属性。
   - `/.../xx.tcl` 表示属性值。

   提示：`--advanced` 和 `--vivado` 选项均可在 `v++` 命令行上指定，或者在由 `--config` 选项所指定的配置文件中指定。以上示例显示了命令行的使用方式，以下示例显示的则是配置文件的使用方式。如需了解更多信息，请参阅 Vitis 编译器配置文件。

2. 在运行 (run)、文件 (file) 和文件集 (fileset) 设计对象上设置属性。

   这与上述 Tcl 脚本传递方式非常类似，但此处则是将值传递给多个设计对象上的不同属性。例如，要在布局期间使用特定实现策略（如 `Performance_Explore`）和禁用全局缓冲器插入，可按如下所示方式定义属性：

   ```
   [vivado]
   prop=run.impl_1.STEPS.OPT_DESIGN.ARGS.DIRECTIVE=Explore
   prop=run.impl_1.STEPS.PLACE_DESIGN.ARGS.DIRECTIVE=Explore
   prop=run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-no_bufg_opt}
   prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
   prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
   prop=run.impl_1.STEPS.ROUTE_DESIGN.ARGS.DIRECTIVE=Explore
   ```

在以上示例中，对实现运行过程中的各步骤的 STEPS.XXX.DIRECTIVE 属性赋值 Explore。请注意，这些属性的定义语法为：

```
<object>.<instance>.property=<value>
```

其中：

- <object> 可以是设计运行、文件或文件集对象。
- <instance> 表示对象的特定实例。
- <property> 指定要分配的属性。
- <value> 定义属性值。

在此示例中，对象为运行，实例为默认实现运行 impl_1，而属性则是不同步骤名称的实参（在此处为 DIRECTIVE、IS_ENABLED 和 {MORE OPTIONS}）。如需了解有关命令语法的更多信息，请参阅 --vivado 选项。

3. 在 Vivado 实现进程中启用可选步骤。

   构建进程会运行 Vivado 综合与实现来生成器件二进制文件。在默认构建进程中会启用并运行部分实现步骤，您也可以自行决定是否启用部分可选实现步骤。

   可选步骤可使用 --list_steps 命令列出，包括：vpl.impl.power_opt_design、vpl.impl.post_place_power_opt_design、vpl.impl.phys_opt_design 和 vpl.impl.post_route_phys_opt_design。

   可选步骤可使用 --vivado.prop 选项来启用。例如，要启用 PHYS_OPT_DESIGN 步骤，请使用以下配置文件内容：

   ```
   [vivado]
   prop=run.impl_1.steps.phys_opt_design.is_enabled=1
   ```

   如上所示，启用可选步骤后，该步骤可指定为 -from_step/-to_step 命令的一部分（如下文《运行 --to_step 或 --from_step》中所述），或者也可以启用 Tcl 脚本以在此步骤之前或之后运行（如 --linkhook 选项 中所述）。

4. 向工具传递参数用于控制处理。

   --vivado 选项还允许您向 Vivado 工具传递参数。这些参数用于配置该工具启动前的功能或行为。指定参数的语法采用如下形式：

   ```
   --vivado.param <object><parameter>=<value>
   ```

   关键字 param 表示您正在为 Vivado 工具传递参数而不是为设计对象传递属性。您还必须定义参数所适用的 <object>、您正在指定的 <parameter> 以及要为该参数赋予的 <value>。

   以下工程示例表示参数正在传递到当前 Vivado 工程 writeIntermedateCheckpoints，并且参数值为 1，即表示启用此布尔值参数。

   ```
   --vivado.param project.writeIntermediateCheckpoints=1
   ```

5. 管理综合与实现期间生成的报告。

   重要提示！在自定义由 Vivado 工具生成的报告时，您还必须在 v++ 命令行上指定 --save-temps，以保留综合与实现期间所创建的临时文件（包括所有生成的报告）。

在 Vitis 工具构建进程中运行 Vivado 工具时，您可能想要生成或保存由该工具所提供的标准报告。您可使用 `--advanced.misc` 选项来自定义所生成的报告，如下所示：

```
[advanced]
misc=report=type report_utilization name
synth_report_utilization_summary steps {synth_design} runs {__KERNEL__}
options {}
misc=report=type report_timing_summary name
impl_report_timing_summary_init_design_summary steps {init_design} runs
{impl_1} options {-max_paths 10}
misc=report=type report_utilization name
impl_report_utilization_init_design_summary steps {init_design} runs
{impl_1} options {}
misc=report=type report_control_sets name
impl_report_control_sets_place_design_summary steps {place_design} runs
{impl_1} options {-verbose}
misc=report=type report_utilization name
impl_report_utilization_place_design_summary steps {place_design} runs
{impl_1} options {}
misc=report=type report_io name impl_report_io_place_design_summary
steps {place_design} runs {impl_1} options {}
misc=report=type report_bus_skew name
impl_report_bus_skew_route_design_summary steps {route_design} runs
{impl_1} options {-warn_on_violation}
misc=report=type report_clock_utilization name
impl_report_clock_utilization_route_design_summary steps {route_design}
runs {impl_1} options {}
```

如下示例对命令行的语法进行了解释：

```
misc=report=type report_bus_skew name
impl_report_bus_skew_route_design_summary steps {route_design} runs
{impl_1} options {-warn_on_violation}
```

- `misc=report=`：指定 `--advanced.misc` 选项（如 --advanced 选项 中所述）并定义 Vivado 工具的报告配置。命令行其余部分是以名称/值对的形式来指定的，以反映 `create_report_config` Tcl 命令的选项，如《Vivado Design Suite Tcl 命令参考指南》(UG835) 中所述。
- `type report_bus_skew`：与 `-report_type` 实参有关，指定报告类型为 `report_bus_skew`。大部分 `report_*` Tcl 命令均可指定为报告类型。
- `name impl_report_bus_skew_route_design_summary`：与 `-report_name` 实参有关，指定报告名称。请注意，这并非报告的文件名，通常可跳过该选项，因为该工具将自动生成报告名称。
- `steps {route_design}`：与 `-steps` 选项有关，指定报告适用于的综合与实现步骤。报告可指定为搭配多个步骤一起使用，以便在每个步骤中都生成此报告，在此情况下将自动定义报告名称。
- `runs {impl_1}`：与 `-runs` 选项有关，指定报告要应用到的设计运行的名称。
- `options {-warn_on_violation}`：指定生成报告时要使用的 `report_*` Tcl 命令的各选项。在此示例中，`-warn_on_violation` 选项是 `report_bus_skew` 命令的一项功能。

> **重要提示！**没有任何错误检查功能可用于确保指定选项正确且适用于指定的报告类型。如果您指示选项错误，那么报告运行时将返回错误。

运行多项实现策略来实现时序收敛

对于较为困难的设计，需要使用多种不同策略来多次迭代 Vivado 实现才能达成时序收敛。本主题为您演示了如何在硬件构建 (`-t hw`) 中同时启动多项实现策略，以及如何识别并使用成功的运行来生成器件二进制文件并完成构建。

正如 --vivado 选项 中所述，`--vivado.impl.strategies` 命令支持您指定在单次构建进程中指定运行多项策略。命令行如下所示：

```
v++ --link -s -g -t hw --platform xilinx_zcu102_base_202010_1 -I . \
--vivado.impl.strategies "Performance_Explore,Area_Explore" -o
kernel.xclbin hello.xo
```

在以上示例中，在 Vivado 构建中同时运行 `Performance_Explore` 和 `Area_Explore` 策略，以查看哪项策略可返回最佳结果。您可以指定 `ALL` 以在工具中运行所有可用策略。

您也可以在配置文件中采用如下形式来决定该选项：

```
#Vivado Implementation Strategies
[vivado]
impl.strategies=Performance_Explore,Area_Explore
```

Vitis 编译器会自动提取首个成功满足时序的运行结果来继续执行构建进程并生成器件二进制文件。但您也可以指令该工具等待运行全部完成后，再从完成的运行中提取最佳结果以继续执行。这将需要使用 `--advanced.compiler` 指令，如下所示：

```
[advanced]
param=compiler.multiStrategiesWaitOnAllRuns=1
```

`compiler.multiStrategiesWaitOnAllRuns=0` 表示默认行为。如果您希望 `v++` 等待运行全部完成，从而获取其报告文件，请将该参数值改为 1。

如 链接汇总：多种策略和时序报告 中所述，Vitis 分析器会显示允许运行完成的所有策略的实现结果。这包括实现结果的综述以及“时序汇总 (Timing Summary)”报告。您可以使用该功能来复查不同策略和结果。

您也可以在所有实现策略全部完成后，手动复查其结果。然后，通过使用 `--reuse_impl` 选项来使用任意实现运行的结果，如 使用 -to_step 并以交互方式启动 Vivado 中所述。

使用 -to_step 并以交互方式启动 Vivado

Vitis 编译器支持您在完成指定步骤 (`--to_step`) 之后停止构建进程、以某种方式手动干预设计或文件，然后通过指定用于恢复构建的步骤 (`--from_step`) 来继续执行构建。`--from_step` 会指令 Vitis 编译器从 `--to_step` 停止位置的步骤或者该进程中更早的步骤开始恢复编译。如需了解有关 `--to_step` 和 `--from_step` 的信息，请参阅 Vitis 编译器常规选项。

> **重要提示！** `--to_step` 和 `--from_step` 选项为顺序构建选项，要求您在启动 `v++ --link --from_step` 时所使用的工程目录与您在使用 `v++ --link --to_step` 时指定的工程目录相同。

Vitis 编译器还会提供 `--list_steps` 选项以列出指定构建目标的编译或链接进程的可用步骤。例如，通过以下命令即可找到硬件构建的链接进程的步骤列表：

```
v++ --list_steps --target hw --link
```

此命令会返回多个步骤，包括 Vitis 编译器在硬件构建的链接进程中执行的默认步骤和可选步骤。部分默认步骤包括：`system_link`、`vpl`、`vpl.create_project`、`vpl.create_bd`、`vpl.generate_target`、`vpl.synth`、`vpl.impl.opt_design`、`vpl.impl.place_design`、`vpl.impl.route_design` 和 `vpl.impl.write_bitstream`。

可选步骤包括：`vpl.impl.power_opt_design`、`vpl.impl.post_place_power_opt_design`、`vpl.impl.phys_opt_design` 和 `vpl.impl.post_route_phys_opt_design`。

提示：可选步骤必须先启用，然后才能为其指定 `--from_step` 或 `--to_step`，如 使用 -vivado 和 -advanced 选项 中所述。

启动 Vivado IDE 以执行交互式设计

例如，通过 `--to_step` 命令，即可启动构建进程以执行 Vivado 综合，然后在工程上启动 Vivado IDE 以手动对设计进行布局布线。要执行此操作，请使用以下命令语法：

```
v++ --target hw --link --to_step vpl.synth --save-temps --platform
<PLATFORM_NAME> <XO_FILES>
```

提示：如以上示例所示，使用 `--to_step` 时，您还必须指定 `--save-temps` 以保留此构建进程创建的所有临时文件。

此命令可指定硬件构建的链接进程、通过综合步骤运行构建，并保存由构建进程生成的临时文件。

您可在由 Vitis 编译器使用 `--interactive` 命令构建的工程上直接启动 Vivado 工具。这样会打开 Vivado 工程（此工程位于构建目录的 `<temp_dir>/link/vivado/vpl/prj` 下），以供您对设计进行交互式编辑。

```
v++ --target hw --link --interactive --save-temps --platform
<PLATFORM_NAME> <XO_FILES>
```

在此模式下调用 Vivado IDE 时，您可打开综合或实现运行以对该工程进行管理和修改。您可以按需更改运行详细信息，以达成时序收敛，并尝试不同的实现方法。您可将结果保存到设计检查点 (DCP)，或者生成工程比特流 (`.bit`)，以供在 Vitis 环境中用于生成器件二进制文件。

保存来自 Vivado IDE 的 DCP 后，即可关闭该工具并返回 Vitis 环境。`--reuse_impl` 选项可在 `v++` 命令中应用先前实现的 DCP 文件来生成 `xclbin`。

重要提示！ `--reuse_impl` 选项属于增量构建选项，要求您在使用 `--reuse_impl` 恢复 Vitis 编译器时，应用的工程目录与您使用 `--to_step` 启动构建时所指定的工程目录相同。

以下命令通过使用来自 Vivado 工具的指定 DCP 文件，基于输入文件创建 `project.xclbin`，从而完成链接进程。

```
v++ --link --platform <PLATFORM_NAME> -o'project.xclbin' project.xo --
reuse_impl ./_x/link/vivado/routed.dcp
```

您还可使用 Vivado 工具生成的比特流文件来创建 `project.xclbin`：

```
v++ --link --platform <PLATFORM_NAME> -o'project.xclbin' project.xo --
reuse_bit ./_x/link/vivado/project.bit
```

其它 Vivado 选项

在 `v++` 命令行或配置文件中可使用的部分其它开关包括：

- `--export_script`/`--custom_script` 编辑和使用 Tcl 脚本来修改编译或链接进程。
- `--remote_ip_cache` 为 Vivado 综合指定远程 IP 高速缓存目录。
- `--no_ip_cache` 为 Vivado 综合关闭 IP 高速缓存。这将导致在构建进程中重新综合所有 IP，并擦除已缓存的数据。

控制报告生成

`v++-R` 选项（或者 `--report_level`）可用于控制编译或链接期间要为硬件仿真和系统目标报告的信息级别。通常如果构建生成的报告越少，则运行得越快。

命令行选项如下：

```
$ v++ -R <report_level>
```

其中 `<report_level>` 为以下选项之一：

- `-R0`：最少量的报告，无中间设计检查点 (DCP)。
- `-R1`：包括 R0 报告，以及：
  - 识别要为每个内核复查的设计特征 (`report_failfast`)。
  - 识别要为整个最优化后设计复查的设计特征。
  - 保存最优化后的设计检查点 (DCP) 文件，以便稍后在 Vivado Design Suite 内进行检验或使用。

  提示： `report_failfast` 实用工具可以突显潜在的器件使用困难、时钟约束问题以及可能无法达成的目标频率 (MHz)。
- `-R2`：包括 R1 报告，以及：
  - 包含来自 Vivado 工具的所有标准报告，包括每个实现步骤后保存的 DCP。
  - 布局后要为每个 SLR 复查的设计特征。
- `-Restimate`：强制 Vitis HLS 生成“系统估算 (System Estimate)”报告，如 系统估算报告 中所述。

  提示： 该选项对于软件仿真构建 (`-t sw_emu`) 很有用。

XILINX

第 14 章

封装系统

编译并链接内核代码以构建 `.xclbin` 后，您需要对器件二进制文件以及所有必需的支持文件进行封装，以构建可在软件或硬件仿真时运行的程序包，或者可在硬件器件上启动并运行的程序包。`v++ --package` 步骤或 `-p` 会在 `v++` 编译和链接进程结束时封装最终产品。这是所有 Versal™ 平台的必需步骤，包括 AI 引擎平台和嵌入式处理器平台。

如 --package 选项 中所述，此命令允许您对设计进行封装，并定义启动和配置赛灵思器件所需的各种文件，以便在仿真期间或者在量产系统中使用。它会收集各种元素以创建 SD 卡，或者通过其它方式进行器件编程，以定义操作系统，并加载应用与内核代码。

嵌入式平台封装

对于嵌入式平台，`--package` 命令支持各种工具流程和平台，包括 Versal、AI 引擎和 Zynq® 器件。命令行如下所示：

```
v++ --package -t [sw_emu | hw_emu | hw] --platform <platform> input.xclbin
[ -o output.xclbin ]
```

注释：如果未指定输出选项 (`-o`)，该工具会以默认名称 `a.xclbin` 创建输出文件。

对于 AI 引擎平台，封装进程还会将 `aiecompiler` 命令生成的 `libadf.a` 文件集成到输出器件二进制文件中。如需了解更多信息，请参阅《Versal ACAP AI 引擎编程环境用户指南》(UG1076)。

`--package` 命令具有各种选项可搭配 Vitis 工具支持的不同平台和构建目标一起使用。在 Vitis IDE 中，封装进程会自动执行，该工具会按需创建所需文件。但在命令行流程中，您必须指定 `v++ --package` 命令，或者在 `config` 文件中使用适合此作业的选项来添加 `[package]` 标签。以下是硬件仿真的命令示例，此命令用于为基于 ZCU104 的应用运行封装进程：

```
v++ --package -t hw_emu --platform xilinx_zcu104_base_202010_1 --save-temps
\
./input.xclbin ./output.xclbin --config package.cfg
```

其中 `--config package.cfg` 选项用于指定 Vitis 编译器的配置文件，可搭配针对封装进程指定的各种选项一起使用。以下是配置文件示例的内容：

```
[package]
out_dir=sd_card
boot_mode=sd
image_format=ext4
rootfs=/tmp/platforms/sw/zynqmp/xilinx-zynqmp-common-v2020.1/rootfs.ext4
sd_file=/tmp/platforms/sw/zynqmp/xilinx-zynqmp-common-v2020.1/Image
sd_file=host.elf
sd_file=output.xclbin
sd_file=xrt.ini
sd_file=launch_app.sh
```

对于软件和硬件仿真，此命令以 .xclbin 文件作为输入、生成脚本以启动仿真（launch_sw_emu.sh 或 launch_hw_emu.sh），并将所需的支持文件写入指定的输出文件夹 --package.out_dir。

运行此应用所需的其它文件都必须包含在输出文件中，例如，作为应用输入或用于确认应用所需的数据文件，或者用于剖析和调试的 xrt.ini 文件；这些文件可使用 sd_file 选项单独进行传输，或者可以使用 sd_dir 选项作为一个目录来进行传输，如 --package 选项 中所述。

根据通过 --package.boot_mode 选项指定的启动模式，对于硬件构建，--package 命令会创建 sd_card 文件夹或者 QSPI.img。

提示：对于 PS 核上运行的裸机 ELF 文件，还应在命令行中添加以下选项：

```
--package.ps_elf <elf>,<core>
```

此 package 命令会创建名为 sd_card 的输出文件夹，其中包含为应用运行硬件仿真所需的所有文件，用于对 sd_card 的启动进程进行建模。对于硬件构建，它包含创建 SD 卡以启动器件所需的文件。

创建 sd_card 文件夹后，请将内容复制到 SD 卡上以创建启动镜像。

注释：在 Windows 操作系统上，您必须使用第三方工具（例如，Etcher）来写入 SD 卡，以供在启动赛灵思器件时使用。

当封装进程完成后，您可使用 Vitis 分析器工具通过运行以下命令来可视化并浏览相关报告和日志文件：

```
vitis_analyzer ./<output>.package_summary
```

数据中心平台的封装

提示：数据中心加速器卡无需 --package 命令。

--package 命令会复制器件二进制文件 (.xclbin) 以生成输出 .xclbin。命令行如下所示：

```
v++ --package -t [sw_emu | hw_emu | hw] --platform <platform> input.xclbin
[ -o output.xclbin ]
```

如果未指定输出选项 (-o)，该工具会以默认名称 a.xclbin 创建输出文件。

当封装进程完成后，您可使用 Vitis 分析器工具通过运行以下命令来可视化并浏览相关报告和日志文件：

```
vitis_analyzer ./<output>.package_summary
```

v++ 命令的输出目录

命令行流程所生成的目录结构已经过组织，以便您轻松查找和访问来自工程的文件。通过浏览各 compile、link、logs 和 reports 目录，您可轻松查找生成的文件。每个内核也同样创建有目录结构。

在命令行上使用 v++ 时，默认情况下它会在编译和链接期间创建目录结构。XO 和 xclbin 文件始终在当前工作目录中生成。所有中间文件都在 --temp_dir 选项所指定的目录下创建，如果不指定 --temp_dir，则默认目录为 _x。link、logs 和 reports 目录默认位于 temp_dir 内，并包含有关构建的相应信息。

您可以选择使用以下 v++ 选项更改目录结构：

```
--temp_dir <dir_name>
--log_dir <dir_name>
--report_dir <dir_name>
```

应用示例使用的是以下命令行：

```
## Kernel Compilation command:
v++ -t hw_emu --config design.cfg -c -k mmult -I'../src' \
-o'mmult.hw_emu.xilinx_u200_xdma_201830_3.xo' '../src/mmult.cpp'

## Device Binary Linking Command:
v++ -t hw_emu --config design.cfg -l \
-o'mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin'
mmult.hw_emu.xilinx_u200_xdma_201830_3.xo
```

design.cfg 文件包含：

```
platform=xilinx_u200_xdma_201830_3
debug=1
save-temps=1
temp_dir=temp_dir

[connectivity]
nk=mmult:1:mmult_1
```

后接输出目录结构，其中 $cwd 表示当前工作目录，这些命令都是从该目录中启动的。

```
### V++ Command Line Directory Structure
$cwd
    >design.cfg

    >emconfig.json -- emulation platform generated by emconfigutil

    >emulation_debug.log

    >host.exe  -- host executable
    >mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin  -- device binary
generated by v++ --link
```

```
    >mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin.info  -- device binary
information file generated by kernelinfo utility

    >mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin.link_summary  -- summary
report of the v++ link command viewable in Vitis analyzer

    >mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin.run_summary  -- summary
report of the hardware emulation run viewable in Vitis analyzer

    >mmult.hw_emu.xilinx_u200_xdma_201830_3.xo  -- compiled kernel object
file generated by v++ --compile

    >mmult.hw_emu.xilinx_u200_xdma_201830_3.xo.compile_summary  --  summary
report of the v++ compile command viewable in Vitis analyzer

    >opencl_summary.csv  -- OpenCL summary produced by XRT during the host/
kernel run

    >temp_dir  -- The build directory specified by the --temp_dir option.
This defaults to _x when not specified.

        >link  --  the output files of the link process
            >activetask.json
            >int
                >address_map.xml
                >appendSection.rtd
                >behav_waveform
                    >xsim  --  the Vivado simulator contents with results
from hardware emulation
                >behav.xse
                >cf2sw_full.rtd
                >cf2sw.rtd
                >consolidated.cf
                >dr.bd.tcl
                >kernel_info.dat
                >_kernel_inst_paths.dat
                >kernel_service.json
                >mmult
                >mmult.hw_emu.xilinx_u200_xdma_201830_3_build.rtd
                >mmult.hw_emu.xilinx_u200_xdma_201830_3.gpp_so.log
                >mmult.hw_emu.xilinx_u200_xdma_201830_3.rtd
                >mmult.hw_emu.xilinx_u200_xdma_201830_3.so
                >mmult.hw_emu.xilinx_u200_xdma_201830_3.xml
                >mmult.hw_emu.xilinx_u200_xdma_201830_3_xml.rtd
                >_new_clk_freq
                >sdsl.dat
                >syslinkConfig.ini
                >systemDiagramModel.json
                >systemDiagramModelSlrBaseAddress.json
                >vplConfig.ini
                >vplsettings.json
                >xclbin_orig.1.xml
                >xclbin_orig.xml
                >xclbin_orig.xml.tmp
                >xo  -- the temporary kernel files created by the v++ --
compile command
                    >ip_repo
                    >mmult
            >link.spr
            >link.steps.log
            >run_link
                >gen_run.xml
                >htr.txt
```

```
                >vpl.pb
            >sys_link  -- files related to the platform used during linking
                >bd
                >cfgraph
                >dr.xml
                >emu
                >iprepo
                >sdsl.dat
                >_sysl
                >xilinx_u200_xdma_201830_3.hpfm
            >vivado  --  the Vivado Design Suite files for synthesis,
implementation, and bitstream generation
                >vivado.spr
                >vpl
                    >gen_run.xml
                    >htr.txt
                    >ipirun.tcl
                    >ISEWrap.js
                    >ISEWrap.sh
                    >openprj.tcl
                    >output
                        >emu_ooc_copy.xdc
                        >insert_debug_profiling.tcl
                        >_post_sys_link_gen_constrs.xdc
                        >resource.json
                    >prj  --  the Vivado Design Suite project files
                        >prj.cache
                        >prj.gen
                        >prj.hw
                        >prj.ip_user_files
                        >prj.sim
                        >prj.srcs
                        >prj.xpr
                    >rundef.js
                    >runme.bat
                    >runme.log
                    >runme.sh
                    >scripts
                        >_vivado_params.tcl
                    >vivado_config_hw_emu.tcl
                    >vivado.jou
                    >vivado.log
                    >vivado.pb
                    >vpl.tcl
        >logs

            >link  -- Logs from the linking process
                >link.steps.log
                >v++.log
            >mmult.hw_emu.xilinx_u200_xdma_201830_3  -- Logs from the
compiltion process
                >mmult.hw_emu.xilinx_u200_xdma_201830_3.steps.log
                >mmult_vitis_hls.log
                >v++.log
            >optraceViewer.html
        >mmult.hw_emu.xilinx_u200_xdma_201830_3
            >mmult
                >htr.txt
                >ISEWrap.js
                >ISEWrap.sh
                >mmult  -- Vits HLS files used and genrated during
compilation
                    >hls.app
```

```
                    >ip
                    >kernel.xml
                    >kernel.xml.orig
                    >mmult.design.xml
                    >solution
                >mmult.tcl
                >rundef.js
                >runme.bat
                >runme.log
                >runme.sh
                >vitis_hls.log
                >vitis_hls.pb
            >mmult.hw_emu.xilinx_u200_xdma_201830_3.spr
            >mmult.hw_emu.xilinx_u200_xdma_201830_3.steps.log

        >reports

            >link  -- Reports generated during the linking process
                >system_estimate_mmult.hw_emu.xilinx_u200_xdma_201830_3.xtxt
                >v++
+_link_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.html
            >mmult.hw_emu.xilinx_u200_xdma_201830_3  -- Reports generated
during the compilation process
                >hls_reports
                >system_estimate_mmult.hw_emu.xilinx_u200_xdma_201830_3.xtxt
                >v++
+_compile_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.html
        >v++_compile_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.json
-- Design guidance generated during compilation

        >v++_compile_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.pb

        >v++_link_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.json  --
Design guidance generated during linking

        >v++_link_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.pb
    >opencl_trace.csv  -- OpenCL of events generated by XRT during run
time.

    >v++_mmult.hw_emu.xilinx_u200_xdma_201830_3.log  --  Log file generated
from the v++ commands

    >xcd.log

    >xclbin.run_summary  -- secondary run_summary report

    >xilinx_u200_xdma_201830_3-0-
mmult.hw_emu.xilinx_u200_xdma_201830_3_simulate.log

    >xilinx_u200_xdma_201830_3-0-
mmult.hw_emu.xilinx_u200_xdma_201830_3_xsc_report.log

    >xrc.log

    >xrt.ini
```

XILINX

第 16 章

运行仿真

以 FPGA 为目标的用户应用和硬件内核的开发需要采用分阶段开发方法。由于 FPGA、Versal™ ACAP 和 Zynq UltraScale+ MPSoC 均为可编程器件，因此为硬件构建器件二进制文件需要一些时间。为了在无需执行整个硬件编译流程的前提下加速迭代，Vitis™ 工具提供了可供运行应用与内核的仿真目标。编译仿真目标比编译实际硬件要快得多。此外，仿真目标可以提供对应用或加速器的完整可视性，从而使执行调试变得更简单。在仿真中设计通过后，在开发的后续阶段内，您即可在硬件平台上编译和运行应用。

Vitis 工具可提供 2 种仿真目标：

- 软件仿真 (sw_emu)：软件仿真构建可以快速编译和链接，主机程序可在 x86 处理器上本机运行或者也可以在 QEMU 仿真环境内运行。PL 内核在主机上本机编译和运行。此构建目标支持您在主机代码和内核逻辑上快速迭代。
- 硬件仿真 (hw_emu)：主机程序在 `sw_emu` 下运行（在 x86 上本机运行或者在 QEMU 内运行），但内核代码编译为 RTL 行为模型，此模型在 Vivado® 仿真器内运行或者在其它受支持的第三方仿真器内运行。该构建和运行循环需要更长时间，但是提供周期精确的内核逻辑视图。

任一仿真目标的编译均无缝集成到 Vitis 命令行和 IDE 流程中。您无需对源代码进行任何更改即可对任一仿真目标的主机和内核源代码进行编译。对于主机代码，您无需为仿真采用不同编译方法，因为在仿真中可使用相同的主机可执行文件或 PS 应用 ELF 二进制文件。仿真目标支持大部分功能，包括 XRT API、缓冲器传输、平台存储器 SP 标签、内核到内核连接等。

运行仿真目标

仿真目标有其自己的特定于目标的驱动程序，通过 XRT 来加载。因此，CPU 二进制文件无需重新编译即可按现状运行，只需在运行时期间更改目标模式即可。XRT 可基于 `XCL_EMULATION_MODE` 环境变量的值，加载特定于目标的驱动程序，并使应用与硬件的仿真模型相连。`XCL_EMULATION_MODE` 允许的值为 `sw_emu` 和 `hw_emu`。如未设置 `XCL_EMULATION_MODE`，则 XRT 将加载硬件驱动程序。

重要提示！运行仿真时，必须设置 `XCL_EMULATION_MODE`。

图 25：**XRT 驱动程序**

您也可以使用 `xrt.ini` 文件来配置适用于仿真的各项选项。在 `xrt.ini` 中专为 `[Emulation]` 提供了一个部分，如 xrt.ini 文件 中所述。

数据中心平台对比嵌入式平台

数据中心平台和嵌入式平台均支持仿真。对于数据中心平台，主机应用是针对 x86 服务器编译的，而器件是作为仿真硬件的独立 x86 进程来建模的。用户主机代码和器件模型进程使用 RPC 调用来进行通信。对于嵌入式平台，CPU 代码在嵌入式 Arm 处理器上运行，仿真流程使用 QEMU (Quick Emulator) 来模拟基于 Arm 的 PS 子系统。在 QEMU 中，您可以在仿真目标上启动嵌入式 Linux 并运行 Arm 二进制文件。

要运行数据中心应用的软件仿真 (`sw_emu`) 和硬件仿真 (`hw_emu`)，必须使用 `emconfigutil` 命令编译加速器卡的仿真模型，并设置 `XCL_EMULATION_MODE` 环境变量，然后才能启动应用。如需了解有关这些步骤的详细信息，请参阅 在数据中心加速器卡上运行仿真。

要运行嵌入式应用的 `sw_emu` 或 `hw_emu`，您必须在 x86 处理器上启动 QEMU 仿真环境，以便对 Arm 处理器的执行环境进行建模。这需要使用 `launch_emulator.py` 命令或构建进程期间生成的 shell 脚本。如需了解有关此流程的详细信息，请参阅 在嵌入式处理器平台上运行仿真。

QEMU

QEMU 表示 Quick Emulator，即快速仿真器。它是通用的开源机器仿真器。赛灵思提供了自定义的 QEMU 模型，此模型模拟 Versal ACAP、Zynq® UltraScale+™ MPSoC 和 Zynq-7000 SoC 器件上存在的基于 Arm 的处理器系统。QEMU 模型提供了几乎实时执行 CPU 指令的能力，无需真实硬件。如需了解更多信息，请参阅《赛灵思快速仿真器用户指南：QEMU》。

对于硬件仿真，Vitis 仿真目标为针对设计其余部分使用 QEMU 并将其与 RTL 和基于 SystemC 的模型进行协同仿真，以便为整个平台提供完整的执行模型。您可在此模型上启动嵌入式 Linux 内核，并运行基于 XRT 的加速器应用。由于 QEMU 可以执行 Arm 指令，因此，您可以在仿真流程中按现状运行 Arm 二进制文件，无需重新编译。QEMU 还允许您使用来自赛灵思系统调试器 (XSDB) 的基于 GDB 和 TCF 的目标连接。

Vitis 仿真流程还使用 QEMU 来对 MicroBlaze™ 处理器进行仿真，以便对器件的平台管理模块（PLM 和 PMU）进行仿真。在 Versal 器件上，PLM 固件用于将 PDI 加载到 PS 和 AI 引擎模型的程序段上。

为了确保 QEMU 配置与平台相匹配，在 Vitis 平台的 `sw` 目录中还必须提供附加文件。`qemu_args.txt` 和 `pmc_args.txt` 这两个常用文件包含在启动 QEMU 时要使用的命令行实参。创建定制平台时，这两个文件会随默认内容一起添加到您的平台上。您可以复查这些文件并按需对其进行编辑，以便对您的定制平台进行建模。请参阅赛灵思嵌入式平台以获取示例。

由于 QEMU 属于通用模型，它使用 Linux 设备树风格的 DTB 格式文件来启用和配置各种硬件模块。Vitis 工具随附有默认 QEMU 硬件 DTB 文件，包含在 `<vitis_installation>/data/emulation/dtbs` 文件夹内。但如果您的平台需要其它 QEMU DTB，那么您也可将其封装在自己的平台内。

提示：QEMU DTB 会提供 QEMU 的硬件配置，且它不同于 Linux 内核所使用的 DTB。

在数据中心加速器卡上运行仿真

提示：运行构建前，请按 设置 Vitis 环境 中所述设置命令 shell 或窗口。

1. 在 `xrt.ini` 文件中设置期望的运行时设置。这是可选步骤。

   如 xrt.ini 文件 中所述，运行主机应用和内核执行时，此文件可指定各种参数用于控制 XRT 中的调试、剖析和消息日志记录。这样即可支持运行时在运行应用时捕获调试和剖析数据。`xrt.ini` 中的 `Emulation` 组可以提供影响仿真运行的功能特性。

   提示：为仿真 (emulation) 模式编译内核代码时，请务必使用 `v++ -g` 选项。

2. 从目标平台创建 `emconfig.json` 文件，如 emconfigutil 实用工具 中所述。此步骤对于运行硬件或软件仿真而言是必需的。

   仿真配置文件 `emconfig.json` 是使用 `emconfigutil` 命令从指定平台生成的，可提供信息以供 XRT 库在仿真期间使用。以下示例为指定目标平台创建了 `emconfig.json` 文件：

   ```
   emconfigutil --platform xilinx_u200_xdma_201830_2
   ```

   在仿真模式下，运行时会在主机可执行文件所在的目录内查找 `emconfig.json` 文件，并读入目标配置用于运行仿真。

   提示：您必须拥有最新 JSON 文件才能在自己的目标平台上运行仿真。

3. 将 `XCL_EMULATION_MODE` 环境变量设置为相应的 `sw_emu`（软件仿真）或 `hw_emu`（硬件仿真）。这样即可将应用执行模式更改为仿真模式。

   以下语法可用于为 C shell (csh) 设置环境变量：

   ```
   setenv XCL_EMULATION_MODE sw_emu
   ```

   Bash shell：

   ```
   export  XCL_EMULATION_MODE=sw_emu
   ```

   重要提示！如不正确设置 `XCL_EMULATION_MODE` 环境变量，仿真目标将不会运行。

4. 运行应用。

设置好运行时初始化文件 (`xrt.ini`)、仿真配置文件 (`emconfig.json`) 和 `XCL_EMULATION_MODE` 环境变量后，请使用所需的命令行实参来运行主机可执行文件。

重要提示！ INI 文件与 JSON 文件必须与可执行文件位于相同目录内。

例如：

```
./host.exe kernel.xclbin
```

提示： 此命令行假定主机程序编写时采用 `xclbin` 文件的名称作为实参，就像大部分 Vitis 示例和教程一样。但您的应用可能已将 `xclbin` 文件的名称硬编码到主机程序中，或者可能要求采用其它方法来运行此应用。

在嵌入式处理器平台上运行仿真

建议： 机器的文件大小限制应设置为无限或者较高的值（高于 16 GB），因为嵌入式硬件仿真 (HW Emulation) 为存储器创建的文件大小可能较大。

提示： 运行构建前，请按 设置 Vitis 环境 中所述设置命令 shell 或窗口。

1. 在 `xrt.ini` 文件中设置期望的运行时设置。

   如 xrt.ini 文件 中所述，运行主机应用和内核执行时，此文件可指定各种参数用于控制 XRT 中的调试、剖析和消息日志记录。如 在应用中启用剖析 中所述，这样即可支持运行时在运行应用时捕获调试和剖析数据。

   `xrt.ini` 文件和运行应用所需的所有其它文件都必须包含在输出文件内，如 嵌入式平台封装 中所述。

   提示： 为仿真 (emulation) 模式编译内核代码时，请务必使用 `v++ -g` 选项。

2. 运行 `launch_sw_emu.sh` 脚本或 `launch_hw_emu.sh` 脚本即可启动 QEMU 仿真环境。

   ```
   launch_sw_emu.sh -forward-port 1440 22
   ```

   此脚本是在封装进程期间在仿真 (emulation) 目录中创建的，它使用 `launch_emulator.py` 命令来设置和启动 QEMU。启动仿真 (emulation) 脚本时，您还可以为 `launch_emulator.py` 命令指定选项。例如，指定 `-forward-port` 选项即可将 QEMU 端口转发到本地系统上打开的端口。尝试从 QEMU 复制文件时，需使用该选项，如以下步骤 5 中所述。请参阅 launch_emulator 实用工具 以获取有关此命令的详细信息。

   另一个示例是指定 `launch_hw_emu.sh -enable-debug` 以配置要打开的其它 XTERM，以供 QEMU 和 PL 进程观测命令执行时处于活动状态的脚本以帮助调试应用。默认不启用该选项，但它对于调试很有用。

3. 使用所需设置来装载和配置 QEMU shell。

   赛灵思嵌入式基本平台将 `rootfs` 包含在 SD 卡的独立 EXT4 分区内。启动 Linux 后，需装载此分区。如果手动运行仿真 (emulation)，则需要从 QEMU shell 运行以下命令：

   ```
   mount /dev/mmcblk0p1 /mnt
   cd /mnt
   export LD_LIBRARY_PATH=/mnt:/tmp:$LD_LIBRARY_PATH
   export XCL_EMULATION_MODE=hw_emu
   export XILINX_XRT=/usr
   export XILINX_VITIS=/mnt
   ```

提示：您可以将 XCL_EMULATION_MODE 环境变量设置为 sw_emu（用于软件仿真）或 hw_emu（用于硬件仿真）。这样即可将主机应用配置为在仿真 (emulation) 模式下运行。

4. 从 QEMU shell 内运行应用。

   通过运行时初始化 (xrt.ini) 设置好 XCL_EMULATION_MODE 环境后，按主机应用要求，通过命令行来运行可执行文件。例如：

   ```
   ./host.elf kernel.xclbin
   ```

   提示：此命令行假定主机程序编写时采用 xclbin 文件的名称作为实参，就像大部分 Vitis 示例和教程一样。但您的应用可能已将 xclbin 文件的名称硬编码到主机程序中，或者可能要求采用其它方法来运行此应用。

5. 应用运行完成后，运行时可能已生成某些文件，例如，opencl_summary.csv、opencl_trace.csv 和 xclbin.run_summary。在 QEMU 环境内的 /mnt 文件夹下可能可以找到这些文件。但是，要查看这些文件，就必须将其从 QEMU Linux 系统重新复制回本地系统。可使用 scp 命令复制这些文件，如下所示：

   ```
   scp -P 1440 root@<host-ip-address>:/mnt/<file> <dest_path>
   ```

   其中：

   - 1440 是要连接到的 QEMU 端口。
   - root@<host-ip-address> 是位于指定 IP 地址的 QEMU 下运行的 PetaLinux root 用户登录信息。默认 root 用户密码为“root”。
   - /mnt/<file> 是要从 QEMU 环境复制的文件的路径和文件名。
   - <dest_path> 用于指定将文件复制到本地系统上的路径和文件名。

   例如：

   ```
   scp -P 1440 root@172.55.12.26:/mnt/xclbin.run_summary
   ```

6. 当应用完成仿真 (emulation) 并且您已复制所有必要文件后，单击“Ctrl + a + x”键即可终止 QEMU shell 并返回到 Linux shell。

   注释：如果无法终止 QEMU 环境，可结束 QEMU 启动的用于运行此环境的进程。该工具会在脚本启动时报告进程 ID (pid)，或者您可以指定 -pid-file 选项以在启动仿真 (emulation) 时捕获 pid。

硬件仿真的速度和精度

硬件仿真 (Hardware emulation) 通过混用 SystemC 和 RTL 协同仿真在仿真 (simulation) 精度与速度之间实现平衡。SystemC 模型由纯功能模型与性能近似模型混合而成。硬件仿真模拟硬件的精度无法达到 100%，因此可以预测运行仿真与在硬件上执行应用之间存在一定的行为差异。这可能导致应用性能出现显著差异，有时也可能出现功能差异。

硬件功能差异通常表明设计中存在竞争条件或者某些不可预测的行为。因此，硬件中出现的问题在硬件仿真中可能并非总能复现，但主机与加速器之间或者加速器与存储器之间的交互的大部分相关行为都能在硬件仿真中得到复现。这使硬件仿真成为用于在硬件上运行加速器之前对其问题进行调试的最适合的工具。

下表列出了用于模拟硬件平台的模型及其精度级别。

表 15：**硬件平台**

硬件功能	描述
主机到卡的 PCIe® 连接和 DMA（XDMA 和 SlaveBridge）	对于数据中心平台，通过 PCIe 连接到 x86 主机服务器的操作是作为纯功能模型来执行的，不包含任何性能建模。因此，与 PCIe 带宽有关的任何问题都无法反映在硬件仿真运行中。
UltraScale™ DDR 存储器，SmartConnect	用于 DDR 存储器控制器、AXI SmartConnect 和其它数据路径 IP 的 SystemC 模型通常为吞吐量近似模型。这些模型通常无法对硬件 IP 的精确时延进行建模。此类模型可用于测量您修改应用或加速器内核过程中的相对性能趋势。
AI 引擎	AI 引擎 SystemC 模型是周期近似模型，但它并非旨在实现 100% 周期精确。在 AI 引擎仿真器 (Simulator) 与硬件仿真之间使用公用模型，因此支持对这两个阶段进行合理比较。
Versal NoC 和 DDR 模型	Versal NoC 和 DDR SystemC 模型均为周期近似模型。
Arm 处理子系统（PS 和 CIPS）	Arm PS 是使用 QEMU 建模的，属于纯功能性执行模型。如需了解更多信息，请参阅 QEMU。
用户内核（加速器）	硬件仿真 (Hardware emulation) 会对用户加速器使用 RTL。如下所示，加速器本身行为精度为 100%。但加速器周围有其它近似模型。
其它 I/O 模型	对于硬件仿真，有基于 Python 或 C 语言的通用流量生成器 (Traffic Generator) 可与仿真进程相连。您可在 AXI 协议级别生成抽象流量，用于模拟设计中的 I/O。由于这些模型均为抽象模型，因此特定硬件开发板上观测到的任何问题都不会出现在硬件仿真中。

由于硬件仿真使用 RTL 协同仿真作为其执行模型，因此执行速度相比真实硬件要慢多个数量级。赛灵思建议使用较小的数据缓冲器。例如，如果您已添加可配置矢量，并且在硬件中执行含 1024 个元素的 `vadd`，那么在仿真中可以将其限制为 16 个元素。这样您即可通过加速器来测试自己的应用，同时仍可在合理时间内完成执行。

在硬件仿真中使用仿真器

仿真器支持

Vitis 工具使用 Vivado 逻辑仿真器 (`xsim`) 作为所有平台的默认仿真器，包括 Alveo 数据中心加速器卡、Versal 和 Zynq UltraScale+ MPSoC 嵌入式平台。但对于 Versal 嵌入式平台（如 `xilinx_vck190_base` 或与之相似的定制平台），Vitis 工具还支持使用第三方仿真器进行硬件仿真：Mentor Graphics Questa Advanced Simulator、Xcelium 和 VCS。受支持的仿真器的具体版本与 Vivado Design Suite 支持的版本相同。

提示：对于数据中心平台，硬件仿真支持使用 Questa Advanced Simulator 的 U250_XDMA 平台。除非明确说明，否则此支持不包括诸如点对点 (P2P)、SlaveBridge 或其它功能特性。

启用第三方仿真器需在生成器件二进制文件 (`.xclbin`) 以及支持性的 Tcl 脚本期间实现额外配置选项。每个仿真器的具体要求如下所述。另请注意，针对第三方仿真器应首先运行 Vivado 设置，然后才能在 Vitis 中使用这些仿真器。具体来说，您必须使用 `compile_sim_lib` Tcl 命令对仿真模型进行预编译。如需了解更多详情，请参阅《Vivado Design Suite 用户指南：逻辑仿真》(UG900)，以获取第三方仿真器的设置信息。

- Questa：将以下高级参数和 Vivado 属性添加到配置文件中以供在链接期间使用：

```
## Final set of additional options required for running simulation using
Questa Simulator
[advanced]
param=hw_emu.simulator=QUESTA
[vivado]
prop=project.__CURRENT__.simulator.questa_install_dir=/tools/gensys/
questa/2020.4/bin/
prop=project.__CURRENT__.compxlib.questa_compiled_library_dir=<install_dir
>/clibs/questa/2020.4/lin64/lib/
prop=fileset.sim_1.questa.compile.sccom.cores={4}
```

  生成配置文件后，您可在 `v++` 命令行中按如下方式使用该文件：

```
v++ -link --config questa_sim.cfg
```

- Xcelium：将以下高级参数和 Vivado 属性添加到配置文件中以供在链接期间使用：

```
## Final set of additional options required for running simulation using
Xcelium Simulator
[advanced]
param=hw_emu.simulator=XCELIUM
[vivado]
prop=project.__CURRENT__.simulator.xcelium_install_dir=/tools/dist/xlm/
20.09.006/tools.lnx86/xcelium/bin/
prop=project.__CURRENT__.compxlib.xcelium_compiled_library_dir=/clibs/
xcelium/20.09.006/lin64/lib/
prop=fileset.sim_1.xcelium.elaborate.xmelab.more_options={-timescale 1ns/
1ps}
```

  生成配置文件后，您可在 `v++` 命令行中按如下方式使用该文件：

```
v++ -link --config xcelium.cfg
```

- VCS：将以下高级参数和 Vivado 属性添加到配置文件中以供在链接期间使用：

```
## Final set of additional options required for running simulation using
VCS Simulator
[advanced]
param=hw_emu.simulator=VCS
[vivado]
prop=project.__CURRENT__.simulator.vcs_install_dir=/tools/gensys/vcs/
R-2020.12/bin/
prop=project.__CURRENT__.compxlib.vcs_compiled_library_dir=/clibs/vcs/
R-2020.12/lin64/lib/
prop=project.__CURRENT__.simulator.vcs_gcc_install_dir=/tools/installs/
synopsys/vg_gnu/2019.06/amd64/gcc-6.2.0_64/bin
```

  生成配置文件后，您可在 `v++` 命令行中按如下方式使用该文件：

```
v++ -link --config vcs_sim.cfg
```

您可使用来自 `launch_emulator.py` 命令的 `-user-pre-sim-script` 和 `-user-post-sim-script` 选项来指定要在仿真开始前或者仿真完成后运行的 Tcl 脚本。例如，在这些脚本中，您可以使用 `$cwd` 命令来获取仿真器的运行目录，并在仿真前复制所需的任意文件，或者在仿真结束后复制生成的任意输出文件。

要启用硬件仿真，必须在 Vivado Design Suite 中设置仿真环境。此设置的关键步骤之一是对 RTL 和 SystemC 模型进行预编译以供搭配仿真器使用。为此，您必须在 Vivado 工具中运行 `compile_sim_lib` 命令。如需了解有关仿真模型预编译的更多信息，请参阅《Vivado Design Suite 用户指南：逻辑仿真》(UG900)。

创建 Versal 平台以准备仿真时，Vivado 工具会生成仿真封装器，您必须在自己的仿真测试激励文件中例化此封装器。这样如果最顶层的设计模块为 `<top>`，那么在 Vivado 工具中调用 `launch_simulation` 时，它将生成 `<top>_sim_wrapper` 模块，并生成 `xlnoc.bd`。这些生成的文件将作为仅限仿真的源文件，只要在 Vivado 工具中调用 `launch_simulation`，就会覆盖这些文件。平台开发者需要在测试激励文件而不是自己的 `<top>` 模块中例化此模块。

使用仿真器波形查看器

硬件仿真是使用 RTL 和 SystemC 模型来执行的。基于常规应用和 HLS 的内核开发者无需留意硬件级别的细节。Vitis 分析器可以提供充足的硬件执行模型详细信息。但对于熟悉硬件信号与协议的高级用户，则可在运行仿真器波形时启动硬件仿真，如 波形视图和实时波形查看器 中所述。

默认情况下，运行 `v++ --link -t hw_emu` 时，该工具会以最优化模式来编译仿真模型。但当您同时指定 `-g` 开关时，则可启用在调试模式下编译硬件仿真模型。在应用运行时期间，将 `-g` 开关与 `launch_hw_emu.sh` 命令搭配即可在 GUI 模式下以交互方式运行仿真器，同时显示波形图。默认情况下，硬件仿真流程会在波形图窗口中添加感兴趣的常见信号。但您可以暂停仿真器来添加您感兴趣的信号，然后恢复并继续运行仿真。

XSIM 波形中显示的 AXI 传输事务

硬件仿真中的许多模型都使用 SystemC 传输事务级建模 (TLM)。在此类情况下，各模型之间的交互无法作为 RTL 波形来查看。但是，Vivado 仿真器 (xsim) 可以提供传输事务级查看器。对于标准平台，可按类似于添加 RTL 信号的方式将这些接口对象添加到波形视图中。例如，要将 AXI 接口添加到波形中，请在 `xsim` 中使用以下 Tcl 命令：

```
add_wave <HDL_objects>
```

使用 `add_wave` 命令可以指定到 HDL 对象的完整路径或相对路径。如需了解有关如何解读 TLM 波形以及如何在 GUI 中添加接口的更多详细信息，请参阅《Vivado Design Suite 用户指南：逻辑仿真》(UG900)。

处理 SystemC 模型

Vitis 应用加速开发流程中的 SystemC 模型允许您对 RTL 算法快速建模，以供在软件和硬件仿真中进行快速分析。如果在 RTL 内核仍在开发时，您想要直接进行某些系统分析，那么您可使用此方法来对部分系统进行建模。

SystemC 模型功能支持使用 `ap_ctrl_hs` 和 `ap_ctrl_chain` 的所有 XRT 管理的内核执行模型。它还支持对 AXI4 接口 (`m_axi`) 和 AXI4-Stream 接口 (`axis`) 进行建模，并支持对 `s_axilite` 接口进行寄存器读写操作。

您可在 SystemC TLM 模型内对自己的内核代码进行建模、提供到其它内核与主机应用的接口，并在仿真期间使用自己的内核代码。您可以创建赛灵思对象文件 (XO) 将 SystemC 模型链接到 `xclbin` 中的其它内核。后续章节探讨了如何创建 SystemC 模型、如何使用 `create_sc_xo` 命令创建 XO，以及如何使用 `v++` 命令生成 `xclbin`。

提示：请谨记，SystemC 模型并非周期精确模型，因此会影响仿真的时序结果。它不影响 RTL 代码的实际带宽、时延或吞吐量。

对 SystemC 模型进行编码

定义 SystemC 模型的过程步骤如下：

1. 包含头文件："xtlm_ap_ctrl.h" 和 "xtlm.h"。
2. 基于支持的内核类型（如 ap_ctrl_chain、ap_ctrl_hs 等），从预定义的类衍生出内核。
3. 声明并定义内核上使用的 AXI 接口。
4. 添加所需的内核实参，其中包含正确的地址偏移和大小。
5. 在 main() 线程中写入内核主体。

以下代码示例演示了此过程。

创建 SystemC 模型时，基于受支持的控制协议（如 xtlm_ap_ctrl_chain、xtlm_ap_ctrl_hs 和 xtlm_ap_ctrl_none）从类衍生出内核。请使用以下结构来创建 SystemC 模型。

提示：以下示例基于简单矢量加法 (VADD) 设计示例，您可从 GitHub 上的 Vitis_Accel_Examples 中找到该示例。

```
#include "xtlm.h"
#include "xtlm_ap_ctrl.h"

class vadd : public xsc::xtlm_ap_ctrl_hs
{
    public:
        SC_HAS_PROCESS(vadd);
        vadd(sc_module_name name, xsc::common_cpp::properties& _properties):
        xsc::xtlm_ap_ctrl_hs(name)
        {
            DEFINE_XTLM_AXIMM_MASTER_IF(in1, 32);
            DEFINE_XTLM_AXIMM_MASTER_IF(in2, 32);
            DEFINE_XTLM_AXIMM_MASTER_IF(out_r, 32);

           ADD_MEMORY_IF_ARG(in1,   0x10, 0x8);
           ADD_MEMORY_IF_ARG(in2,   0x18, 0x8);
           ADD_MEMORY_IF_ARG(out_r, 0x20, 0x8);
           ADD_SCALAR_ARG(size,     0x28, 0x4);

            SC_THREAD(main_thread);
        }


        //! Declare aximm interfaces..
        DECLARE_XTLM_AXIMM_MASTER_IF(in1);
        DECLARE_XTLM_AXIMM_MASTER_IF(in2);
        DECLARE_XTLM_AXIMM_MASTER_IF(out_r);

        //! Declare scalar args...
        unsigned int size;

        void main_thread()
        {
            wait(ev_ap_start); //! Wait on ap_start event...

            //! Copy kernel args configured by host...
            uint64_t  in1_base_addr = kernel_args[0];
            uint64_t  in2_base_addr = kernel_args[1];
            uint64_t  out_r_base_addr = kernel_args[2];
            size = kernel_args[3];
```

```
            unsigned data1, data2, data_r;
            for(int i = 0; i < size; i++) {
                in1->read(in1_base_addr + (i*4), (unsigned
char*)&data1);  //! Read from in1 interface
                in2->read(in2_base_addr + (i*4), (unsigned
char*)&data2);  //! Read from in2 interface

                //! Add data1 & data2 and write back result
                data_r = data1 + data2;                 //! Add
                out_r->write(out_r_base_addr + (i*4), (unsigned
char*)&data_r); //! Write the result
            }

            ap_done(); //! completed Kernel computation...
        }

};
```

包含的文件位于 Vitis 安装层级的 `$XILINX_Vivado/data/systemc/simlibs/` 文件夹下。

内核名称是在为 SystemC 模型定义类时指定的（如上所述），继承自 `xtlm_ap_ctrl_hs` 类。

您必须声明并定义与内核实参关联的 AXI 接口，如以下结构所示：

```
DECLARE_XTLM_AXIMM_MASTER_IF(in1);
DEFINE_XTLM_AXIMM_MASTER_IF(in1, 32);
```

声明会将接口类型与实参相关联。定义则用于定义此接口的数据宽度。您还必须声明内核实参的寄存器偏移和大小，如下所示：

```
ADD_MEMORY_IF_ARG(in1, 0x10, 0x8);
```

指定内核实参、偏移和大小时，这些值应与 XRT 管理的内核的 AXI4-Lite 接口中反映的值相匹配，如 软件可控内核 和 XRT 管理的内核的控制要求 中所述。

低于 `0x10` 的地址为保留地址，供 XRT 用于管理内核执行。内核实参可从 `0x10` 开始向上指定。最重要的是，SystemC 模型中指定的实参、偏移和大小应与 Vitis HLS 或 RTL 内核中使用的值相匹配。

内核是在 SystemC `main_thread` 中执行的。此线程会等待至主机应用或 XRT 的 `ap_start` 位完成置位后才开始执行，届时内核即可按如下方式处理实参值：

1. 内核等待来自 XRT (`ev_ap_start`) 的开始信号。
2. 将内核实参映射至 SystemC 模型中的变量。
3. 读取输入。
4. 添加矢量并捕获结果。
5. 将输出写回主机。
6. 将完成信号发送给 XRT (`ap_done`)。

创建 XO

要从 SystemC 模型生成 XO 文件，请使用 `create_sc_xo` 命令。这样会使用 SystemC 内核源文件作为输入，创建 IP 以生成 XO，用于通过 Vitis 编译器与目标平台和其它内核进行链接。例如：

```
create_sc_xo vadd.cpp
```

从源文件生成 XO 文件涉及多个中间步骤，如生成“封装 IP (Package IP)”脚本和运行 `package_xo` 命令。如果需要，这些中间命令可用于调试。

以上 `create_sc_xo` 命令的输出为 `vadd.xo`。

使用 v++ 命令建立链接

通过将内核添加到 `v++ --link` 命令行中来链接 SystemC 模型 XO 文件。

```
v++ --link --platform <platform> --target hw_emu \
--config ./vadd.cfg --input_files ../vadd.xo --output ../vadd.link.xclbin \
--optimize 0 --save-temps --temp_dir ./hw_emu
```

SystemC 模型可用于软件仿真和硬件仿真，但硬件构建目标不支持此模型。

在 `xclbin` 内包含 SystemC 模型时，受 TLM 所限，设计无法再实现时钟周期精确。

使用 I/O 流量生成器

部分用户应用（如视频流媒体和基于以太网的应用）使用平台上的 I/O 端口执行进出平台的数据流传输。对于此类应用，对设计执行硬件仿真需要一种机制来模拟 I/O 端口的硬件行为以及对通过这些端口运行的数据流量进行仿真。I/O 流量生成器允许您对于在 Vitis 应用加速开发流程中执行硬件仿真期间或者在 Vivado Design Suite 中执行逻辑仿真期间流经 I/O 端口的流量进行建模。此机制支持 AXI4-Stream 和 AXI4 存储器映射接口 I/O 仿真。

为设计添加流量生成器

赛灵思器件具有丰富的 I/O 接口。Alveo 加速器卡主要包含 PCIe 和 DDR 存储器接口，这些接口具有其自己的专用模型。但是，您的平台也可包含其它 I/O，例如，基于 GT 内核的通用 I/O、视频流传输和传感器数据。I/O 流量生成器 (Traffic Generator) 内核提供了一种方法，可供平台和应用用于在仿真期间将流量注入 I/O。

该解决方案需要在设计中包含数据流传输 I/O 内核 (XO) 或 IP 并使用赛灵思所提供的 Python/C++/C 来注入流量或者捕获来自仿真进程的输出数据。赛灵思提供的 Python/C++/C 库可用于将流量生成器代码集成到您的应用中，将其作为独立进程来运行，并使其与仿真进程相连。当前，赛灵思提供的库支持在 AXI4-Stream 级别进行连接，以模仿任意数据流传输 I/O，并支持 AXI3/AXI4 接口以模仿任意存储器映射 I/O。

用于数据流传输流量的 AXI4-Stream I/O 模型

以下章节主要围绕 AXI4-Stream 展开。数据流传输 I/O 模型可用于对平台上的数据流传输流量进行仿真，并且支持延迟建模。您可以将数据流传输 I/O 添加到自己的应用中，或者也可以将其添加到您的定制平台设计中，如下所述：

- 数据流传输 I/O 内核可像任何其它已编译的内核对象 (XO) 文件一样，使用 `v++ --link` 命令添加到器件二进制 (`xclbin`) 文件中。Vitis 安装为各种不同数据宽度的 AXI4-Stream 接口提供了内核。这些内核可在位于 `$XILINX_VITIS/data/emulation/XO` 处的软件安装中找到。

  您可使用以下命令示例将这些内核添加到自己的设计中：

  ```
  v++ -t hw_emu --link $XILINX_VITIS/data/emulation/XO/
  sim_ipc_axis_master_32.xo $XILINX_VITIS/data/emulation/XO/
  sim_ipc_axis_slave_32.xo ...
  ```

  在以上示例中，`sim_ipc_axis_master_32.xo` 和 `sim_ipc_axis_slave_32.xo` 提供了 32 位主内核与从内核，这些内核能够与目标平台和您的设计中的其它内核相链接，以创建 `.xclbin` 文件用于硬件仿真构建。

- 使用 Versal 和 Zynq UltraScale+ MPSoC 定制平台的 Vivado IP integrator 还可将 IPC 模块添加到平台块设计中。该工具提供了 `sim_ipc_axis_master_v1_0` 和 `sim_ipc_axis_slave_v1_0` IP 用于添加到您的平台设计中。这些内核可在位于 `$XILINX_VIVADO/data/emulation/hw_em/ip_repo` 处的软件安装中找到。

  以下提供的 Tcl 脚本示例可用于将 IPC IP 添加到您的平台设计中，这样您即可将来自以 Python 或 C++ 编写的外部进程的数据流量注入自己的仿真中。

  ```
  #Update IP Repository path if required
  set_property  ip_repo_paths $XILINX_VIVADO/data/emulation/hw_em/ip_repo
  [current_project]
  ## Add AXIS Master
  create_bd_cell -type ip -vlnv xilinx.com:ip:sim_ipc_axis_master:1.0
  sim_ipc_axis_master_0
  #Change Model Property if required
  set_property -dict [list CONFIG.C_M00_AXIS_TDATA_WIDTH {64}]
  [get_bd_cells sim_ipc_axis_master_0]

  ##Add AXIS Slave
  create_bd_cell -type ip -vlnv xilinx.com:ip:sim_ipc_axis_slave:1.0
  sim_ipc_axis_slave_0
  #Change Model Property if required
  set_property -dict [list CONFIG.C_S00_AXIS_TDATA_WIDTH {64}]
  [get_bd_cells sim_ipc_axis_slave_0]
  ```

以 Python 编写流量生成器

进行应用仿真时，您必须同时包含流量生成器进程，以在 I/O 流量生成器上生成数据流量或者捕获来自仿真进程的输出数据。赛灵思提供的 Python 或 C++ 库可用于创建流量生成器代码，如下所述。并且，每个应用均可与多个 I/O 接口进行通信。不必将 I/O 实用工具的每个实例单独包含在一个独立的进程/线程内。如果您的应用有此需求，您可以考虑采用非阻塞版本 API（下一章节中提供了相关详细信息）。

- 对于 Python，请在命令终端上设置 `$PYTHONPATH`：

  ```
  setenv PYTHONPATH $XILINX_VIVADO/data/emulation/hw_em/lib/python:\
  $XILINX_VIVADO/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/python/
  ```

- 用于与 `gt_master` 实例相连接的 Python 样本如下所示：

  ```
  Blocking Send
  from xilinx_xtlm import ipc_axis_master_util
  from xilinx_xtlm import xtlm_ipc
  import struct

  import binascii

  #Instantiating AXI Master Utilities
  master_util = ipc_axis_master_util("gt_master")
  ```

```
#Create payload
payload = xtlm_ipc.axi_stream_packet()
payload.data = "BINARY_DATA" # One way of getting "BINARY_DATA" from
integer can be like payload.data = bytes(bytearray(struct.pack("i",
int_number))) More info @ https://docs.python.org/3/library/struct.html
payload.tlast = True #AXI Stream Fields
#Optional AXI Stream Parameters
payload.tuser = "OPTIONAL_BINARY_DATA"
payload.tkeep = "OPTIONAL_BINARY_DATA"

#Send Transaction
master_util.b_transport(payload)
master_util.disconnect() #Disconnect connection between Python & Emulation
```

- 用于与 `gt_slave` 实例相连接的 Python 样本如下所示：

```
Blocking Receive
from xilinx_xtlm import ipc_axis_slave_util
from xilinx_xtlm import xtlm_ipc

#Instantiating AXI Slave Utilities
slave_util = ipc_axis_slave_util("gt_slave")


#Sample payload (Blocking Call)
payload = slave_util.sample_transaction()
slave_util.disconnect() #Disconnect connection between Python & Emulation
```

- 对于 Python 下的非阻塞版本 API，可在以下位置找到：

```
$XILINX_VIVADO/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/python/
xilinx_xtlm.py
```

以 C++ 编写流量生成器

- 对于 C++，API 可从以下位置获取：

```
$XILINX_VIVADO/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/cpp/inc/axis
```

C++ API 可提供阻塞和非阻塞函数支持。以下片段显示了其用法。

提示：此外还提供了一个 Makefile 样本用于生成可执行文件。

- 阻塞发送：

如果您倾向于不使用细粒度控制（建议），则可使用一个简单的 API：

```
#include "xtlm_ipc.h" //Include file
void send_data()
{
 //! Instantiate IPC socket with name matching in IPI diagram...
 xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::BLOCKING>
socket_util("gt_master");
 const unsigned int NUM_TRANSACTIONS = 8;
 std::vector<char> data;
 std::cout << "Sending " << NUM_TRANSACTIONS << " data transactions..."
<<std::endl;
 for(int i = 0; i < NUM_TRANSACTIONS; i++) {
```

```
 data = generate_data();
 print(data);
 socket_util.transport(data.data(), data.size());
 }
}
```

对于需要对 AXI4-Stream 进行细粒度控制的高级用户，可使用：

```
#include "xtlm_ipc.h" //Include file

void send_packets()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::BLOCKING>
socket_util("gt_master");

    const unsigned int NUM_TRANSACTIONS = 8;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Sending " << NUM_TRANSACTIONS << " Packets..."
<<std::endl;
    for(int i = 0; i < NUM_TRANSACTIONS; i++) {
        xtlm_ipc::axi_stream_packet packet;
        // generate_data() is your custom code to generate traffic
        std::vector<char> data = generate_data();
        //! Set packet attributes...
        packet.set_data(data.data(), data.size());
        packet.set_data_length(data.size());
        packet.set_tlast(1);
        //Additional AXIS attributes can be set if required
        socket_util.transport(packet); //Blocking transport API to send
the transaction
    }
}
```

- 阻塞接收：

  如果您倾向于不使用细粒度控制（建议），则可使用一个简单的 API：

```
#include "xtlm_ipc.h" //Include file
void receive_data()
{
 //! Instantiate IPC socket with name matching in IPI diagram...
 xtlm_ipc::axis_target_socket_util<xtlm_ipc::BLOCKING>
socket_util("gt_slave");
 const unsigned int NUM_TRANSACTIONS = 100;
 unsigned int num_received = 0;
 std::vector<char> data;
 std::cout << "Receiving " << NUM_TRANSACTIONS << " data transactions..."
<<std::endl;
 while(num_received < NUM_TRANSACTIONS) {
 socket_util.sample_transaction(data);
 print(data);
 num_received += 1;
 }
}
```

对于需要对 AXI4-Stream 进行细粒度控制的高级用户，可使用：

```
#include "xtlm_ipc.h"

void receive_packets()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_target_socket_util<xtlm_ipc::BLOCKING>
socket_util("gt_slave");

    const unsigned int NUM_TRANSACTIONS = 8;
    unsigned int num_received = 0;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Receiving " << NUM_TRANSACTIONS << " packets..."
<<std::endl;
    while(num_received < NUM_TRANSACTIONS) {
        socket_util.sample_transaction(packet); //API to sample the
transaction
        //Process the packet as per requirement.
        num_received += 1;
    }
}
```

- 非阻塞发送：

```
#include <algorithm>    // std::generate
#include "xtlm_ipc.h"

//A sample implementation of generating random data.
xtlm_ipc::axi_stream_packet generate_packet()
{
    xtlm_ipc::axi_stream_packet packet;
    // generate_data() is your custom code to generate traffic
    std::vector<char> data = generate_data();

    //! Set packet attributes...
    packet.set_data(data.data(), data.size());
    packet.set_data_length(data.size());
    packet.set_tlast(1);
    //packet.set_tlast(std::rand()%2);
    //! Option to set tuser tkeep optional attributes...

    return packet;
}
 //Simple Usage

void send_data()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::NON_BLOCKING>
socket_util("gt_master");

    const unsigned int NUM_TRANSACTIONS = 8;
    std::vector<char> data;

    std::cout << "Sending " << NUM_TRANSACTIONS << " data
transactions..." <<std::endl;
    for(int i = 0; i < NUM_TRANSACTIONS/2; i++) {
        data = generate_data();
        print(data);
        socket_util.transport(data.data(), data.size());
    }
```

```
    std::cout<< "Adding Barrier to complete all outstanding
transactions..." << std::endl;
    socket_util.barrier_wait();
    for(int i = NUM_TRANSACTIONS/2; i < NUM_TRANSACTIONS; i++) {
        data = generate_data();
        print(data);
        socket_util.transport(data.data(), data.size());
    }
}
void send_packets()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::NON_BLOCKING>
socket_util("gt_master");
    // Instantiate Non Blocking specialization

    const unsigned int NUM_TRANSACTIONS = 8;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Sending " << NUM_TRANSACTIONS << " Packets..."
<<std::endl;
    for(int i = 0; i < NUM_TRANSACTIONS; i++) {
        packet = generate_packet(); // Or user's test patter / live data
etc.
        socket_util.transport(packet);
    }
}
```

- 非阻塞接收：

```
#include <unistd.h>
#include "xtlm_ipc.h"
//Simple Usage
void receive_data()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_target_socket_util<xtlm_ipc::NON_BLOCKING>
socket_util("gt_slave");

    const unsigned int NUM_TRANSACTIONS = 8;
    unsigned int num_received = 0, num_outstanding = 0;
    std::vector<char> data;

    std::cout << "Receiving " << NUM_TRANSACTIONS << " data
transactions..." <<std::endl;
    while(num_received < NUM_TRANSACTIONS) {
        num_outstanding = socket_util.get_num_transactions();
        num_received += num_outstanding;

        if(num_outstanding != 0) {
            std::cout << "Outstanding data transactions = "<<
num_outstanding <<std::endl;
            for(int i = 0; i < num_outstanding; i++) {
                socket_util.sample_transaction(data);
                print(data);
            }
        }
        usleep(100000);
    }
}
void receive_packets()
{
```

```
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_target_socket_util<xtlm_ipc::NON_BLOCKING>
socket_util("gt_slave");

    const unsigned int NUM_TRANSACTIONS = 8;
    unsigned int num_received = 0, num_outstanding = 0;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Receiving " << NUM_TRANSACTIONS << " packets..."
<<std::endl;
    while(num_received < NUM_TRANSACTIONS) {
        num_outstanding = socket_util.get_num_transactions();
        num_received += num_outstanding;

        if(num_outstanding != 0) {
            std::cout << "Outstanding packets = "<< num_outstanding
<<std::endl;
            for(int i = 0; i < num_outstanding; i++) {
                socket_util.sample_transaction(packet);
                print(packet);
            }
        }
        usleep(100000); //As transaction is non-blocking we would like to
give some delay between consecutive samplings
    }
}
```

- 以下是用于上述阻塞接收的 Makefile 示例：

```
GCC=/usr/bin/g++
IPC_XTLM=$(XILINX_VIVADO)/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/
cpp/
PROTO_PATH=$(XILINX_VIVADO)/data/simmodels/xsim/2021.1/lnx64/6.2.0/ext/
protobuf/
BOOST=$(XILINX_VIVADO)/tps/boost_1_64_0/

SRC_FILE=b_receive.cpp
.PHONY: run all

default: all

all : b_receive

b_receive: $(SRC_FILE)
    $(GCC)   $(SRC_FILE) $(IPC_XTLM)/src/common/xtlm_ipc.pb.cc $
(IPC_XTLM)/src/axis/*.cpp $(IPC_XTLM)/src/common/*.cpp -I$(IPC_XTLM)/inc/
-I$(PROTO_PATH)/include/ -L$(PROTO_PATH) -lprotobuf -o $@ -lpthread -I$
(BOOST)/
```

- 对于 C API，可从以下位置获取此文件。

```
$XILINX_VIVADO/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/C/inc/axis/
c_axis_socket.h
```

可基于以下位置的预编译库来链接此文件：

```
$XILINX_VIVADO/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/C/lib/
```

在以下位置可找到完整的系统级示例：https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/emulation。

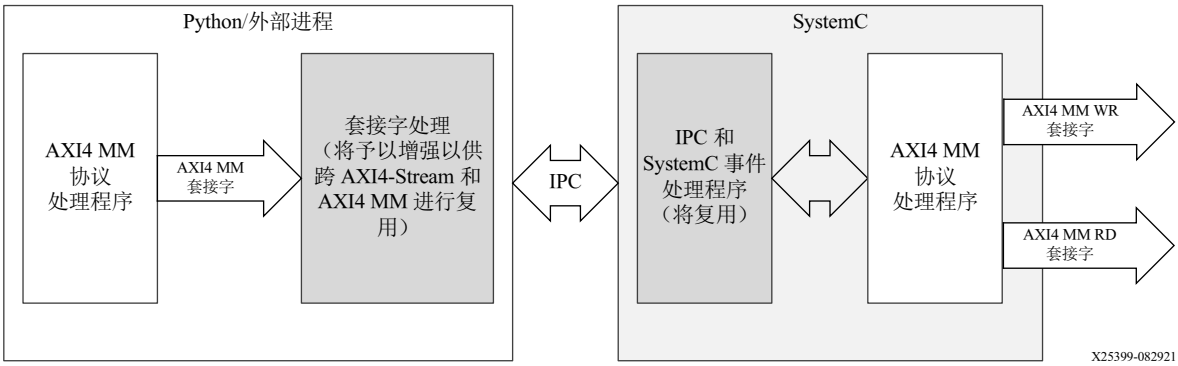
通过 Python/C++ 的 AXI4 存储器映射外部流量

AXI4 存储器映射外部流量具有如下规范：

- 仅支持传输事务级粒度。
- 不支持传输事务重新排序。
- 不支持并行读写传输事务（传输事务将序列化）。
- 不支持未对齐的传输事务。

下图显示了高层次设计。

图 26：**AXI4 存储器映射外部流量设计**



用例

以下提供了常见用例：

- 通过外部进程（例如，Python/C++）对 AXI4 存储器映射主/从接口进行仿真。这样有助于您借助 AXI4 主/从接口的快速设计能力来执行设计仿真，而无需投入资源用于 AXI4 主接口开发。
- 2 个 FPGA 之间的芯片到芯片 (Chip-to-Chip) 连接可通过 AXI4 存储器映射进程间通信来进行仿真。

API/伪代码

对于 API/伪代码，将 AXI4 存储器映射传输事务的单个实例用于整个传输事务。这与赛灵思 SystemC 模块中有效载荷的使用方式相符。对于 AXI4 主接口，可使用 `b_transport(aximm_packet)` API。完成调用后，将以 AXI4 从接口提供的响应来更新 `aximm_packet`。对于 AXI4 从接口，可使用 `sample_transaction()` 和 `send_response(aximm_packet)` API。

以下代码片段显示了 C++ 上下文中的 API 使用方式。

- C++ 主机的代码片段：

```
auto payload = generate_random_transaction(); //Custom Random transaction
generator. Users can configure AXI propeties on the payload.
/* Or User can set the AXI transaction properties as follows
payload->set_addr(std::rand() * 4);
payload->set_len(1 + (std::rand() % 255));
payload->set_size(1 << (std::rand() % 3));
*/
```

```
master_uti.b_transport(*payload.get(), std::rand() % 0x10); //A blocking
call. Response will be updated in the same payload. Each AXI MM
transaction will use same payload for whole transaction
std::cout << "-----------Transaction Response------------" << std::endl;
std::cout << *payload << std::endl; //Prints AXI transaction info
```

- C++ 从机的代码片段：

```
auto& payload = slave_util.sample_transaction(); // Sample the transaction

//If it is read transaction, give read data
if(payload.cmd() == xtlm_ipc::aximm_packet_command_READ)
{
    rd_resp.resize(payload.len()*payload.size());
    std::generate(rd_resp.begin(), rd_resp.end(), []()
    {   return std::rand()%0x100;});
}

//Set AXI response (for Read & Write)
payload.set_resp(std::rand()%4);
slave_util.send_response(payload); //Send the response to the master
```

以下代码片段显示了 Python 上下文中的 API 使用方式。

您需要按如下方式设置 PYTHONPATH：

- 例如，在 C Shell 上：

```
setenv PYTHONPATH $XILINX_VIVADO/data/emulation/hw_em/lib/python:
$XILINX_VIVADO/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/python
```

- Python 主机的代码片段：

```
aximm_payload = xtlm_ipc.aximm_packet()
random_packet(aximm_payload) # Custom function to set AXI Properties
randomly
#Or user can set AXI properties as required
#aximm_payload.addr = int(random.randint(0, 1000000)*4)
#aximm_payload.len = random.randint(1, 64)
#aximm_payload.size = 4

master_util.b_transport(aximm_payload)
#After this call aximm_payload will have updated response as set by the
AXI Slave.
```

- Python 从机的代码片段：

```
aximm_payload = slave_util.sample_transaction()
aximm_payload.resp = random.randint(0,3)
if not aximm_payload.cmd: #if it is a read transction set Random data
    tot_bytes = aximm_payload.len * aximm_payload.size
    for i in range(0, int(tot_bytes/SIZE_OF_EACH_DATA_IN_BYTES)):
        aximm_payload.data += bytes(bytearray(struct.pack(">I",
random.randint(0,60000)))) # Binary data should be aligned with C struct

slave_util.send_resp(aximm_payload)
```

平台中的 AXI4 存储器映射 I/O 限制

以下显示的是平台中的 AXI4 存储器映射 I/O 限制：

- 在平台开发期间，AXI4 存储器映射 I/O 可连接到任意主/从接口。
- 由于内核无法提供额外的从接口，因此，AXI4 存储器映射 I/O 主接口无法连接到内核。
- AXI4 存储器映射 I/O 从接口可供使用且无任何限制。
- 如需将数据从外部进程驱动到存储器/从接口，则可使用 AXI4 存储器映射 I/O 主接口。

XO 的使用

AXI4 存储器映射 I/O XO 的用例不同于 AXI4-Stream I/O XO 的用例。AXI4 存储器映射 XO 在 Vitis 的链接阶段具有下列使用限制：

- 只能使用 AXI4 存储器映射主接口 I/O。
- AXI4 存储器映射主接口 I/O 只能与平台中可用的从接口相连。
- AXI4 存储器映射主接口 I/O 不能与设计中的内核进行通信。

对于链接阶段的 XO 使用：

- 要生成 XO，开发者可以使用位于以下位置的脚本：`$XILINX_VITIS/data/emulation/XO/scripts/aximm_xo_creation.sh`
- 所需 XO 配置可使用以上脚本来生成。

```
$XILINX_VITIS/data/emulation/XO/scripts/aximm_xo_creation.sh --
address_width <adr_width> --data_width <data_width> --id_width <id_width>
--output_path <output_path>.xo
$XILINX_VITIS/data/emulation/XO/scripts/aximm_xo_creation.sh --
address_width 64 --data_width 64 --id_width 4 --output_path
sim_ipc_aximm_master.xo
```

- 生成 XO 后，可在设计中通过如下所示配置来使用（此配置仅为使用样本，实际连接应基于需求来完成）：

```
[connectivity]
nk=sim_ipc_aximm_master:1:aximm_master
sp=aximm_master.M_AXIMM:HBM[0]
```

运行流量生成器

如上所述使用 `$XILINX_VIVADO/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/<supported_language>` 中提供的头文件和源文件生成外部进程二进制文件后，您可使用以下步骤来运行仿真：

1. 使用标准流程启动 Vitis 硬件仿真或 Vivado 仿真并等待仿真启动。
2. 在另一个或多个终端上，启动外部进程，例如 Python/C++/C。

XILINX

第 17 章

运行应用硬件构建

运行应用硬件构建允许您查看加速器卡上运行的应用，例如，Alveo™ 数据中心加速器卡或者以 Versal™ ACAP 或 Zynq® 器件为目标的嵌入式处理器平台。此处捕获的性能数据和结果均为加速应用的实际性能。但此运行生成的剖析数据可能仍表示有机会进一步最优化设计。

提示：要使用加速器卡，必须按《Alveo 数据中心加速器卡入门指南》(UG1301) 中所述方式完成卡安装。

1. 按 xrt.ini 文件 中所述，编辑 `xrt.ini` 文件。

   这是可选操作，但在硬件上运行以进行仿真时建议执行此操作。您可通过 `xrt.ini` 文件来配置 XRT 以在应用运行时捕获调试和剖析数据。要在运行硬件时捕获事件追踪数据，请参阅 在应用中启用剖析。要调试运行硬件，请参阅 硬件执行期间的调试。

   提示：编译内核代码以进行调试时，请务必使用 `v++ -g` 选项。

2. 取消设置 `XCL_EMULATION_MODE` 环境变量。

   重要提示！如果 `XCL_EMULATION_MODE` 环境变量设置为仿真目标，那么硬件构建将无法运行。

3. 对于嵌入式平台，请启动 SD 卡。

   提示：仅限使用赛灵思嵌入式器件的平台（如，Versal ACAP 或 Zynq UltraScale+ MPSoC）才需要此步骤。

   对于嵌入式处理器平台，请将 `v++ --package` 命令生成的 `./sd_card` 文件夹内容复制到 SD 卡，作为系统的启动器件。从 SD 卡启动系统。

4. 运行应用。

   用于运行应用的具体命令行取决于您的主机代码。以下提供了赛灵思教程中使用的常用实现及示例：

   ```
   ./host.exe kernel.xclbin
   ```

提示：此命令行假定主机程序编写时采用 `xclbin` 文件的名称作为实参，就像大部分 Vitis 示例和教程一样。但您的应用可能已将 `xclbin` 文件的名称硬编码到主机程序中，或者可能要求采用其它方法来运行此应用。

第四部分

对应用进行剖析、最优化和调试

无论是在仿真中还是在系统硬件上运行系统，都会带来一系列潜在的挑战和机会。首次运行系统时，您可通过剖析应用来识别瓶颈或者性能问题，从中寻找机会对设计进行最优化，如下文所述。当然，运行应用也可能会遇到编码错误或设计错误，须加以调试才能使系统按期望方式正常运行。

本部分包含以下章节：

XILINX

第 18 章

剖析应用

Vitis™ 核开发套件会在编译期间生成各种系统和内核资源性能报告。这些报告可帮助您确立应用性能基线、识别瓶颈，并帮助识别可在硬件内核中加速的目标函数，如 器件加速型应用的架构方法论 中所述。赛灵思的 Xilinx Runtime (XRT) 可在执行应用期间收集仿真和硬件构建中的剖析数据。可报告的剖析和事件数据示例包括：

- 主机和器件时间线事件
- OpenCL™ 或 XRT 本机 API 调用序列
- 内核执行序列
- 内核开始和停止信号
- FPGA 追踪数据，包括 AXI 传输事务
- 加速器卡的功耗剖析数据
- AI 引擎剖析和事件追踪
- 用户事件和范围剖析

剖析报告和数据可用于确定应用中的性能瓶颈、识别系统中的问题以及最优化设计以改善性能。应用最优化需要同时对应用主机代码和任何硬件加速内核进行最优化。必须通过主机代码最优化来改善数据传输和内核执行，同时应对内核进行最优化以改善性能和资源利用率。

在 Vitis 中执行算法最优化时，有 4 个不同领域需要考量：系统资源利用率和性能、内核最优化、主机最优化以及数据传输最优化。以下 Vitis 报告和图形工具支持您对这些领域进行剖析和最优化：

- 指南
- 系统估算报告
- HLS 报告
- 剖析汇总报告
- 时间线轨迹
- 波形视图和实时波形查看器

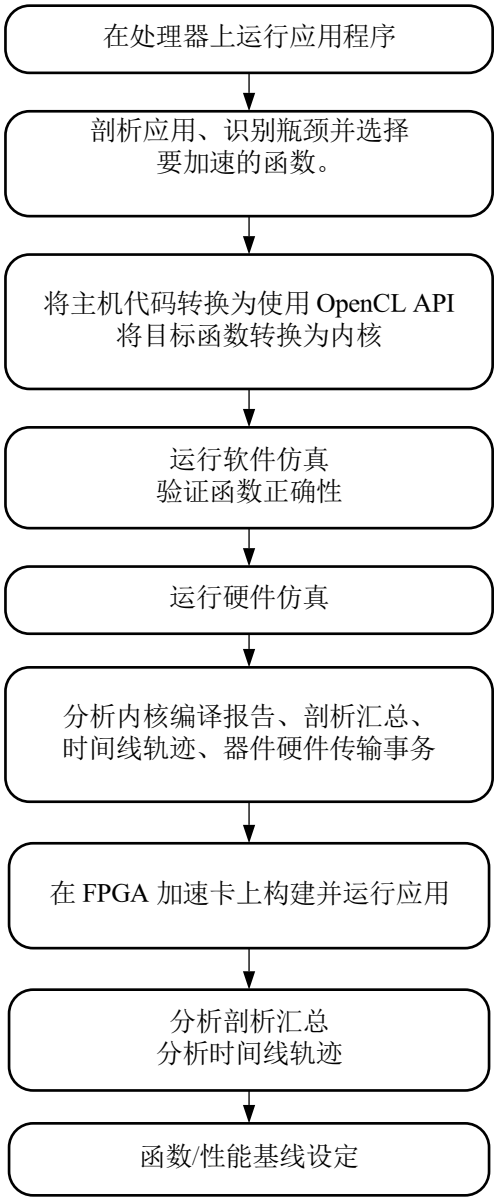
按 在应用中启用剖析 中所述方式正确启用后，即可在运行活动构建时通过命令行（如 第三部分：构建和运行应用 中所述）或者通过 Vitis 集成设计环境 (IDE) 来自动生成报告。针对不同构建目标将生成不同报告，并且可在相应的报告目录下找到这些报告。如需了解有关查找这些报告的更多信息，请参阅 v++ 命令的输出目录 或 Vitis IDE 的输出目录。

这些报告可在 Vitis 分析器中查看，或者在某些情况下也可通过 Vitis IDE 来查看。要从 Vitis 分析器访问这些报告，请打开 `run_summary` 报告，如 第六部分：使用 Vitis 分析器 中所述。

功能和性能基线设定

使用 Vitis 软件平台加速应用的方法论 提供了应用设计概述，从剖析应用以识别要加速的功能开始，直至建议的 C/C++ 加速器开发方法为止。正如本指南中所述，在开始任何最优化操作前，了解应用的架构和性能至关重要。这是通过对应用进行功能和性能的基线设定来实现的。

图 27：**功能和性能基线设定流程**



识别瓶颈

第一步是识别目标平台上正在运行的应用的瓶颈。最有效的方法是通过各种剖析工具来运行应用，此类工具包括 主机应用的定制剖析 中所述的用户剖析功能或者 `valgrind`、`callgrind` 和 GNU `gprof` 等。这些工具生成的剖析数据可显示调用图形，包括所有函数的调用次数及其执行时间。

运行软件和硬件仿真

按 运行仿真 中所述方式对加速应用运行软件和硬件仿真以验证功能正确性并生成有关主机代码和内核的剖析数据。使用 Vitis 分析器复查内核编译报告、剖析汇总信息、时间线轨迹和器件硬件传输事务，以了解时序间隔、时延和资源利用率的基线性能估算（如 DSP 和块 RAM）。

构建和运行应用

基线设定的最后一步是在 Alveo™ 数据中心加速器卡之类的 FPGA 加速卡上构建和运行应用，如 运行应用硬件构建 中所述。分析系统编译所生成的报告以及来自应用执行的剖析数据，以了解硬件上的实际性能和资源利用率。

提示：在基线设定过程中保存所有报告，以便您在最优化期间复查这些报告并比较结果。

在应用中启用剖析

要在应用执行期间启用剖析和捕获事件追踪数据，您必须指令自己的应用执行此任务。您必须启用额外逻辑、耗用额外的器件资源来跟踪主机和内核执行步骤，并捕获事件数据。（可选）此进程要求修改您的主机应用，以捕获定制数据、在编译期间修改内核 XO 并在链接期间修改 `xclbin` 以从器件侧活动中捕获不同类型的剖析数据，并按 xrt.ini 文件 中所述方式配置赛灵思的 Xilinx Runtime (XRT) 以在应用运行时期间捕获数据。

提示：虽然捕获剖析数据是剖析和最优化进程中用于构建加速应用的关键一环，但它确实会耗用额外的资源，从而对性能产生不利影响。您应确保从最终量产构建中清除掉这些元素。

根据系统所包含的元素以及要捕获的数据类型，对于应用而言，有多种不同类型的剖析可供使用。下表显示了可启用的部分剖析级别，并探讨了各自的实用性。

表 16：**主机和内核剖析**

剖析/轨迹	描述	评述
主机应用 OpenCL API 和部分受限的器件侧（内核）剖析。	通过在 `xrt.ini` 文件中使用 `opencl_summary` 和 `opencl_trace` 选项来指定。	生成 `opencl_summary.csv` 和 `opencl_trace.csv` 文件。
主机应用 XRT 本机 API	通过在 `xrt.ini` 文件中使用 `native_xrt_trace` 选项来指定。	为 XRT API 生成追踪事件。
主机应用用户事件剖析	要求在主机应用中添加额外代码，如 主机应用的定制剖析 中所述。	为主机应用生成用户范围数据和用户事件。
低开销剖析	通过在 `xrt.ini` 文件中使用 `lop_trace` 选项来指定。	生成 `lop_trace.csv` 文件，如 启用低开销剖析 中所述。 通过在 `xrt.ini` 文件中设置 `opencl_summary=true` 来禁用。
器件侧剖析	通过在 `v++` 编译和链接期间使用 `--profile` 选项来启用，如 --profile 选项 中所述。	启用对主机与内核之间的数据流量、内核停滞、内核与计算单元 (CU) 的执行时间以及 Versal AI 引擎中的监控活动进行捕获。

表 16：**主机和内核剖析** (续)

剖析/轨迹	描述	评述
AI 引擎 Graph 和内核	通过在 `xrt.ini` 文件中使用 `aie_profile` 和 `aie_trace` 选项来指定。这些选项可以结合在一起指定，也可以各自单独指定。	生成 `aie_profile_<device>.csv` 和 `aie_trace_##_<stream id>.txt` 报告。 无法在 `xrt.ini` 文件中配合 `profile=true` 一起使用。 在主机应用中如果存在用户事件剖析，则同样会禁用此项。
功耗剖析	通过在 `xrt.ini` 文件中使用 `power_profile` 选项来指定。	生成 `power_profile_<device>.csv` 报告。
Vitis AI 剖析	通过在 `xrt.ini` 文件中使用 `vitis_ai_profile` 选项来指定。	启用 DPU 计数器剖析，以生成 `opencl_summary.csv` 文件。 通过在 `xrt.ini` 文件中设置 `opencl_summary=true` 来禁用。

默认情况下，器件二进制文件 (`xclbin`) 配置为捕获有限的器件侧剖析数据。但在 Vitis 编译器链接进程中使用 `--profile` 选项则会对器件二进制文件进行检测，方法是在系统中添加 Acceleration Monitor 和 AXI Performance Monitor。该选项还具有多个检测选项：`--profile.data`、`--profile.stall` 和 `--profile.exec`，如 --profile 选项 中所述。

例如，请将 `--profile.data` 添加到 `v++` 链接命令行中：

```
v++ -g -l --profile.data all:all:all ...
```

提示：编译内核代码用于调试软件或硬件仿真 (emulation) 时，请务必一并使用 `v++ -g` 选项。

在 `v++` 编译和链接进程中启用应用剖析后，还必须通过编辑 `xrt.ini` 文件，在 XRT 中启用应用运行时期间的数据收集，如上文所述。例如，以下 `xrt.ini` 文件支持在运行应用时进行 OpenCL 剖析、功耗剖析以及事件和停滞追踪捕获。

```
[Debug]
opencl_summary=true
opencl_trace=true
power_profile=true
data_transfer_trace=coarse
stall_trace=all
```

要启用内核内部数据剖析，还必须在 `xrt.ini` 的 `[Emulation]` 部分中添加 `debug_mode` 标签：

```
[Emulation]
debug_mode=batch
```

如果要收集大量追踪数据，可以通过在 `v++` 链接期间指定 `--trace_memory` 选项并在 `xrt.ini` 文件中添加 `trace_buffer_size` 关键字来增加可用于捕获数据的存储空间量。

- `--trace_memory`：表示用于捕获追踪数据的存储器类型，如 Vitis 编译器常规选项 中所述。
- `trace_buffer_size`：指定应用运行时期间用于捕获追踪数据的存储空间量。

最后，如 连续追踪捕获 中所述，您可以启用连续追踪捕获以在运行应用时连续卸载器件追踪数据，以便在发生应用或系统崩溃的情况下，有部分追踪数据可用于帮助调试应用。

连续追踪捕获

Vitis 工具支持在运行应用时记录连续追踪数据。应用可能因长时间运行而导致捕获大量追踪数据，这可能导致诸如追踪数据不完整等问题，当用于追踪数据的存储器资源不足时更是如此。通过使用连续追踪，即可在应用仍在运行时或者在应用完成前即崩溃的情况下执行追踪分析。

借助连续捕获追踪数据的功能，在 Vitis 分析器工具中即可在运行应用的同时动态更新“时间线轨迹 (Timeline Trace)”报告和“应用时间线 (Application Timeline)”报告。在 Vitis 分析器中加载这些报告后，有一个超链接可用于表示当前报告正在磁盘上进行修改。如需加载新数据，功能区上的“Reload”或“Auto-Reload”选项可支持您在运行应用并生成追踪数据的同时查看更新的报告。

默认情况下，不启用连续追踪。此外，FPGA 的存储器资源并非无限。因此，如果应用生成大量追踪数据，可使用圆形缓冲器来存储数据。圆形缓冲器可写入、下载至主机并可复用。通过为连续追踪启用圆形缓冲器，可进一步减少所需的存储器资源，从而节省器件上的可用资源。但应用随连续追踪/圆形缓冲器一起运行可能导致生成多个设备树文件。

提示：对于硬件仿真，仅限主机侧连续追踪可用，而对于硬件运行，主机侧和器件侧连续追踪均可用。

在如下情况下，建议使用存储器资源作为圆形缓冲器。

在 `xrt.ini` 中启用连续追踪时，自动启用圆形缓冲器实现。此流程需如下设置才能启用连续追踪。

- 在 `xrt.ini` 文件中，`continuous_trace` 设置为 TRUE
- v++ 链接选项 `--trace_memory` 设置为 DDR 或 HBM

您可选择如下设置：

- 在 `xrt.ini` 文件中使用 `trace_buffer_size` 来设置追踪缓冲器的大小。此大小默认值为 1 MB。
- 在 `xrt.ini` 文件中使用 `trace_buffer_offload_interval_ms` 来设置从器件卸载追踪缓冲器的时间间隔。默认值为 10 ms。
- 通过设置 `trace_file_dump_interval_s` 来设置文件转储的时间间隔。默认值为 3 秒。

重要提示！可通过将 `trace_buffer_offload_interval_ms` 设置为 0 ms 来强制启用圆形缓冲器。

例如，如果您启用 `continuous_trace` 并将 `trace_buffer_size` 设为 8k，`trace_buffer_offload_interval_ms` 采用默认值 10 ms，那么追踪数据速率为每秒 819200 字节，小于默认的 100 MB/s。在此情况下，圆形缓冲器默认不启用，并报告 XRT 警告：

```
[XRT] WARNING: Unable to use circular buffer for continuous trace offload.
Please increase trace buffer size and/or reduce continuous
trace interval. Minimum required offload rate (bytes per second) :
104857600 Requested offload rate : 819200
```

以下是 `xrt.ini` 设置示例：

```
[Debug]
opencl_summary=true
opencl_trace=true
data_transfer_trace=coarse
stall_trace=all
continuous_trace=true
```

```
// The following are optional and needed only in rare circumstances

trace_buffer_size=20M
trace_buffer_offload_interval_ms=10
trace_file_dump_interval_s=2
```

以下是上述设置的结果：

- `opencl_summary`：启用生成主机相关 OpenCL API 剖析汇总报告，创建 `opencl_summary.csv` 文件。
- `opencl_trace`：启用生成主机相关 OpenCL API 追踪，创建 `opencl_trace.csv` 文件。
- `data_transfer_trace`：启用收集内核活动，这些活动将被添加到剖析汇总和轨迹报告中，并创建 `device_trace_0.csv` 文件（器件编号为 0）。
- `stall_trace`：启用硬件在计算单元中生成停滞。
- `continuous_trace`：启用将器件数据的追踪和连续读取对应的文件连续转储到主机中。
- `trace_buffer_offload_interval_ms`：控制从器件到主机的器件数据读取（以毫秒为单位）。
- `trace_file_dump_interval_s`：控制追踪文件转储的时间间隔（以秒为单位）。

由此导致在运行应用时，使用以上 `xrt.ini` 文件不仅可以生成 `xclbin.run_summary`，还可生成多个 CSV 文件。Vitis 分析器只需生成的 `run_summary` 文件即可，并且将使用相关 CSV 文件来显示剖析汇总和时间线轨迹信息。

以下是有关设置应用以进行追踪数据转储的一些建议：

1. 默认情况下，使用 8k FIFO 来保存追踪数据。FIFO 大小可以增加，但不建议超过 64k，并且需在 v++ 链接步骤中进行预分配。此外，最好使用器件存储器来保存追踪数据。如果您为追踪指定存储体 (memory bank)，那么您可在 `xrt.ini` 中使用 `trace_buffer_size` 选项来控制在运行时生成的追踪量。对于器件存储器，默认大小为 1M，最大大小为 4095M。
2. 如果仍无法转储最大追踪数据，则可通过设置 `stall_trace=off` 或 `stall_trace=on` 和 `data_transfer_trace=coarse` 来禁用停滞追踪。
3. 如果应用需要更大的追踪缓冲器大小，则可通过设置 `continuous_trace=true` 并采用默认设置 `trace_buffer_offload_interval_ms=10` 和 `trace_file_dump_interval_s=5` 来启用圆形缓冲器。理想情况下，连续追踪功能应在下列情况下使用：
   - 设计长时间运行并生成少量追踪数据。
   - 调试应用时发生崩溃，其中部分 `.csv` 文件仍可用于调试。
4. 如果应用运行仍无法转储最大追踪，可进一步增加 `trace_buffer_size`。
5. 如果应用仍创建海量追踪数据，导致主机无法满足需求，请减小所用的 `trace_file_dump_interval` 大小，这样会创建等同于所提供的时间间隔的多个文件。
6. 最后，在运行应用时，连续追踪不仅会创建 `xclbin.run_summary` 文件，还会创建多个追踪文件。Vitis 分析器只需生成的 `run_summary` 文件即可，它可选取生成的相关 CSV 文件来显示剖析汇总和时间线轨迹信息，以提供更好的体验。

主机应用的定制剖析

来自主机应用的所有 XRT 相关操作均通过 OpenCL API 调用或 XRT API 调用来进行自动追踪以供剖析之用。但您也可以对超出 XRT 相关事件范围的主机应用进行剖析，即，基于用户指定的操作或事件来捕获事件数据。

提示：您可在设计进程早期使用这些功能（如 功能和性能基线设定 中所述），甚至可在拆分函数以供在赛灵思器件硬件中运行之前使用这些功能。

该功能可提供两种类型的定制剖析：

- 用户范围：对某一范围内指定的代码开始/结束时间进行剖析。这样即可捕获主机应用中发生某项操作的时间段。
- 用户事件：在时间线中标记某一事件。在发生用户事件的任意时间点，将其添加到时间线波形中。

user_range 和 user_event 数据可捕获到“Profile Summary”和“Timeline Trace”报告中，以供显示在 Vitis 分析器中。如下图所示，“Profile Summary”显示了给定事件的发生次数和范围。“用户范围 (User Ranges)”表还可报告主机代码中用户定义的范围的最小 (Min)/最大 (Max)/平均 (Avg)/总计 (Total) 持续时间。在“Timeline Trace”报告中，主机代码中的 user_range 元素单独显示在一行中，user_event 标记则被添加到时间线上的各特定时间点上。

图 28：**剖析汇总 - 用户范围**

使用定制剖析需要对主机应用源代码和构建进程进行些许更改。您必须在代码中使用 C 或 C++ API（如下所述），并且链接主机应用时，必须包含 xrt_coreutil 库。

- C/C++ API 如下所述，但也可通过以下 URL 获取：https://github.com/Xilinx/XRT/blob/master/src/runtime_src/core/include/experimental/xrt_profile.h。
- 对于 C 和 C++，您必须添加：

```
#include experimental/xrt_profile.h
```

- 链接主机代码时，请将 -lxrt_coreutil 添加到编译器命令行中。

提示：如需获取 user_range 和 user_event 的示例，请参阅以下主机代码：https://github.com/Xilinx/Vitis_Accel_Examples/blob/master/host/debug_profile/src/host.cpp。

C++ 代码剖析

对于 C++ 代码，提供的对象包括：

- user_range：此对象可捕获含指定 ID 的活动的测量范围的开始时间和结束时间。对象构造函数为：

```
user_range(const char* label, const char* tooltip);
```

- user_event：此对象用于标记在单一时间点发生的事件，并在时间线轨迹上添加指定标签。对象构造函数为：

```
user_event()
```

user_range 用于构造对象，并在开始构造时立即开始记录时间。user_range 对象的使用详情：

- 如果使用默认构造函数来例化 `user_range`，则从用户以标签和工具提示来调用 `user_range.start()` 开始，才会标记时间。
- 您可将传递标签和工具提示字符串的 `user_range` 对象例化。这样即可立即开始监控时间范围。
- 您必须调用 `user_range.start()` 和 `user_range.end()` 以捕获您感兴趣的时间范围。
- 如果不调用 `user_range.end()`，那么对任何范围进行的跟踪都将持续至 `user_range` 对象被析构为止。
- 通过调用主机代码中的 `user_range.start()`/`user_range.end()` 对，即可复用 `user_range` 对象，复用次数无限。
- 对 `user_range.start()` 执行顺序调用会忽略除第一次调用外的所有调用，直至 `user_range.end()` 终止此范围为止。
- 对 `user_range.end()` 执行顺序调用会忽略除第一次调用外的所有调用，直至 `user_range.start()` 开始新范围为止。

`user_event` 对象的使用：

- `user_event` 对象必须使用默认构造函数进行例化。
- 调用 `user_event.mark()` 会在时间线轨迹上的该特定时间创建用户标记。
- `user_event.mark()` 可接受可选 `const char*` 实参，该实参在时间线轨迹上显示为标签。

Vitis_Accel_Examples 的 debug_profile 示例演示了主机应用中的用户事件剖析。XRT 正确检测您的主机应用后，即可从这些用户定义的范围和事件以及基于 XRT API 的标准事件中捕获剖析数据。您必须在 `xrt.ini` 文件中启用剖析，如前所述。

C 语言代码剖析

对于 C 语言代码，提供的函数包括：

- `xrtURStart()`：此函数用于确定含指定 ID 的活动的测量范围的开始时间。函数特征符为：

```
void xrtURStart(unsigned int id, const char* label, const char* tooltip)
```

- `xrtUREnd()`：此函数用于标记含指定 ID 的活动的测量范围的结束时间。函数特征符为：

```
void xrtUREnd(unsigned int id)
```

- `xrtUEMark()`：此函数用于标记在单一时间点发生的事件，并在时间线轨迹上添加指定标签。函数特征符为：

```
void xrtUEMark(const char* label)
```

`xrtURStart()` 和 `xrtUREnd()` 函数可用于立即开始记录时间、指定 ID 以与开始/结束调用配对，并定义用户范围。`user_range` 函数的使用详情：

- 任一 ID 的开始/结束范围都可嵌套在其它 ID 的其它开始/结束范围内。
- 您自行负责确保您要剖析的开始/结束范围的各 ID 匹配。

> **重要提示！**使用同一个 ID 多次调用 `xrtURStart` 和 `xrtUREnd` 可能导致出现意外行为。

- 在时间线上可以为用户范围添加一个标签，当您将光标置于此用户范围上时，可显示工具提示。

调用 `xrtUEMark()` 将在时间线轨迹上事件发生的时间点创建一个用户标记。

- `xrtUEMark()` 允许您为该事件指定标签。此标签将随标记一起显示在时间线上。
- 您可以为此标签使用 `NULL` 来添加无标签的标记。

代码示例如下所示：

```
int main(int argc, char* argv[]) {
 58
 59   xrtURStart(0, "Software execution", "Whole program execution") ;
 60 ...
 61   //TARGET_DEVICE macro needs to be passed from gcc command line
 62   if(argc != 2) {
 63       std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
 64       return EXIT_FAILURE;
 65    }
....
153     q.enqueueTask(krnl_vector_add);
154
155     // The result of the previous kernel execution will need to be
retrieved in
156     // order to view the results. This call will transfer the data from
FPGA to
157     // source_results vector
158
q.enqueueMigrateMemObjects({buffer_result},CL_MIGRATE_MEM_OBJECT_HOST);
159 ····
160     q.finish();
161
162     xrtUEMark("Starting verification") ;
163
```

启用低开销剖析

Vitis 软件平台支持低开销剖析，此类剖析可提供最少量的信息且对执行时间影响最小。在运行时期间使用该选项时，时间线轨迹仍可用，但信息量会减少。低开销剖析会捕获有关 OpenCL 事件的最少量信息，并在执行结束时转储一个 CSV 文件，名为 `lop_trace.csv`。低开销剖析可在全部 3 个流程（硬件流程、硬件仿真流程和软件仿真流程）中运行。

要启用低开销剖析，可使用 xrt.ini 文件 的“Debug”部分中名为 `lop_trace` 的新标记。默认情况下，`lop_trace` 设为 FALSE，必须通过将 `ini` 参数设为 TRUE 才能启用。

```
xrt.ini file
[Debug]
lop_trace=true
```

提示： `lop_trace` 参数可随其它剖析参数一起启用，但这样也会捕获所有剖析数据，从而使低开销剖析的优势荡然无存。

启用 `lop_trace=true` 时，运行时将生成 `lop_trace.csv`，此文件可在 Vitis 分析器的“Run Summary”中查看。

```
vitis_analyzer <project>.run_summary
```

为尽可能降低开销，正常 OpenCL 剖析中收集的信息将省略。具体来说，在低开销剖析追踪中，预计下列信息不可用：

- 器件事件，例如，计算单元执行或内核存储器传输
- 有关存储器读写的信息，例如，目标地址或大小

- 有关内核队列的信息，例如，内核名称或 NDRange 大小
- 缓冲器传输与内核队列之间的依赖关系

指南

Vitis 核开发套件具有全面的设计指南工具，可即时提供切实可行的指导信息，以帮助软件开发者解决设计中检测到的问题。这些问题可能与源代码相关，或者也可能是由于未进行工具最优化而导致的。此外，其中所含规则均为基于广泛的参考设计制定的通用规则。因此，这些规则可能不适用于您的特定设计。您应自行理解特定指南规则，并根据您自己的具体算法和要求来采取相应的行动。

这些指南是从 `v++` 编译器调用的 Vitis HLS、Vitis Profiler 和 Vivado Design Suite 生成的。生成的设计指南可能包含多个严重性级别，警告消息、参考消息和设计规则检查是在软件仿真、硬件仿真和系统构建期间提供的。剖析设计指南可帮助您解读剖析结果，从而使您能够专注于提高性能。

指南包含有关已报告的违例的消息文本、简要的建议解决方案以及以 Web 链接形式提供的详细解决方案。您可以根据建议的解决方案来确定自己接下来的行动方针。它会快速高亮问题并指引您获取有关使用 Vitis 技术的更多信息，从而帮助您提升工作效率。

设计指南是在通过命令行或 Vitis IDE 来构建或运行应用之后自动生成的。

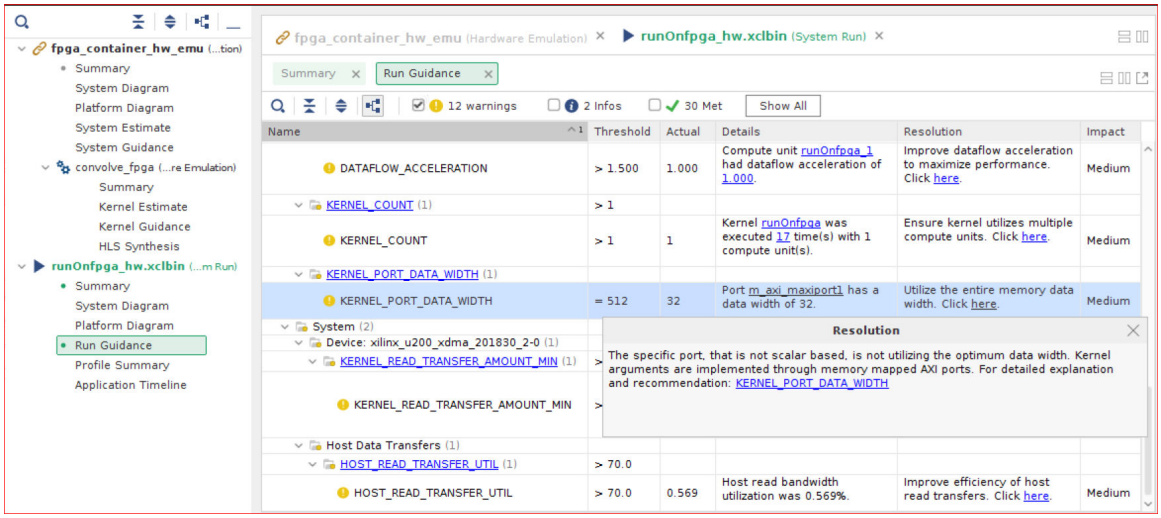
您可按 第六部分：使用 Vitis 分析器 中所述方式打开“指南 (Guidance)”报告。要访问“Guidance”报告，请打开“编译汇总 (Compile Summary)”、“链接汇总 (Link Summary)”或“运行汇总 (Run Summary)”，然后打开“Guidance”报告。

- “内核指南 (Kernel Guidance)”是在使用 `v++` 编译命令构建内核之后，由 Vitis HLS 工具生成的。可在 Vitis 分析器中通过打开“Compile Summary”报告来查看此报告。针对编译的每个内核，都会生成内核指南和编译汇总文件。内核指南包含有关使用数据流的建议以及无法实现期望的吞吐量的可能原因。
- “系统指南 (System Guidance)”是在使用 `v++` 链接命令构建内核之后生成的。可在 Vitis 分析器中通过打开“Link Summary”报告来查看此报告。系统指南包含所有内核指南检查，并提供了应用运行前的全方位审查结果。
- “运行指南 (Run Guidance)”是在运行所生成的 `.xclbin` 时生成的，它属于 XRT 的一项功能特性。可在 Vitis 分析器中通过打开“Run Summary”来查看此报告。运行指南包含“内核停滞 (Kernel Stall)”是否超过 50% 之类的检查以及是否可以使用 PLRAM 代替 DDR 等建议。

打开“Guidance”报告后，“Guidance”视图会显示消息列和解决方案列。解决方案还可包含展开的 Web 链接帮助。

下图显示了 Vitis 分析器中显示的“Guidance”报告示例。例如，单击“名称 (Name)”列中的链接即可打开规则检查描述。“详情 (Details)”列中的链接可打开源代码、选择设计对象（例如，内核）或者浏览其它报告。

图 29：**设计指南示例**



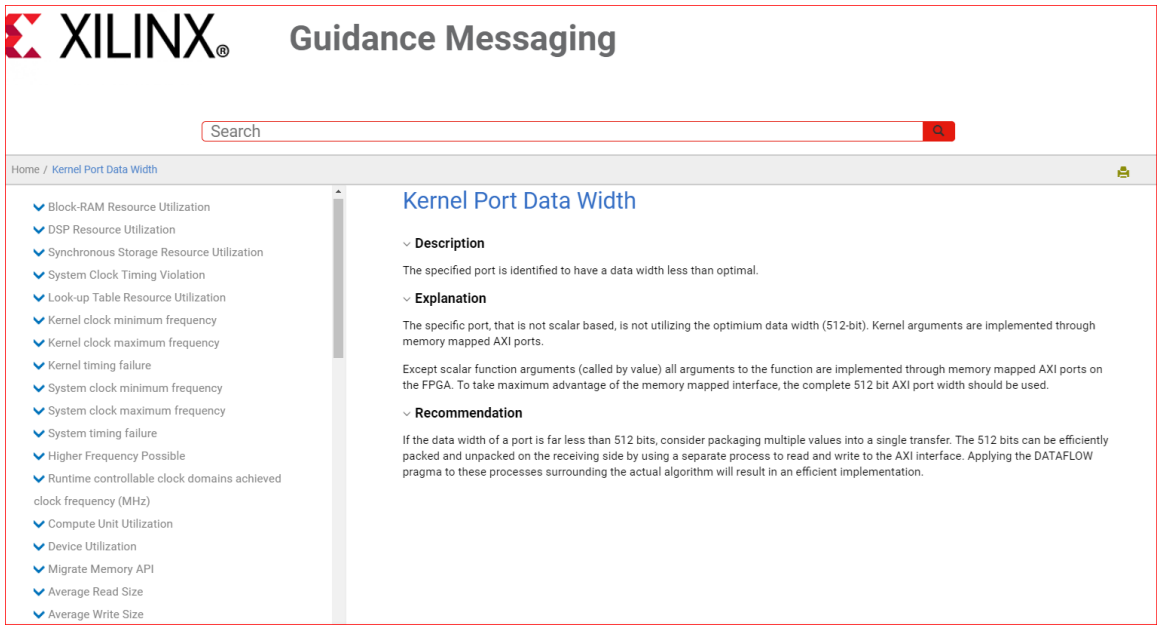
提示：如 设置指南阈值 中所述，您可以手动编辑“Run Guidance”报告的“Threshold”列中的值以自定义此报告。

针对运行的每条 `v++` 命令（包括编译和链接），都会生成一份 HTML 指南报告。报告文件位于 `--report_dir` 中的特定输出名称下。例如：

- `v++_compile_<output>_guidance.html` 对应 `v++` 编译
- `v++_link_<output>_guidance.html` 对应 `v++` 链接

您可单击“解决方案 (Resolution)”列中的 Web 链接以获取有关解决方案的更多详细信息。Guidance Messaging 网页列出了所有当前消息以供您复查。

图 30：**“Guidance Messaging”网页**



内核和计算单元对象以及剖析报告的数据值还可以与其它视图（例如，“系统框图 (System Diagram)”或“剖析报告 (Profile Report)”）进行交叉探测。如需了解更多信息，请参阅 使用报告。

打开指南报告

当内核完成编译并且 FPGA 二进制文件完成链接后，`v++` 命令会自动生成指南报告。您可在 Vitis 分析器中查看这些报告，方法是打开应用工程的 `<output_filename>.compile_summary` 或 `<output_filename>.link_summary`。`<output_filename>` 是 `v++` 命令的输出。

例如，启动 Vitis 分析器并使用以下命令打开报告：

```
vitis_analyzer <output_filename>.link_summary
```

打开 Vitis 分析器时，它会显示链接汇总报告、编译汇总以及编译和链接进程中生成的各种报告的集合。编译和链接步骤都会生成指南报告，可单击左侧“Build”标题查看这些报告。如需了解更多信息，请参阅 第六部分：使用 Vitis 分析器。

解读指南数据

“Guidance”视图将每个输入项放置在独立的行中。每行可能含有指南规则的名称、阈值、实际值以及该规则的简要而具体的描述。最后一个字段提供了指向参考资料的链接，这些参考材料的目的是为了帮助理解和解决任何规则违例。

在 GUI 的“Guidance”视图中，指南规则在“名称 (Name)”列中按类别和唯一 ID 进行分组，并带有表示严重性的符号注释。这些规则在 HTML 报告内单独列出。此外，由于 HTML 报告不显示工具提示，因此 HTML 报告中还包含一个“全名 (Full Name)”列。

以下列表描述了 HTML 指南报告中包含的所有字段及其目的。

- “Id”：为每条指南规则分配一个唯一 ID。此 ID 可用于唯一标识指南报告中的特定消息。
- “Name”：“名称 (Name)”列显示了一个助记符名称，用于唯一标识该指南规则。这些名称旨在帮助用户记住视图中的特定指南规则。
- “Severity”：“严重性 (Severity)”列可便于识别指南规则的重要性。
- “Full Name”：“全名 (Full Name)”提供了比“Name”列中的助记符名称更易于辨识的名称。
- “Categories”：大部分消息已按不同类别进行分组。这样 GUI 即可在“Guidance”视图中的通用树节点下的逻辑类别中显示消息组。
- “Threshold”：“阈值 (Threshold)”列显示了期望的阈值，该阈值用于判定是否满足规则。该阈值是基于诸多遵循良好的设计和编码实践的应用来决定的。
- “Actual”：“实际值 (Actual)”列显示具体设计上实际遇到的值。此值将与期望值做比较以确定是否满足规则。
- “Details”：“详情 (Details)”列提供了一条简要消息，用于描述当前规则的具体信息。
- “Resolution”：“解决方法 (Resolution)”列提供了一个指向常用方法的指针，用户可根据这些常用方法修改模型源代码或工具变换以满足当前规则。单击此链接会显示一个弹出窗口或文档，其中包含可供您应用于具体问题的提示信息和代码片段。

系统估算报告

执行时间最长的进程步骤包括构建硬件系统和 FPGA 二进制文件以在赛灵思器件上运行。构建时间还受到目标器件和例化到 FPGA 互连结构上的计算单元数量的影响。因此，无需为系统硬件构建应用即可估算应用性能的能力是很有用的。

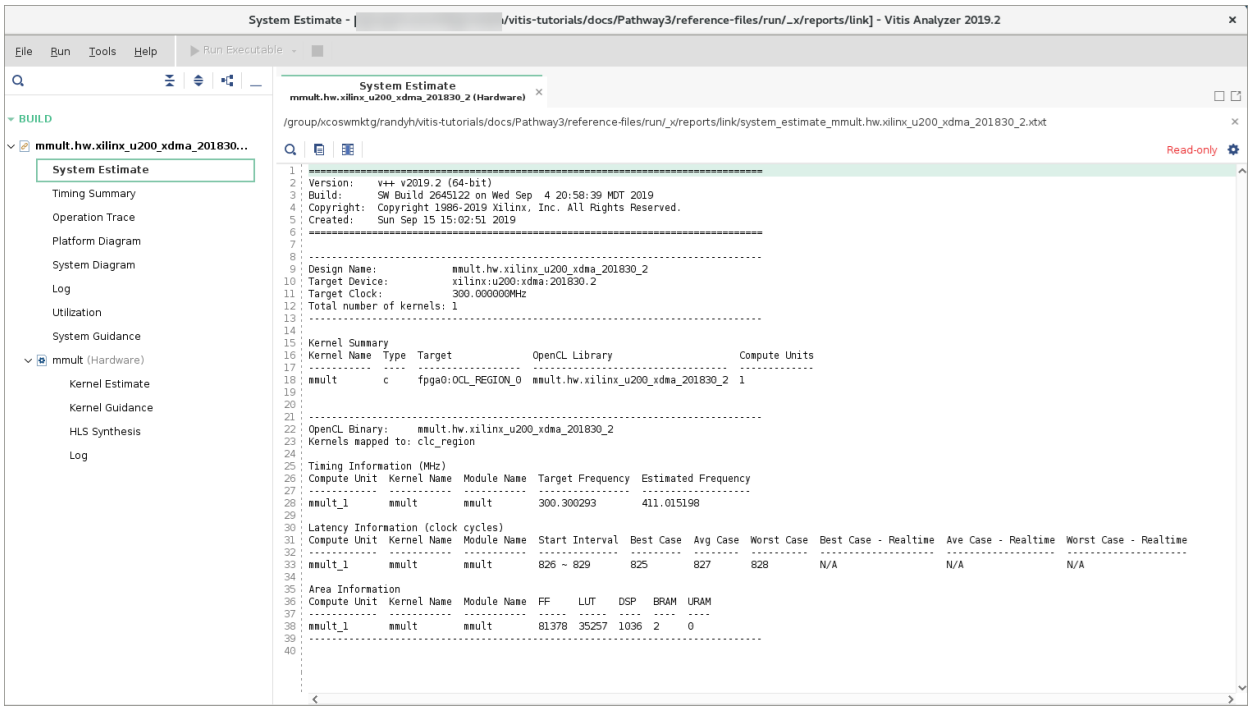
“系统估算 (System Estimate)”报告可提供有关 FPGA 资源使用情况的估算结果以及硬件加速内核可运行的估算频率信息。对于硬件仿真和系统硬件构建，会自动生成此报告。此报告包含用户内核的高层次详细信息，包括资源使用情况和估算的频率。此报告可用于指导设计最优化。

您也可以通过以下选项来强制生成“System Estimate”报告：

```
v++ .. --report_level estimate
```

报告示例如下图所示：

图 31：**系统估算**



打开系统估算报告

“系统估算 (System Estimate)”报告可在 Vitis 分析器工具中打开，在构建应用时，此工具可用于查看 Vitis 编译器生成的报告，在运行应用时，则可用于查看 XRT 库生成的报告。您可以使用以下命令启动 Vitis 分析器并打开此报告：

```
vitis_analyzer <output_filename>.link_summary
```

`<output_filename>` 是 `v++` 命令的输出。这样即可在 Vitis 分析器工具中打开应用工程的“链接汇总 (Link Summary)”。随后即可选择“System Estimate”报告。如需了解更多信息，请参阅 第六部分：使用 Vitis 分析器。

解读系统估算报告

由 v++ 命令所生成的“系统估算 (System Estimate)”报告可以提供有关应用中每个二进制容器的信息，以及有关设计中每个计算单元的信息。该报告的结构如下：

- 目标器件信息
- 应用中每个内核的汇总信息
- 有关解决方案中每个二进制容器的详细信息

以下报告文件示例提供了针对估算报告生成的信息：

```
--------------------------------------------------------------------------------
Design Name:             mmult.hw_emu.xilinx_u200_xdma_201830_2
Target Device:           xilinx:u200:xdma:201830.2
Target Clock:            300.000000MHz
Total number of kernels: 1
--------------------------------------------------------------------------------

Kernel Summary
Kernel Name  Type  Target              OpenCL Library                            Compute Units
-----------  ----  ------------------  ----------------------------------------  -------------
mmult        c     fpga0:OCL_REGION_0  mmult.hw_emu.xilinx_u200_xdma_201830_2  1


--------------------------------------------------------------------------------
OpenCL Binary:     mmult.hw_emu.xilinx_u200_xdma_201830_2
Kernels mapped to: clc_region

Timing Information (MHz)
Compute Unit  Kernel Name  Module Name  Target Frequency  Estimated Frequency
------------  -----------  -----------  ----------------  -------------------
mmult_1       mmult        mmult        300.300293        411.015198

Latency Information (clock cycles)
Compute Unit  Kernel Name  Module Name  Start Interval  Best Case  Avg Case  Worst Case
------------  -----------  -----------  --------------  ---------  --------  ----------
mmult_1       mmult        mmult        826 ~ 829       825        827       828

Area Information
Compute Unit  Kernel Name  Module Name  FF     LUT    DSP   BRAM  URAM
------------  -----------  -----------  -----  -----  ----  ----  ----
mmult_1       mmult        mmult        81378  35257  1036  2     0
--------------------------------------------------------------------------------
```

设计和目标器件摘要

所有设计估算报告均以应用汇总信息和有关目标器件的信息开始。在报告的以下部分中提供了器件信息：

```
--------------------------------------------------------------------------------
Design Name:             mmult.hw_emu.xilinx_u200_xdma_201830_2
Target Device:           xilinx:u200:xdma:201830.2
Target Clock:            300.000000MHz
Total number of kernels: 1
--------------------------------------------------------------------------------
```

对于设计汇总信息，所提供的信息包括：

- “Target Device”：目标平台上赛灵思器件的名称，该器件运行由 Vitis 编译器构建的 FPGA 二进制文件。
- “Target Clock”：指定映射到 FPGA 互连结构的计算单元 (CU) 的目标工作频率。

内核汇总

本章节列出了针对应用工程定义的所有内核。以下示例显示了内核汇总信息：

```
Kernel Summary
Kernel Name  Type  Target              OpenCL Library                              Compute Units
-----------  ----  ------------------  ----------------------------------------  -------------
mmult        c     fpga0:OCL_REGION_0  mmult.hw_emu.xilinx_u200_xdma_201830_2  1
```

除了内核名称，该汇总信息还提供了执行目标和输入源的类型。由于 OpenCL™、C 和 C/C++ 源文件的编译和最优化方法存在差异，因此指定了内核源文件的类型。

“内核汇总”章节是报告中的最后汇总信息。此处提供了有关每个计算单元二进制容器的详细信息。

时序信息

对于每个二进制容器，详细信息部分从所有计算单元 (CU) 的执行目标开始。其中还提供了每个 CU 的时序信息。一般情况下，如果 FPGA 二进制文件的估算频率高于目标频率，则 CU 将能够在器件中运行。如果估算频率低于目标频率，则需要进一步最优化 CU 的内核代码，以使其能够在 FPGA 互连结构上正确运行。此信息显示在以下示例中：

```
OpenCL Binary:      mmult.hw_emu.xilinx_u200_xdma_201830_2
Kernels mapped to: clc_region

Timing Information (MHz)
Compute Unit  Kernel Name  Module Name  Target Frequency  Estimated Frequency
------------  -----------  -----------  ----------------  -------------------
mmult_1       mmult        mmult        300.300293        411.015198
```

了解目标频率和估算频率之间的差别非常重要。CU 并非被单独布局到 FPGA 互连结构中。CU 作为有效 FPGA 设计的一部分进行布局，该设计可包括由器件开发者定义的其它组件，以便支持各类应用。

由于每个内核每次生成一项 CU 定制逻辑，估算频率高于目标频率表明 CU 可以更高的估算频率运行。因此，实现 FPGA 二进制文件期间，CU 应满足目标频率的时序要求。

时延信息

时延信息表示二进制容器中每个 CU 的执行剖析。分析此数据时，重要的是确认所有值都是通过定制逻辑从 CU 边界测量所得的。报告的这些值中并不包含与数据传输到全局存储器相关联的系统内时延。此外，报告的时延数值仅适用于以 FPGA 互连结构为目标的 CU。以下是时延报告的示例：

```
Latency Information (clock cycles)
Compute Unit  Kernel Name  Module Name  Start Interval  Best Case  Avg Case  Worst Case
------------  -----------  -----------  --------------  ---------  --------  ----------
mmult_1       mmult        mmult        826 ~ 829       825        827       828
```

时延报告分为以下字段：

- 起始时间间隔
- 最佳情况时延
- 平均情况时延
- 最坏情况时延

起始时间间隔定义了给定内核的两次 CU 调用之间必须经历的时间量。

最佳、平均和最坏情况时延数量是指 CU 为内核的某一 ND 范围数据块生成结果所需的时间。对于内核没有依赖数据的计算循环的情况，时延值将是相同的。依赖数据的循环的执行会导致数据特定的时延变化，该变化将被时延报告捕获。

对于具有以下所列一个或多个条件的内核，间隔或时延数值将被报告为“undef”：

- 没有明确 `reqd_work_group_size(x,y,z)` 的 OpenCL 内核
- 具有含变量边界的循环的内核

注释：时延信息根据对循环变换和已使用模型并行度的分析来反映估算值。这些高级变换（如流水打拍和数据流）可以极大改变实际吞吐量数值。因此，时延仅可用作不同运行之间的相关指南。

区域信息

虽然您可将 FPGA 视为空白的计算画布，但每个 FPGA 中提供的基本构建块的数量是有限的。Vitis 编译器使用这些基本块（FF、LUT、DSP、块 RAM）来为设计中的每个 CU 生成定制逻辑。为单个 CU 实现定制逻辑所需的基本资源数量将决定能够同时加载到 FPGA 互连结构中的 CU 数量。以下示例显示了针对单一 CU 所报告的区域信息：

```
Area Information
Compute Unit   Kernel Name   Module Name   FF      LUT     DSP   BRAM   URAM
------------   -----------   -----------   -----   -----   ----   ----   ----
mmult_1        mmult         mmult         81378   35257   1036   2      0
-------------------------------------------------------------------------
```

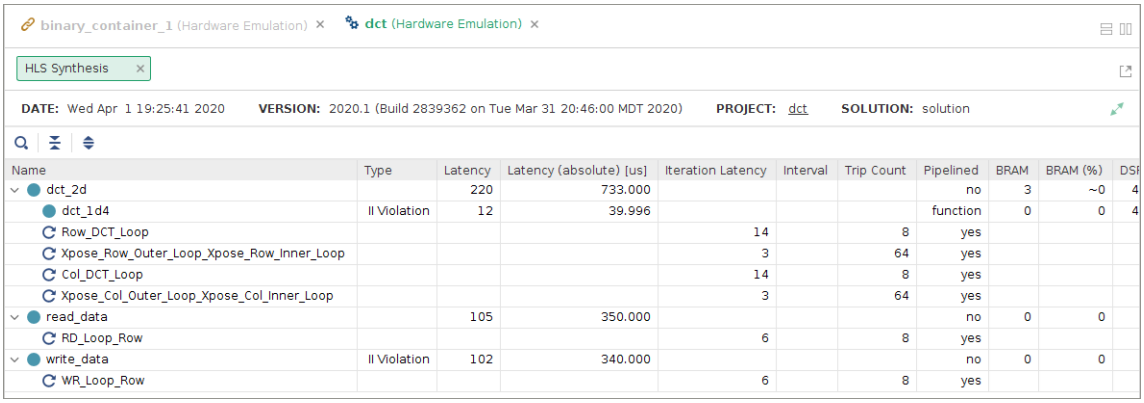
HLS 报告

HLS 报告可以提供有关用户内核的高层次综合 (HLS) 进程的详细信息，在硬件仿真和系统构建的编译进程中会生成此报告。此进程会将 C/C++ 和 OpenCL 内核转换为硬件描述语言，用于在 FPGA 上实现内核逻辑。此报告可以提供定制生成的硬件逻辑的 FPGA 资源使用情况、运行频率、时延和接口信号的估算结果。这些详细信息可以为用户提供诸多见解，指导其执行内核最优化。

如果从 Vitis IDE 运行此报告，此报告可在以下目录中找到：`_x/<kernel_name>.<target>.<platform>/<kernel_name>/<kernel_name>/solution/syn/report`

在 Vitis 分析器中打开“编译汇总 (Compile Summary)”或者“链接汇总 (Link Summary)”即可打开 HLS 报告，如 第六部分：使用 Vitis 分析器 中所述。HLS 报告示例如下所示。

图 32：**HLS 报告**



生成并打开 HLS 报告

重要提示！您必须在构建进程中指定 `--save-temps` 选项以保留 Vitis HLS 所生成的中间文件（包括报告）。HLS 报告和 HLS 指南是专为针对 C 语言内核与 OpenCL 内核的硬件仿真和系统构建而生成的。对于软件仿真或 RTL 内核，则不会生成此报告和指南。

HLS 报告可通过 Vitis 分析器来查看，方法是打开应用工程的 `<output_filename>.compile_summary` 或 `<output_filename>.link_summary`。`<output_filename>` 是 `v++` 命令的输出。

您可以使用以下命令启动 Vitis 分析器并打开此报告：

```
vitis_analyzer <output_filename>.compile_summary
```

打开 Vitis 分析器时，它会显示“编译汇总 (Compile Summary)”以及编译进程中生成的报告集合。如需了解更多信息，请参阅 第六部分：使用 Vitis 分析器。

解读 HLS 报告

“HLS 综合 (HLS Synthesis)”报告是一张电子数据表，其中左列中列出了模块层级。本节主要描述 HLS 报告中的一部分：性能和资源估算。HLS 运行所生成的每个模块和循环都显示在此层级中。“HLS Synthesis”报告包含以下列：

- 违例类型 (Violation Type)
- 以时钟周期数表示的时延 (Latency in clock cycles)
- 以绝对时间表示的时延 (Latency in absolute time (µs))
- 迭代时延 (Iteration latency)
- 迭代时间间隔 (Iteration Interval)
- 循环次数 (Loop Tripcount)
- 流水打拍 (Pipelined)
- BRAM、DSP、FF 和 LUT 利用率估算 (Utilization Estimates of BRAM, DSP, FF, and LUT)
- 负时序裕量 (Negative Slack)

如果此信息是层级块的一部分，则它将汇总层级中包含的块的信息。因此，只要明确哪个实例参与了总体设计，即可在此报告内浏览层级。

注意！周期和时延数量的绝对计数基于 HLS 综合期间确定的估算值，对于高级变换（例如，流水打拍和数据流）尤其如此。因此，这些数值可能并不能精确反映最终结果。如果您在报告中遇到问号，这可能是由于变量边界循环造成的，我们鼓励您为这些循环设置循环次数，以便能够在此报告中显示某些相关估算值。

剖析汇总报告

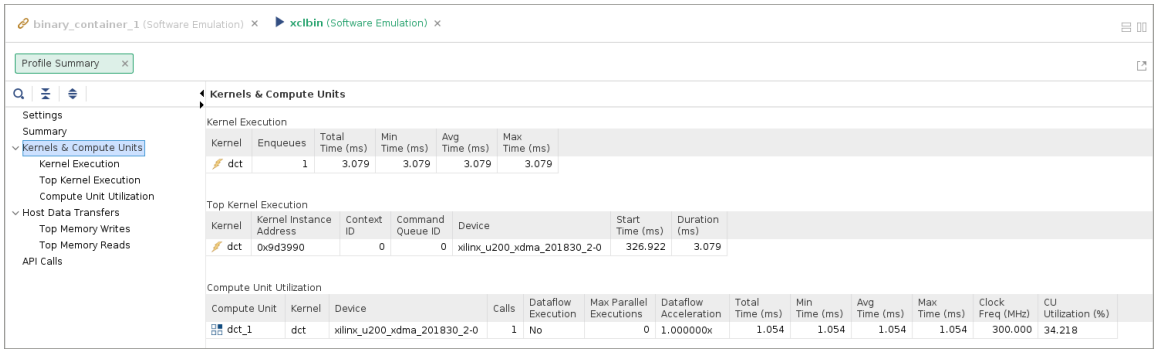
赛灵思的 Xilinx Runtime (XRT) 经正确配置后，即可收集有关主机应用与内核的数据。XRT 会在通过 OpenCL 或 XRT API 调用来调用运行时的过程中，自动为主机应用捕获剖析数据。您可以向自己的主机应用添加用户调用，以捕获更多剖析信息，如 主机应用的定制剖析 中所述。要捕获内核操作的详细信息，必须使用 `--profile` 选项来检测内核，请参阅下一章节获取相关说明。

应用完成执行后，“剖析汇总 (Profile Summary)”报告将另存为 `.csv` 文件，并保存在用于执行已编译的主机代码的目录中。“Profile Summary”报告可提供有关整体应用性能的详细信息（含注释）。应用执行期间生成的所有数据将按类别进行分组。“Profile Summary”允许您检验内核执行情况和数据传输统计数据。

提示：用户可为所有构建配置生成“Profile Summary”报告。但是，对于软件仿真构建，此报告将不包括内核执行效率和数据传输效率方面的任何数据传输详细信息。只有在硬件仿真或系统构建时才会生成这些信息。

以下显示了“Profile Summary”报告的示例。

图 33：**剖析汇总**



生成和打开剖析汇总报告

在实际运行应用之前，首先需要执行多个步骤来捕获“剖析汇总 (Profile Summary)”报告所需的数据。

1. 默认情况下，FPGA 二进制文件 (`xclbin`) 文件配置为捕获剖析数据。但在链接进程中使用 `v++ --profile` 选项能够显著提升捕获的剖析数据的详细程度。如需了解更多信息，请参阅 --profile 选项。
2. 运行时要求 `xrt.ini` 文件已存在（如 xrt.ini 文件 中所述），并且其中包含用于捕获剖析数据的关键字。

```
[Debug]
opencl_summary = true
opencl_device_counter=true
```

3. 要启用内核内部数据剖析，还必须在 `xrt.ini` 的 `[Emulation]` 部分中添加 `debug_mode` 标签：

```
[Emulation]
debug_mode = batch
```

在器件二进制文件以及 `xrt.ini` 文件中启用剖析后，运行时就会在运行应用时创建 `opencl_summary.csv` 报告文件，并在启用“Kernel Internals”时创建 `profile_kernels.csv` 文件。这些文件链接至“Profile Summary”报告，此报告可在 Vitis 分析器工具内通过“Run Summary”来查看。使用以下命令可打开“Run Summary”：

```
vitis_analyzer <project>.run_summary
```

相关信息

运行应用硬件构建
第六部分：使用 Vitis 分析器

解读剖析汇总

剖析汇总 (Profile Summary) 包含主机应用和内核的大量实用统计数据。此报告能够笼统展示您应用中的功能瓶颈。下表显示了剖析汇总描述。

“Settings”

显示此报告和 XRT 配置设置。

“Summary”

显示汇总统计数据，包括器件执行时间和器件电源。

“Kernels & Compute Units”

下表显示了已调度和已执行的所有内核功能的剖析汇总数据。

表 17： **“Kernel Execution”**

名称	描述
“Kernel”	内核名
“Enqueues”	内核排队次数。如果内核仅排队一次，则以下统计数据全部相同。
“Total Time”	所有队列的运行时间总和（按 OpenCL 执行模式下的 START 到 END 来测量）（以 ms 为单位）
“Minimum Time”	所有队列的最小运行时间
“Average Time”	平均内核运行时间（以 ms 为单位） (总时间) / (队列数)
“Maximum Time”	所有队列的最大运行时间（以 ms 为单位）

下表显示了顶层内核函数的剖析汇总数据。

表 18： **“Top Kernel Execution”**

名称	描述
“Kernel”	内核名
“Kernel Instance Address”	内核实例的主机地址（十六进制）
“Context ID”	主机上的上下文 ID
“Command Queue ID”	主机上的命令队列 ID
“Device”	在其中执行内核的器件的名称（格式：`<device>-<ID>`）
“Start Time”	执行开始时间（以 ms 为单位）
“Duration”	执行持续时间（以 ms 为单位）

下表显示了器件上所有计算单元的剖析汇总数据。

表 19： **“Compute Unit Utilization”**

名称	描述
“Compute Unit”	计算单元名
“Kernel”	与此计算单元关联的内核

表 19： **“Compute Unit Utilization”** (续)

名称	描述
“Device”	器件名称（格式：<device>-<ID>）
“Calls”	计算单元的调用次数
“Dataflow Execution”	指定 CU 是否随数据流一起执行
“Max Parallel Executions”	数据流区域中的执行次数
“Dataflow Acceleration”	显示因数据流执行而提升的性能
“CU Utilization (%)”	显示 CU 耗用的内核总计运行时间的百分比
“Total Time”	所有调用的运行时间总和（以 ms 为单位）
“Minimum Time”	所有调用的最小运行时间（以 ms 为单位）
“Minimum runtime of all calls”	(总时间) / (工作组数)
“Maximum Time”	所有调用的最大运行时间（以 ms 为单位）
“Clock Frequency”	用于指定加速器的时钟频率（以 MHz 为单位）

下表显示了器件上计算单元的运行时间和停滞的剖析汇总数据。

表 20： **“Compute Unit Running Times & Stalls”**

名称	描述
“Compute Unit”	计算单元名
“Execution Count”	计算单元的执行计数
“Running Time”	计算单元运行的总时间（以 µs 为单位）
“Intra-Kernel Dataflow Stalls (%)”	计算单元停滞处理内核内部数据流的时间百分比
“External Memory Stalls (%)”	计算单元停滞处理外部存储器访问的时间百分比
“Inter-Kernel Pipe Stalls (%)”	计算单元停滞处理内核间流水线访问的时间百分比

“Kernel Data Transfers”

下表显示了内核到全局存储器的数据传输。

表 21： **“Data Transfer”**

名称	描述
“Compute Unit Port”	计算单元/端口的名称
“Kernel Arguments”	连接到此端口的内核实参的列表
“Device”	器件名称（格式：<device>-<ID>）
“Memory Resources”	该端口访问的存储器资源
“Transfer Type”	内核数据传输的类型
“Number of Transfers”	AXI 传输事务内的内核数据传输次数 **注释：**这可能包含 printf 传输。
“Transfer Rate”	内核数据传输速率（以 MB/s 为单位）： 传输速率 = (总字节数) / (总计 CU 执行时间) 其中总计 CU 执行时间是 CU 保持活动状态的总时间

表 21： **“Data Transfer”** (续)

名称	描述
“Avg Bandwidth Utilization (%)”	内核数据传输的平均带宽： 带宽使用率 (%) = (100 * 传输速率) / (0.6 * 最大理论速率)
“Avg Size”	内核数据传输的平均大小（以 KB 为单位）： 平均大小 = (总 KB 数) / (传输次数)
“Avg Latency”	内核数据传输的平均时延（以 ns 为单位）

下表显示了内核到全局存储器的顶层数据传输。

表 22： **“Top Data Transfer”**

名称	描述
“Compute Unit”	计算单元名
“Device”	器件名
“Number of Transfers”	写入和读取数据传输数量
“Avg Bytes per Transfer”	内核数据传输的平均字节数： 平均字节数 = (总字节数) / (传输次数)
“Transfer Efficiency (%)”	内核数据传输的效率： 效率 = (平均字节数) / min((存储器字节宽度 * 256), 4096)
“Total Data Transfer”	内核传输的数据总量（以 MB 为单位）： 数据总量 = (写入总量) + (读取总量)
“Total Write”	内核写入的数据总量（以 MB 为单位）：
“Total Read”	内核读取的数据总量（以 MB 为单位）：
“Total Transfer Rate”	平均总数据传输速率（以 MB/s 为单位）： 总传输速率 = (数据传输总量) / (总计 CU 执行时间) 其中总计 CU 执行时间是 CU 保持活动状态的总时间

下表显示了数据传输流。

注释： 仅当存在数据流传输时，该表才会显示。

表 23： **“Data Transfer Streams”**

名称	描述
“Master Port”	主计算单元和端口的名称
“Master Kernel Arguments”	连接到此端口的内核实参的列表
“Slave Port”	从计算单元和端口的名称
“Slave Kernel Arguments”	连接到此端口的内核实参的列表
“Device”	器件名称（格式：`<device>-<ID>`）
“Number of Transfers”	流传输数据包数
“Transfer Rate”	数据流传输速率（以 MB/s 为单位）： 传输速率 = (总字节数) / (总计 CU 执行时间) 其中总计 CU 执行时间是 CU 保持活动状态的总时间

表 23： **“Data Transfer Streams”** (续)

名称	描述
“Avg Size”	内核数据传输的平均大小（以 KB 为单位）： 平均大小 = (总 KB 数) / (传输次数)
“Link Utilization (%)”	链接使用率 (%)： 链接使用率 = 100 * (链接忙碌周期数 - 链接停滞周期数 - 链接匮乏周期数) / (链接忙碌周期数)
“Link Starve (%)”	链接匮乏 (%)： 链接匮乏 = 100 * (链接匮乏周期数) / (链接忙碌周期数)
“Link Stall (%)”	链接停滞 (%)： 链接停滞 = 100 * (链接停滞周期数) / (链接忙碌周期数)

“Host Data Transfers”

下表显示了主机与器件存储器之间通过 PCI Express® 链路的所有写入传输的剖析数据。

表 24： **“Top Memory Writes”**

名称	描述
“Buffer Address”	指定缓冲器的地址位置
“Context ID”	主机上的 OpenCL 上下文 ID
“Command Queue ID”	主机上的 OpenCL 命令队列 ID
“Start Time”	写入操作开始时间（以 ms 为单位）
“Duration”	写入操作持续时间（以 ms 为单位）
“Buffer Size”	正在传输的数据量（以 KB 为单位）
“Writing Rate”	数据传输速率（以 MB/s 为单位）： (缓冲器大小)/(持续时间)

下表显示了主机与器件存储器之间通过 PCI Express® 链路的所有读取传输的剖析数据。

表 25： **“Top Memory Reads”**

名称	描述
“Buffer Address”	指定缓冲器的地址位置
“Context ID”	主机上的上下文 ID
“Command Queue ID”	主机上的命令队列 ID
“Start Time”	读取操作开始时间（以 ms 为单位）
“Duration”	读取操作持续时间（以 ms 为单位）
“Buffer Size”	正在传输的数据量（以 KB 为单位）
“Reading Rate”	数据传输速率（以 MB/s 为单位）： (缓冲器大小) / (持续时间)

下表显示了主机到全局存储器的数据传输。

表 26： **“Data Transfer”**

名称	描述
“Context:Number of Devices”	上下文 ID 和上下文中的器件数
“Transfer Type”	内核主机传输的类型
“Number of Buffer Transfers”	主机缓冲器传输次数 **注释：**这可能包含 printf 传输。
“Transfer Rate”	主机缓冲器传输速率（以 MB/s 为单位）： 传输速率 = (总字节数) / (以 µs 为单位的总时间)
“Avg Bandwidth Utilization (%)”	主机缓冲器传输的平均带宽： 带宽使用率 (%) = (100 * 传输速率) / (最大理论速率)
“Avg Size”	主机缓冲器传输的平均大小（以 KB 为单位）： 平均大小 = (总 KB 数) / (传输次数)
“Total Time”	主机缓冲器传输持续时间总和（以 ms 为单位）
“Avg Time”	主机缓冲器传输平均持续时间（以 ms 为单位）

“API Calls”

下表显示在主机应用中执行的所有 OpenCL 主机 API 函数调用的剖析数据。顶部显示的条形图表示 API 调用时间占总时间的百分比。

表 27： **“API Calls”**

名称	描述
“API Name”	API 函数名称（例如，`clCreateProgramWithBinary`、`clEnqueueNDRangeKernel`）
“Calls”	由主机应用对此 API 执行的调用次数
“Total Time”	所有调用的运行时间总和（以 ms 为单位）
“Minimum Time”	所有调用的最小运行时间（以 ms 为单位）
“Average Time”	平均时间（以 ms 为单位） (总时间) / (调用次数)
“Maximum Time”	所有调用的最大运行时间（以 ms 为单位）

“Device Power”

下表显示了器件电源的剖析数据。

表 28： **“Device Power”**

名称	描述
“Power Used By Platform”	显示数据中心加速卡上 3 条电源轨的折线图： · 12V 辅助电源 · 12V PCIe · 内部电源 显示卡的长期功耗 (W) 使用情况。
“Temperature”	针对温度读数为非零值的每个器件创建一张图表。针对读出以 (°C) 为单位的每个温度传感器显示一条折线。
“Fan Speed”	针对风扇速度读数为非零值的每个器件创建一张图表。风扇速度以 RPM 来计量。

“Kernel Internals”

下表显示了以微秒 (µs) 为单位的计算单元运行时间，并报告停滞时间占运行时间的百分比。

提示： “Kernel Internals”选项卡报告以 µs 为单位的时间，而“Profile Summary”报告其余部分则以毫秒 (ms) 为单位来报告时间。

表 29： **“CU Runtime and Stalls”**

名称	描述
“Compute Unit”	表示计算单元实例名称
“Running Time”	报告 CU 的总运行时间（以 µs 为单位）
“Intra-Kernel Dataflow Stalls (%)”	报告在内核之间进行数据流传输时，停滞中耗用的运行时间百分比
“External Memory Stalls (%)”	报告在 CU 外部进行存储器传输时，停滞中耗用的运行时间百分比
“Inter-Kernel Pipe Stalls (%)”	报告进出 CU 的数据流传输期间，停滞中耗用的运行时间百分比

下表显示了计算单元上特定端口的数据传输。

表 30： **“CU Port Data Transfers”**

名称	描述
“Port”	表示计算单元上的端口名称
“Compute Unit”	表示计算单元实例名称
“Write Time”	指定端口上的总计数据写入时间（以 µs 为单位）
“Outstanding Write (%)”	指定写入进程中耗用的运行时间百分比
“Read Time”	指定端口上的总计数据读取时间（以 µs 为单位）
“Outstanding Read (%)”	指定读取进程中耗用的运行时间百分比

下表显示了计算单元上功能端口的数据传输。

表 31： **“Functional Port Data Transfers”**

名称	描述
“Port”	端口名称
“Function”	功能名称
“Compute Unit”	计算单元名
“Write Time”	端口包含未完成的写入的总时间（以 µs 为单位）
“Outstanding Write (%)”	端口包含未完成的写入的时间百分比
“Read Time”	端口包含未完成的读取的总时间（以 µs 为单位）
“Outstanding Read (%)”	端口包含未完成的读取的时间百分比

下表显示了计算单元上运行时间和停滞时间。

表 32： **“Functions”**

名称	描述
“Compute Unit”	计算单元名
“Function”	功能名称
“Running Time”	总计功能运行时间（以 ms 为单位）
“Intra-Kernel Dataflow Stalls”	功能停滞处理内核内部数据流的时间百分比（以 ms 为单位）
“External Memory Stalls”	功能停滞处理外部存储器访问的时间百分比（以 ms 为单位）
“Inter-Kernel Pipe Stalls”	功能停滞处理内核间流水线访问的时间百分比（毫秒）

“Shell Data Transfers”

下表显示 DMA 数据传输。

表 33： **“DMA Data Transfer”**

名称	描述
“Device”	器件名称（格式：<device>-<ID>）
“Transfer Type”	数据传输的类型
“Number of Transfers”	AXI 传输事务内的数据传输次数
“Transfer Rate”	数据传输速率（以 MB/s 为单位）： 传输速率 = (总字节数) / (以 µs 为单位的总时间)
“Total Data Transfer”	已传输的数据总量（以 MB 为单位）
“Total Time”	数据传输的总计持续时间（以 ms 为单位）
“Avg Size”	数据传输的平均大小（以 KB 为单位）： 平均大小 = (总 KB 数) / (传输次数)
“Avg Latency”	数据传输的平均时延（以 ns 为单位）

对于 DMA 旁路和全局存储器到全局存储器数据传输，请参阅以上“DMA 数据传输”表。

“NoC Counters”

NoC 计数器显示的是 NoC 计数器读取和 NoC 计数器写入信息。仅当存在非零 NoC 计数器数据时，才会显示这些部分。

每个部分都有 1 个表，其中包含汇总数据以及传输速率和时延的折线图。这些图形可包含多个 NoC 计数器，以便您通过该表格的“图表 (Chart)”列中的复选框来开关计数器。

根据设计，可能可以将 NoC 计数器关联到 CU 端口。在此情况下，CU 端口会显示在表格中，选中此端口即可对系统框图、剖析汇总以及包含 CU 端口作为可选对象的任何其它视图进行交叉探测。

表 34： **“NoC Counters Read or Write”**

名称	描述
“Compute Unit Port”	计算单元/端口的名称
“Name”	NoC 端口名称
“Traffic Class”	流量类的类型

表 34： **“NoC Counters Read or Write”** (续)

名称	描述
“Requested QoS”	QoS (MB/s)：请求的服务质量（以 MB/s 为单位）
“Min Transfer Rate”	数据传输的最小速率（以 MB/s 为单位）
“Avg Transfer Rate”	数据传输的平均速率（以 MB/s 为单位）
“Max Transfer Rate”	数据传输的最大速率（以 MB/s 为单位）
“Avg Size”	数据传输的平均大小（以 KB 为单位）： 平均大小 = (总 KB 数) / (传输次数)
Min Latency	数据传输的最小时延（以 ns 为单位）
Avg Latency	数据传输的平均时延（以 ns 为单位）
Max Latency	数据传输的最大时延（以 ns 为单位）

“AI Engine Counters”

如果存在非零 AI 引擎计数器数据，就会显示 AI 引擎计数器。如果存在不兼容的 AI 引擎计数器配置，则这部分会显示 1 条消息称此配置不支持性能剖析。

这部分包含 1 个表，其中包含汇总数据以及活动时间和使用情况的折线图。仅当启用停滞剖析时，使用情况图表才可用。

这些图形可包含多个 AI 引擎计数器，以便您通过该表格的“图表 (Chart)”列中的复选框来开关计数器。

可以对拼块 (tile) 和 AI 引擎阵列与 Graph 视图进行交叉探测。

注释： 根据 AI 引擎计数器的配置方式，可能显示一个或多个指标列。这些指标列包括存储器停滞、数据流停滞、调用实例时间、组错误时间等。欲知详情，请参阅 《Versal ACAP AI 引擎编程环境用户指南》(UG1076)。

表 35： **“AI Engine Counters”**

名称	描述
“Tile”	AI 引擎 拼块 (tile) [列, 行]
“Clock Frequency (MHz)”	用于 AI 引擎拼块的时钟频率（以 MHz 为单位）

时间线轨迹

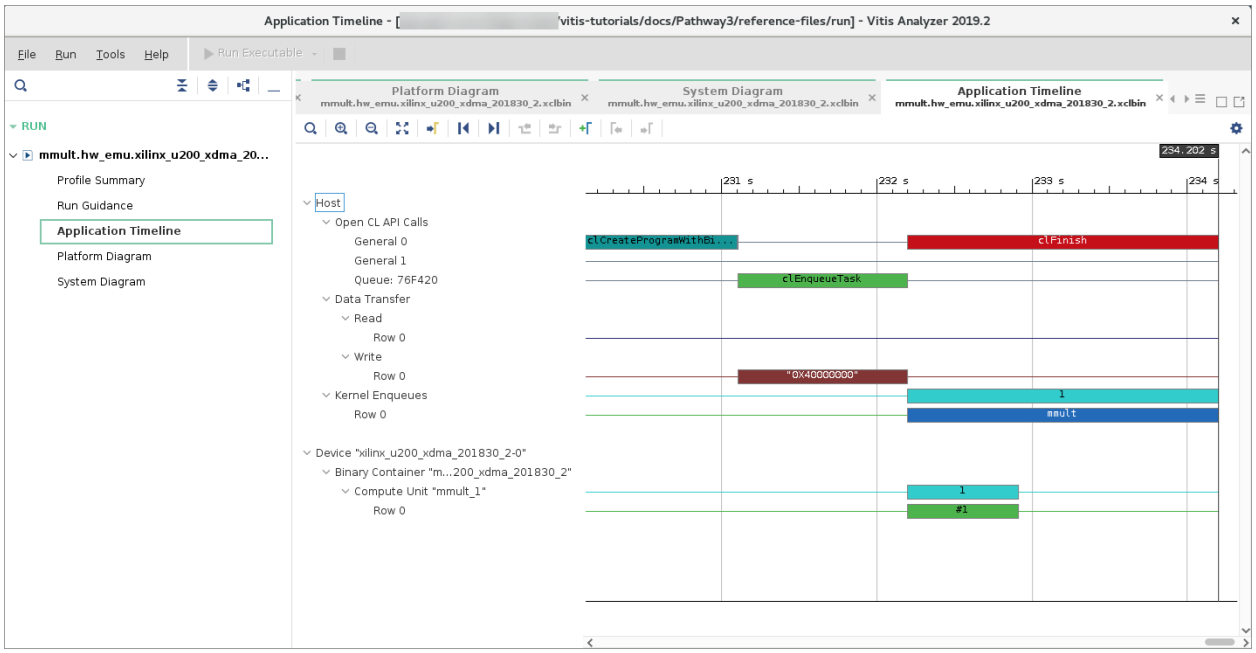
“时间线轨迹 (Timeline Trace)”在公共的时间线上收集并显示主机和内核事件，以便帮助您了解和可视化各类系统的总体运行状况和性能。图形化表示法能够便于您查看有关内核同步和高效并发执行的问题。显示的事件包括：

- 来自主机代码的 OpenCL API 调用。
- 器件追踪数据，包括计算单元、AXI 传输事务启动/停止。
- 主机事件和内核启动/停止。

虽然时间线和器件追踪数据对于应用调试和剖析很有用，但是默认不收集这些数据，因为收集这些数据会增加应用执行时间，从而影响性能。但在内核中通过专用资源来收集追踪数据，并且不会影响内核功能。默认情况下，数据在运行结束时卸载（`v++ --trace_memory` 选项）。启用连续卸载后，会更改默认行为。

以下是“时间线轨迹 (Timeline Trace)”窗口的快照，其中在公共时间线上显示主机和器件事件。主机活动显示在图像的顶部，内核活动显示在图像的底部。主机活动包括创建程序、运行内核以及全局存储器与主机之间的数据传输。内核活动包括读写访问以及全局存储器与内核之间的传输。此信息可帮助您了解应用程序执行的详细信息并确定潜在可改进之处。

图 34：**时间线轨迹**



时间线数据可通过命令行流程来启用和收集。但必须在 Vitis 分析器内查看这些数据，如 第六部分：使用 Vitis 分析器 中所述。

生成并打开时间线轨迹

要生成“时间线轨迹 (Timeline Trace)”报告，必须完成以下步骤，在命令行流程中启用时间线和设备树数据收集：

1. 链接期间，使用 `v++ --profile` 选项向内核添加 Acceleration Monitor 和 AXI Performance Monitor 以检测 FPGA 二进制文件，如 --profile 选项 中所述。例如，请将 `--profile.data` 添加到 `v++` 链接命令行中：

   ```
   v++ -g -l --profile.data all:all:all ...
   ```

2. 在构建进程中完成内核检测后，在执行应用运行时期间还必须通过编辑 `xrt.ini` 文件以启用数据收集。如需了解更多信息，请参阅 xrt.ini 文件。

   以下 `xrt.ini` 文件可在运行应用时启用最大量的信息收集。

   ```
   [Debug]
   opencl_summary=true
   opencl_trace=true
   data_transfer_trace=coarse
   stall_trace=all
   ```

   提示：如果您收集大量追踪数据，您可能需要在 `xrt.ini` 中使用 `--trace_memory` 搭配 `v++` 命令，并使用 `trace_buffer_size` 关键字。

   运行应用后，将在名为 `opencl_trace.csv` 和 `device_0.csv` 的 CSV 文件中捕获“时间线轨迹”数据。

3. 在 Vitis 分析器工具中，可打开应用执行期间生成的“Run Summary”来查看此 CSV 报告。您可以使用以下命令启动 Vitis 分析器并打开“Run Summary”：

```
vitis_analyzer <project>.run_summary
```

解读时间线轨迹

“时间线轨迹 (Timeline Trace)”窗口可在公用时间线上显示主机和器件事件。此信息可帮助您了解应用程序执行的详细信息并确定潜在可改进之处。“Timeline Trace”报告具有 2 个主要部分：“主机 (Host)”和“器件 (Device)”。“Host”部分显示的是源自主机侧的所有活动的追踪轨迹。“Device”显示的则是 FPGA 上的 CU 活动。

此报告具有如下结构：

- “Host”
  - “OpenCL API Calls”：在此追踪所有 OpenCL API 调用。从主机透视图角度测量活动时间。
    - “General”：在此追踪所有常规 OpenCL API 调例，如 `clCreateProgramWithBinary`、`clCreateContext` 和 `clCreateCommandQueue`。
    - “Queue”：在此追踪与特定命令队列相关联的 OpenCL API 调用。包括诸如 `clEnqueueMigrateMemObjects` 和 `clEnqueueNDRangeKernel` 等命令。如果用户应用创建了多个命令队列，则此部分会显示所有队列和活动。
    - “Data Transfer”：在此处追踪从主机到器件存储器的 DMA 传输。有多个 DMA 线程在 OpenCL 运行时实现，通常有相同数量的 DMA 通道。DMA 传输由用户应用通过调用 OpenCL API（例如，`clEnqueueMigrateMemObjects`）来进行初始化。这些 DMA 请求将被转发至运行时，随后被委派至其中一个线程。从主机到器件的数据传输显示在“Write”下，因为这些传输由主机写入，而从器件到主机的传输则显示在“Read”下。
    - “Kernel Enqueues”：此处显示的是由主机程序排队的内核。此处的内核不应与器件上的内核或 CU 混淆。此处的内核表示 `NDRangeKernels` 以及由 OpenCL 的 `clEnqueueNDRangeKernels` 和 `clEnqueueTask` 命令创建的任务。这些内核是根据从主机视角测量所得时间来绘制的。用户可将多个内核调度为同时执行，并且可从调度运行的时间点开始追踪，直至内核执行结束为止。这就是存在多个输入的原因。行数取决于重叠的内核执行数。

      注释：内核的重叠并非表示在器件上实际并行执行，因为进程可能尚未准备好立即执行。
- “Device "name"”
  - “Binary Container "name"”：二进制容器名称。
    - “Accelerator "name"”：FPGA 上计算单元的名称（又称为“加速器”）。
      - “User Functions”：对于 Vitis HLS 工具内核而言，在此追踪作为数据流进程执行的函数。这些函数的追踪会显示当前正在并行执行的函数的活动实例数。当启用波形时，会在硬件仿真中生成这些名称。

        注释：函数级别活动仅在硬件仿真中才可能出现。
        - “Function: "name a"”
        - “Function: "name b"”
      - “Read”：CU 通过 AXI-MM 端口从 DDR 中读取。此处显示由 CU 读取的数据的轨迹。活动显示为传输事务，每个传输事务的工具提示都会显示有关此 AXI 传输事务的更多详情。当为 CU 指定 `--profile.data` 时，就会生成这些名称。
      - “Write”：CU 通过 AXI-MM 端口写入 DDR。此处显示由 CU 写入的数据的轨迹。活动显示为传输事务，每个传输事务的工具提示都会显示有关此 AXI 传输事务的更多详情。当为 CU 指定 `--profile.data` 时，就会生成此信息。

波形视图和实时波形查看器

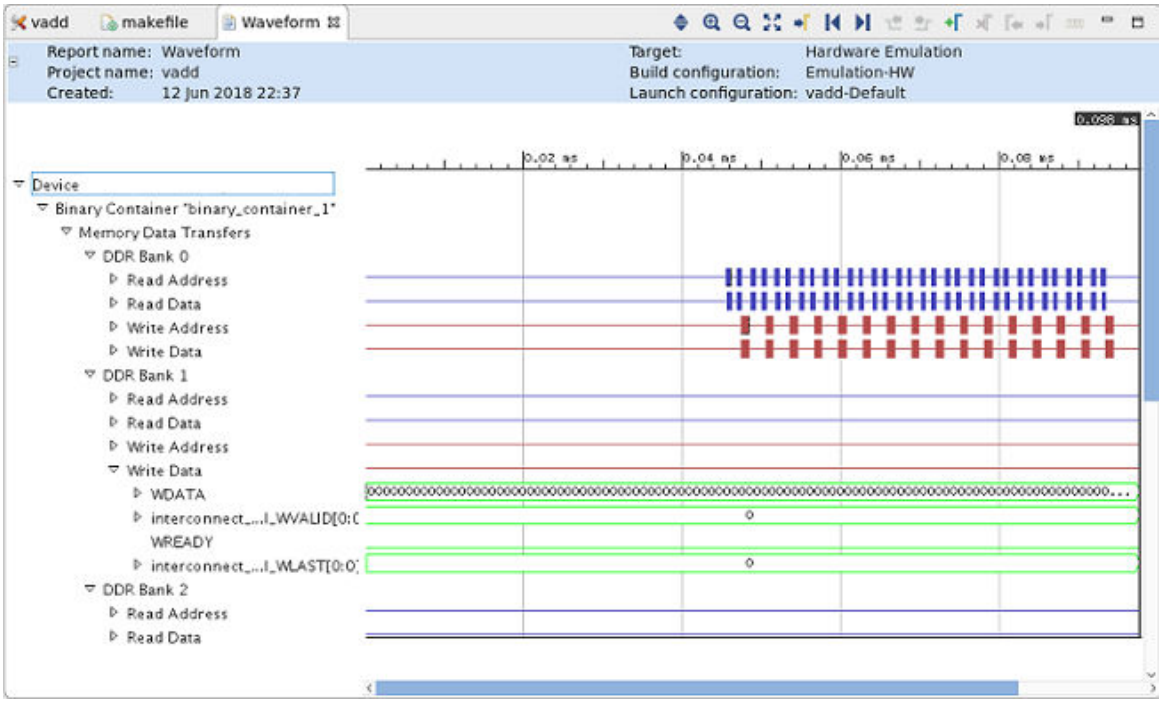
Vitis 核开发套件可在运行硬件仿真时生成“波形 (Waveform)”视图。它可显示系统级别、CU 级别和函数级别的深度详细信息。详细信息包括内核和全局存储器之间的数据传输以及通过内核间管道的数据流。这些详细信息对性能瓶颈提供了诸多有助于应用最优化的深入洞察，上至系统级别，下至个别函数调用。

“实时波形查看器 (Live Waveform Viewer)”与“Waveform”视图类似，但是前者可以提供更低级别的细节，并具有一定程度的交互性。“Live Waveform Viewer”还可使用 Vivado 逻辑仿真器 `xsim` 来打开。

注释： “Waveform”视图允许您从 Vitis 分析器内部检验器件传输事务，如 第六部分：使用 Vitis 分析器 中所述。相反，“Live Waveform Viewer”则可打开 Vivado 仿真波形查看器，以检验除任何用户选定的信号以外的硬件传输事务。

默认情况下不收集波形数据，因为它需要运行时才能在硬件仿真期间生成仿真波形，而这会耗用更多时间和磁盘空间。请参阅 生成并打开波形报告 以获取有关启用这些功能的指示信息。

图 35： **“Waveform”视图**



您也可以使用 Vivado 逻辑仿真器通过 Linux 命令行打开波形数据库 (`.wdb`) 文件：

```
xsim -gui <filename.wdb> &
```

提示： `.wdb` 文件写入的目录与执行已编译的主机代码的目录相同。

生成并打开波形报告

在硬件仿真期间遵循以下指示信息，即可启用从命令行收集波形数据，并打开查看器：

1. 在编译和链接期间，使用 `-g` 选项启用调试代码生成。

```
v++ -c -g -t hw_emu ...
```

2. 在主机可执行文件所在目录中创建 `xrt.ini` 文件，其中包含以下内容（请参阅 xrt.ini 文件 以获取更多信息）：

```
[Emulation]
debug_mode=batch
```

`debug_mode=batch` 支持通过以批处理模式运行仿真来捕获波形数据 (`.wdb`)。您也可以通过在 `xrt.ini` 中使用以下设置启用“实时波形查看器 (Live Waveform Viewer)”，以交互模式启动仿真：

```
[Emulation]
debug_mode=gui
```

提示：如果已启用“Live Waveform Viewer”，那么运行硬件仿真期间，会打开仿真波形。

3. 按 运行应用硬件构建 中所述方式，运行应用的硬件仿真构建。这样即可在波形数据库文件 `<hardware_platform>-<device_id>-<xclbin_name>.wdb` 中收集硬件传输事务数据。如需了解有关查找这些报告的更多信息，请参阅 v++ 命令的输出目录 或 Vitis IDE 的输出目录。

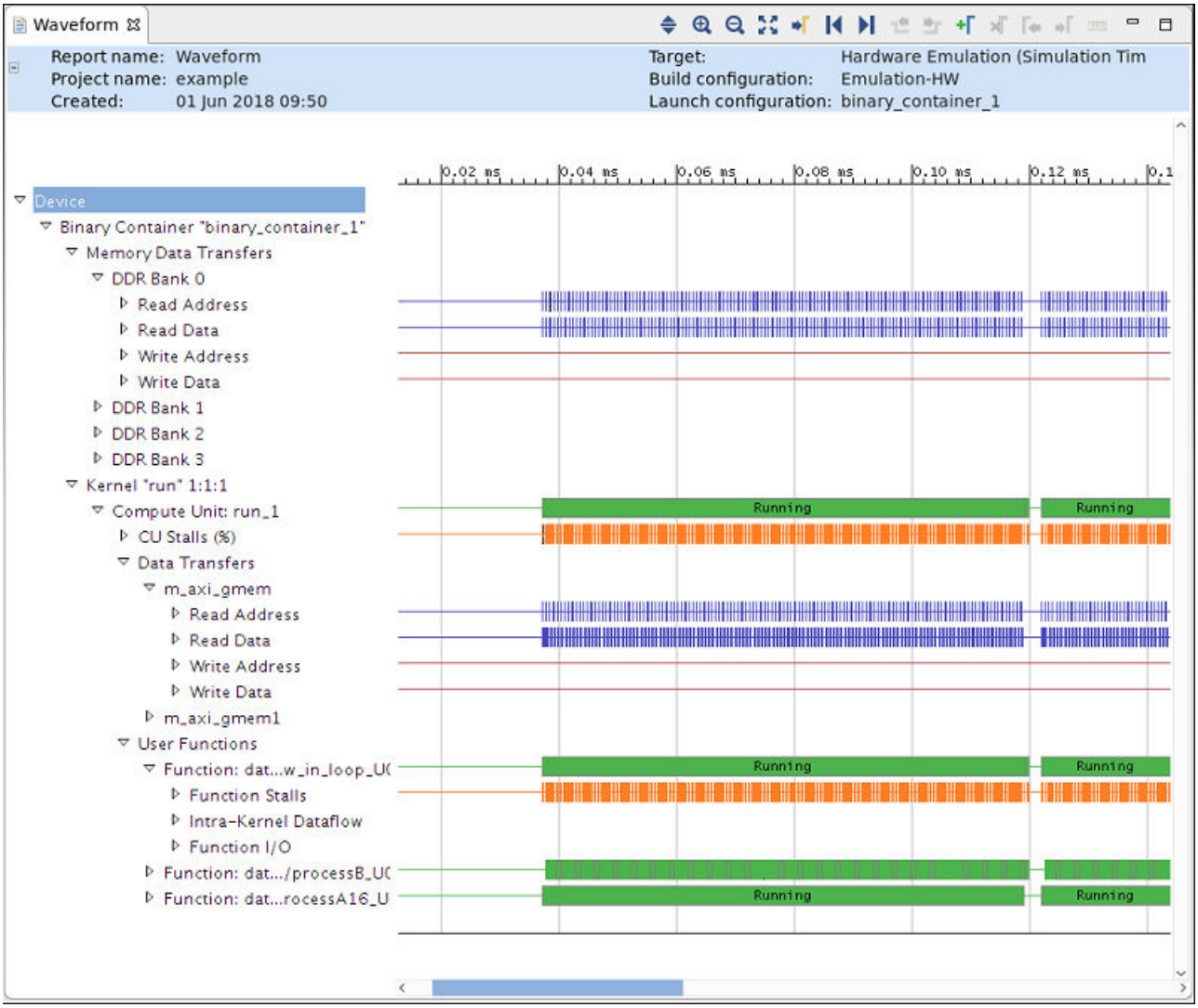
4. 在 Vitis 分析器中打开“运行汇总 (Run Summary)”报告并打开“波形 (Waveform)”报告即可打开“波形 (Waveform)”视图。

```
vitis_analyzer <project>.run_summary
```

解读“Waveform”视图中数据

以下图像显示的是“Waveform”视图：

图 36： **“Waveform”视图**



“波形 (Waveform)”视图和“实时波形 (Live Waveform)”视图按层级组织，以方便导航。

- “Waveform”视图基于硬件仿真（内核追踪）期间生成的实际波形来显示的。这样即可使查看器从上到下逐级衍生，直至负责处理抽象数据的单个信号为止。但由于“Waveform”视图是从后处理数据生成的，无法向报告中添加其它信号，而且部分运行时分析无法可视化，例如，DATAFLOW 传输事务。
- “实时波形 (Live Waveform)”查看器显示的是 Vivado 逻辑仿真器 (`xsim`) 的运行情况，因此您可以在实时视图中添加额外信号以及寄存器传输级 (RTL) 设计内部信息。如需了解有关使用“Waveform”查看器的信息，请参阅《Vivado Design Suite 用户指南：逻辑仿真》(UG900)。

“Waveform”视图和“Live Waveform”视图的层级包括：

- “Device "name"”：目标器件名。
  - “Binary Container "name"”：二进制容器名称。
    - “Memory Data Transfers”：对于每个 DDR 存储体，这里显示的是从主机到达该存储体的所有读写请求传输事务的追踪信息。

- “Kernel "name" 1:1:1”：对于每个内核和该内核的每个计算单元，此部分细分了源自计算单元的各种活动。
  - “Compute Unit: "name"”：计算单元名。
  - “CU Stalls (%)”：当由于外部存储器访问、内部流（即数据流）传输或外部流（即 OpenCL 流水线）传输等原因造成电路的一部分停滞时，Vitis HLS 工具会提供停滞信号来提醒您。详细内核追踪中显示的停滞总线会对所有最低级别的停滞信号进行编译，并报告在任意时间点停滞的百分比。这样可以提供仿真中任意时间点处于停滞状态的内核数量因子。

    例如，在给定时钟周期内，如果有 100 个最低级别的停滞信号，有 10 个信号处于活动状态，那么 CU 停滞百分比为 10%。如果其中任一信号变为不互动，则百分比变为 9%。
  - “Data Transfers”：可显示从每个计算单元的 AXI 主端口到 DDR 发起的读写数据传输访问数。
  - “User Functions”：此信息可用于 HLS 内核，并可显示用户函数。
    - “Function: "name"”：函数名。
      - “Dataflow/Pipeline Activity”：如果函数作为数据流进程来实现，那么此信息可显示函数的并行执行数。
        - “Active Iterations”：此信息可显示数据流的当前活动迭代数。行数将动态递增以调整任何并发执行的可视化显示。
        - “StallNoContinue”：这是一个停滞信号，表示数据流进程是否遇到任何输出停滞（函数已执行，但是未能从相邻的数据流进程接收到继续执行信号）。
        - “RTL Signals”：这些是底层 RTL 控制信号，用于解读上面的数据流进程传输事务视图。
      - “Function Stalls”：显示该进程遇到的不同类型的停滞。
        - “External Memory”：访问 DDR 存储器时遇到停滞。
        - “Internal-Kernel Pipe”：如果计算单元彼此之间通过流水线来进行通信，那么此信息可显示相关停滞情况。
      - “Intra-Kernel Dataflow”：内核内部的 FIFO 活动。
      - “Function I/O”：实际接口信号。

XILINX

第 19 章

性能最优化

主机最优化

本节侧重于演示主机程序的最优化，主机程序使用 OpenCL™ API 来调度各个计算单元的执行以及往来 FPGA 开发板的数据传输。为了对数据传输和计算调用执行最优化，您需要思考如何通过一个或多个 OpenCL 命令队列来并发执行各项任务。本节主要探讨常见的错误做法及其识别和解决方法。

减少内核排队的开销

基于 OpenCL 的执行模型支持数据并行和任务并行编程模型。OpenCL 主机通常需要多次调用不同内核。这些内核会在命令队列中按特定顺序排队，或者在无序命令队列中排队。随后，根据计算资源和任务数据可用性，将在器件上调度这些内核的执行。

用户可在命令队列上使用 `clEnqueueTask` 来对内核调用进行排队。分派过程是在主机处理器上执行的。分派会在将内核实参传输到器件上运行的加速器之后调用内核执行。分派器使用赛灵思的低层级 Xilinx Runtime (XRT) 库来传输内核实参并发出开始计算的触发命令。根据为内核设置的实参数量，向加速器分派命令和实参的开销可能在 30 µs 到 60 µs 之间。您可通过最大程度减少内核需执行的次数和最大程度减少 `clEnqueueTask` 调用次数来降低此开销的影响。理想情况下，应在单次 `clEnqueueTask` 调用中完成所有计算。

您可通过将数据分批并调用一次内核来最大程度减少 `clEnqueueTask` 调用次数，并使循环环绕在原始实现周围以避免多次排队调用的开销。这样还可以提升主机与加速器之间的数据传输性能，因为传输的是少量大数据包，而不是传输大量小数据包。如需了解有关减少内核执行开销的更多信息，请参阅 内核执行。

以下示例显示的是用于处理给定工作量或数据大小的简单内核。

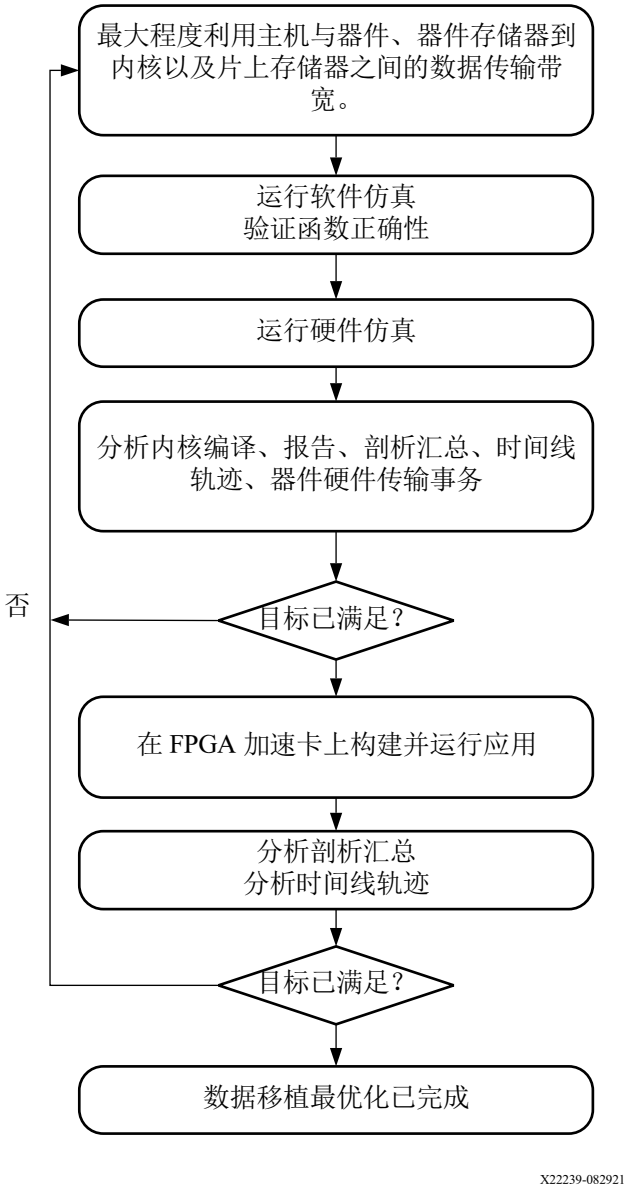
```
#define SIZE 256
extern "C" {
    void add(int *a , int *b, int inc){
        int buff_a[SIZE];
        for(int i=0;i<size;i++)
        {
            buff_a[i] = a[i];
        }
        for(int i=0;i<size;i++)
        {
            b[i] = a[i]+inc;
        }
    }
}
```

以下示例显示的是与以上示例相同的简单内核，但此处将其优化为处理分批数据。根据 num_batches 实参，内核可在单次调用内处理多项大小为 256 的输入，避免了多次调用 clEnqueueTask 的开销。主机应用可更改为按 SIZE * num_batches 区块来分配数据和缓冲器，实际上即对主机全局存储器与器件存储器之间的存储器分配和数据传输进行分批处理。

```
#define SIZE 256
extern "C" {
    void add(int *a , int *b, int inc, int num_batches){
        int buff_a[SIZE];
        for(int j=0;j<num_batches;j++)
        {
            for(int i=0;i<size;i++)
            {
                buff_a[i] = a[i];
            }
            for(int i=0;i<size;i++)
            {
                b[i] = a[i]+inc;
            }
       }
    }
}
```

最优化数据移动

图 37：**最优化数据移动流程**



在 OpenCL 执行模型中，首先所有数据都从主机主存储器传输到全局器件存储器，然后从全局器件存储器传输到内核用于执行计算。计算结果从内核重新写入全局器件存储器，最后从全局存储器重新写入主机主存储器。确定内核数据移动最优化策略的关键要素之一是了解如何在不同级别的存储器之间有效移动数据，从而最大程度地有效利用所有存储器接口上的带宽。

建议：执行计算最优化前，先对应用中的数据移动进行最优化。

数据移动最优化过程中，将数据传输代码与计算代码隔离非常重要，因为计算效率低下可能会导致数据移动停滞。在此最优化步骤中，应侧重于修改主机和内核代码中的数据传输逻辑。这个步骤的目标是通过最大程度提升数据传输带宽和器件全局存储器带宽利用率，来最大程度提升系统级别的数据吞吐量。通常运行软件仿真、硬件仿真以及硬件执行需要经过多次迭代才能达成最优性能。

将数据传输与内核计算重叠

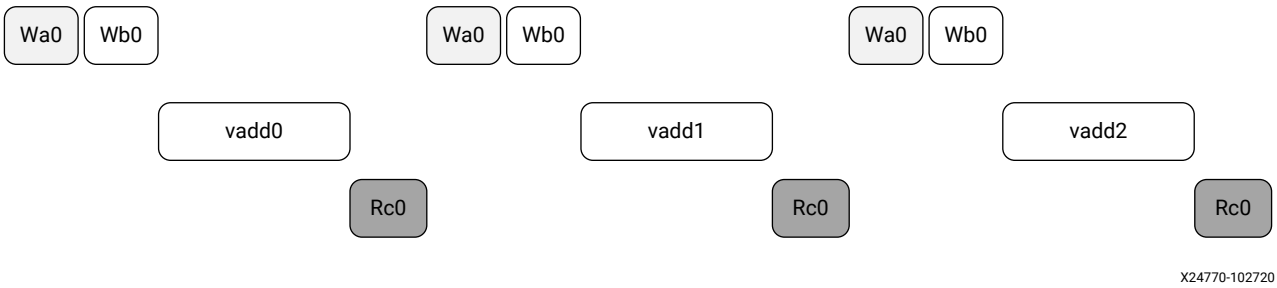
诸如数据库分析等应用所含的数据集大小远超加速器件上可用的全局器件存储器中可存储的量。此类应用要求以块的形式来传输和处理完整的数据。将数据传输与计算加以重叠的技巧对于为这类应用实现高性能而言，显得至关重要。

在 GitHub 的 Vitis 加速示例的主机类别下的重叠示例的 `vadd` 内核中可找到相应的示例。此示例演示了在应用中重叠主机 (CPU) 与 FPGA 计算的技巧。在此示例中，内核对 2 个阵列的处理方式是将其添加到一起并写入输出。从主机角度来看，在此示例中执行 4 项任务：

1. 写入缓冲器 a (`Wa`)
2. 写入缓冲器 b (`Wb`)
3. 执行 `vadd` 内核
4. 读取缓冲器 c (`Rc`)

如果使用简单的按顺序执行命令队列（无数据传输最优化），整个执行时间线轨迹应如下所示：

图 38：**主机任务视图**

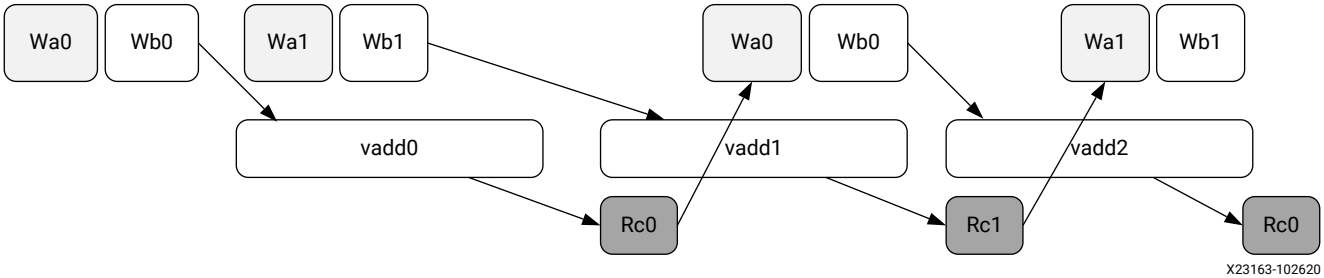


如果使用无序命令队列，则数据传输和内核执行可以重叠，如下图所示。在此示例的主机代码中，针对所有缓冲器使用双重缓冲，以便在内核处理一组缓冲器的同时，主机可以在另一组缓冲器上运行。

OpenCL `event` 对象提供了一种简单的方法，可用于设置复杂的操作依赖关系并同步主机线程和器件操作。事件均为 OpenCL 对象，可追踪操作状态。事件对象是由存储器对象上的内核执行命令 `read`、`write` 和 `copy` 命令或者使用 `clCreateUserEvent` 创建的用户事件来创建的。

您可通过查询这些命令返回的事件来确保操作完成。下图中的箭头演示了如何设置事件触发以实现最优性能。

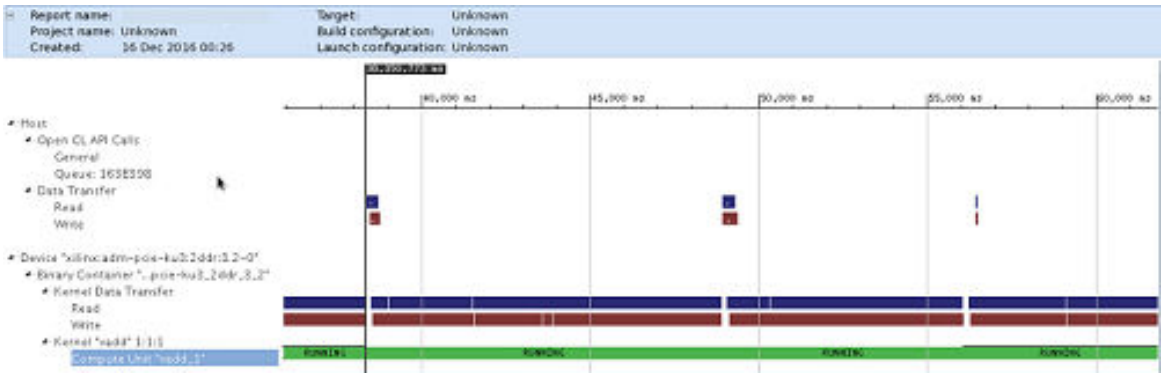
图 39：**事件触发设置**



在此示例中，主机代码 (host.cpp) 通过一个循环来对这 4 项任务进行排队，以便处理整个数据集。它还会设置不同任务之间的事件同步，以确保满足每项任务的数据相依赖关系。通过将不同存储器对象值传递给 clEnqueueMigrateMemObjects API 即可设置双重缓冲。事件同步可通过让每个 API 调用等待其它事件以及在 API 完成时触发其自己的事件来予以实现。

以下“应用时间线 (Application Timeline)”视图清晰展示了数据传输时间被完全隐藏，而计算单元 vadd_1 则持续运行的过程。

图 40： **“Application Timeline”视图中隐藏的数据传输时间**



缓冲存储器分段

存储缓冲器的分配和取消分配可能会导致 DDR 控制器中发生存储器分段。这可能会导致计算单元性能欠佳，即使理论上这些单元能并行执行也是如此。

当不同计算单元使用多个线程时，经常会出现这个问题，此时在线程每次执行内核排队时，都会分配和释放具有不同大小的许多器件缓冲器。这种情况下，时间线轨迹将在内核执行之间出现间隙，看起来就像进程处于休眠状态。

由运行时分配的每个缓冲器在硬件中应该是连续的。对于较大的存储器，当分配和取消分配许多缓冲器时，可能需要等待一段时间才能释放该空间。这个问题可以通过分配器件缓冲器并在内核的不同队列之间复用缓冲器来解决。

计算单元调度

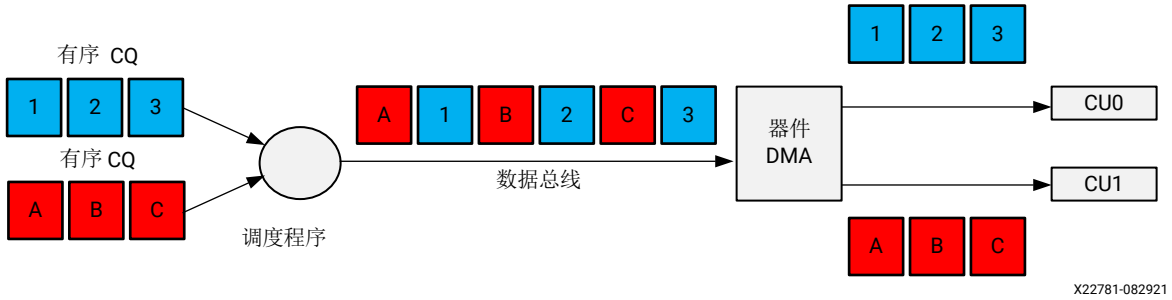
调度内核运算对于整体系统性能非常关键。当执行多个计算单元（同一个内核或不同内核）时，这变得更加重要。本节将考察负责调度内核的不同命令队列。

多个有序命令队列

建议：为了提升性能，赛灵思建议使用单个无序命令队列，并显式管理事件依赖关系和同步，而不是使用多个命令队列。

下图所示示例包含 2 个有序命令队列：CQ0 和 CQ1。调度器从每个队列中按顺序分派命令，但是调度器可以按任何顺序从 CQ0 和 CQ1 中提取出命令。如果需要，您必须管理 CQ0 与 CQ1 之间的同步。

图 41：**含 2 个有序命令队列的示例**



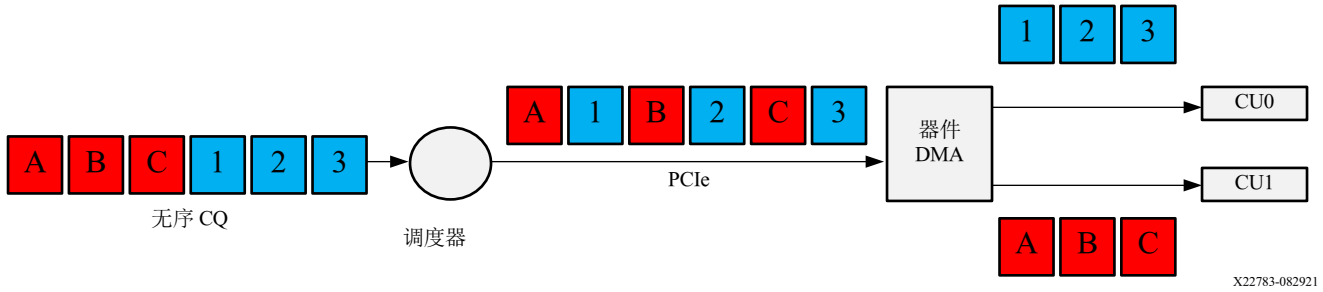
以下代码提取自 concurrent_kernel_execution_c 示例的 `host.cpp`，用于设置多个有序命令队列，并在每个队列中对命令执行排队。

```
   OCL_CHECK(
        err,
        cl::CommandQueue ooo_queue(context,
                                   device,
                                   CL_QUEUE_PROFILING_ENABLE |

CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
                                   &err));
...
    printf("[OOO Queue]: Enqueueing scale kernel\n");
    OCL_CHECK(
        err,
        err = ooo_queue.enqueueTask(
            kernel_mscale,nullptr,&ooo_events[0]));
    set_callback(ooo_events[0], "scale");
...
    // This is an out of order queue, events can be executed in any order.
Since
    // this call depends on the results of the previous call we must pass
the
    // event object from the previous call to this kernel's event wait list.
    printf("[OOO Queue]: Enqueueing addition kernel (Depends on scale)\n");
    kernel_wait_events.resize(0);
    kernel_wait_events.push_back(ooo_events[0]);
    OCL_CHECK(err,
              err = ooo_queue.enqueueTask(
                  kernel_madd,
                  &kernel_wait_events, // Event from previous call
                  &ooo_events[1]));
    set_callback(ooo_events[1], "addition");
...
    // This call does not depend on previous calls so we are passing nullptr
    // into the event wait list. The runtime should schedule this kernel in
    // parallel to the previous calls.
    printf("[OOO Queue]: Enqueueing matrix multiplication kernel\n");
    OCL_CHECK(err,
              err = ooo_queue.enqueueTask(
                  kernel_mmult,
                  nullptr,
                  &ooo_events[2]));
    set_callback(ooo_events[2], "matrix multiplication");
```

单个无序命令队列

下图所示示例包含单个无序命令队列。调度器可以任何顺序从队列中分派命令。如果需要，您必须手动定义事件依赖关系并同步。

图 42：**含单个无序命令队列的示例**



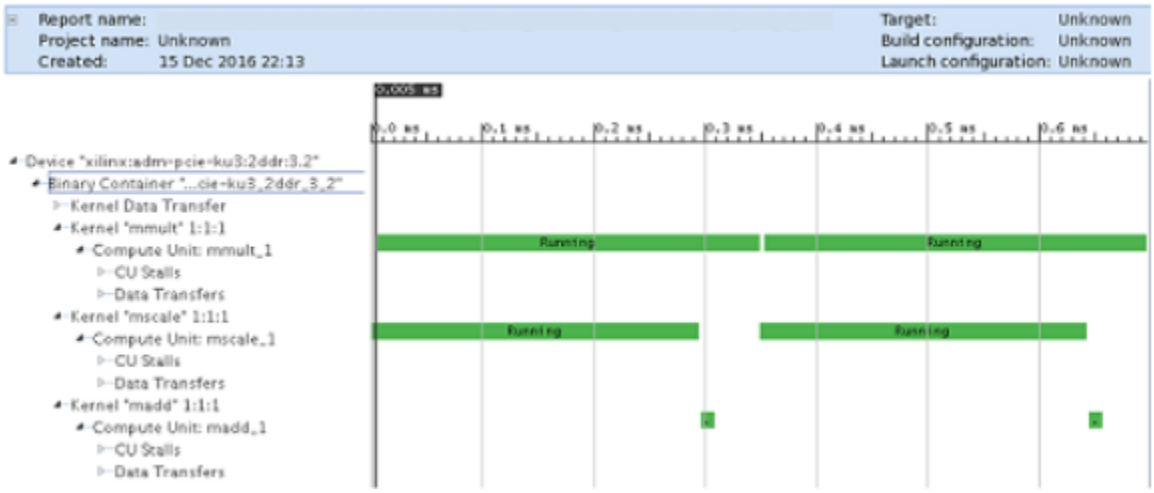
以下代码提取自 concurrent_kernel_execution_c 示例的 `host.cpp`，用于设置单个无序命令队列，并按需对命令执行排队。

```
    OCL_CHECK(
        err,
        cl::CommandQueue ooo_queue(context,
                                   device,
                                   CL_QUEUE_PROFILING_ENABLE |

CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
                                   &err));
...
    printf("[OOO Queue]: Enqueueing scale kernel\n");
    OCL_CHECK(
        err,
        err = ooo_queue.enqueueTask(
            kernel_mscale,nullptr, &ooo_events[0]));
    set_callback(ooo_events[0], "scale");
...
    // This is an out of order queue, events can be executed in any order.
Since
    // this call depends on the results of the previous call we must pass
the
    // event object from the previous call to this kernel's event wait list.
    printf("[OOO Queue]: Enqueueing addition kernel (Depends on scale)\n");
    kernel_wait_events.resize(0);
    kernel_wait_events.push_back(ooo_events[0]);
    OCL_CHECK(err,
              err = ooo_queue.enqueueTask(
                  kernel_madd,
                  &kernel_wait_events, // Event from previous call
                  &ooo_events[1]));
    set_callback(ooo_events[1], "addition");
    // This call does not depend on previous calls so we are passing nullptr
    // into the event wait list. The runtime should schedule this kernel in
    // parallel to the previous calls.
    printf("[OOO Queue]: Enqueueing matrix multiplication kernel\n");
    OCL_CHECK(err,
```

```
            err = ooo_queue.enqueueTask(
                kernel_mmult,
                nullptr,
                &ooo_events[2]));
    set_callback(ooo_events[2], "matrix multiplication");
```

“应用时间线 (Application Timeline)”视图显示计算单元 `mmult_1` 当前与计算单元 `mscale_1` 和 `madd_1` 并行运行，并且同时使用多个有序队列方法和单个无序队列方法。

图 43：**显示 mult_1 与 mscale_1 和 madd_1 并行运行的“Application Timeline”视图**

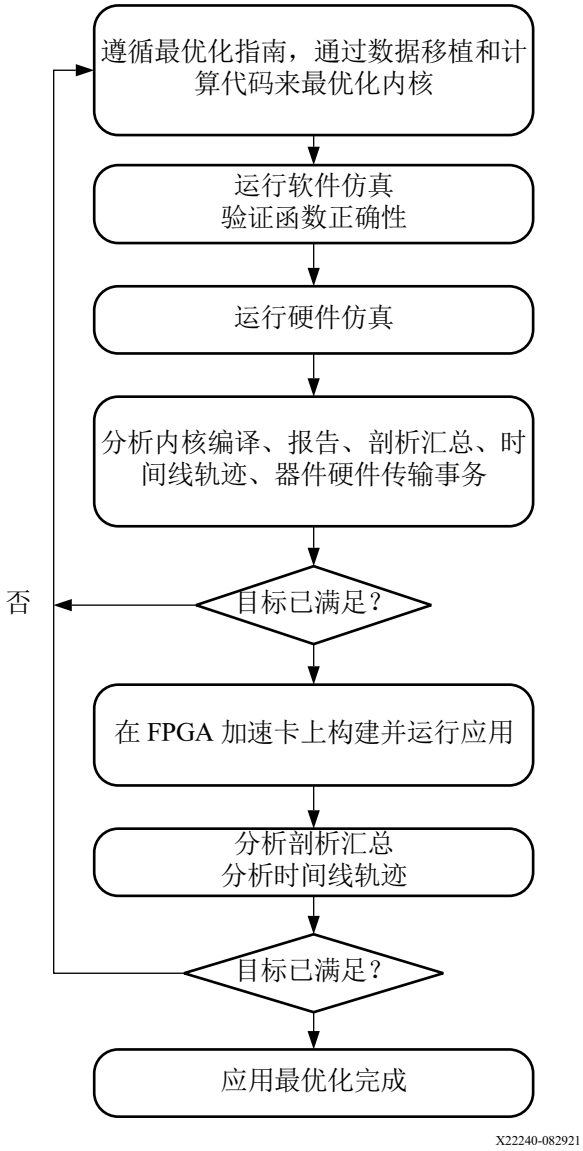


内核最优化

FPGA 的关键优势之一是它具备相应的能力和灵活性，能够专为您的算法创建自定义设计。这样即可提供各种实现选择，从而在算法吞吐量与功耗之间进行利弊取舍。以下准则有助于管理设计复杂性和实现所需的设计目标。

最优化内核计算

图 44：**最优化内核计算流程**

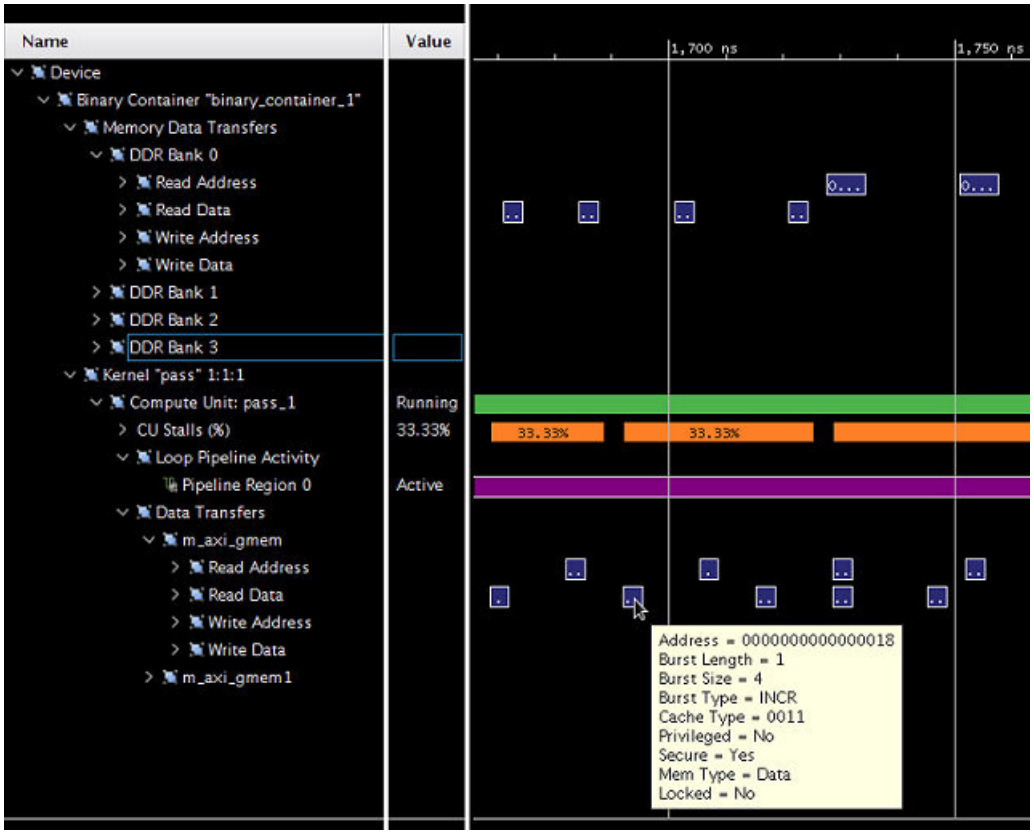


内核最优化的目标是创建处理逻辑，此逻辑可在数据到达内核接口时立即使用所有数据。此处关键指标是启动时间间隔 (II) 或者内核可接受新输入数据之前经过的时钟周期数。II 最优化的常用方法是展开处理代码，并将数据路径与诸如函数流水打拍、循环展开、阵列分区、数据流等方法搭配使用。

接口属性（详细的内核追踪）

详细的内核追踪支持轻松访问 AXI 传输事务及其属性。对于全局存储器以及 AXI Interconnect 的内核侧 (Kernel "pass" 1:1:1)，都会显示 AXI 传输事务。下图显示了一个新加速算法的典型内核追踪。

图 45：**已加速的算法内核追踪**



最值得关注的是以下性能相关字段：

- 突发长度 (Burst Length)：描述在一个传输事务内发送的数据包数量。
- 突发量 (Burst Size)：描述每个数据包中传输的字节数。

给定突发长度为 1 且每个数据包仅 4 个字节的情况下，它将需要大量独立的 AXI 传输事务来传输任意合理数量的数据。

注释：Vitis 核开发套件创建的突发量永远不会小于 4 个字节，即使发射的数据量不足 4 个字节也是如此。在此情况下，如果在未启用 AXI 突发的情况下访问多个连续项，那么可以观测到对同一地址执行多次 AXI 读取。

因此，突发长度越短以及突发量越小（远小于 512 位）的传输事务，就越适合进行接口性能最优化。

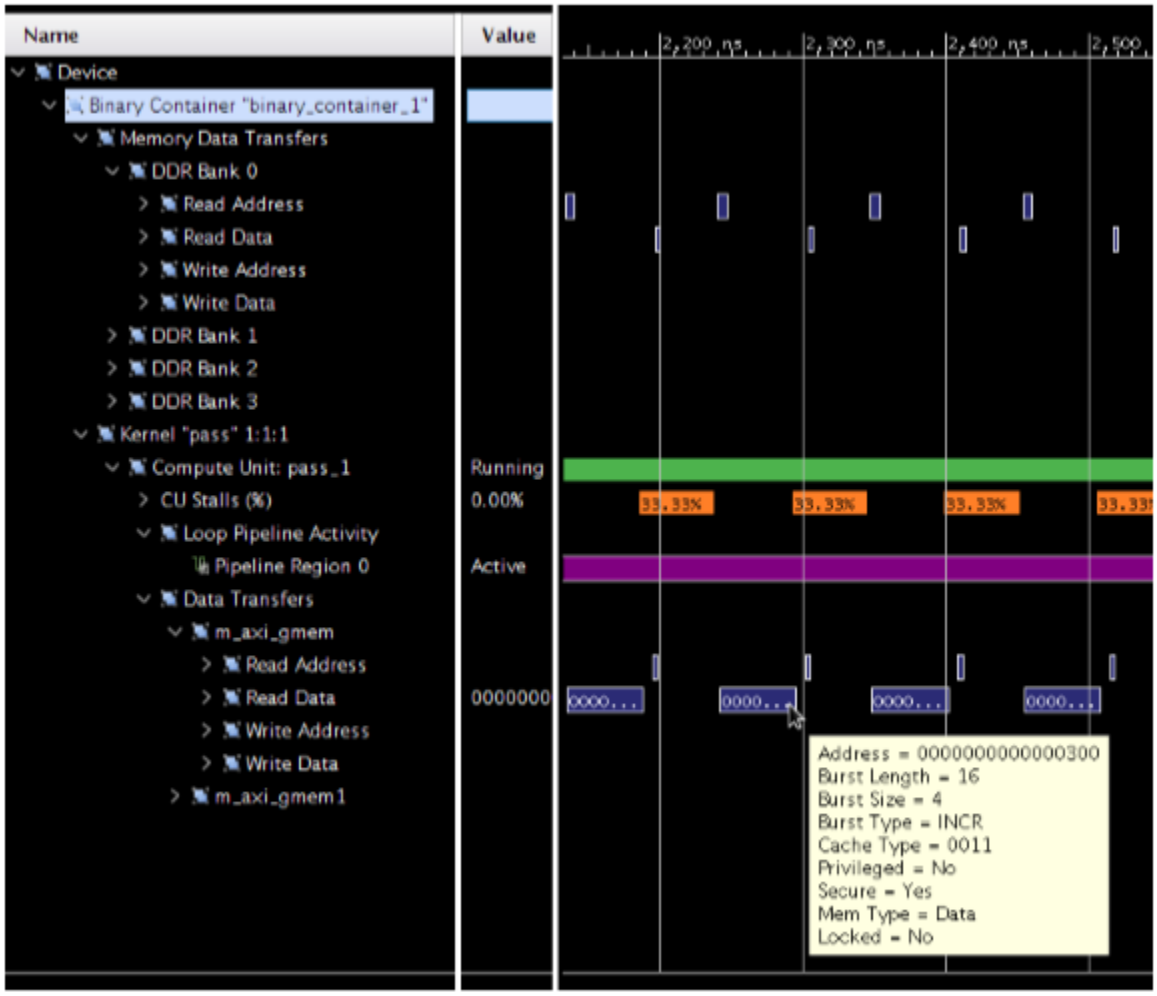
使用突发数据传输

以突发方式传输数据可隐藏存储器访问时延，并提高存储器控制器的带宽使用率和效率。同时，请检查 HLS 报告以获取突发信息。

建议：根据来自连续的地址位置的连续数据请求来推断突发传输。如需了解更多详情，请参阅 突发访问全局存储器。

如果出现突发数据传输，详细的内核追踪将把较高的突发速率反映为较大的突发长度数字：

图 46：**带详细内核追踪的突发数据传输**



在上图中，还可以观察到 AXI Interconnect 后的存储器数据传输实际上是以不同方式实现的（传输事务时间更短）。悬停于这些传输事务上时，可以看到 AXI Interconnect 已将 16 x 4 字节的传输事务封装到 1 个 1 x 64 字节的封装传输事务中。这样可以有效使用 AXI4 带宽，这种方法更有利。下一节将更加详细地讨论此最优化技术。

突发推断在很大程度上依赖于编码样式和访问模式。但是，您可以通过隔离数据传输和计算来方便突发检测和提升性能，如以下代码片段所示：

```
void kernel(T in[1024], T out[1024]) {
    T tmpIn[1024];
    T tmpOu[1024];
    read(in, tmpIn);
    process(tmpIn, tmpOut);
    write(tmpOut, out);
}
```

总之，`read` 函数负责从 AXI 输入读取到内部变量 (`tmpIn`)。计算由 `process` 函数来实现，该函数负责处理内部变量 `tmpIn` 和 `tmpOut`。`write` 函数会获取产生的输出并写入 AXI 输出。如需了解有关突发的更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399)。

将读写函数与计算隔离可带来：

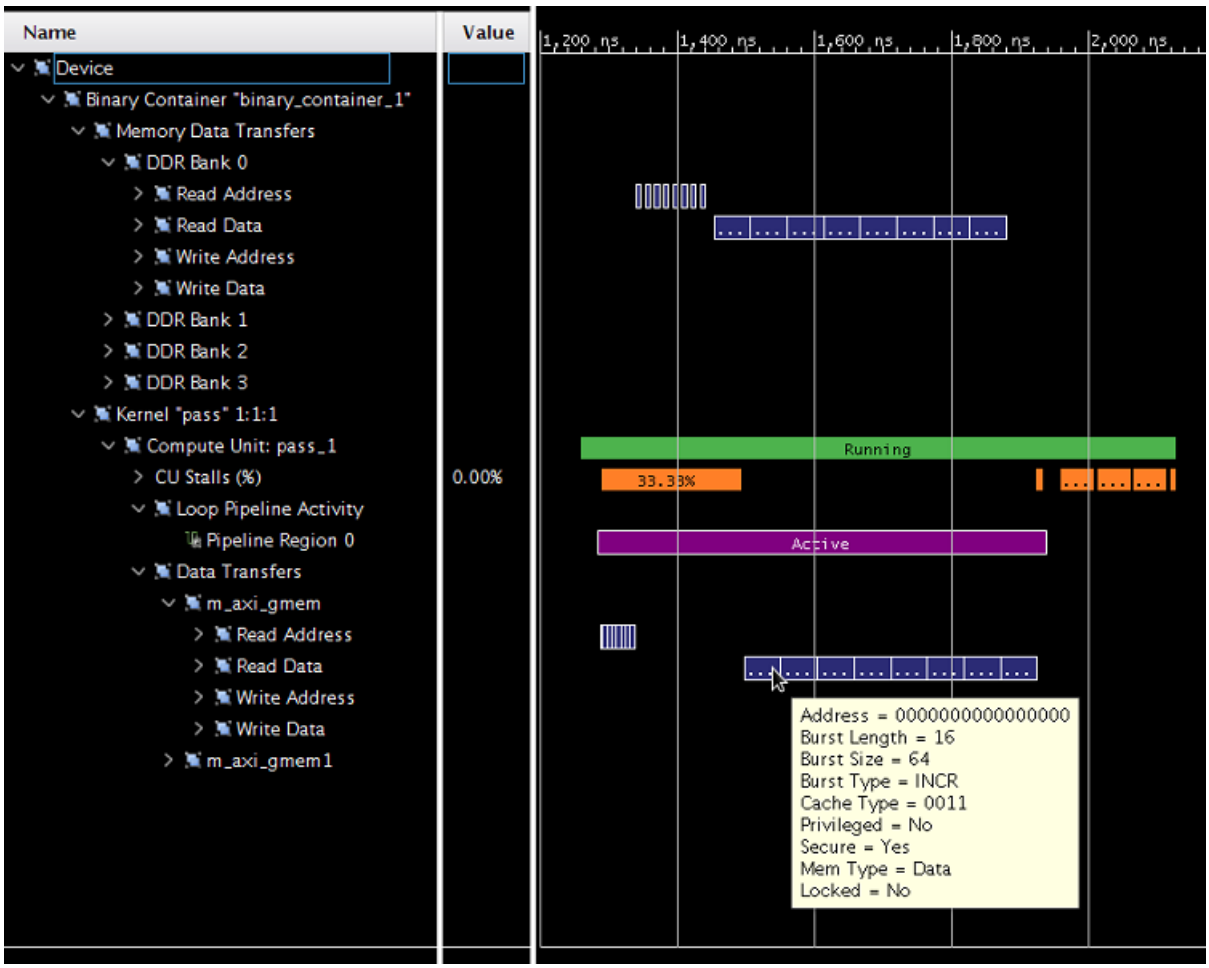
- 读取/写入函数中的简单控制结构（循环），使突发检测更简单。
- 计算函数与 AXI 接口隔离，可简化潜在的内核最优化。如需了解更多信息，请参阅 内核最优化。
- 内部变量将映射到片上存储器，相比 AXI 传输事务，这样访问速度更快。Vitis 核开发套件中支持的加速平台可具有多达 10 MB 的片上存储器，可用作管道、本地存储器和专用存储器。有效使用这些资源可极大地提升应用的效率和性能。

使用完整的 AXI 数据宽度

Vitis 编译器可根据内核实参的数据类型来配置内核与存储器控制器之间的用户数据宽度。为了最大程度提升数据吞吐量，赛灵思建议您选择映射到存储器控制器上的完整数据宽度的数据类型。所有受支持的加速卡中的存储器控制器都支持 512 位用户接口，此接口可映射到 C/C++ 任意精度数据类型 `ap_int<512>` 或 OpenCL 矢量数据类型（如 `int16`）。

如 存储器接口宽度约束 中所述，Vitis HLS 的默认行为是自动调整内核接口端口大小（最大 512 位）以改善突发访问。如下图所示，您可以观察到突发 AXI 传输事务（突发长度为 16）和 512 位封装大小（突发大小为 64 字节）。

图 47：**突发 AXI 传输事务**



此示例演示的是良好的接口配置，因为它最大程度提升了 AXI 数据宽度和实际突发传输事务。

由于存储器布局和数据打包差异，用于声明接口的复杂结构体或类可能导致硬件接口极为复杂。这可能会带来在复杂系统中很难调试的潜在问题。

建议：对于可打包到 32 位边界的内核实参，请使用简单的结构体。请参阅 GitHub 上的赛灵思快速入门示例的《kernel_to_gmem》 类别下的《定制数据类型示例》，以获取结构体的使用方法建议。

内核间通信最优化

支持硬件加速器流水线通过数据流传输来进行通信是 FPGA 和基于 FPGA 的 SoC 的主要优势之一，这在 DSP 和图像处理应用领域以及通信系统内已经得到广泛使用。如 内核之间 (K2K) 的数据流传输 中所述，AXI4-Stream 接口可用于在不同内核之间进行数据流传输，而无需使用外部存储器，这样即可显著改善总体系统时延。

数据流传输中所涉及的内核端口均在内核中定义，主机程序不会对其进行寻址。在转发数据以供处理之前，无需将数据发送回全局存储器。内核之间的连接是在 v++ 链接进程中直接定义的，如 在计算单元之间指定数据流传输连接 中所述。

最优化存储器架构

存储器架构是实现的关键。由于访问带宽有限，可能会严重影响整体性能，如以下示例所示：

```
void run (ap_uint<16> in[256][4],
          ap_uint<16> out[256]
         ) {
  ...
  ap_uint<16> inMem[256][4];
  ap_uint<16> outMem[256];

  ... Preprocess input to local memory

  for( int j=0; j<256; j++) {
    #pragma HLS PIPELINE OFF
    ap_uint<16> sum = 0;
    for( int i = 0; i<4; i++) {

      sum += inMem[j][i];
    }
    outMem[j] = sum;
  }

  ... Postprocess write local memory to output
}
```

此代码添加与二维输入阵列内部维度相关联的四个值。如果在不进行任何其它修改的情况下实现，则会产生以下估算：

图 48：**性能估算**

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5908	5908	5908	5908	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	4608	4608	18	-	-	256	no
+ Loop 2.1	16	16	4	-	-	4	no
- Loop 3	257	257	3	1	1	256	yes

4608（循环 2）的总体时延应该为 18 个周期的 256 次迭代（16 个周期消耗在内部循环中，加上总和复位，加上写入的输出）。此结果可在 HLS 工程的调度查看器中查看。当展开内部循环时，该估值结果会有明显改善。

图 49：**性能估算**

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2580	2580	2580	2580	none

Detail

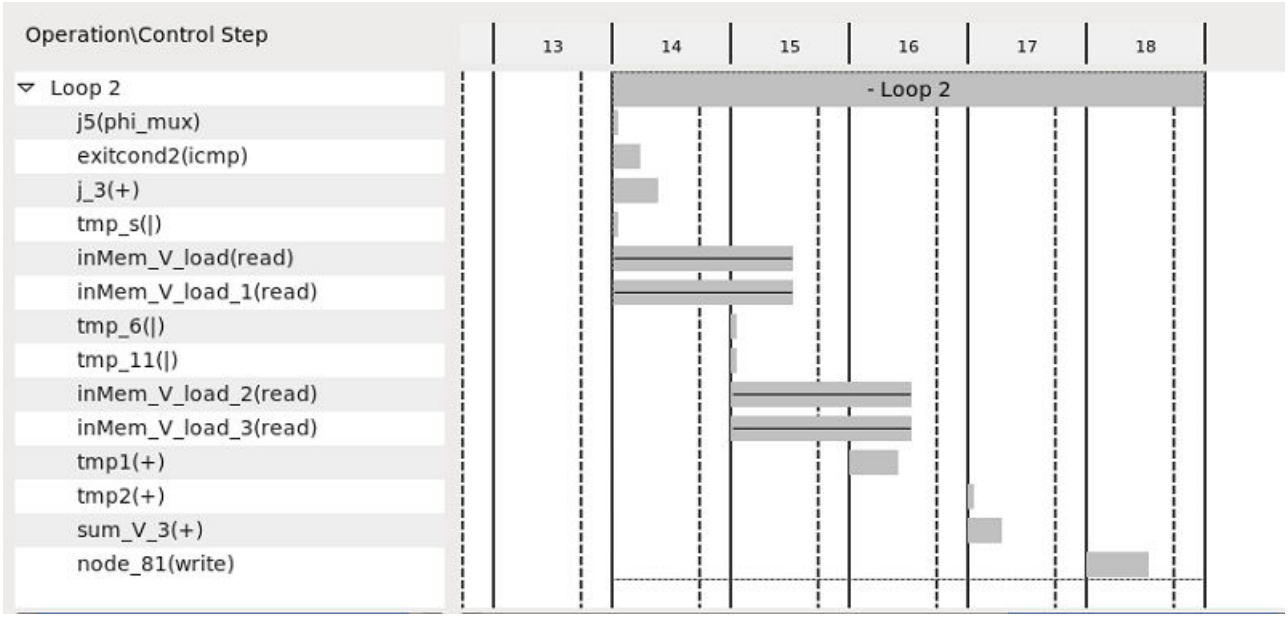
Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	1280	1280	5	-	-	256	no
- Loop 3	257	257	3	1	1	256	yes

但此改善主要原因在于使用了双端口存储器的两个端口。此结果可在 HLS 工程的调度查看器中查看。

图 50：**调度查看器**



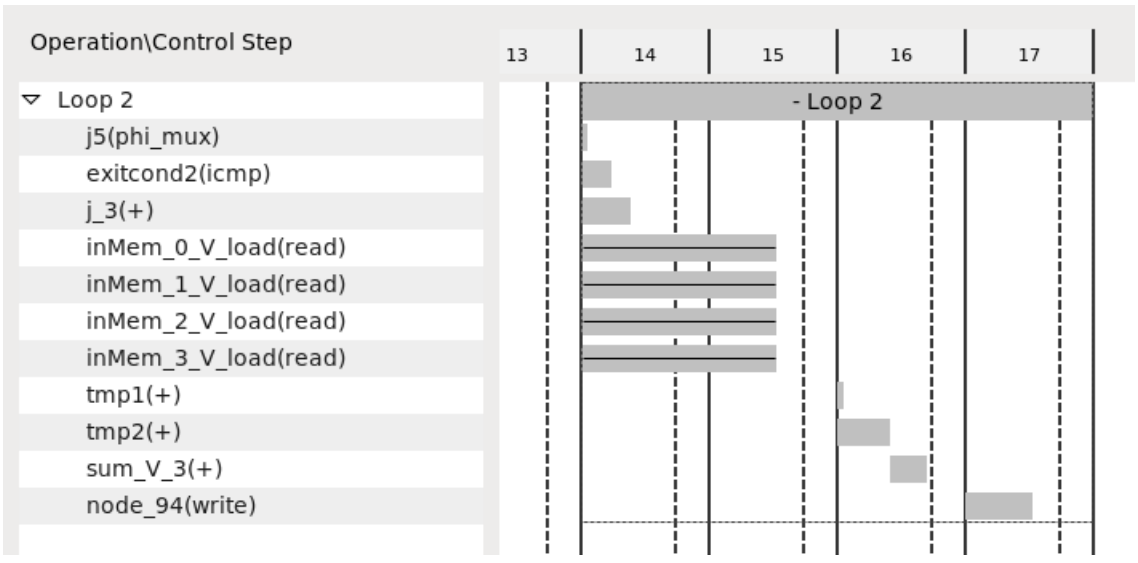
每个周期执行两次读取操作，以访问来自该存储器的所有值并计算总和。这通常并非期望的结果，因为这样完全阻塞了对于存储器的访问。为了进一步改善结果，可将该存储器沿第二个维度拆分为 4 个更小的存储器。

```
#pragma HLS ARRAY_PARTITION variable=inMem complete dim=2
```

如需了解更多信息，请参阅 pragma HLS array_partition。

这样会导致 4 次阵列读取，并且读取操作全部在使用单一端口的不同存储器上执行：

图 51：**执行 4 个阵列的结果**



使用的周期总数为 256 * 4 = 1024 个周期（循环 2）。

图 52：**性能估算**

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2324	2324	2324	2324	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	1024	1024	4	-	-	256	no
- Loop 3	257	257	3	1	1	256	yes

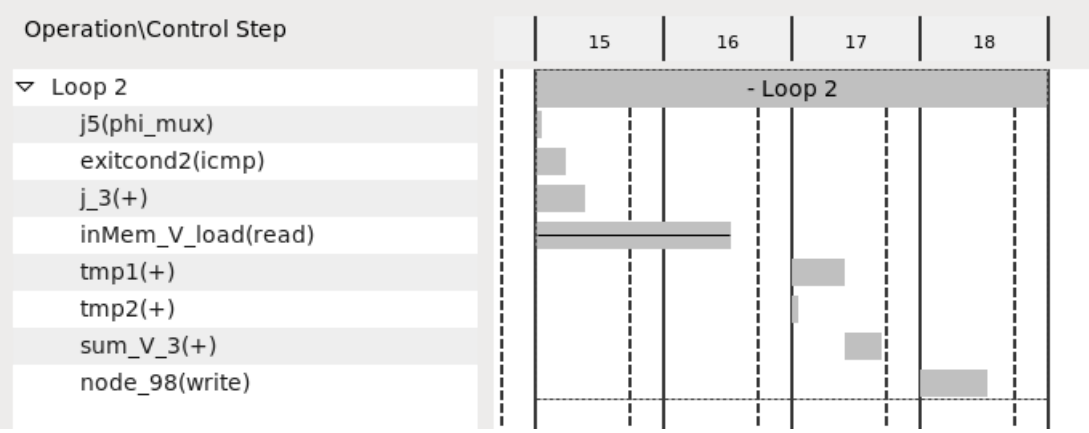
或者，存储器可以重构为带有 4 个并行字的单个存储器。这可通过如下编译指示来执行：

```
#pragma HLS array_reshape variable=inMem complete dim=2
```

如需了解更多信息，请参阅 pragma HLS array_reshape。

这样会导致与阵列分区相同的时延，但是只有一个存储器且使用单个端口：

图 53：**时延结果**



虽然上述任何一种解决方案在总体时延和利用率上可产生类似的结果，但是重塑阵列可带来更清洁的接口和更少的布线拥塞，因而使其成为首选解决方案。

注释：这样即可完成阵列最优化，在实际设计中，可通过利用循环并行化来进一步改善时延（请参阅 循环并行化）。

```
void run (ap_uint<16> in[256][4],
      ap_uint<16> out[256]
      ) {
  ...

  ap_uint<16> inMem[256][4];
  ap_uint<16> outMem[256];
  #pragma HLS array_reshape variable=inMem complete dim=2

  ... Preprocess input to local memory

  for( int j=0; j<256; j++) {
    #pragma HLS PIPELINE OFF
    ap_uint<16> sum = 0;
    for( int i = 0; i<4; i++) {
      #pragma HLS UNROLL
      sum += inMem[j][i];
    }
    outMem[j] = sum;
  }

  ... Postprocess write local memory to output

}
```

最优化计算并行度

默认情况下，C/C++ 不会为计算并行度建模，因为它始终按顺序执行任何算法。然而，FPGA 之类的可完全配置的计算引擎支持以更自由的方式来利用计算并行度。

对数据并行度进行编码

需要指出的是，在 FPGA 上实现算法期间，为了充分利用计算并行度，综合工具需能够首先从源代码中识别计算并行度。为了反映源描述中的计算并行度和计算单元，主要使用的是循环和函数。但是即使如此，验证实现能否充分利用计算并行度仍至关重要，因为在某些情况下，Vitis 技术可能因源代码结构问题而无法应用所需的变换。

某些计算并行度可能一开始就未反映在源代码中，这是很常见的。在此情况下，需要在源代码中添加计算并行度。典型示例如下：内核可能基于单一输入值来运行，而 FPGA 实现则可能基于多个值来并行执行计算时更有效。如需了解此类并行建模的描述，请参阅 任务并行化。

使用 OpenCL 矢量数据类型（如 `int16`）或 C/C++ 任意精度数据类型 `ap_int<512>` 可创建 512 位接口。这些矢量类型也可用作为在内核内部进行数据并行度建模的有效方法，使用 `int16` 的情况下最多将有 16 条数据路径并行运行。请参阅 GitHub 上赛灵思快速入门示例下的《愿景》类别中的《中值滤波器示例》，以了解有关使用矢量的推荐方法。

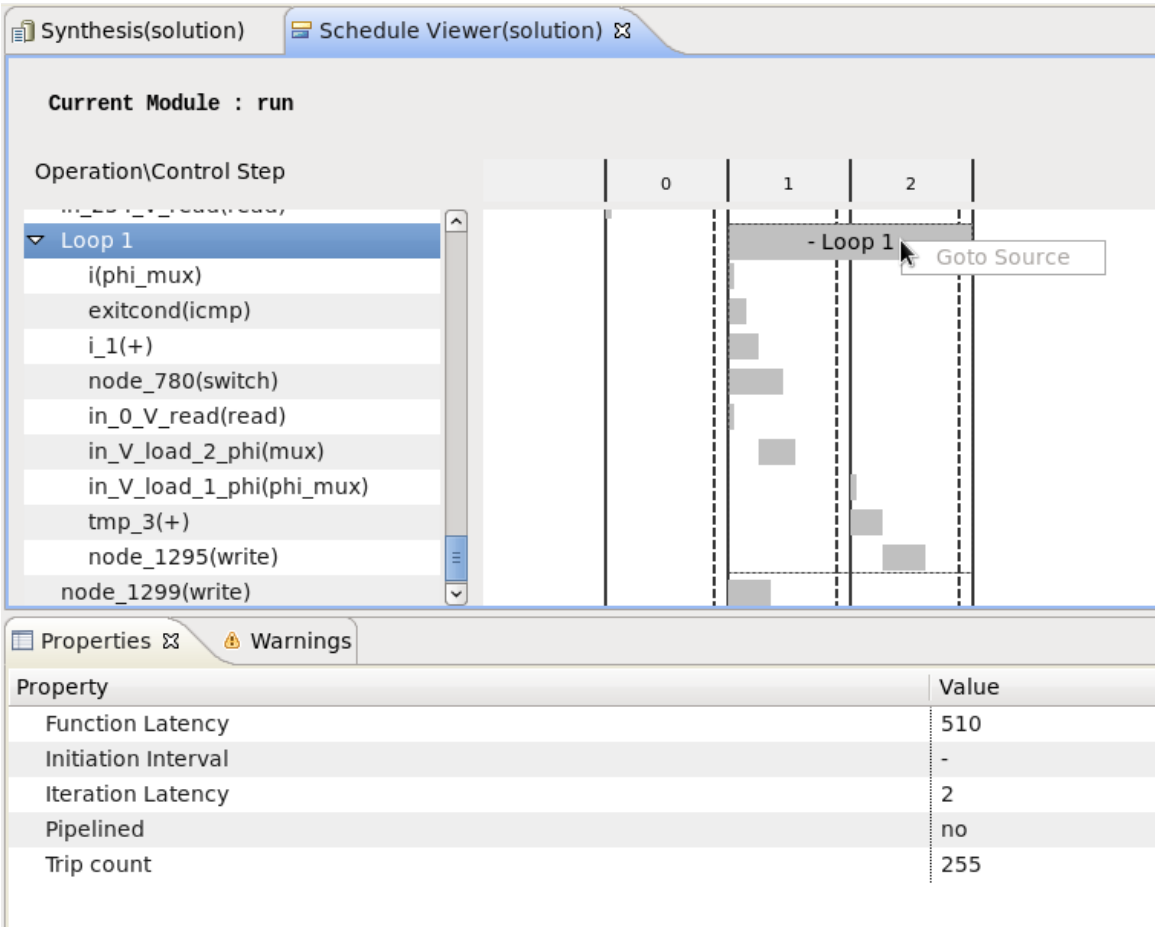
循环并行化

循环是表示重复算法代码的基本 C/C++/OpenCL API 方法。以下示例展示了循环结构的各个实现方面：

```
  for(int i = 0; i<255; i++) {
    out[i] = in[i]+in[i+1];
  }
  out[255] = in[255];
```

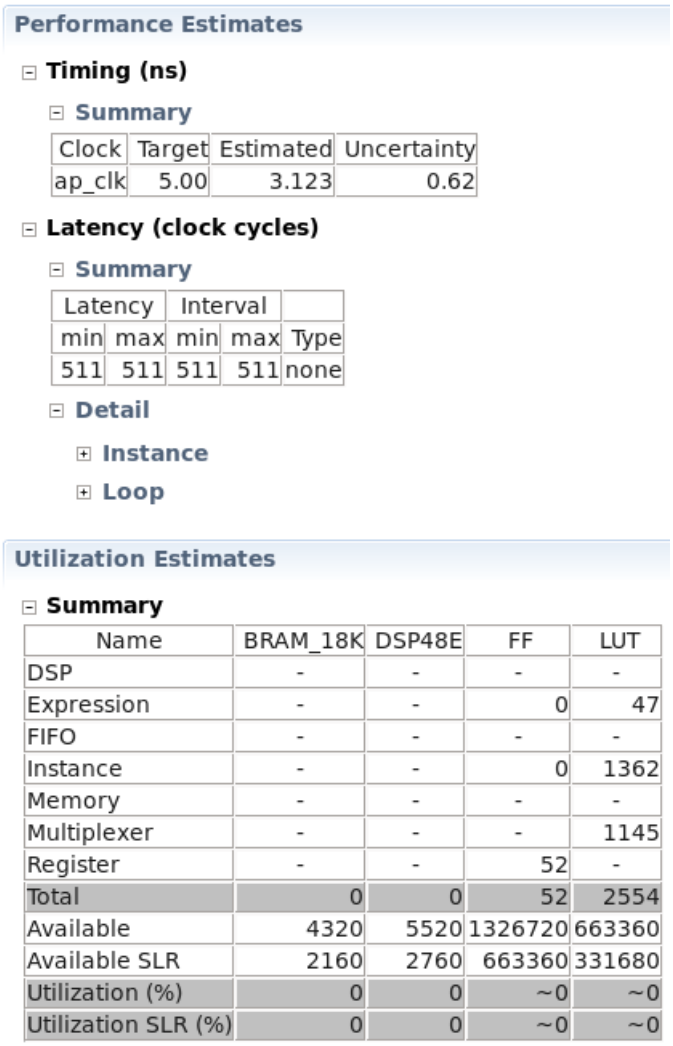
此代码基于值阵列进行迭代并添加连续值，最后一个值除外。如果此循环作为写入来执行，则每个循环迭代需要两个实现周期，这将导致总共 510 个实现周期。可以通过 HLS 工程的调度查看器对此进行详细分析。

图 54：**调度查看器中已实现的循环结构**



也可通过 Vivado 综合结果对总数和时延进行分析：

图 55：**综合结果性能估算**



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
511	511	511	511	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	47
FIFO	-	-	-	-
Instance	-	-	0	1362
Memory	-	-	-	-
Multiplexer	-	-	-	1145
Register	-	-	52	-
Total	0	0	52	2554
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

此处的关键数值是时延数量和 LUT 使用总量。例如，根据配置，结果可能是时延数量为 511，LUT 使用总量为 47。因此，这些值可能根据所选实现而异。虽然此实现所需面积可能非常小，但可能产生巨大的时延。

展开循环

展开循环能为要使用的模型启用完全并行化。要执行此操作，请标记要展开的循环，该工具将以尽可能最大的并行化来创建实现。要标记要展开的循环，可以使用 UNROLL 属性来标记 OpenCL 循环：

```
__attribute__((opencl_unroll_hint))
```

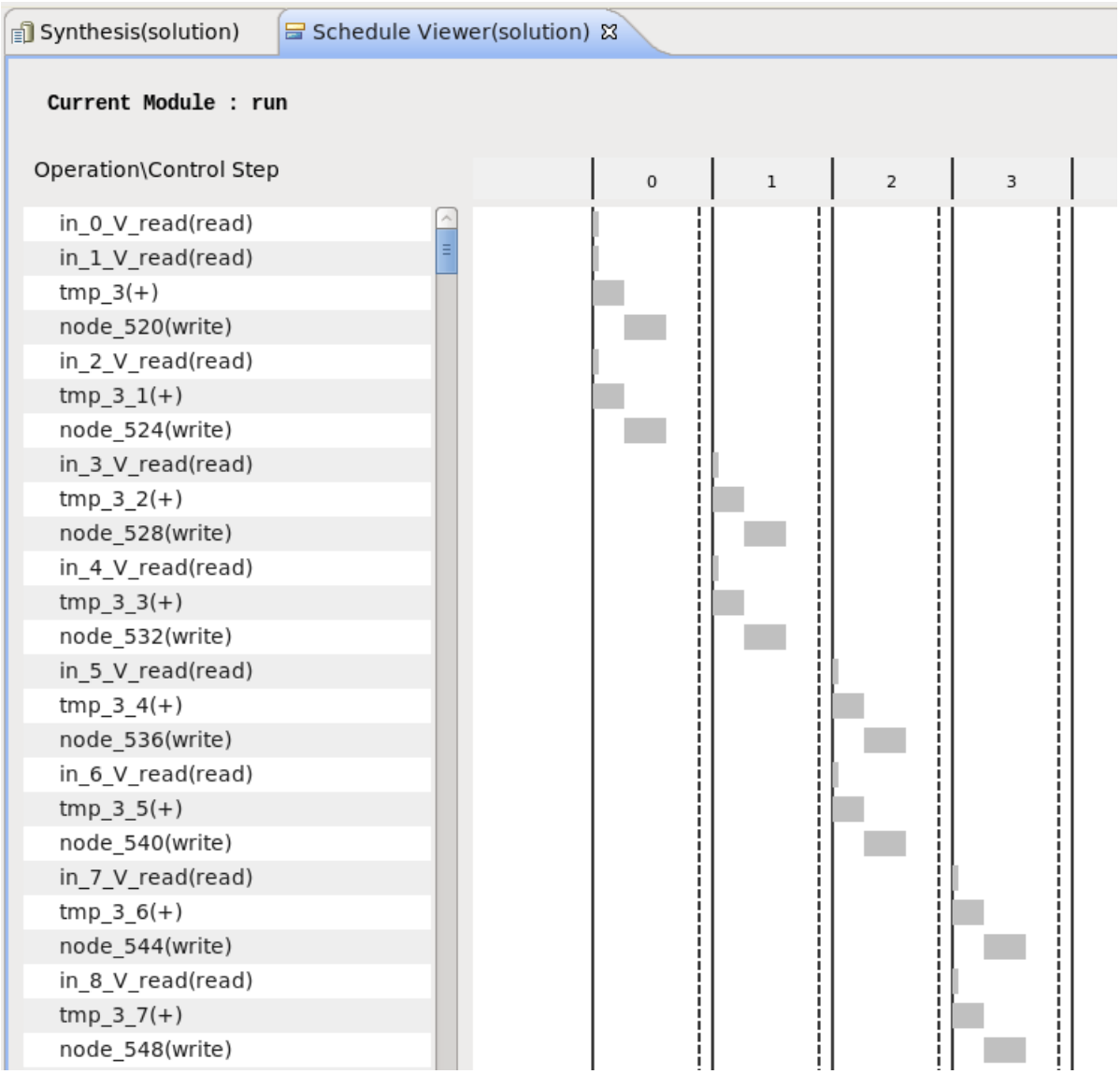
或者 C/C++ 循环可以使用 unroll 编译指示：

```
#pragma HLS UNROLL
```

如需了解更多信息，请参阅 循环展开。

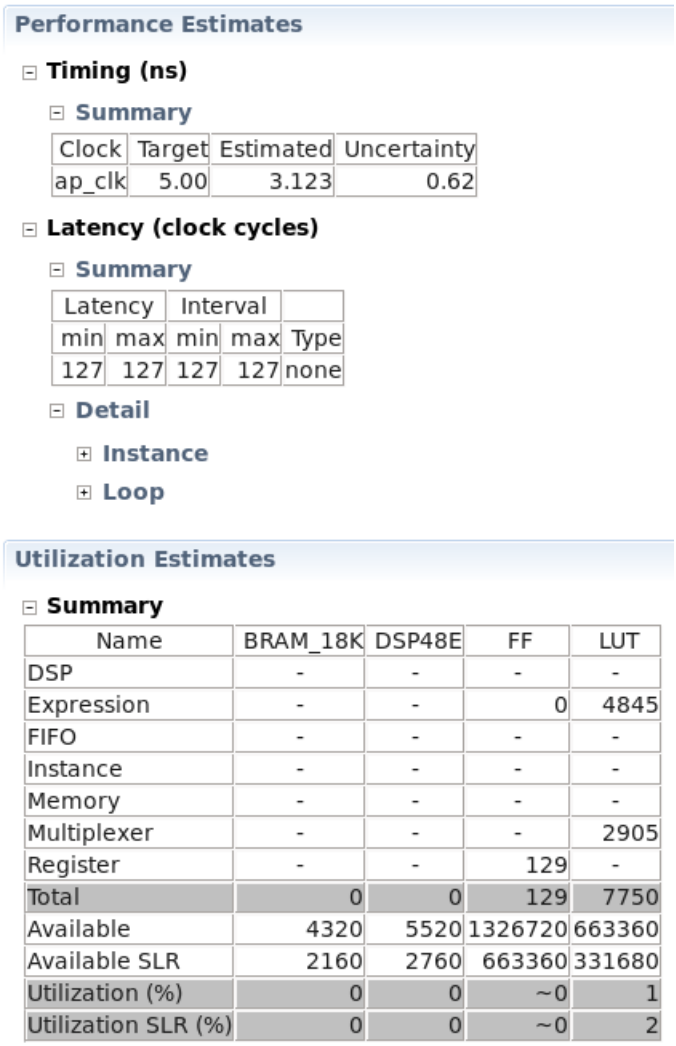
HLS 工程中的“调度查看器 (Schedule Viewer)”应用于此特定示例后，如下所示：

图 56：**调度查看器**



下图显示了估算的性能：

图 57：**性能估算**



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
127	127	127	127	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4845
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	2905
Register	-	-	129	-
Total	0	0	129	7750
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	1
Utilization SLR (%)	0	0	~0	2

由此可见，总时延得到了显著改善，缩短至达 127 个周期，且正如预期，计算硬件增加至 4845 个 LUT，这样即可并行执行相同的计算。

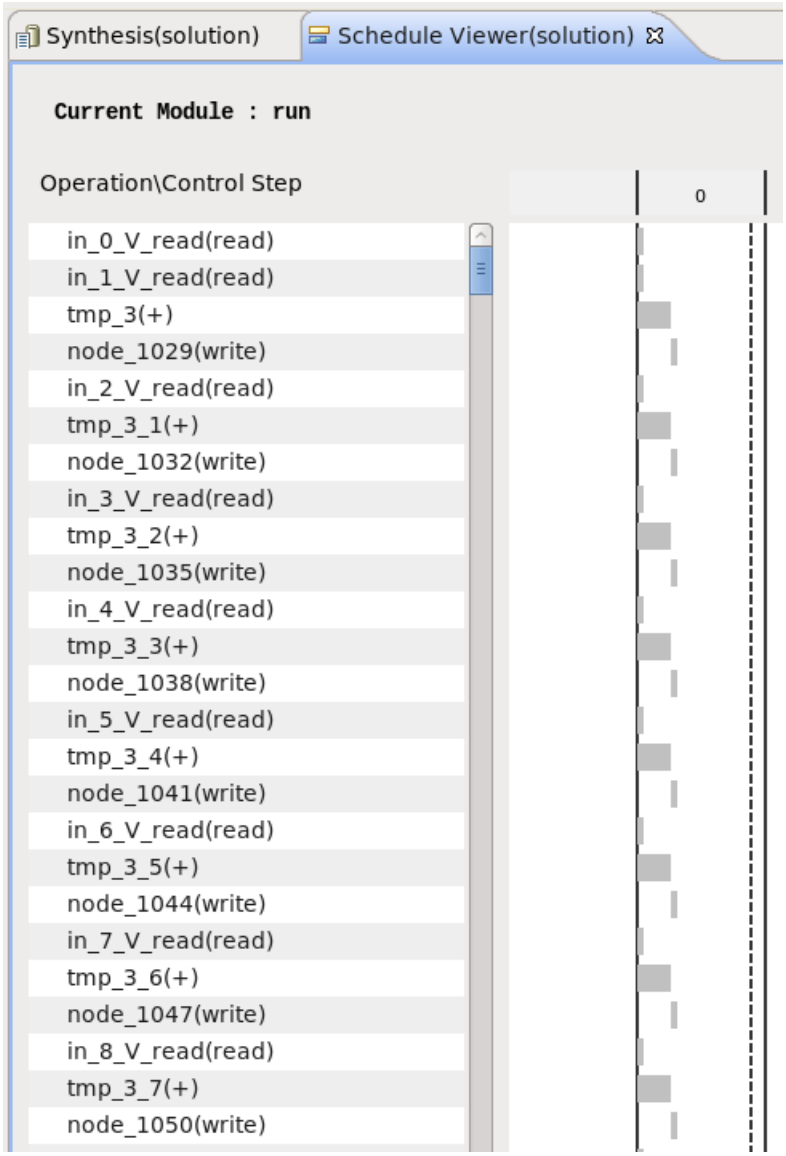
但是，如果您分析 for 循环，您可能会问，为什么此算法不能在单个周期中实现，因为每个加法都完全独立于前面的循环迭代。其原因在于，存储器接口将用于 `out` 变量。Vitis 核开发套件默认针对任一阵列使用双端口存储器。但是，这表示每个周期最多可将两个值写入存储器。因此，要达成完全并行实现，必须指定在寄存器中应保留 `out` 变量，如下示例所示：

```
#pragma HLS array_partition variable= out complete dim= 0
```

如需了解更多信息，请参阅 pragma HLS array_partition。

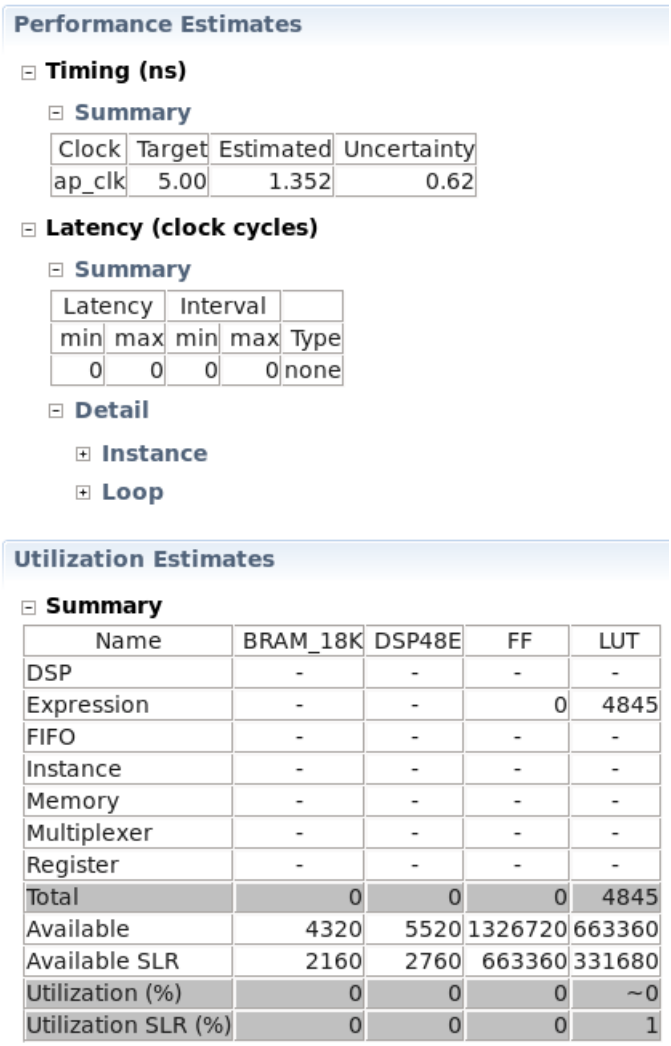
此变换的结果可在以下调度查看器中进行观测：

图 58：**调度查看器中的变换结果**



相关联的估算为：

图 59：**变换结果性能估算**



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	1.352	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	0	0	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4845
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	4845
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	0	~0
Utilization SLR (%)	0	0	0	1

因此，此代码可作为组合函数来实现，完成此代码所需的周期数大幅减少。

循环流水打拍

循环流水打拍允许您将一段时间内的循环迭代重叠，如 循环流水打拍 中所述。允许循环迭代并发运行通常是个好办法，因为可在迭代之间共享资源（降低资源利用率），同时所需执行时间比未展开的循环所需时间更短。

在 C/C++ 中通过 pragma HLS pipeline 启用流水打拍：

```
#pragma HLS PIPELINE
```

当 OpenCL API 使用 xcl_pipeline_loop 属性时：

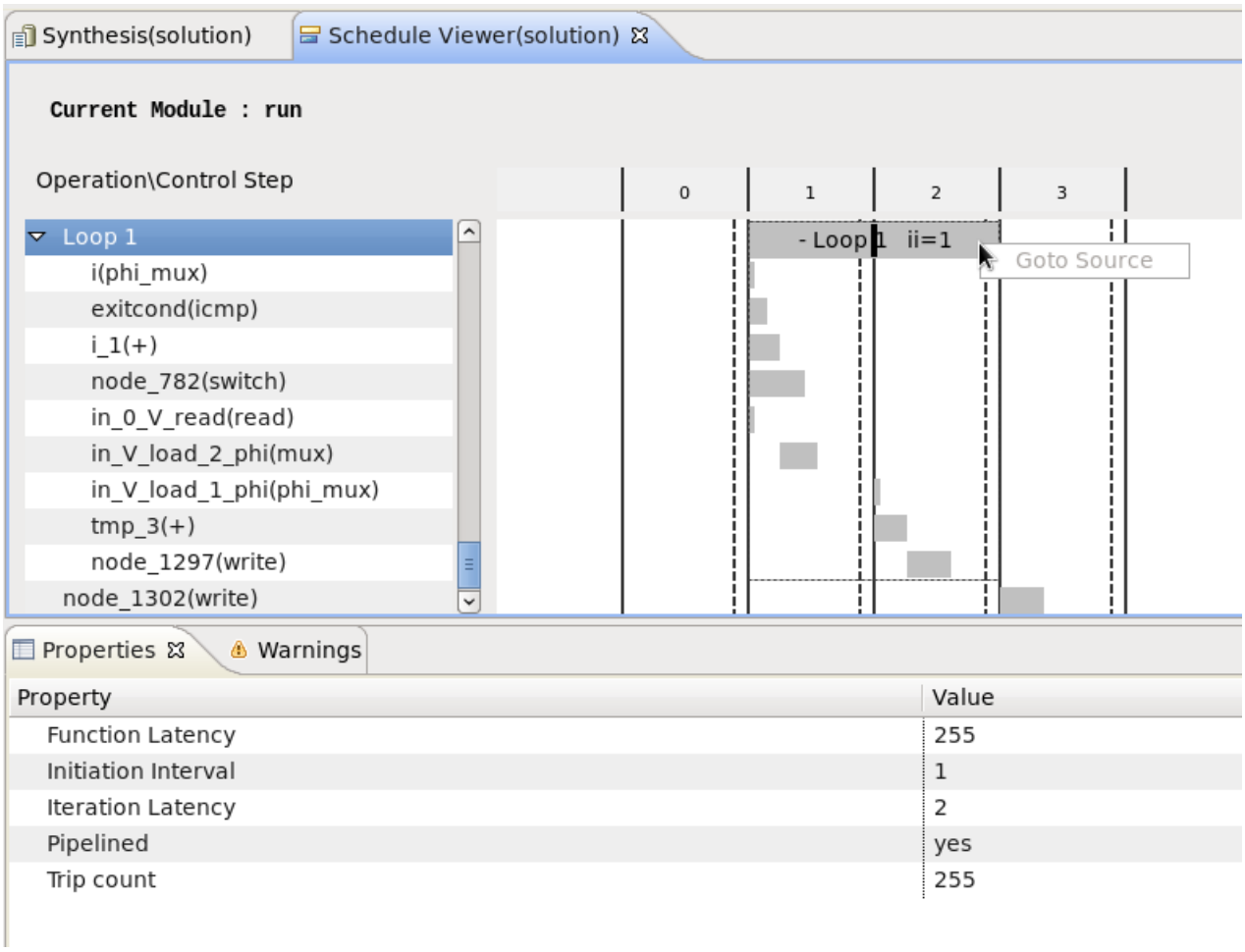
```
__attribute__((xcl_pipeline_loop))
```

注释：OpenCL API 还有另一种指定循环流水打拍的方法，请参阅 xcl_pipeline_workitems。原因在于工作项循环并未被明确声明，对这些循环进行流水打拍需要以下属性：

```
__attribute__((xcl_pipeline_workitems))
```

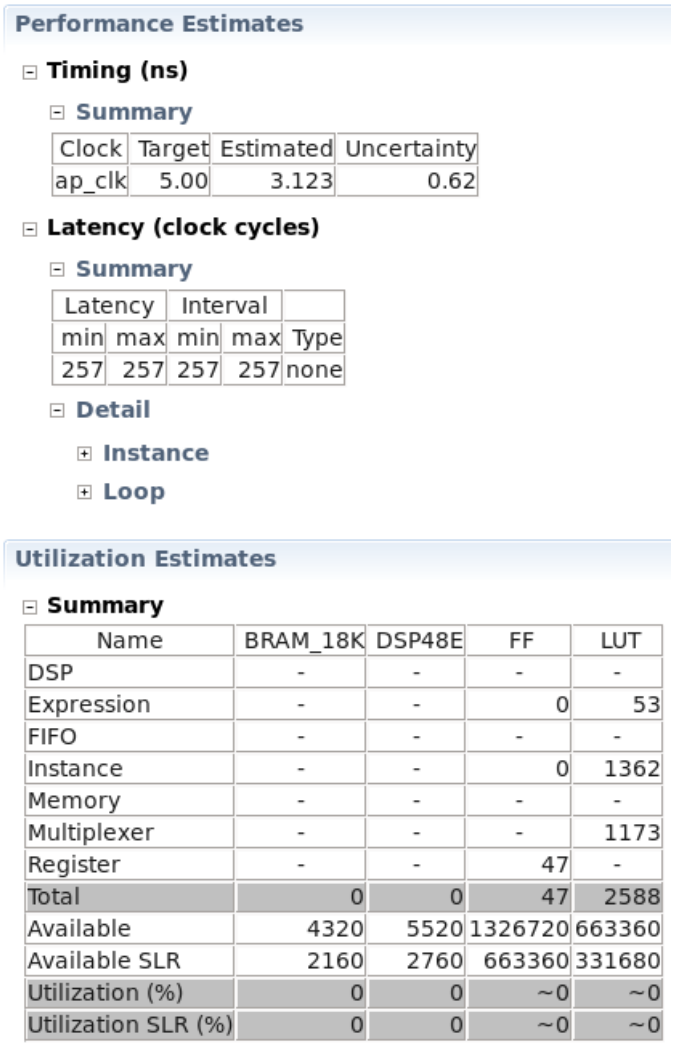
在此示例中，“HLS 工程 (HLS Project)”中的“调度查看器 (Schedule Viewer)”可产生以下信息：

图 60：**“Schedule Viewer”中的循环流水打拍**



其中总体估算为：

图 61：**性能估算**



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
257	257	257	257	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	53
FIFO	-	-	-	-
Instance	-	-	0	1362
Memory	-	-	-	-
Multiplexer	-	-	-	1173
Register	-	-	47	-
Total	0	0	47	2588
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

由于循环的每次迭代仅耗用两个时延周期，因此只能有单个迭代重叠。这样使总时延相比原时延减半，最终总时间为 257 个周期。由此不仅降低了时延，而且所占用的资源相比循环展开时所占用的资源更少。

在多数情况下，循环流水打拍本身可提升总体性能。但是流水打拍的有效性取决于循环结构。部分常用迭代为：

- 如果存储器端口或进程通道之类的资源可用性受限，则可能限制迭代的重叠（启动时间间隔）。
- 例如，由任一迭代中计算所得的变量条件会影响下一个迭代，诸如此类的循环进位依赖关系可能会导致流水线的启动时间间隔 (II) 增加。

在高层次综合期间，该工具会报告这些影响，并且可在“Schedule Viewer”中对其进行观测和检验。为了尽可能提升性能，可能需要修改代码以移除这些限制因素，或者可能需要指示工具通过重构阵列的存储器实现来消除部分依赖关系或者切断全部依赖关系。

任务并行化

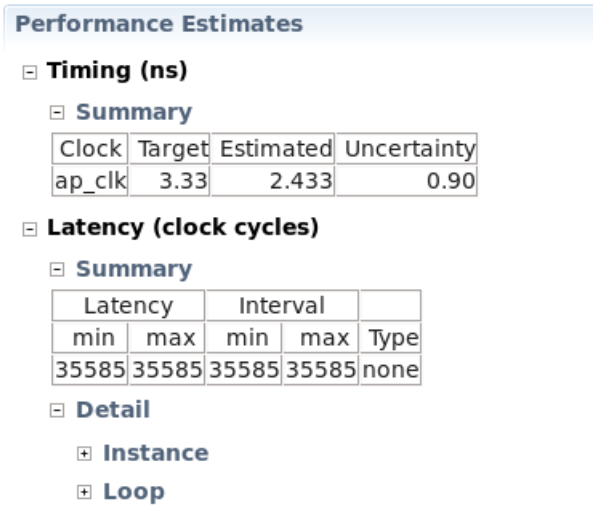
任务并行化使您能够充分利用数据流并行化。与循环并行化相反，当部署任务并行化后，即可允许完全执行单元（任务）并行运行，从而充分利用任务之间带来的额外缓冲。

请参阅以下示例：

```
void run (ap_uint<16> in[1024],
      ap_uint<16> out[1024]
      ) {
  ap_uint<16> tmp[128];
  for(int i = 0; i<8; i++) {
    processA(&(in[i*128]), tmp);
    processB(tmp, &(out[i*128]));
  }
}
```

执行此代码时，processA 函数和 processB 函数将按顺序连续执行 128 次。如果 processA 与 processB 存在组合时延，该循环设置为 278 且总时延可通过如下方式来估算：

图 62：**性能估算**



额外的周期是因循环建立而产生，并可在“调度查看器 (Schedule Viewer)”中查看。

对于 C/C++ 代码，可通过将 DATAFLOW 编译指示添加到 for 循环中来执行任务并行化：

```
#pragma HLS DATAFLOW
```

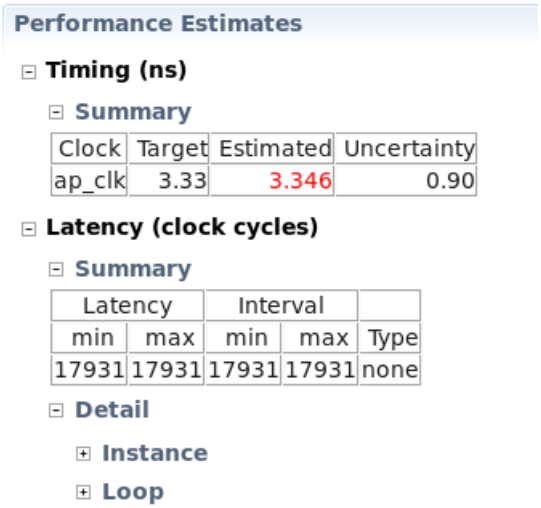
对于 OpenCL API 代码，请将该属性添加到 for 循环之前：

```
__attribute__ ((xcl_dataflow))
```

如需了解有关该主题的更多信息，请参阅 数据流最优化、HLS 编译指示 和 OpenCL 属性。

如 HLS 报告中的估算所示，应用此变换将能够通过在任务之间使用双（乒乓）缓冲器方案来有效地显著提升总体性能。

图 63：**性能估算**

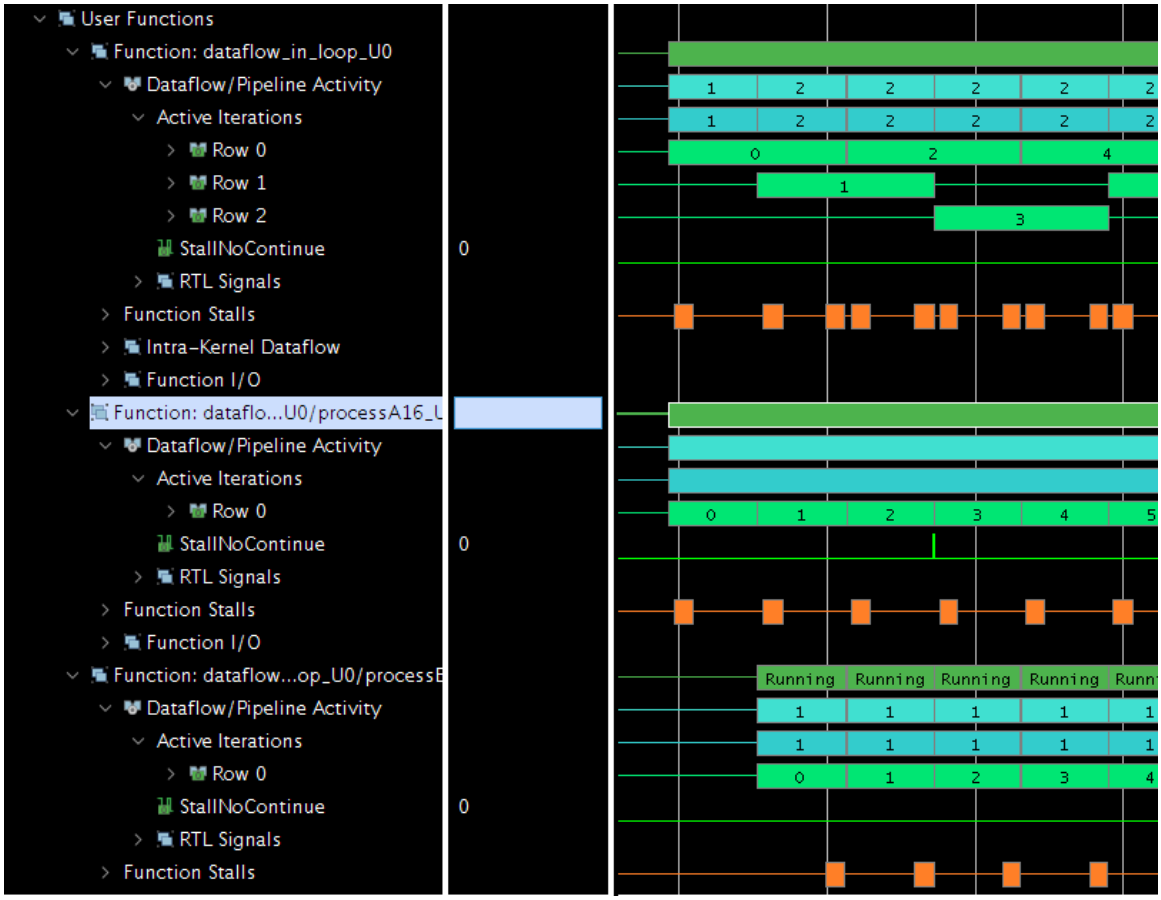


在此案例中，由于并发执行不同迭代的不同任务，使设计的总体时延几乎减半。考虑到每个处理函数有 139 个循环，且 128 个迭代完全重叠，由此所得总时延为：

```
(1x only processA + 127x both processes + 1x only processB) * 139 cycles =
17931 cycles
```

使用任务并行度是一种在实现时提升性能的强大方法。但是，将 DATAFLOW 编译指示应用于代码的特定片段和应用于任意片段的效果可能截然不同。通常有必要通过观察个别任务的执行模式来了解 DATAFLOW 编译指示的最终实现结果。最后，Vitis 核开发套件提供了“Detailed Kernel Trace”用于展示并发执行情况。

图 64：**详细的内核追踪**



对于此“Detailed Kernel Trace”，该工具会显示数据流循环的启动，如上图所示。它会演示 `processA` 在循环开始时立即启动，而 `processB` 则等待至 `processA` 完成后才启动其首次迭代的方式。但在 `processB` 完成循环的首次迭代的同时，`processA` 就会开始操作第二次迭代，以此类推。

在 时间线轨迹 中为主机和器件活动提供了有关此信息的更抽象的演示。

器件资源最优化

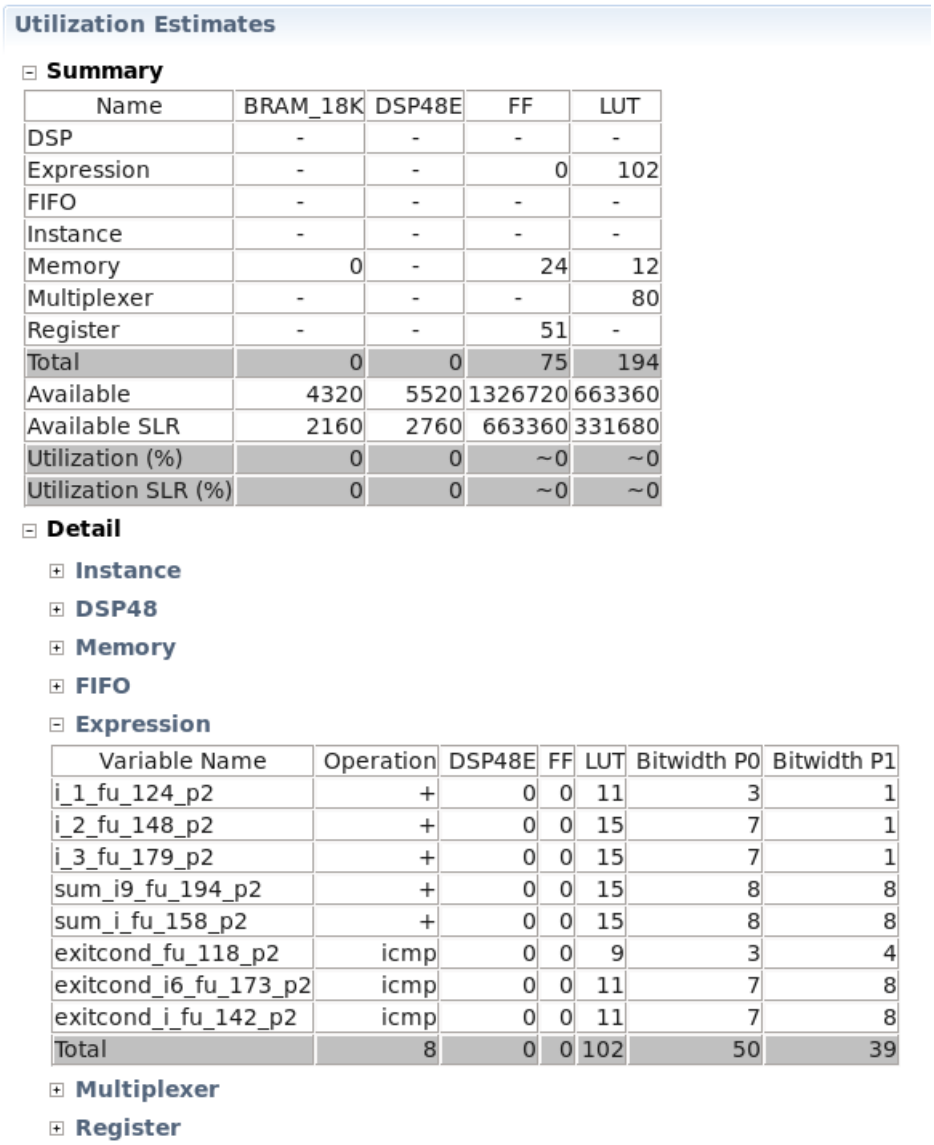
数据宽度

实现所需的数据宽度是性能最重要的方面之一。该工具在整个算法过程中会传输端口宽度。在某些情况下（尤其是以算法描述为起点时），C/C++/OpenCL 代码可能仅使用大型数据类型（如整数），即使在设计的端口处也是如此。但由于算法已映射到完全可配置的实现，因此有时 10 位或 12 位的小型数据类型可能就足够了。在最优化期间，最好检查“HLS 综合 (HLS Synthesis)”报告中的基本运算大小。

总之，当 Vitis 核开发套件将算法映射到 FPGA 上时，需要进一步的处理才能理解 C/C++/OpenCL API 结构并提取运算之间的依赖关系。因此，为了执行此映射，Vitis 核开发套件通常会将源代码分区为不同运算单元，然后将其映射到 FPGA 上。通过该工具可以发现，这些运算单元 (ops) 的数量和大小受到多方面的影响。

下图报告了基本运算及其位宽。

图 65：**运算利用率估算结果**



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	102
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	24	12
Multiplexer	-	-	-	80
Register	-	-	51	-
Total	0	0	75	194
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

Detail

Instance

DSP48

Memory

FIFO

Expression

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
i_1_fu_124_p2	+	0	0	11	3	1
i_2_fu_148_p2	+	0	0	15	7	1
i_3_fu_179_p2	+	0	0	15	7	1
sum_i9_fu_194_p2	+	0	0	15	8	8
sum_i_fu_158_p2	+	0	0	15	8	8
exitcond_fu_118_p2	icmp	0	0	9	3	4
exitcond_i6_fu_173_p2	icmp	0	0	11	7	8
exitcond_i_fu_142_p2	icmp	0	0	11	7	8
Total	8	0	0	102	50	39

Multiplexer

Register

查找算法描述中常用的位宽（16 位、32 位和 64 位），并验证来自 C/C++/OpenCL API 源代码的关联运算确实需要这样大的位宽。这样可以显著改善算法实现，因为运算越小，所需的计算时间越短。

定点运算

某些应用使用浮点运算的原因只是因为这些应用是专为其它硬件架构而最优化的。针对深度学习之类的应用使用定点算法可以显著节省能耗和面积，同时保持精度等级不变。

建议：赛灵思建议在决定使用浮点运算之前，请先为您的应用探索使用定点运算的可能性。

宏运算

考虑更大的计算元素有时非常有利。该工具将在独立于其余的源代码的源代码上运行，有效地将算法映射到 FPGA 上，而无需考虑周围的运算。应用 Vitis 技术时，此技术可以保留运算边界，从而为特定代码有效创建宏运算。这使用了以下原理：

- 映射进程的运算区域
- 降低启发式方法的复杂性

应用此技术可能会产生巨大的不同结果。在 C/C++ 中，宏运算是借助 `#pragma HLS inline off` 的帮助来创建的。而在 OpenCL API 中，通过在定义函数时不指定以下属性即可生成同样种类的宏运算：

```
__attribute__((always_inline))
```

如需了解更多信息，请参阅 pragma HLS inline。

使用最优化的库

OpenCL 规范提供了许多内置数学函数。含 `native_` 前缀的所有内置数学函数都映射到一条或多条本地器件指令，并且与不含 `native_` 前缀的对应函数相比，通常性能更优。这些函数的精度和输入范围（某些情况下）由实现来定义。在 Vitis 技术中，这些 `native_` 内置函数使用 Vitis HLS 工具的数学库中的等效函数，这些等效函数在面积和性能方面都已针对赛灵思 FPGA 经过了最优化。

建议：赛灵思建议使用 `native_` 内置函数或 HLS 工具的数学库，前提是其精度能够满足应用要求。

探索如何使用 Vitis HLS 来进行内核最优化

使用 OpenCL 或 C/C++ 进行的所有内核最优化均可在 Vitis 核开发套件内执行。主要的性能最优化措施（如本章节所述的函数和循环流水打拍、应用数据流提升函数和循环之间的并发度、以及展开循环等）均由 Vitis HLS 工具来执行。

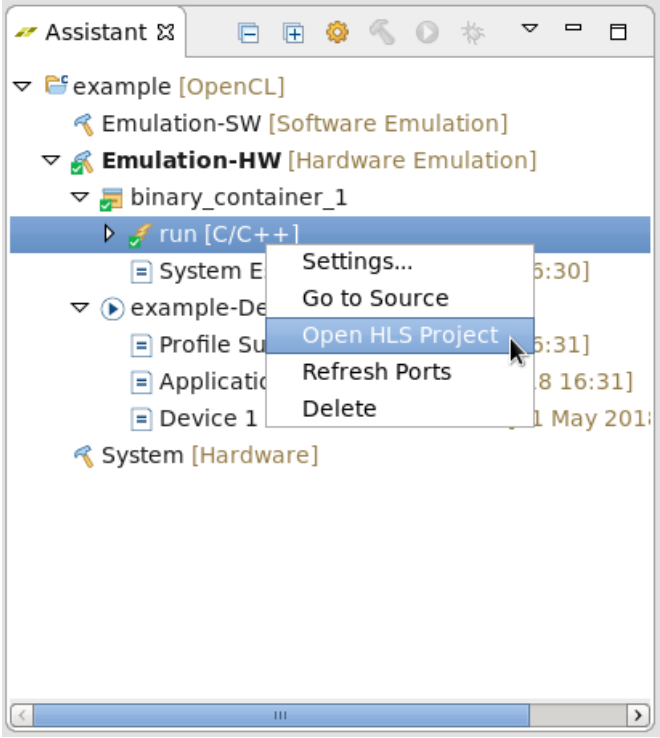
Vitis 核开发套件会自动调用 HLS 工具。但是，要使用 GUI 分析功能，您必须在 Vitis 内直接启动 HLS 工具。以独立模式使用 HLS 工具（如 使用 Vitis HLS 编译内核 中所述）可在最优化方法的基础上实现如下几个方面的提升：

- 支持单独聚焦内核最优化，因为无需执行仿真。
- 能够创建多种解决方案、比较其结果以及通过探索解空间来查找最优设计。
- 能够使用“交互式分析透视图 (Analysis Perspective)”来分析设计性能。

重要提示！将内核源代码单独重新整合到 Vitis 核开发套件中。尝试完各种可能的最优化后，请确保将所有最优化都作为 OpenCL 属性或 C/C++ 编译指示应用到内核源代码中。

要在独立模式下打开 HLS 工具，请在“Assistant”窗口中右键单击硬件函数对象，然后选择“Open HLS Project”，如下图所示。

图 66：**打开 HLS 工程**



拓扑结构最优化

本节聚焦拓扑结构最优化。其中探讨了与多个计算单元的大体布局和实现相关的属性及其对性能的影响。

多个计算单元

根据目标器件上的可用资源，可以创建同一个内核（或不同内核）的多个计算单元以并行运行，这样可以缩短系统处理时间并提升吞吐量。欲知详情，请参阅 创建内核的多个实例。

使用多个 DDR 存储体

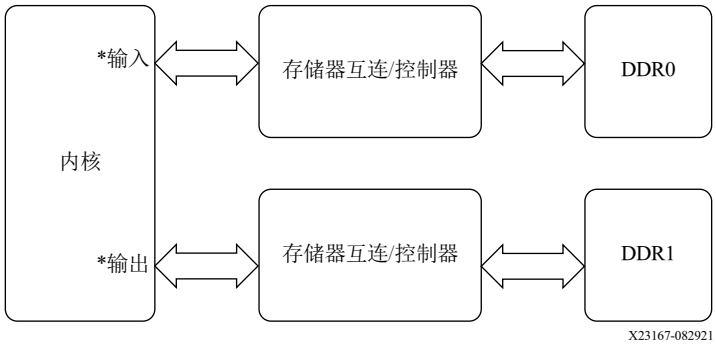
Vitis 技术中支持的加速卡可提供一个、两个或四个 DDR 存储体和最大 80 GB/s 原始 DDR 带宽。对于在 FPGA 和 DDR 之间传输大量数据的内核，赛灵思推荐您引导 Vitis 编译器和运行库使用多个 DDR 存储体。

除了 DDR 存储体，主机应用可访问 PLRAM 来将数据直接传输到内核。该功能是使用 `connnectivity.sp` 选项启用的，该选项包含在通过 `v++ --config` 选项指定的配置文件中。请参阅 将内核端口映射到存储器 以获取有关实现此最优化的更多信息，另请参阅 存储器映射接口 以获取有关传输到全局存储体的数据传输的信息。

为了充分利用多个 DDR 存储体，您需要将 CL 存储缓冲器分配到主机代码中的不同存储体，并配置 `xclbin` 文件以便匹配 `v++` 命令行中的存储体分配。

以下模块框图显示了 GitHub 上的 Vitis 示例中的《全局存储器双存储体 (C)》 示例。此示例将内核的输入指针接口连接到 DDR 存储体 0，并将输出指针接口连接到 DDR 存储体 1。

图 67：**全局存储器双存储体示例**



在主机代码中分配 DDR 存储体

重要提示！这是可选操作，仅在下述特殊情况下才需要执行。

在 Vitis 工具流程中，内核端口到存储体的连接可使用 `--connectivity.sp` 开关来建立，如 将内核端口映射到存储器 中所述。由 `v++` 生成的 `xclbin` 包含有关内核端口到存储器的连接信息，以便 XRT 能够正确分配缓冲器。在主机代码中创建缓冲器时，XRT 会自动将缓冲器从内核 `xclbin` 分配到存储器，并在内部管理这些缓冲器。如果将单个内核端口连接到多个存储体，那么 XRT 始终从编号较低的存储体开始操作。

在大部分情况下，此方法足以满足需求。但在某些特殊情况下，您可能需要在主机代码中手动分配缓冲器位置（或特殊属性）。为此，赛灵思 OpenCL 供应商扩展程序提供了名为 `CL_MEM_XRT_PTR_XILINX` 的缓冲器扩展程序，专用于管理主机代码中的存储体分配。以下代码示例显示了将输入和输出缓冲器分配到 DDR 存储体 0 和存储体 1 所需的头文件和代码：

```
#include <CL/cl_ext.h>
...
int main(int argc, char** argv)
{
...
    cl_mem_ext_ptr_t inExt, outExt;  // Declaring two extensions for both
buffers
    inExt.flags  = 0|XCL_MEM_TOPOLOGY; // Specify Bank0 Memory for input
memory
    outExt.flags = 1|XCL_MEM_TOPOLOGY; // Specify Bank1 Memory for output
Memory
    inExt.obj = 0    ; outExt.obj = 0; // Setting Obj and Param to Zero
    inExt.param = 0 ; outExt.param = 0;

    int err;
    //Allocate Buffer in Bank0 of Global Memory for Input Image using
Xilinx Extension
    cl_mem buffer_inImage = clCreateBuffer(world.context, CL_MEM_READ_ONLY
| CL_MEM_EXT_PTR_XILINX,
            image_size_bytes, &inExt, &err);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" << std::endl;
        return EXIT_FAILURE;
    }
    //Allocate Buffer in Bank1 of Global Memory for Input Image using
Xilinx Extension
    cl_mem buffer_outImage = clCreateBuffer(world.context,
CL_MEM_WRITE_ONLY | CL_MEM_EXT_PTR_XILINX,
            image_size_bytes, &outExt, NULL);
```

```
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" << std::endl;
        return EXIT_FAILURE;
    }
...
}
```

扩展指针 `cl_mem_ext_ptr_t` 为 `struct`，定义如下：

```
typedef struct{
    unsigned flags;
    void *obj;
    void *param;
  } cl_mem_ext_ptr_t;
```

- `flags` 的有效值为：
  - XCL_MEM_DDR_BANK0
  - XCL_MEM_DDR_BANK1
  - XCL_MEM_DDR_BANK2
  - XCL_MEM_DDR_BANK3
  - <id> | XCL_MEM_TOPOLOGY

    注释：<id> 是通过观察生成的 `xxx.xclbin.info` 文件（位于 `xxx.xclbin` 文件旁）的“存储器配置 (Memory Configuration)”部分来确定的。在 `xxx.xclbin.info` 文件中，列出的全局存储器（DDR、HBM、PLRAM 等）都带有表示 <id> 的索引。
- `obj` 是指向关联主机存储器的指针，仅当 `CL_MEM_USE_HOST_PTR` 标记传递到 `clCreateBuffer` API 时才会为 CL 存储缓冲器分配主机存储器，否则它设置为 NULL。
- `param` 保留以供将来使用。始终为其赋值 0 或 NULL。

以下是您可能需要使用扩展指针的部分特殊情况：

- P2P 缓冲器：要获取解释和示例，请参阅 https://xilinx.github.io/XRT/master/html/p2p.html。
- 主机存储缓冲器：要获取解释和示例，请参阅 https://xilinx.github.io/XRT/master/html/sb.html。
- 当内核端口连接到多个存储体时，请向特定存储体分配主机缓冲器：例如，DDR[0:1]。在 《Vitis 最优化 FPGA 加速应用：布隆过滤器示例》教程的使用多个 DDR 存储体实践中详述了此用例。

向特定存储体分配主机缓冲器的示例

上述第三个用例的示例是，当主机和内核同时访问 DDR 存储体时，如果您想要进行数据拆分以使内核与主机以乒乓方式来访问存储体，则可能需要使用 `cl_mem_ext_ptr_t`。当主机对特定存储体进行读写时，内核从另一个存储器执行读写，这样这些主机/内核访问就不会产生争用而影响性能。在此情况下，您必须自行管理缓冲器分配。

`xclbin` 中的内核端口连接到 DDR bank1 和 bank2，并从其中任一存储体读取数据。此连接是在链接期间由 Vitis 编译器使用 `--connectivity.sp` 开关来建立的：

```
[connectivity]
sp=runOnfpga_1.input_words:DDR[1:2]
```

从主机代码，可向 DDR 存储体 1 或 2 发送 input_words 数据。这样会创建 2 个赛灵思扩展指针 (cl_mem_ext_ptr_t) 对象，如以下代码示例所示。对象标记将确定每个缓冲器将分配到哪个 DDR 存储体以供内核进行访问。内核实参可设置为 input_words[0] 和 input_words[1] 以供执行连续内核排队。

```
#include <CL/cl_ext.h>
...
int main(int argc, char** argv)
{
cl_mem_ext_ptr_t buffer_words_ext[2];

buffer_words_ext[0].flags = 1 | XCL_MEM_TOPOLOGY; // DDR[1]
buffer_words_ext[0].param = 0;
buffer_words_ext[0].obj   = input_doc_words;
buffer_words_ext[1].flags = 2 | XCL_MEM_TOPOLOGY; // DDR[2]
buffer_words_ext[1].param = 0;
buffer_words_ext[1].obj   = input_doc_words;
...
```

为内核代码分配全局存储器

创建多个 AXI 接口

OpenCL 内核、C/C++ 内核和 RTL 内核采用不同方法来向 AXI 接口分配函数参数。

- 对于 OpenCL 内核，需要使用 --max_memory_ports 选项来为内核实参上的每个全局指针生成一个 AXI4 接口。AXI4 接口名称是根据实参列表上的全局指针顺序来命名的。

  以下代码取自 GitHub 上的 Vitis 加速示例的 ocl_kernels 类别中的 gmem_2banks_ocl 示例：

  ```
  __kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
  void apply_watermark(__global const TYPE * __restrict input,
  __global TYPE * __restrict output, int width, int height) {
   ...
  }
  ```

  在此示例中为第一个全局指针 input 分配的 AXI4 名称是 M_AXI_GMEM0，为第二个全局指针 output 分配的名称是 M_AXI_GMEM1。

- 对于 C/C++ 内核，通过在 HLS INTERFACE 编译指示中为不同全局指针指定不同的“bundle”名称，来生成多个 AXI4 接口。如需了解更多信息，请参阅 内核接口。

  以下代码片段来自 gmem_2banks 示例，该示例将 input 指针分配给 bundle gmem0 并将 output 指针分配给 bundle gmem1。bundle 名称可以是任意有效的 C 字符串，而生成的 AXI4 接口名称将为 M_AXI_<bundle_name>。例如，输入指针的 AXI4 接口名为 M_AXI_gmem0，而输出指针的接口则为 M_AXI_gmem1。如需了解更多信息，请参阅 pragma HLS interface。

  ```
  #pragma HLS INTERFACE m_axi port=input  offset=slave bundle=gmem0
  #pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem1
  ```

- 对于 RTL 内核，在“RTL Kernel”Wizard 执行导入过程中，会生成端口名称。“RTL Kernel”Wizard 建议的默认名称为 m00_axi 和 m01_axi。如果不更改这些名称，那么在配置文件中通过 connectivity.sp 选项分配 DDR 存储体时，必须使用这些名称。如需了解更多信息，请参阅 将内核端口映射到存储器。

将 AXI 接口分配至 DDR 存储体

重要提示！使用多个 DDR 接口时，赛灵思要求您为每个内核/CU 指定 DDR 存储体，并指定用于放置内核的 SLR。如需了解更多信息，请参阅 将内核端口映射到存储器 和 将计算单元分配给 SLR。

以下配置文件示例中指定了 `connectivity.sp` 选项以及 `v++` 命令行，此命令行用于将输入指针 (`M_AXI_GMEM0`) 连接到 DDR 存储体 0，并将输出指针 (`M_AXI_GMEM1`) 连接到 DDR 存储体 1：

`config_sp.cfg` 文件：

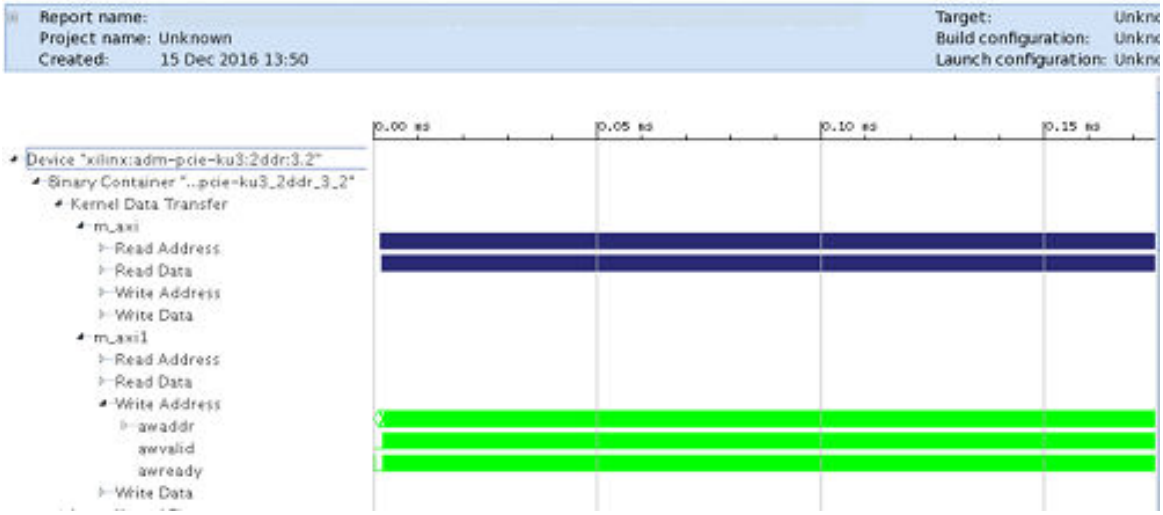
```
[connectivity]
sp=apply_watermark_1.m_axi_gmem0:DDR[0]
sp=apply_watermark_1.m_axi_gmem1:DDR[1]
```

`v++` 命令行：

```
v++ apply_watermark --config config_sp.cfg
```

您可以使用“Device Hardware Transaction”视图来观察实际的 DDR 存储体通信，并分析 DDR 的使用情况。

图 68：**“Device Hardware Transaction”视图 DDR 存储体上的传输事务**



将 AXI 接口分配至 PLRAM

某些平台支持 PLRAM。在此类情况下，可按 将 AXI 接口分配至 DDR 存储体 中所述的方式使用 `--connectivity.sp` 选项，但使用的名称应为 PLRAM[id]。在随 `xclbin` 一起生成的 `xclibin.info` 文件的“Memory Configuration”部分中可以找到特定平台支持的有效名称。

内核 SLR 和 DDR 存储器分配

就频率和资源而言，内核计算单元 (CU) 实例和 DDR 存储器资源布局规划是满足设计结果质量的关键。布局规划包括将 CU（内核实例）显式分配至 SLR 和将 CU 映射至 DDR 存储器资源。在进行布局规划时，需要考虑 CU 资源使用情况和 DDR 存储器带宽要求。

最大的赛灵思 FPGA 由多个堆叠硅片裸片组成。每个堆栈被称为一个超级逻辑区域 (SLR)，具有固定量的资源和存储器，包括 DDR 接口。如需了解可用于定制逻辑的可用器件 SLR 资源，请参阅 Vitis 软件平台版本说明，或者可使用 `platforminfo` 实用工具来显示这些资源，如 platforminfo 实用工具 中所述。

您可以使用实际内核资源利用率值来帮助跨 SLR 分配 CU，从而降低任一 SLR 中的拥塞。系统会在设计周期早期估算内核所使用的资源（LUT、触发器、BRAM 等）的数量。在硬件仿真和系统编译期间，可通过命令行或 GUI 生成报告，如 系统估算报告 中所述。

该信息可搭配可用的 SLR 资源一起使用，以帮助将 CU 分配至 SLR，以免过度使用任一 SLR。SLR 中的拥塞越少，工具就能更有效地将设计映射至 FPGA 资源并满足您的性能目标。如需了解有关映射存储器资源和 CU 的信息，请参阅 将内核端口映射到存储器 和 将计算单元分配给 SLR。

注释：虽然计算单元可以连接至任何可用的 DDR 存储器资源，但是在分配至 SLR 时还必须考虑内核的带宽要求。

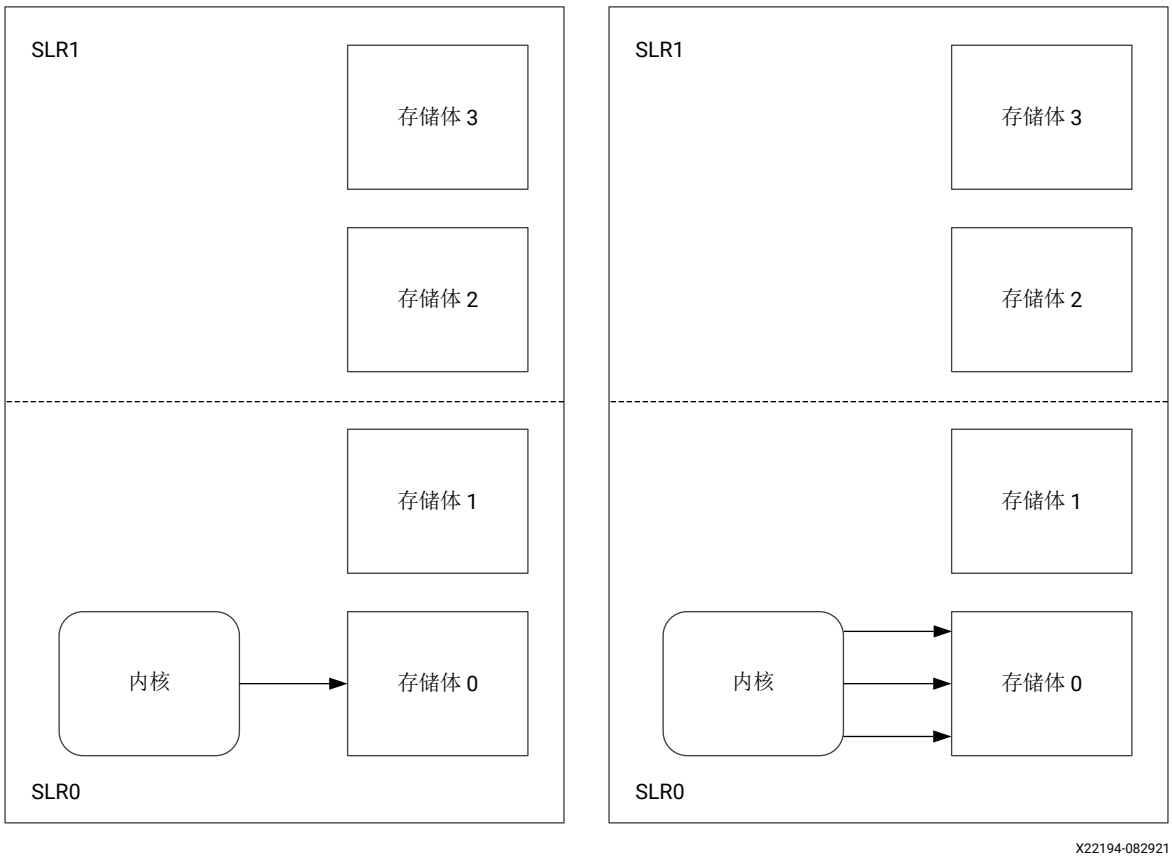
在将您的 CU 分配至 SLR 后，请将所有 CU AXI 主端口映射至 DDR 存储器资源。赛灵思推荐将这些 CU 主端口连接至与该 CU 相同的 SLR 中的 DDR 存储器资源。这样可以为有限的 SLR 交汇连接资源减少竞争。此外，SLR 之间的连接使用超长线路 (SLL) 布线资源，这会造成了比标准 SLR 内部布线更大的延迟。

可能需要跨 SLR 区域，以连接到其它 SLR 中的 DDR 资源。但是，如果 `connectivity.sp` 和 `connectivity.slr` 指令都已显式定义，则工具会自动添加额外的交汇逻辑，以便将 SLL 延迟的影响降到最低并促进时序收敛。

有关访问多个存储体 (memory bank) 的内核准则

DDR 存储器资源分布在平台的超级逻辑区域 (SLR) 上。由于可用于跨 SLR 的连接数量有限，因此一般准则是：任一内核都应与连接最多的 DDR 存储器资源布局在同一个 SLR 内。这样即可减少跨 SLR 的连接竞争，并避免消耗与跨 SLR 关联的额外逻辑资源。

图 69：**内核和存储器位于同一 SLR 中**



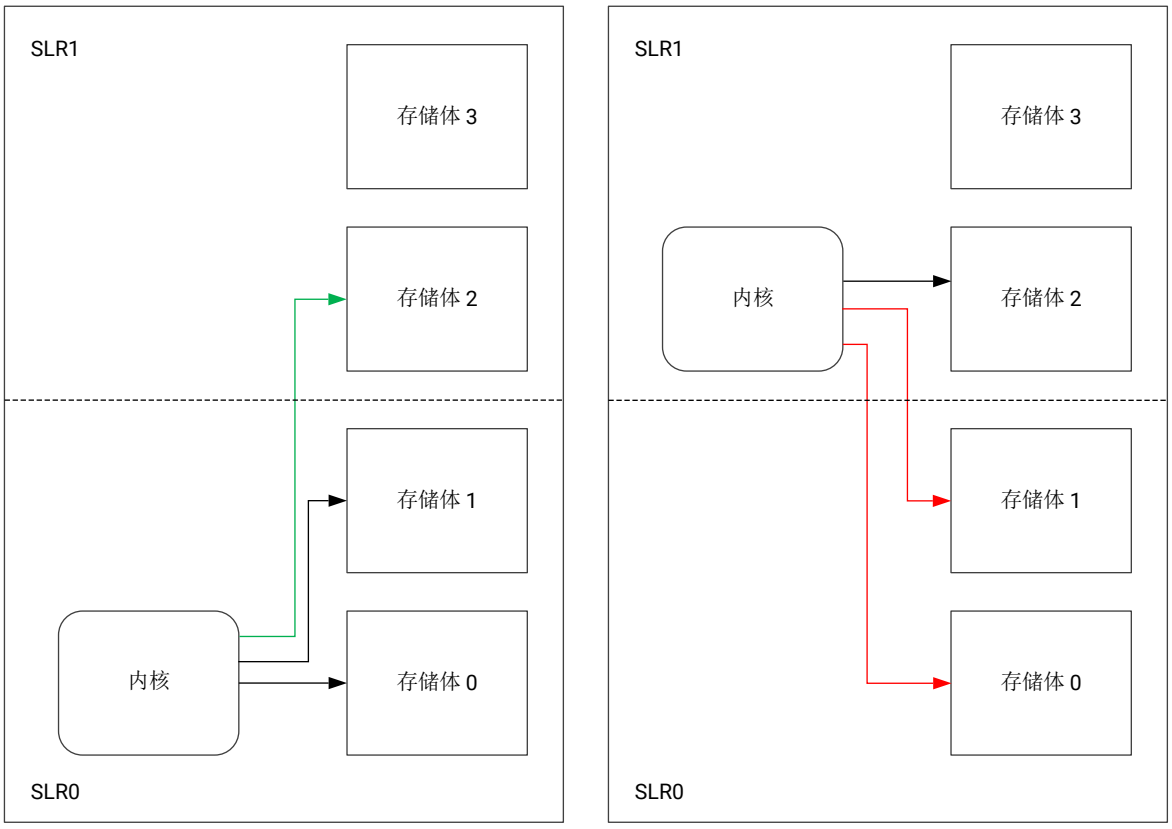
注释：左图显示了映射到单个存储体的单个 AXI 接口。右图显示了映射到同一存储体的多个 AXI 接口。

如上图所示，当内核具有仅映射单个存储体的单一 AXI 接口时，platforminfo 实用工具 中所描述的 `platforminfo` 实用工具会列出与该内核的存储体关联的 SLR，因此，最好将该内核布局在此 SLR 中。在此情况下，设计工具可能会自动将内核布局在该 SLR 中，而无需额外输入；但是，在以下条件下您可能需要为某些内核提供显式 SLR 分配：

- 设计包含大量访问同一存储体的内核。
- 内核需要一些专用逻辑资源，而这些资源在存储体所在的 SLR 中不可用。

当一个内核有多个 AXI 接口并且该内核的所有接口都访问同一个存储体时，处理方式与具有单一 AXI 接口的内核处理方式非常类似，并且该内核应与其 AXI 接口正在映射的存储体驻留在相同 SLR 中。

图 70：**存储体位于相邻 SLR 中**

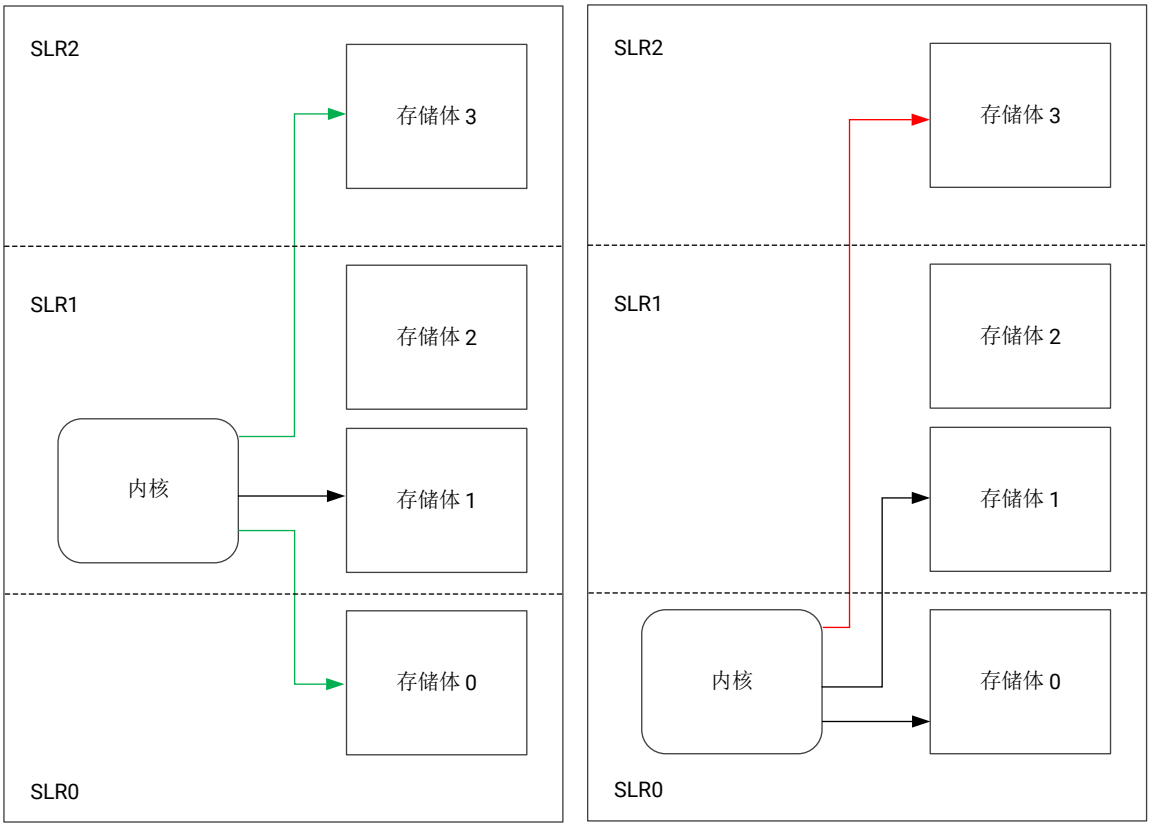


注释：左图显示当内核布局在 SLR0 中时，需要跨一次 SLR。右图显示内核访问多个存储体需要跨两次 SLR。

当内核具有多个 AXI 接口并访问位于不同 SLR 内的多个存储体时，建议将该内核布局在包含其访问的大部分存储体的 SLR 中（如上图所示）。这样可以最大程度减少此内核需要跨 SLR 的次数，从而可保留更多 SLR 交汇资源，以供设计中的其它内核用于访问存储体。

当内核在映射来自不同 SLR 的存储体时，请显式指定 SLR 分配，如 内核 SLR 和 DDR 存储器分配 中所述。

图 71：**相距 2 个 SLR 的存储体**



注释：左图显示，访问所有存储器映射的存储体需要跨两次 SLR。右图显示，访问所有存储器映射的存储体需要跨三次 SLR。

如上图所示，当平台包含 2 个以上的 SLR 时，内核可能会将位于非相邻 SLR 中的存储体映射到其最常用的存储器映射存储体。出现这种情况时，对远处存储体进行的存储器访问必须跨多个 SLR 边界，从而将产生额外的 SLR 交汇资源成本。为了避免此类成本，将内核布局在中间 SLR 中可能会更好，因为在中间 SLR 中跨相邻 SLR 所需的成本更低。

XILINX

第 20 章

调试应用与内核

Vitis™ 统一软件平台可提供应用级调试功能和技巧，支持对主机代码、内核代码以及两者之间的交互进行调试。这些功能和技巧根据软件调试流程和硬件调试流程来区分。

对于软件调试，主机与内核代码可使用 Vitis IDE 来调试，或者也可以使用来自命令行的 GDB 作为独立调试工具来进行调试。

对于硬件调试，硬件上运行的内核可使用通过 PCIe® 总线运行的赛灵思虚拟线缆 (XVC) 来调试（适用于 Alveo™ 数据中心加速器卡），也可以使用 USB-JTAG 线缆来调试（适用于 Alveo 卡和嵌入式处理器平台）。

调试流程

Vitis 统一软件平台可以提供应用级别的调试功能，这些调试功能支持在 Vitis IDE 中或者通过命令行来对主机代码、内核代码及其相互间的交互进行高效调试。建议的调试流程包括三个级别的调试：

- 在软件仿真中调试，用于确认主机程序与内核代码中所呈现的应用的算法功能。
- 在硬件仿真中调试，用于将内核编译到 RTL 中、确认生成的逻辑的行为并评估硬件的仿真性能。
- 硬件执行期间的调试，用于实现 FPGA 二进制文件，并调试硬件中运行的应用。

这种三层式方法支持按不同的抽象层级来对主机、内核代码及其相互间的交互进行调试。每个层级都能对设计提供深入、具体的见解，从而简化调试。上述所有流程都支持通过集成 GUI 流程来完成，也支持通过使用基本编译时和运行时设置选项的批处理流程来完成。

对于在嵌入式处理器平台上运行的应用，则需要一些附加设置，如 在嵌入式处理器平台上进行调试 中所述。

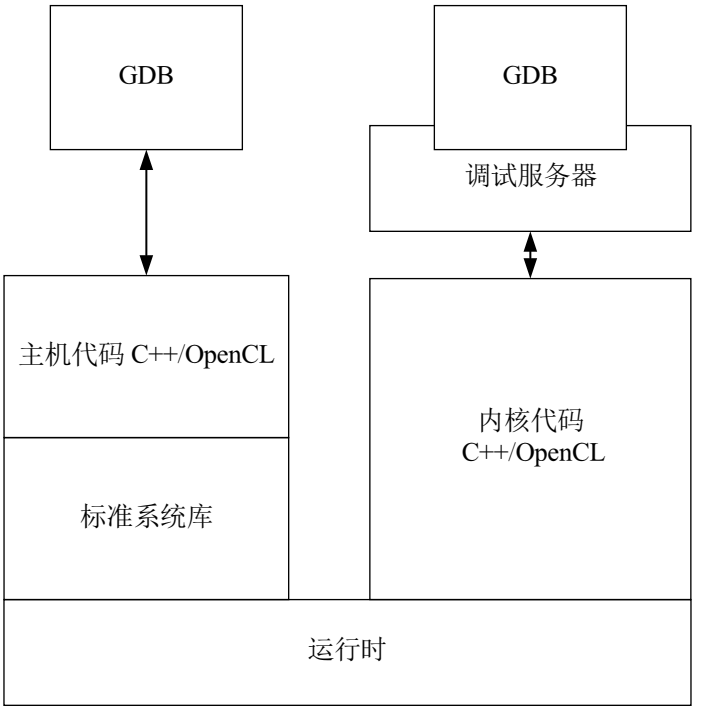
在软件仿真中调试

重要提示！ 以下步骤描述了如何从命令行进行调试。但 Vitis IDE 可提供独立调试环境以搭配从命令行创建的 Vitis 应用加速工程一起使用。如需了解更多信息，请参阅 使用独立调试流程。

Vitis 统一软件平台支持在下列情况下执行典型的软件调试：可随时对主机代码进行调试、在软件仿真模式下对内核代码进行调试，以及在硬件仿真模式期间的不同时间点进行调试。这是标准软件调试流程，此流程使用断点、单步执行代码调试、分析变量并强制代码进入特定状态。

下图显示了使用 GNU 调试 (GDB) 工具来为主机与内核代码（以 C/C++ 或 OpenCL™ 编写）执行软件仿真期间的调试流程。请注意，其中使用 2 个 GDB 实例分别对主机进程与内核进程执行调试，并使用调试服务器 (`xrt_server`)。

图 72：**软件仿真**



赛灵思建议在软件仿真中尽可能迭代设计，因为设计迭代的编译时很短且能快速执行。如需了解有关软件仿真的更多详细信息，请参阅 软件仿真。

基于 GDB 的调试

重要提示！主机与内核代码都必须使用 `-g` 选项进行编译以供调试。

对于 GNU 调试 (GDB)，您可调试内核或主机代码、添加断点并检验变量。这种常见的软件调试流程支持快速设计、编译和调试，以便确认应用功能。Vitis 调试器还可为 GDB 提供扩展，以便您从主机程序检验赛灵思的 Xilinx Runtime (XRT) 库的内容。这些扩展可用于调试主机与内核之间的协议同步问题。

Vitis 核开发套件支持在所有流程中进行 GDB 主机程序调试，但内核调试仅限在软件仿真模式才可用。您需要在编译和链接期间在主机与内核代码中使用 `-g` 选项来启用调试功能。

本节将介绍如何在 GDB 帮助下执行主机和内核调试。由于大多数软件开发者都应熟悉此流程，因此本节将主要侧重于 XRT 库的主机代码调试功能的扩展以及内核调试的要求。

Xilinx Runtime 库 GDB 扩展

Vitis 调试器 (`xgdb`) 支持全新的 GDB 命令，这些命令可便于您从主机应用查看 XRT 库。

注释：如果您在 Vitis 调试器外部启动 GDB，则需使用 `appdebug.py` 脚本启用命令扩展，如 启动主机和内核调试 中所述。

有两种命令可从 `gdb` 命令行调用：

1. `xprint` 命令支持查看 XRT 库数据结构（`cl_command_queue`、`cl_event` 和 `cl_mem`）。这些命令解释如下。
2. `xstatus` 命令支持在硬件执行期间进行调试时查看 Vitis 目标平台上运行的 IP。

您可通过在 `gdb` 命令提示符中使用 `help <command>` 来获取有关 `xprint` 命令和 `xstatus` 命令的更多信息。

典型应用场景是当您发现主机应用挂起时，即可使用这些命令。在此情况下，主机应用可能正在等待命令队列完成，或者正在等待事件列表。使用 `xprint queue` 命令打印命令队列即可告知您有哪些事件尚未完成，以便您分析事件之间的依赖关系。

使用 Vitis IDE 进行调试时，会自动追踪这两种命令的输出。在这种情况下，调试透视图左上角的变量、断点和寄存器的常用选项卡旁边还提供了三个选项卡。这些选项卡标记为“命令队列 (Command Queue)”、“存储缓冲器 (Memory Buffers)”和“平台调试 (Platform Debug)”，分别显示 `xprint queue`、`xprint mem` 以及 `xstatus` 的输出。

xprint 命令

`xprint queue` 和 `xprint mem` 的实参为可选项。如果未指定实参，则应用调试环境会持续跟踪所有 XRT 库对象，并自动打印所有有效队列和 `cl_mem` 对象。此外，这些命令会对所提供的命令 `queue`、`event` 以及 `cl_mem` 实参进行正确确认。

```
xprint queue [<cl_command_queue>]
xprint event <cl_event>
xprint mem [<cl_mem>]
xprint kernel
xprint all
```

xstatus 命令

该功能仅在系统流程（硬件执行）中可用，而在任何仿真 (emulation) 流程中都不可用。

```
xstatus all
xstatus --<ipname>
```

基于 GDB 内核的调试

软件仿真流程支持 GDB 内核调试。当 GDB 可执行程序连接到 IDE 中的内核或命令行流程时，您可以设置断点并查询内核中的变量内容，与正常主机代码调试类似。由于内核 GDB 进程附加到已生成的软件进程上，因此软件仿真流程完全支持该功能。

命令行调试流程

提示：运行工具前，请按 设置 Vitis 环境 中所述方式设置命令 shell 或窗口。

以下描述了在软件仿真中从命令行运行调试流程所需的步骤。如需了解有关在 IDE 中进行调试的信息，请参阅 第七部分：使用 Vitis IDE。在 Vitis 核开发套件中进行调试需使用以下步骤：

1. 在 `g++` 命令行中添加 `-g` 选项以编译并链接用于调试的主机代码，如 构建主机程序 中所述。
2. 在 `v++` 命令行中添加 `-g` 选项以编译并链接用于调试的内核代码，如 构建器件二进制文件 中所述。

   注释：调试 OpenCL 内核时，在编译和链接期间还可执行其它步骤，如 调试 OpenCL 内核 中所述。
3. 启动 GDB 以调试应用。此过程涉及 3 个命令目标平台，如 启动主机和内核调试 中所述。

调试 OpenCL 内核

对于 OpenCL 内核，在软件仿真期间可以执行额外的运行时检查。这些额外检查包括：

- 检查 OpenCL 内核是否对接口缓冲器执行界外访问 (`fsanitize=address`)。
- 检查内核是否访问未初始化的本地存储器 (`fsanitize=memory`)。

这些 Vitis 编译器选项是通过 `--advanced` 编译器选项使用以下命令语法来启用的，如 --advanced 选项 中所述：

```
--advanced.param compiler.fsanitize=address,memory
```

应用这些选项后，仿真运行会生成调试日志，其中包含已写入 `<project_dir>/Emulation-SW/<proj_name>-Default>/emulation_debug.log` 的仿真诊断消息。

`fsanitize` 指令同样可在配置文件中指定，如下所述：

```
[advanced]
#param=<param_type>:<param_name>.<value>
param=compiler.fsanitize=address,memory
```

随后，在 `v++` 命令行上指定配置文件：

```
v++ -l -t sw_emu --config ./advanced.cfg -o bin_kernel.xclbin
```

如需了解有关 `--config` 选项的更多信息，请参阅 Vitis 编译器配置文件。

启动主机和内核调试

在软件仿真中，为了对硬件加速器进行更有效的建模，FPGA 二进制文件的执行将作为独立进程来生成。如果您正在使用 GDB 来调试主机代码，则不会在内核代码中遇到断点，因为在主机代码进程中不运行内核代码。为了支持主机代码和内核代码的并发调试，Vitis 调试器提供了一个系统，可通过使用调试服务器 (`xrt_server`) 来连接到生成的内核。要将主机代码和内核代码连接到调试服务器，您必须使用以下进程打开 3 个终端窗口。

提示：当使用 GDB 图形前端（如 GNU 上提供的数据显示调试器 (DDD)）时，此流程也应该能够正常工作。以下步骤是启动 GDB 的说明。

1. 打开 3 个终端窗口，并按 设置 Vitis 环境 中所述设置每个窗口。这 3 个窗口用于：
   - 运行 `xrt_server`
   - 在主机代码上运行 GDB (`xgdb`)
   - 在内核代码上运行 GDB (`xgdb`)
2. 在第一个终端中，设置终端环境后，请使用以下命令启动 Vitis 调试服务器：

   ```
   xrt_server --sdx-url
   ```

   调试服务器会监听来自主机和内核的调试命令、并将 2 个进程相连以创建单一调试环境。`xrt_server` 会在标准输出上返回 `listener port <num>`。跟踪返回的监听器端口号，因为此端口供 GDB 用于调试内核进程。为了控制该进程，您必须启动新的 GDB 实例，并连接到 `xrt_server`。您可通过下列步骤完成此操作。

   重要提示！运行 `xrt_server` 后，所有生成的 GDB 进程都会等待您进行控制。如果没有任何 GDB 连接到 `xrt_server`，或者没有任何 GDB 提供命令，那么内核代码会显示为挂起。

3. 在第二个终端中，设置终端环境后，请按如下步骤所述为主机代码启动 GDB：
   a. 设置 ENABLE_KERNEL_DEBUG 环境变量。例如，在 C-shell 中使用：

      ```
      setenv ENABLE_KERNEL_DEBUG true
      ```

   b. 将 XCL_EMULATION_MODE 环境变量设置为 sw_emu 模式，如 运行应用硬件构建 中所述。例如，在 C-shell 中使用：

      ```
      setenv XCL_EMULATION_MODE sw_emu
      ```

   c. 必须使用 xrt.ini 文件中的条目来启用运行时调试功能，如 xrt.ini 文件 中所述。在主机可执行文件所在的目录中创建 xrt.ini 文件，并包含下列行：

      ```
      [Debug]
      app_debug=true
      ```

      这样即可告知运行时库，内核已编译完成，可用于调试，并且 XRT 库应启用调试功能。

   d. 通过赛灵思封装启动 gdb：

      ```
      xgdb --args <host> <xclbin>
      ```

      其中 <host> 是主机可执行文件的名称，<xclbin> 是 FPGA 二进制文件的名称。例如：

      ```
      xgdb --args host.exe vadd.xclbin
      ```

      从 xgdb 封装启动 GDB 即可为 Vitis 调试器执行以下设置步骤：

      - 随指定的主机程序加载 GDB。
      - 从 GDB 命令提示符处使用 source 命令获取 Python 脚本，以启用 Vitis 调试器扩展：

        ```
        gdb> source ${XILINX_XRT}/share/appdebug/appdebug.py
        ```

4. 在第三个终端中，设置终端环境后，请启动 xgdb 命令并从 (gdb) 提示符处运行以下命令：
   - 对于软件仿真：

     ```
     file <Vitis_path>/data/emulation/unified/cpu_em/generic_pcie/model/
     genericpciemodel
     ```

     其中 <Vitis_path> 是 Vitis 核开发套件的安装路径。$XILINX_VITIS 环境变量在 GDB 内不可用。
   - 连接到内核进程：

     ```
     target remote :<num>
     ```

     其中 <num> 是 xrt_server 返回的监听器端口号。

当全部 3 个终端窗口都运行 xrt_server、用于主机的 GDB 和用于内核的 GDB 后，您即可在自己的主机或内核上按需设置断点、运行 continue 命令并调试自己的应用。当所有内核调用完成后，主机代码将继续，而 xrt_server 连接将断开。

使用 printf() 或 cout 调试内核

调试算法的基本方法是验证程序执行的整个过程中的关键代码步骤和关键数据值。对于应用开发者而言，打印检查点声明和输出代码中的当前值是识别程序执行中存在的问题的简单而又有效的方法。这可通过使用 printf() 函数或 cout（适用于标准输出）来完成。

C/C++ 内核

对于 C/C++ 内核模型，仅在软件仿真期间支持 `printf()`，在 Vitis HLS 综合步骤中应将其排除。在这种情况下，任何 `printf()` 语句都应包含在以下编译器宏中：

```
#ifndef __SYNTHESIS__
    printf("Checkpoint 1 reached");
#endif
```

对于 C++ 内核，您还可在自己的代码中使用 `cout` 来添加检查点或消息，用于调试内核。例如，您可添加：

```
std::cout << "TEST " << (match ? "PASSED" : "FAILED") << std::endl;
```

OpenCL 内核

赛灵思的 Xilinx Runtime (XRT) 库支持在所有配置中使用内核中内置的 OpenCL™ `printf()` 函数，包括软件仿真、硬件仿真以及硬件执行期间。

提示：在 OpenCL 内核的所有构建配置中，都仅支持 `printf()` 函数。对于 C/C++ 内核，仅在软件仿真中支持 `printf()`。

以下是在内核中使用 `printf()` 的示例以及输出，执行内核时，`global` 大小为 8：

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void hello_world(__global int *a)
{
    int idx = get_global_id(0);

     printf("Hello world from work item %d\n", idx);
     a[idx] = idx;
}
```

输出如下：

```
Hello world from work item 0
Hello world from work item 1
Hello world from work item 2
Hello world from work item 3
Hello world from work item 4
Hello world from work item 5
Hello world from work item 6
Hello world from work item 7
```

重要提示！ `printf()` 消息在全局存储器中进行缓冲，并在内核执行完成后进行卸载。如果在多个内核中使用 `printf()`，那么主机终端上显示的来自每个内核的消息顺序是不确定的。请注意，特别是当在硬件仿真和硬件中运行时，硬件缓冲器的大小可能会限制 `printf` 的输出捕获。

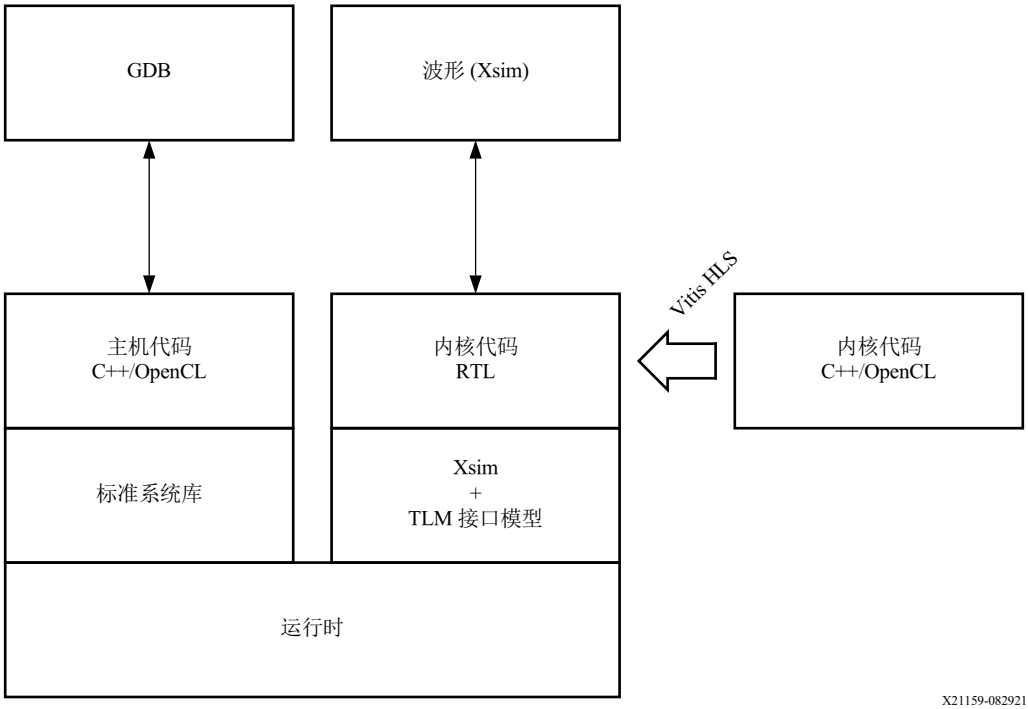
在硬件仿真中调试

重要提示！以下步骤描述了如何从命令行进行调试。但 Vitis IDE 可提供独立调试环境以搭配从命令行创建的 Vitis 应用加速工程一起使用。如需了解更多信息，请参阅 使用独立调试流程。

在硬件仿真期间，内核代码被编译到 RTL 代码中，以便您可先对内核的 RTL 逻辑进行评估，然后再将其实现到赛灵思器件中。主机代码可与内核的 RTL 模型的行为仿真并发执行，也可以直接导入或者通过 Vitis HLS 基于 C/C++/OpenCL 内核代码来创建。如需了解更多信息，请参阅 硬件仿真。

下图显示了硬件仿真流程图，此流程图可在 Vitis 调试器中用于确认主机代码、剖析主机与内核性能、估算 FPGA 资源利用率以及通过使用精确的硬件模型 (RTL) 来验证内核。RTL 内核代码在 Vivado 仿真器或第三方 RTL 仿真器中接受分析。GDB 则用于以更为传统的方式来对主机代码进行软件风格调试。

图 73：**硬件仿真**



通过在数据集上运行硬件仿真来验证主机代码和内核硬件实现是否正确。硬件仿真流程会在 Vitis 核开发套件中调用 Vivado 逻辑仿真器来测试将在 FPGA 互连结构上执行的内核逻辑。模型之间的接口以传输事务级模型 (TLM) 来表示，以限制接口模型对于总体执行时间的影响。硬件仿真的执行时间比软件仿真更长。

提示：赛灵思建议您使用小数据集进行调试和确认。

在硬件仿真期间，您可以选择修改内核代码以改善性能。在硬件仿真中迭代主机和内核代码设计，直至功能正确并对估算的内核性能感到满意为止。

基于波形的内核调试

由于在硬件仿真构建配置中已使用 Vitis HLS 将 C/C++ 与 OpenCL 内核代码综合到 RTL 代码内，因此，您也可以使用 RTL 行为仿真来分析内核逻辑。硬件设计师可能很熟悉这种方法。Vitis 核开发套件支持这种基于波形的 HDL 调试方法，可使用命令行流程或在硬件仿真期间通过 IDE 流程来使用此方法。

提示：基于波形的调试属于高级功能。大多数情况下，无需分析 RTL 逻辑。

利用 Vitis 编译器命令启用波形调试

波形调试进程可通过 `v++` 命令使用以下步骤来启用：

1. 在编译和链接期间，在内核代码中启用调试功能，如 构建器件二进制文件 中所述。

```
v++ -g ...
```

2. 在主机可执行文件所在目录中创建包含以下内容的 `xrt.ini` 文件，如 xrt.ini 文件 中所述。

```
[Emulation]
debug_mode=batch
```

3. 在硬件仿真模式下运行应用、主机与内核。在名为 `<hardware_platform>-<device_id>-<xclbin_name>.wdb` 的文件中会收集反映硬件传输事务数据的波形数据库。在 Vitis 分析器内可直接打开此文件，如 第六部分：使用 Vitis 分析器 中所述。

提示：如果 `xrt.ini` 中包含 `debug_mode=gui`，那么运行应用时，会启动实时波形查看器，如 波形视图和实时波形查看器 中所述。调试 `hw_emu` 挂起问题时，这尤其有用，因为您可以在仿真器内中断仿真进程，并观测截至该时间点的波形。

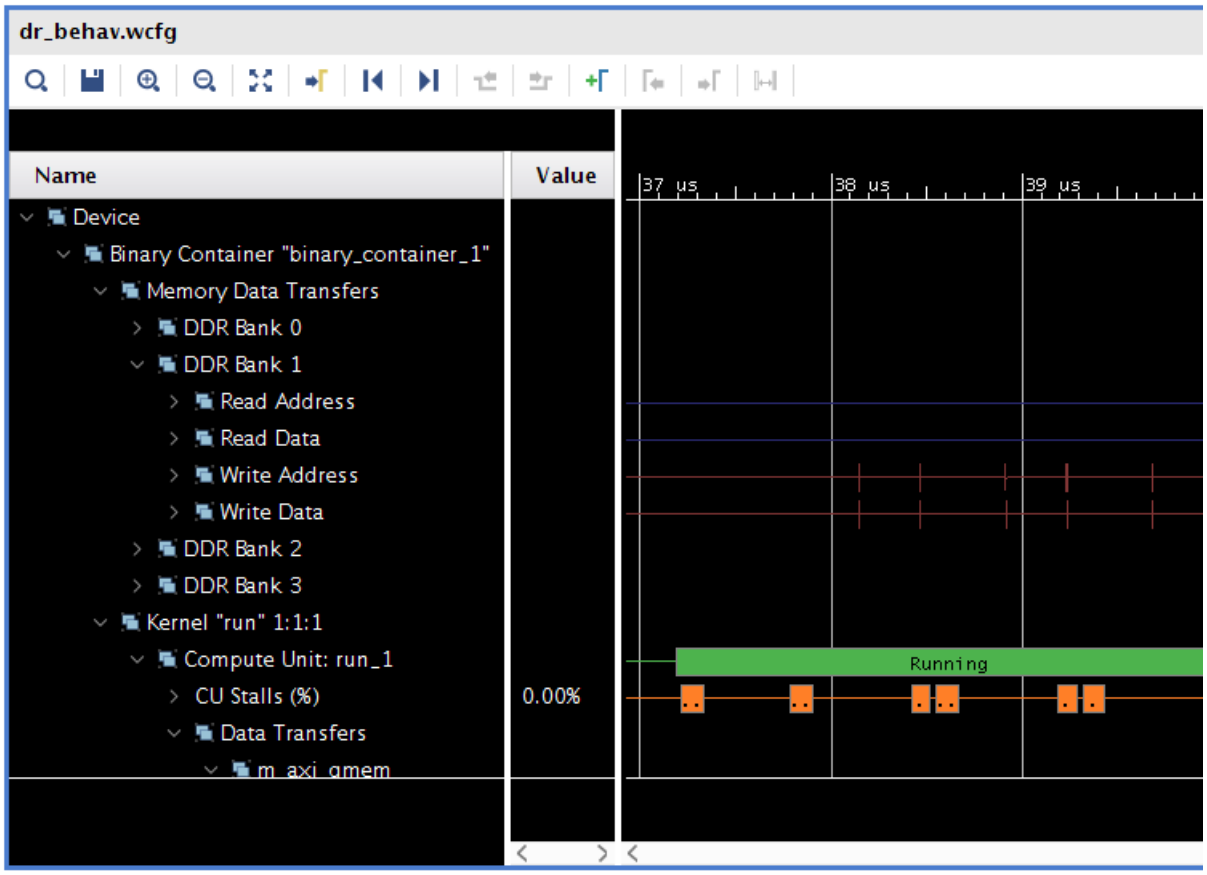
运行基于波形的内核调试流程

Vitis IDE 可在硬件仿真模式下提供基于波形的 HDL 调试。波形可在 Vivado 波形查看器中打开，Vivado 逻辑仿真用户应对此很熟悉。Vitis IDE 允许您显示内核接口、内部信号并包含调试控件（如，重新启动、HDL 断点以及 HDL 代码查找和波形标记）。此外，它还提供顶层 DDR 数据传输（每个 bank）以及特定于内核的详细信息，包括计算单元停滞、循环流水线活动和数据传输。

欲知详情，请参阅 波形视图和实时波形查看器。

如果已激活实时波形查看器，则当运行可执行文件时，波形查看器会自动打开。默认情况下，波形查看器会显示所有接口信号及下列调试层级：

图 74：**波形查看器**



- “Memory Data Transfers”：显示从所有计算单元漏斗经由这些接口进行的数据传输。

  提示：这些接口的位宽可能与计算单元不同。如果是这样，则突发长度可能也不同。例如，在某个计算单元上，突发可能含 16 个 32 位码字，而在 OCL 主接口上，突发可能含 1 个 512 位码字。

- “Kernel <kernel name><workgroup size> Compute Unit<CU name>”：内核名称、工作组大小和计算单元名称。
- “CU Stalls (%)”：显示整个 CU 的停滞汇总信息。其中已创建所有最低级别的停滞信号的总线，该总线在波形中以任意时间点处于活动状态的信号的百分比 (%) 来表示。
- “Data Transfers”：显示 CU 上所有 AXI 主接口的数据传输。
- “User Functions”：列出 CU 层级内的所有函数。
- “Function: <function name>”：这是函数名称。
- “Dataflow/Pipeline Activity”：显示 CU 的函数级别的循环数据流/流水线信号。
- “Function Stalls”：列出此函数内的 3 个停滞信号。
- “Function I/O”：列出该函数的 I/O。这些 I/O 为下列协议的 I/O：`-m_axi`、`ap_fifo`、`ap_memory` 或 `ap_none`。

提示：正如任何波形调试器一样，可通过从范围菜单中选择感兴趣的实例以及从对象菜单中选择感兴趣的信号来添加内部信号的额外调试数据。同样，诸如 HDL 断点之类的调试控件以及 HDL 代码查找和波形标记也都受支持。如需了解有关使用波形查看器的更多信息，请参阅《Vivado Design Suite 用户指南：逻辑仿真》(UG900)。

硬件仿真调试技巧

由于硬件仿真中使用的是近似模型，因此仿真系统的行为可能与硬件不相符。以下列表提供了一些常见问题，以便检验您的应用在硬件仿真期间是否未能提供所期望的结果。

1. 复查主机应用以确保正确捕获不同内核运行之间的事件依赖关系。此类问题可能导致出现意外行为。也有可能应用能够在硬件中通过测试，但在应用中存在逻辑错误，当条件产生些许变化时，在硬件上就可能触发此类逻辑错误。
2. 如果您有 RTL 内核，请在调试模式下运行应用，确保内核仿真中没有“X”（未驱动的值）。这表示存在错误代码，可能在硬件中能正常运行，但在仿真中将运行失败并出现意外行为。如果内核是由 HLS 生成的，请确认所有变量都已初始化为相应的值。
3. 确保硬件仿真中内核处理的数据量小于仿真在合理时间内能够完成的量。否则，可能导致应用永久运行或者“挂起”。在此案例中，在硬件仿真中运行应用时，请在主机应用控制台中查找 `INFO: [Vitis-EM 22]` 消息。检查在全局存储器上读取或写入的数量是否不断增加：
   a. 如果 RD/WR 数据量不断增加，这表明应用和硬件执行正在正常进行中。应用未挂起，但耗费较长时间才能完成。这可能是由于数据大小较大或者由于内核执行存储器读写的方式效率较低。应用和内核需进行最优化。
   b. 如果在连续消息中 RD/WR 数据量未增加，这表明仿真正在运行，但硬件中某处存在死锁，可能在内核内部或者在平台其余部分。复查位于内核边界处、互连（例如，`sdx_memss`）处以及其它位置的 AXI 传输事务，确认是否存在未完成的传输事务，或者是否内核正在生成任何传输事务。
4. 以波形模式运行硬件仿真，同样复查时间线轨迹。通过观测内核的 AXI4-Lite 接口上的流量或者通过观测内核的输出中断来确认内核是否处于“已启动”和“已完成”状态。
5. 复查 xrt.ini 文件 的 `[Emulation]` 部分，启用相应的设置，这些设置有助于缩小应用或内核中的问题范围。

硬件执行期间的调试

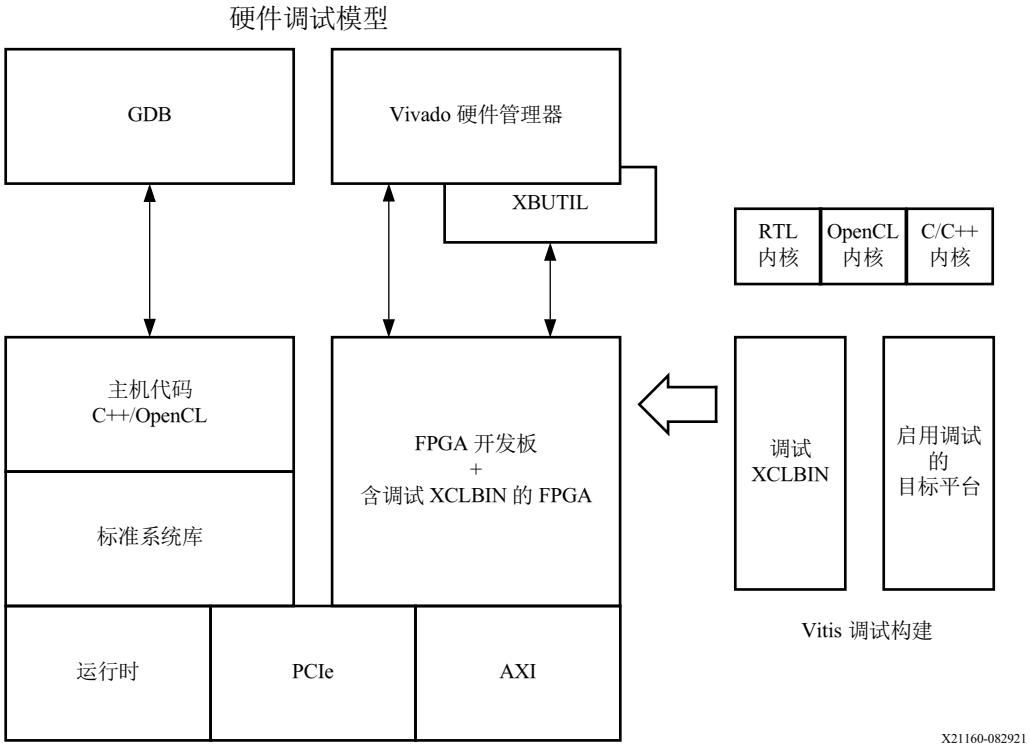
重要提示！以下步骤描述了如何从命令行进行调试。但 Vitis IDE 可提供独立调试环境以搭配从命令行创建的 Vitis 应用加速工程一起使用。如需了解更多信息，请参阅 使用独立调试流程。

在硬件执行期间，实际硬件平台用于执行内核，您只需运行应用即可评估主机程序和加速内核的性能。但要调试硬件构建，则需要将额外的逻辑整合到应用中。这将影响内核耗用的 FPGA 资源以及硬件中运行的内核的性能。硬件构建的调试配置包括特殊的 ChipScope 调试核（如 Integrated Logic Analyzer (ILA) 和 Virtual Input/Output (VIO) 核）以及用于调试的 AXI Performance Monitor。

提示：在最终量产构建中应移除调试硬件所需的额外逻辑。

下图显示的是硬件构建的调试进程，包括使用 GDB 调试主机代码和使用 Vivado 硬件管理器 (Hardware Manager) 配合波形分析、内核活动报告以及存储器访问分析来识别和定位这些硬件问题。

图 75：**硬件执行**



配置系统硬件构建以供进行调试后，即可确认 CPU 上运行的主机程序以及赛灵思器件上运行的 Vitis 加速内核在目标平台的实际硬件上是否正确执行。可识别并分析的部分状况包括：

- 由于协议违例导致的系统挂起：
  - 这些违例可能破坏整个系统。
  - 这些违例可能造成内核获得无效数据或挂起。
  - 很难确定这些违例产生的位置或时间。
  - 要对此状况进行调试，应使用由 AXI Protocol Checker（需在 Vitis 目标平台上进行配置）所触发的 ILA。
- 硬件内核问题包括：
  - 有时因实现导致的问题：时序问题、争用条件和无效的设计约束。
  - 硬件仿真未显示的功能漏洞。
- 性能问题：
  - 例如，每秒处理的帧数并非您的期望值。
  - 您可以检查数据节拍和流水打拍。
  - 通过将 ILA 与触发定序器搭配使用，您可以检查突发量、流水打拍和数据宽度以确定瓶颈的位置。

启用内核以利用 Chipscope 进行调试

System ILA

硬件调试的关键在于以所需的调试逻辑来检测内核。以下主题探讨的是可用于列出可用内核端口 `v++` 连接器选项、在选定端口上启用 System Integrated Logic Analyzer (ILA) 核以及启用 AXI Protocol Checker 调试核用于检查协议违例。

ILA 核可提供对硬件上运行的计算单元实例 (CU) 的传输事务级可视性。ILA 核还可用于捕获和查看感兴趣的 AXI 流量。ILA 可在一个或多个信号上提供定制事件触发，允许以系统速度进行波形捕获。波形可在查看器中进行分析，并用于调试硬件，以便查找协议违例或性能问题。它对于调试各种难题（例如，应用挂起）也同样至关重要。

捕获的数据可使用 Vivado 工具通过赛灵思虚拟线缆 (XVC) 来访问。欲知详情，请参阅《Vivado Design Suite 用户指南：编程和调试》(UG908)。

ILA 核可添加到现有 RTL 内核中，以在该设计内启用调试功能，或者可由 `v++` 编译器在链接阶段自动插入。`v++` 命令可提供 `--debug` 选项（如 --debug 选项 中所述）以将位于接口处的 System ILA 核连接到内核，用于调试和性能监控。

重要提示！ILA 调试核需要各种系统资源（包括逻辑和本地存储器）用于捕获和存储信号数据。因此，ILA 调试核可提供对您的内核的卓越可视性，但可能影响性能和资源使用情况。

`--debug` 选项可启用 ILA IP 核插入，其语法如下：

```
--debug.chipscope <cu_name>[:<interface_name>]>
```

提示：`<interface_name>` 为可选，如果不指定该选项，则将分析 CU 上的所有端口。您可以使用 `--debug.list_ports` 选项来返回内核上的接口名称，以搭配 `--debug` 选项一起使用。

对于扁平设计或者在主模式下包含多个 Debug Bridge 的任何设计，此流程将不会选择其中任一 Debug Bridge 来拼接调试核，需要通过约束来定义连接。以 Samsung Smart SSD U.2 平面 shell 为例，启用通过调试 (ILA) 选项来生成内核时，静态区域与动态区域之间并没有分区。必须指定从需接受调试的内核 AXI 端口到动态区域中的用户 Debug Bridge 的连接。

要指定此连接，您必须在 XOCC 命令行中提供以下选项：

```
--advanced.paramcompiler.userPostDebugProfileOverlayTcl=<path to
post_dbg_profile_overlay.tcl >
```

在 `post_dbg_profile_overlay.tcl` 内，此文件必须通过连接调试核命令来调用 XDC 文件，并提及其处理顺序。

例如，以下提供了 `post_dbg_profile_overlay.tcl` 文件中的内容。

```
read_xdc < path to the connect_debug_core.xdc file>
set_property used_in_implementation TRUE [get_files <path to the
connect_debug_core.xdc file>]
set_property PROCESSING_ORDER EARLY [get_files <path to the
connect_debug_core.xdc file>]]
```

在 `connect_debug_core.xdc` 文件中，您必须指定 `connect_debug_cores` 约束。

例如：

```
connect_debug_cores -master [get_cells -hierarchical -filter {NAME =~
*debug_bridge_xsdbm/inst/xsdbm}]
-slaves [get_cells -hierarchical -filter {NAME =~ level0_i/ulp/
system_ila_0}]
```

AXI Protocol Checker

AXI Protocol Checker 核用于监控 AXI 接口。将其连接到接口上时，它会主动检查协议违例并指示发生了哪些违例。您可在设计中为所有 CU 或者特定 CU 和端口分配该核。

`--debug` 选项用于启用 AXI Protocol Checker 插入，其语法如下：

```
 --debug.protocol all
```

协议检查器可通过关键字 `all` 或 `<cu_name>:<interface_name>` 来指定。

注释： `--debug.list_ports` 选项可指定为返回内核上端口的实际名称，以搭配 `protocol` 或 `chipscope` 一起使用。

以下提供的流程示例可供您用于为设计添加 ILA 或协议检查器：

1. 将内核源文件编译到 XO 文件中，使用 `-g` 选项来检测内核的调试功能：

   ```
   v++ -c -g -k <kernel_name> --platform <platform> -o <kernel_xo_file>.xo
   <kernel_source_files>
   ```

2. 将内核编译到 XO 文件中之后，使用 `--debug.list_ports` 会导致 `v++` 编译器打印内核的有效计算单元和端口组合列表。

   ```
   v++ -l -g --platform <platform> --connectivity.nk
   <kernel_name>:<compute_units>:<kernel_nameN>
   --debug.list_ports <kernel_xo_file>.xo
   ```

3. 在目标端口上通过将 `list_ports` 替换为相应的 `--debug.chipscope` 或 `--debug.protocol` 命令语法来添加 ILA 或 AXI 调试核：

   ```
   v++ -l -g --platform <platform> --connectivity.nk
   <kernel_name>:<compute_units>:<kernel_nameN>
   --debug.chipscope <compute_unit_name>:<interface_name>
   <kernel_xo_file>.xo
   ```

提示： 在每个 `v++` 命令行或配置文件中可多次指定 `--debug` 选项以指定多个 CU 和接口。

构建设计后，您可使用 Vivado 硬件管理器 (Hardware Manager) 来调试设计，如 利用 ChipScope 调试 中所述。

将调试 IP 添加到 RTL 内核

重要提示！ 该调试技巧要求必须熟知 Vivado Design Suite 以及 RTL 设计。

您也可以通过以下方法在 RTL 内核中启用调试：先在自己的 RTL 内核代码中手动添加 ChipScope 调试核（如 ILA 和 VIO），然后再对其进行封装以供在 Vitis 开发流程中使用。在 Vivado Design Suite 内部，编辑 RTL 内核代码以手动例化来自赛灵思 IP 目录的 ILA 调试核或 VIO IP，这与在 Vivado IDE 中使用任何其它 IP 类似。请参阅《Vivado Design Suite 用户指南：编程和调试》(UG908) 中的 HDL 例化流程，以了解有关向设计添加调试核的更多信息。

创建 RTL 内核时最适合为其添加调试核。但是，调试核会耗用器件资源并且可能影响性能，因此最好将一个内核用于调试，将另一个内核用于量产使用。GitHub 上的 RTL 内核示例的 `rtl_vadd_hw_debug` 显示了将 ILA 调试核例化到 RTL 内核源文件中的过程。ILA 按照 `src/hdl/krnl_vadd_rtl_int.sv` 文件中指定的方式来监测组合加法器的输出。

```
    // ILA monitoring combinatorial adder
    ila_0 i_ila_0 (
        .clk(ap_clk),               // input wire         clk
        .probe0(areset),            // input wire [0:0]   probe0
        .probe1(rd_fifo_tvalid_n),  // input wire [0:0]   probe1
        .probe2(rd_fifo_tready),    // input wire [0:0]   probe2
        .probe3(rd_fifo_tdata),     // input wire [63:0]  probe3
        .probe4(adder_tvalid),      // input wire [0:0]   probe4
        .probe5(adder_tready_n),    // input wire [0:0]   probe5
        .probe6(adder_tdata)        // input wire [31:0]  probe6
    );
```

您也可以使用《Vivado Design Suite 用户指南：编程和调试》(UG908) 中所描述的“网表插入”流程。在打开的 Vivado 工程中使用 Tcl 脚本来添加 ILA 调试核，如以下 Tcl 脚本示例所示：

```
create_ip -name ila -vendor xilinx.com -library ip -version 6.2 -
module_name ila_0
set_property -dict [list CONFIG.C_PROBE6_WIDTH {32} CONFIG.C_PROBE3_WIDTH
{64} \
CONFIG.C_NUM_OF_PROBES {7} CONFIG.C_EN_STRG_QUAL {1}
CONFIG.C_INPUT_PIPE_STAGES {2} \
CONFIG.C_ADV_TRIGGER {true} CONFIG.ALL_PROBE_SAME_MU_CNT {4}
CONFIG.C_PROBE6_MU_CNT {4} \
CONFIG.C_PROBE5_MU_CNT {4} CONFIG.C_PROBE4_MU_CNT {4}
CONFIG.C_PROBE3_MU_CNT {4} \
CONFIG.C_PROBE2_MU_CNT {4} CONFIG.C_PROBE1_MU_CNT {4}
CONFIG.C_PROBE0_MU_CNT {4}] [get_ips ila_0]
```

完成 RTL 内核检测并确认可通过相应的调试核进行调试之后，您即可按 利用 ChipScope 调试 中所述，在 Vivado 硬件管理器中进行硬件分析。

启用 ILA 触发器用于硬件调试

要对目标平台上运行的主机程序与内核代码执行硬件调试，必须修改应用主机代码，以确保在将内核编程到器件中之后，且在启动内核之前，您可以设置 ILA 触发器条件。

启动内核前添加 ILA 触发器

通过在代码中使用暂停 (pause) 或等待 (wait) 步骤即可完成主机程序的暂停操作，例如，GitHub 上的 RTL 内核中使用的 `wait_for_enter` 函数。该函数在 `src/host.cpp` 代码中定义，如下所示：

```
void wait_for_enter(const std::string &msg) {
    std::cout << msg << std::endl;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
```

wait_for_enter 函数在 main 函数中按如下方式使用：

```
....
    std::string binaryFile = xcl::find_binary_file(device_name,"vadd");

    cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
    devices.resize(1);
    cl::Program program(context, devices, bins);
    cl::Kernel krnl_vadd(program,"krnl_vadd_rtl");


     wait_for_enter("\nPress ENTER to continue after setting up ILA
trigger...");

    //Allocate Buffer in Global Memory
    std::vector<cl::Memory> inBufVec, outBufVec;
    cl::Buffer buffer_r1(context,CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
            vector_size_bytes, source_input1.data());
    ...

    //Copy input data to device global memory
    q.enqueueMigrateMemObjects(inBufVec,0/* 0 means from host*/);

    //Set the Kernel Arguments
    ...

    //Launch the Kernel
    q.enqueueTask(krnl_vadd);
```

使用 wait_for_enter 函数即可暂停主机程序，为您提供时间，以便您设置所需的 ILA 触发器，并准备捕获来自内核的数据。完成 Vivado 硬件管理器的设置和配置后，请按 Enter 键以继续运行应用。

- 对于 C++ 主机代码，创建 cl::Kernel 对象后请添加暂停，如以上示例所示。
- 对于 C 语言主机代码，请在 clCreateKernel() 函数调用后添加暂停：

```
// Build the program executable
//
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
  size_t len;
  char buffer[2048];

  printf("Error: Failed to build program executable!\n");
  clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
  printf("%s\n", buffer);
  printf("Test failed\n");
  return EXIT_FAILURE;
}

// Create the compute kernel in the program we wish to run
//
kernel = clCreateKernel(program, "vadd", &err);
if (!kernel || err != CL_SUCCESS)
{
  printf("Error: Failed to create compute kernel!\n");
  printf("Test failed\n");
  return EXIT_FAILURE;
}

// PAUSE
wait_for_enter("\nPress ENTER to continue after setting up ILA trigger...");

// Create the input and output arrays in device memory for our calculation
//
d_a = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(int) * LENGTH, NULL, NULL);
d_b = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(int) * LENGTH, NULL, NULL);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY,  sizeof(int) * LENGTH, NULL, NULL);
if (!d_a || !d_b || !d_c)
{
  printf("Error: Failed to allocate device memory!\n");
  printf("Test failed\n");
  return EXIT_FAILURE;
}
```

使用 GDB 暂停主机应用

如果您在内核上执行硬件调试的同时，运行 GDB 来调试主机程序，那么您也可以根据需要在相应的代码行中插入断点，以暂停主机程序。这样您无需更改主机程序以按需暂停应用，而可改为在主机代码中执行内核之前设置断点。当达到此断点时，您可在 Vivado 硬件管理器中设置调试 ILA 触发器、装备该触发器，然后在 GDB 中恢复主机程序。

利用 ChipScope 调试

您可以使用 ChipScope 调试环境和 Vivado 硬件管理器来帮助您更快速高效地调试主机应用与内核。这些工具支持从逻辑到系统级调试在内的各种功能，可供您的内核在硬件中运行时使用。为此，必须满足以下至少一项条件：

- 您的 Vitis 应用工程已使用 `--debug.xxx` 编译器开关，利用调试核完成了设计，如 启用内核以利用 Chipscope 进行调试 中所述。
- 您工程中使用的 RTL 内核必须已利用调试核进行了例化（如 将调试 IP 添加到 RTL 内核 中所述）。

检查 FPGA 开发板所包含的硬件调试支持

支持硬件调试需要平台支持多个 IP 组件，尤其是 Debug Bridge。请咨询您的平台设计师，以确定目标平台中是否包含这些组件。如果使用了赛灵思平台，则可以通过使用 `platforminfo` 实用工具查询该平台来验证调试的可用性。调试能力列在 `chipscope_debug` 对象下。

例如，要查询平台是否包含硬件调试支持，可使用以下 `platforminfo` 命令：

```
$ platforminfo --json="hardwarePlatform.extensions.chipscope_debug"
xilinx_u200_xdma_201830_2
{
    "debug_networks": {
        "user": {
            "name": "User Debug Network",
            "pcie_pf": "1",
            "bar_number": "0",
            "axi_baseaddr": "0x000C0000",
            "supports_jtag_fallback": "false",
            "supports_microblaze_debug": "true",
            "is_user_visible": "true"
        },
        "mgmt": {
            "name": "Management Debug Network",
            "pcie_pf": "0",
            "bar_number": "0",
            "axi_baseaddr": "0x001C0000",
            "supports_jtag_fallback": "true",
            "supports_microblaze_debug": "true",
            "is_user_visible": "false"
        }
    }
}
```

查询的响应显示，目标平台包含 `user` 和 `mgmt` 调试网络、支持调试 MicroBlaze™ 处理器，并且针对管理调试网络 (Management Debug Network) 还支持 JTAG 回退。

运行 XVC 和硬件服务器

以下步骤是运行赛灵思虚拟线缆 (XVC) 和硬件服务器、主机应用以及在 Vivado 硬件管理器中触发并装备调试核所必需的步骤。

1. 如需了解有关向内核添加调试 IP 的信息，请参阅 启用内核以利用 Chipscope 进行调试。
2. 如需了解有关如何修改主机程序以在相应的时间点暂停的信息，请参阅 启用 ILA 触发器用于硬件调试。
3. 硬件调试环境有两种设置方法：使用 硬件调试的自动设置 中描述的自动脚本，或者按 硬件调试的手动设置 中所述方式来手动设置。
4. 使用以下流程即可运行硬件调试流程：
    a. 启动所需的 XVC 和 Vivado 硬件管理器的 `hw_server`。
    b. 运行主机程序并在相应的时间点暂停，以启用 ILA 触发器的设置。
    c. 打开 Vivado 硬件管理器并连接到 XVC 服务器。

d. 设置设计的 ILA 触发条件。

e. 继续执行主机程序。

f. 在 Vivado 硬件管理器中检验内核活动。

g. 按需从上述步骤 b 开始重新迭代运行。

硬件调试的自动设置

1. 按 设置 Vitis 环境 中所述方式设置 Vitis 核开发套件。
2. 使用 `debug_hw` 脚本启动 `xvc_pcie` 和 `hw_server` 应用，如下所示：

```
debug_hw --xvc_pcie /dev/xvc_pub.<driver_id> --hw_server
```

`debug_hw` 脚本会返回：

```
launching xvc_pcie...
xvc_pcie -d /dev/xvc_pub.<driver_id> -s TCP::10200
launching hw_server...
hw_server -sTCP::3121
```

> **提示：** `/dev/xvc_pub.<driver_id>` 驱动程序字符路径是在您的机器上定义的，可通过检查 `/dev` 文件夹来获取。

3. 创建/下载内核后，修改主机代码以包含暂停 (pause) 语句，然后再开始执行内核，如 启用 ILA 触发器用于硬件调试 中所述。
4. 运行您修改后的主机程序。
5. 使用 `debug_hw` 脚本启动 Vivado Design Suite：

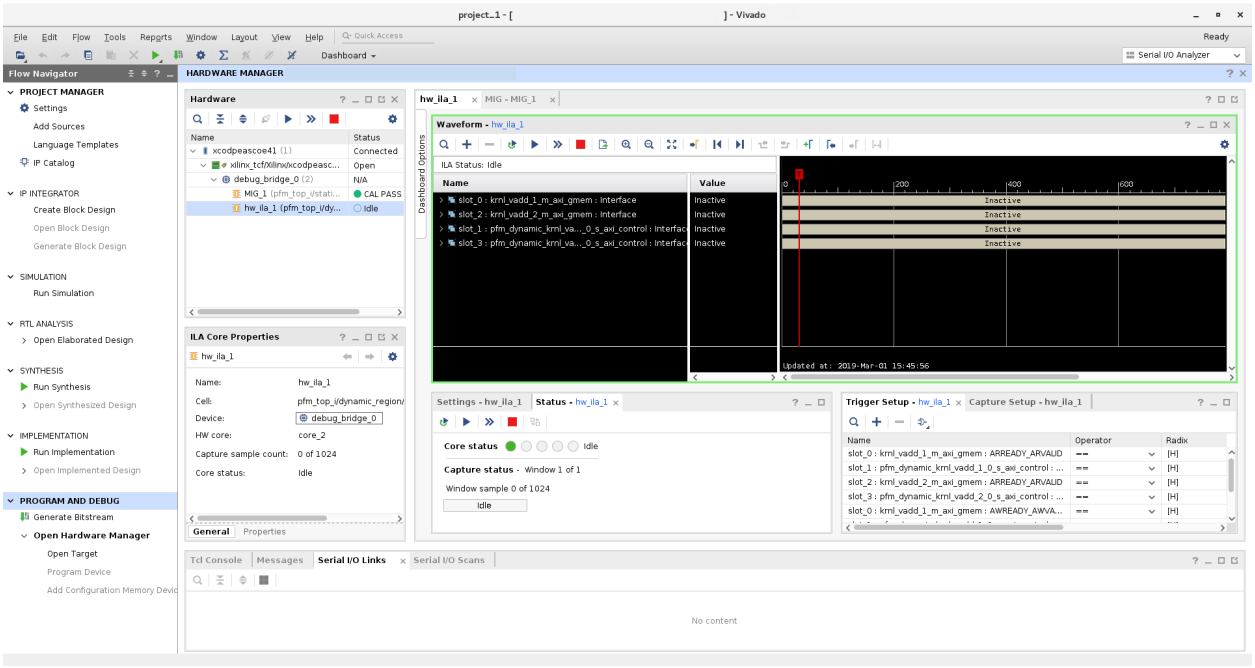
```
debug_hw --vivado --host <host_name> --ltx_file ./_x/link/vivado/vpl/prj/
prj.runs/impl_1/debug_nets.ltx
```

> **提示：** `<host_name>` 是您的系统的名称。

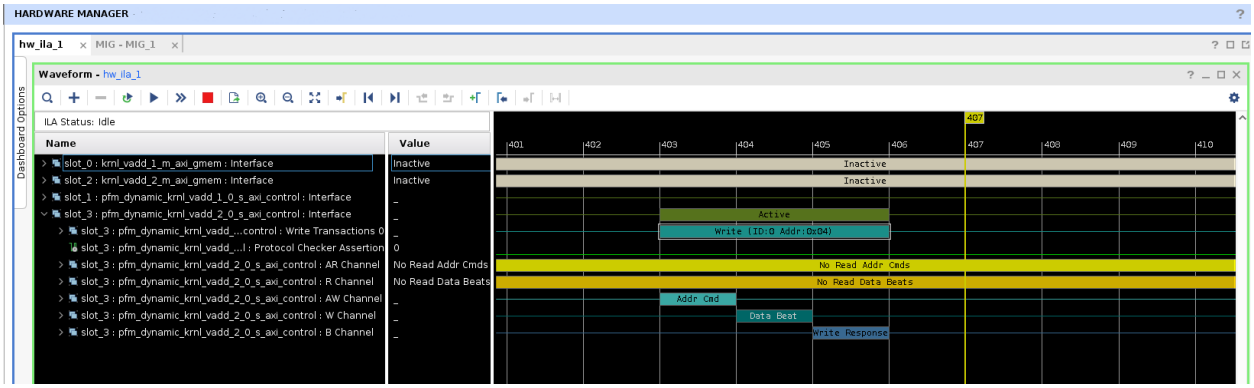
作为示例，命令窗口显示如下结果：

```
launching vivado... ['vivado', '-source', 'vitis_hw_debug.tcl', '-
tclargs',
'/tmp/project_1/project_1.xpr', 'workspace/vadd_test/System/
pfm_top_wrapper.ltx',
'host_name', '10200', '3121']

****** Vivado v2019.2 (64-bit)
  **** SW Build 2245749 on Date Time
  **** IP Build 2245576 on Date Time
    ** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.

start_gui
```

6. 在 Vivado Design Suite 中，运行 ILA 触发器。



7. 按“Enter”键以继续运行主机程序。
8. 在 Vivado 硬件管理器中，查看“Waveform”视图中有关内核计算单元从控制接口的接口传输事务。



硬件调试的手动设置

提示：在设置 Nimbix 和其它云平台时，可以使用以下步骤。

在 Vivado 硬件管理器中调试设计之前，需要先执行几个步骤来启动调试服务器。

1. 按 设置 Vitis 环境 中所述方式设置 Vitis 核开发套件。
2. 启动 `xvc_pcie` 服务器。传递给 `xvc_pcie` 的文件名必须与随内核器件驱动程序安装的字符驱动程序文件名称相匹配，其中 `<driver_id>` 可通过检查 `/dev` 文件来找到。

```
>xvc_pcie -d /dev/xvc_pub.<device_id>
```

提示：`xvc_pcie` 服务器具有很多有用的命令行选项。您可以发送 `xvc_pcie -help` 来获取可用选项的完整列表。

3. 在端口 3121 上启动 hw_server，并使用以下命令连接到端口 10201 上的 XVC 服务器：

```
>hw_server -e "set auto-open-servers xilinx-xvc:localhost:10201" -e "set always-open-jtag 1"
```

4. 启动 Vivado Design Suite 并打开硬件管理器：

```
vivado
```

在 Amazon F1 实例上启动调试服务器

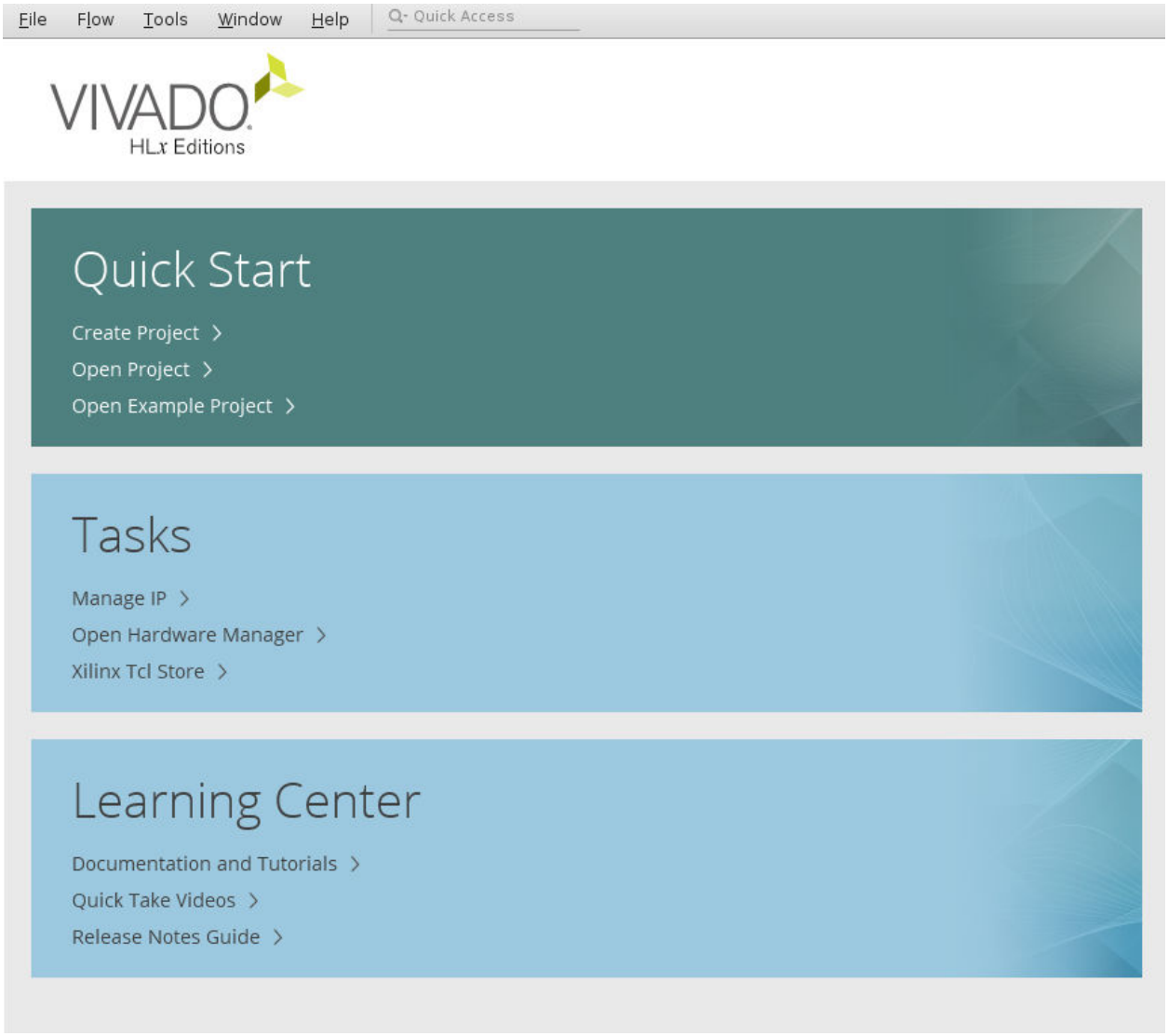
如需获取有关在 Amazon F1 实例上启动调试服务器的说明，请参阅此处：https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md。

使用 Vivado 硬件管理器来调试设计

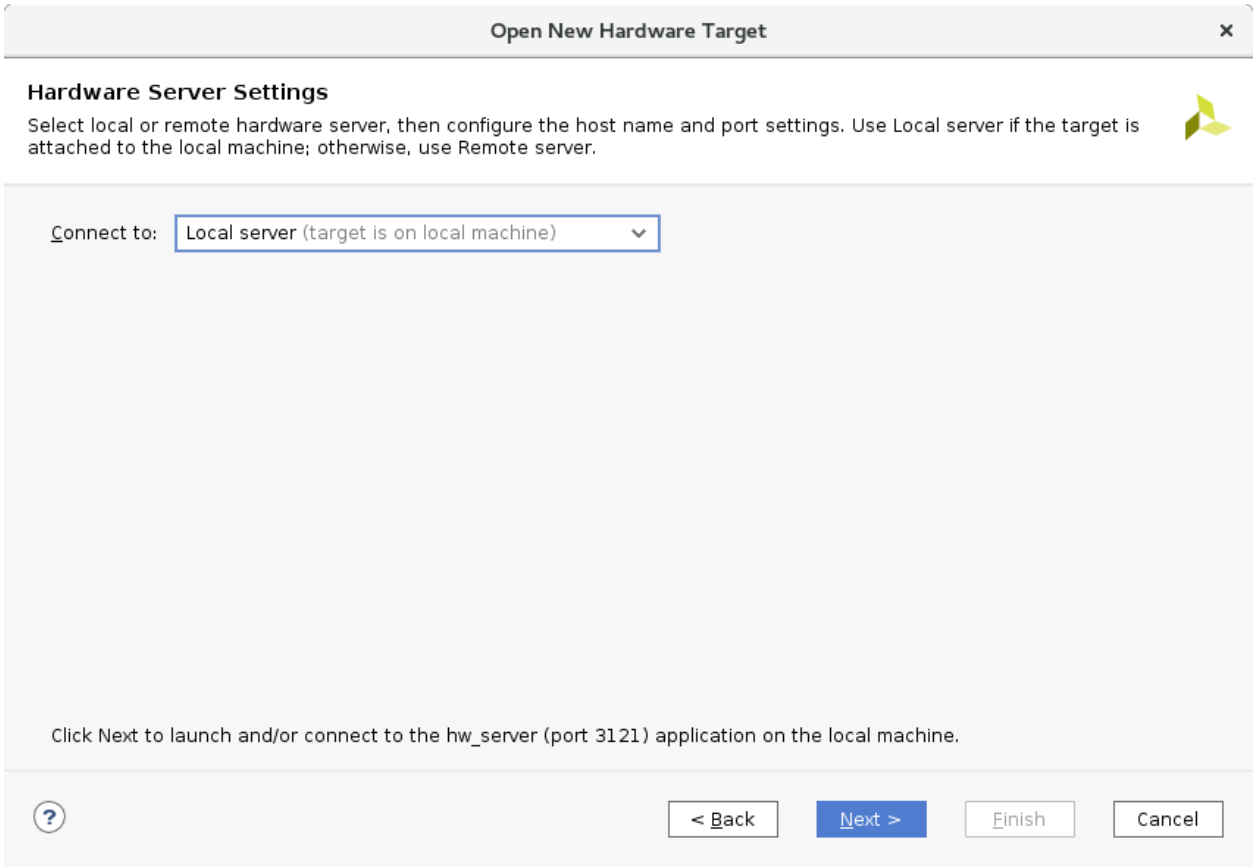
传统上，物理 JTAG 连接用于通过 Vivado 硬件管理器来为赛灵思器件执行硬件调试。Vitis 统一软件平台还使用赛灵思虚拟线缆 (XVC) 在远端加速器卡上执行硬件调试。为充分利用此功能，Vitis 调试器使用 XVC 服务器，它是 XVC 协议的实现，支持 Vivado 硬件管理器使用标准赛灵思调试核（如 ILA 或 VIO IP）连接到本地或远端目标器件以进行调试。

可在目标实例上运行 Vivado 硬件管理器（通过 Vivado Design Suite 或 Vivado 调试功能）或者，也可以在不同主机上远程运行此硬件管理器。运行 Vivado 硬件管理器的主机必须能够访问 XVC 服务器监听的 TCP 端口。若要将 Vivado 硬件管理器连接到目标上的 XVC 服务器，必须在托管 Vivado 工具的机器上执行以下步骤：

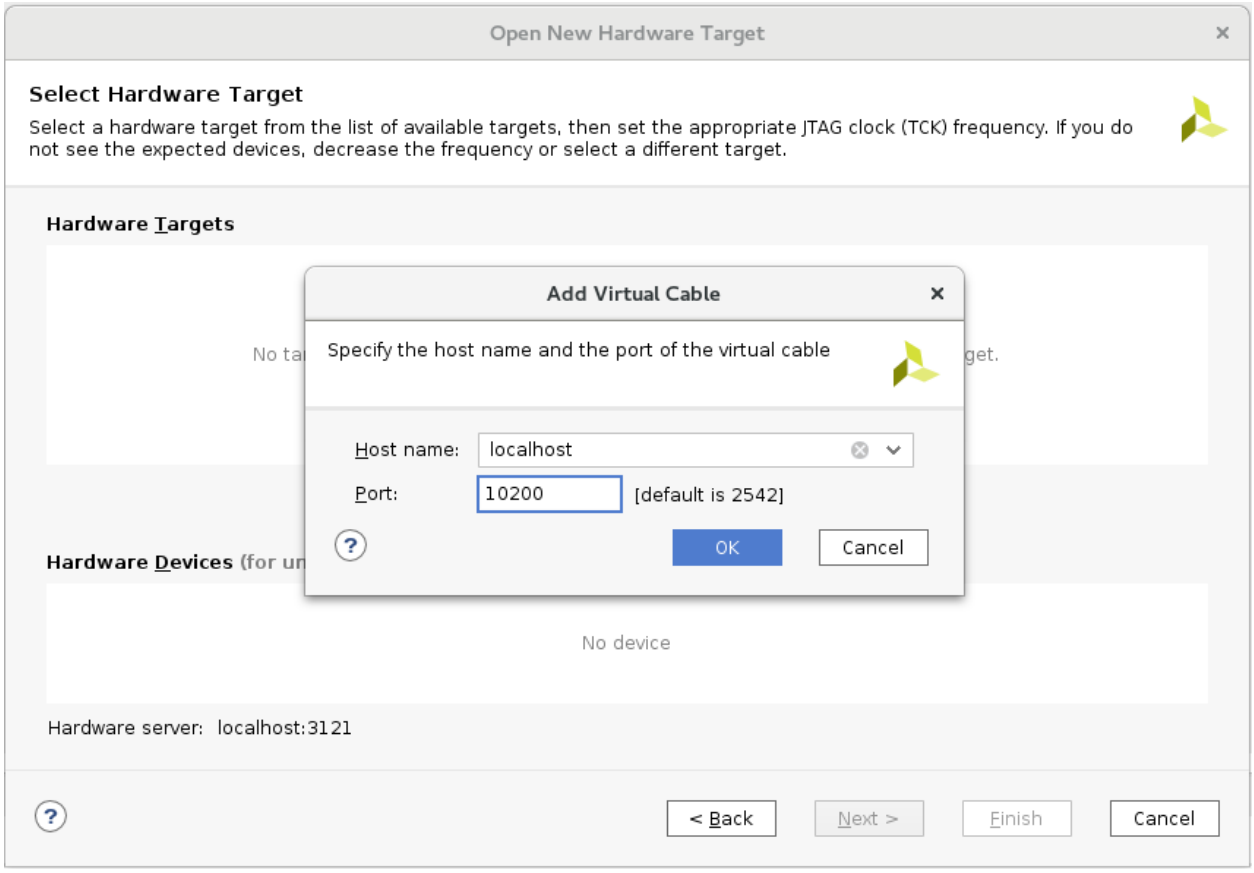
1. 启动 Vivado 调试功能或者完整版 Vivado Design Suite。
2. 从“任务 (Tasks)”菜单中选择“Open Hardware Manager”，如下图所示。



3. 连接到 Vivado 工具 `hw_server`，指定本地或远程连接，以及“Host name”和“Port”，如下所示。



4. 连接到目标实例虚拟 JTAG XVC 服务器。

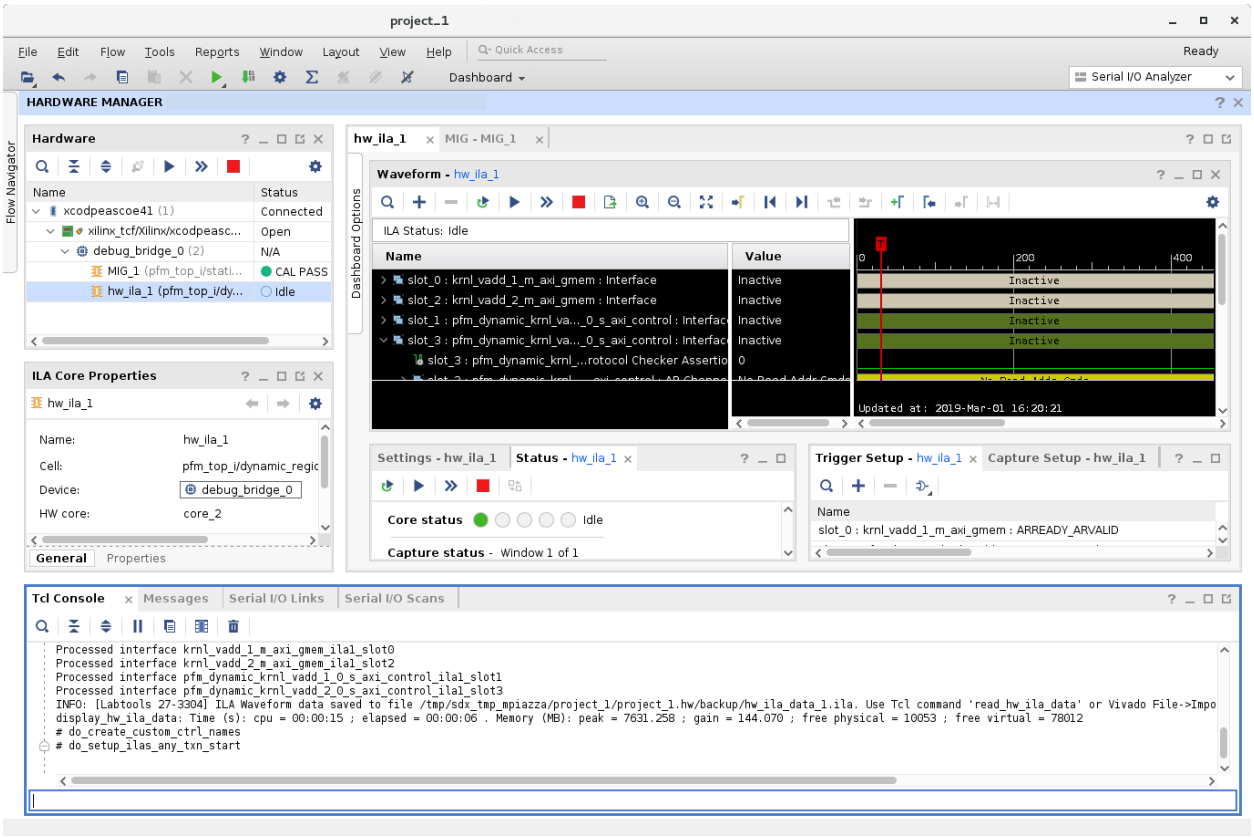


5. 从 Vivado 硬件管理器的“硬件 (Hardware)”窗口中选择 `debug_bridge` 实例。

   为设计指定探针文件 (`.ltx`)，将其添加到“硬件器件属性 (Hardware Device Properties)”窗口中的“Probes” → “File”条目下。添加探针文件会刷新硬件器件，“Hardware”窗口现在应显示设计中的调试核。

   > **提示：**如果内核具有 启用内核以利用 Chipscope 进行调试 中指定的调试核，那么在内核实现期间，Vivado 工具会写出探针文件 (`.ltx`)。

6. 现在，Vivado 硬件管理器可用于调试 Vitis 软件平台上运行的内核了。在内核中装备 ILA 核并运行主机应用。



提示：如需了解有关使用 Vivado 硬件管理器来调试设计的更多信息，请参阅《Vivado Design Suite 用户指南：编程和调试》(UG908)。

专用调试网络的 JTAG 回退

由于物理卡和卡上的 JTAG 连接器不可访问，Alveo 数据中心加速器卡的硬件调试通常使用 XVC-over-PCIe 连接。尽管 XVC-over-PCIe 允许您对目标平台上运行的应用进行远程调试，但诸如 AXI 互连系统挂起等条件可能阻止您访问依赖于这些 PCIe/AXI 功能特性的硬件调试功能。对于平台设计师而言，调试此类状况的能力尤为重要。

JTAG 回退功能是专为支持访问原先只能通过 XVC-over-PCIe 访问的调试网络而设计的。JTAG 回退功能可直接启用，无需更改平台设计中基于 XVC-over-PCIe 的调试网络。

在主机端，当 Vivado 硬件管理器用户通过 `hw_server` 连接到 JTAG 线并且该 JTAG 线已连接到加速器卡的 JTAG 物理管脚或待测器件 (DUT) 时，`hw_server` 会禁用 XVC-over-PCIe 到硬件的路径。这样您即可使用 XVC-over-PCIe 线作为自己的主要调试路径，并在某些情况下按需直接启用通过 JTAG 线进行调试。当您断开与 JTAG 线的连接时，`hw_server` 会重新启用 XVC-over-PCIe 到硬件的路径。

JTAG 回退步骤

以下是启用 JTAG 回退所需的步骤：

1. 启用调试网络的“Debug Bridge”（AXI 到 BSCAN 模式）主控制器的 JTAG 回退功能，您需为该调试网络提供 JTAG 访问。此步骤会在此“Debug Bridge”实例上启用 BSCAN 从接口。
2. 在平台设计的静态逻辑分区中例化另一个“Debug Bridge”（BSCAN Primitive 模式）。

3. 将步骤 2 的“Debug Bridge”的 BSCAN 主端口（BSCAN Primitive 模式）连接到步骤 1 的“Debug Bridge”的 BSCAN 从接口（AXI-to-BSCAN 模式）。

硬件调试工具

在某些情况下，正常的 Vitis IDE 和命令行调试功能在隔离问题上能力有限。当软件或硬件似乎未取得任何进展（挂起）时更是如此。这种系统问题利用本节中所述的工具进行分析效果最好。

使用 Linux dmesg 实用工具

精心设计的内核和模块通过内核环形缓冲器报告问题。此方法对于 Vitis 技术模块同样有效，这些模块允许您在最低 Linux 级别上对与加速器开发板的互动进行调试。

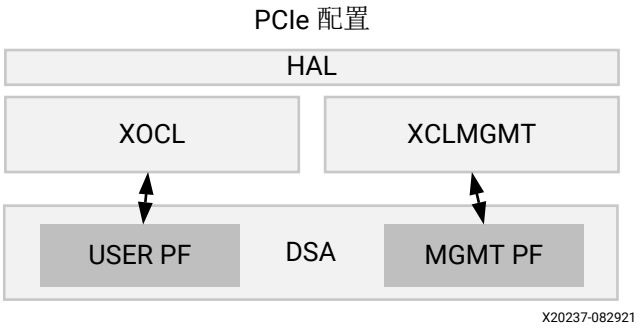
`dmesg` 实用工具是一种 Linux 工具，可让您读取内核环形缓冲器。内核环形缓冲器可将内核参考消息保存在圆形缓冲器内。固定大小尺寸的圆形缓冲器可用于限制资源要求，方法是通过利用下一条传入消息来覆盖最旧的条目。

提示：大多数情况下，利用较简洁的 `xbutil` 功能即可足以将问题局限在某一区域内。如需了解有关如何使用此工具进行调试的更多信息，请参阅 使用赛灵思 xbutil 实用工具。

在 Vitis 技术中，`xocl` 模块和 `xclmgmt` 驱动程序模块会向环形缓冲器写入参考消息。因此，对于应用挂起、崩溃或任何意外行为（例如，无法进行比特流编程等），都应使用 `dmesg` 工具来检查环形缓冲器。

下图显示了与目标平台关联的软件平台的各层次。

图 76：**软件平台层**



要复查来自 Linux 工具的消息，应首先清空环形缓冲器：

```
sudo dmesg -c
```

此操作将从环形缓冲器中刷新所有消息，使其更容易发现来自 `xocl` 和 `xclmgmt` 的消息。完成此操作后，启动您的应用并在另一个终端中运行 `dmesg`。

```
sudo dmesg
```

`dmesg` 实用工具会打印一条记录，如以下示例所示：

图 77：**dmesg 实用工具示例**

在以上示例中，AXI 防火墙 2 已脱扣，最好使用 `xbutil` 实用工具来对其进行检查。

使用赛灵思 xbutil 实用工具

赛灵思开发板实用工具 (`xbutil`) 是一种强大的单机命令行工具，可用于调试更低级别的硬件/软件交互问题。如需获取有关此实用工具的完整描述，请参阅 xbutil 实用工具。

关于调试，请特别关注以下 `xbutil` 选项：

- `query`：提供有关卡的整体状态信息，包括卡的存储器中的内核相关信息。
- `program`：将二进制文件 (`xclbin`) 下载至赛灵思器件的可编程区域内。
- `status`：提取各性能监控器（`aim` 和 `asm`）的状态以及 Lightweight AXI Protocol Checker (`lapc`) 的状态。

应用挂起调试技巧

本节讨论了与主机代码和加速内核互动相关的调试问题。这些互动相关的问题表现为诸如机器挂起或应用挂起之类的现象。尽管 GDB 调试环境可能在某些情况下有助于隔离错误 (`xprint`)，比如与具体内核相关的挂起，但是最好利用 `dmesg` 和 `xbutil` 命令来进行调试，如本文所示。

如果硬件调试进程无法解决问题，则必须使用 ChipScope 功能来执行硬件调试。

AXI 防火墙脱扣

AXI 防火墙应能够预防主机挂起。因此在所有量产 Vitis 平台中都包含 Axi Protocol Firewall IP。当防火墙脱扣时，首先要检查的是确认主机代码与内核是否设置为使用相同的存储体。以下步骤详述了如何执行此项检查：

1. 使用 `xbutil` 对 FPGA 进行编程：

   ```
   xbutil program -p <xclbin>
   ```

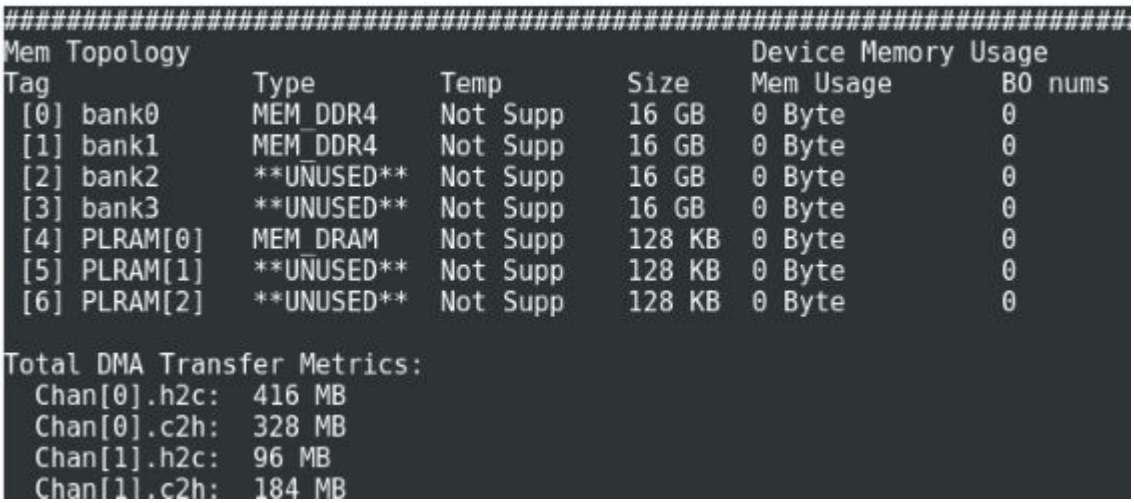
   

   提示：如需了解有关 `xbutil` 的更多信息，请参阅 xbutil 实用工具。

2. 运行 `xbutil` 查询选项，检查存储器拓扑结构：

   ```
   xbutil query
   ```

   在以下示例中，没有与存储体关联的内核：



3. 如果主机代码预计会使用任何 DDR 存储体/PLRAM，则该报告应指示存在问题。在此情况下，有必要检查期望的内核与主机代码。如果主机代码正在使用赛灵思 OpenCL 扩展，则必须检查内核应使用哪些 DDR 存储体。所使用的这些存储体应与指定的 `connectivity.sp` 选项相匹配，如 将内核端口映射到存储器 中所述。

由于 AXI 违例导致内核挂起

内核可能因内核与存储器控制器之间的 AXI 传输事务错误而导致挂起。要调试这些问题，必须对内核进行检测。

1. Vitis 核开发套件提供了 2 个选项，可供要在 `v++` 链接 (`--link`) 期间应用的检测使用。这 2 个选项都会为您的实现添加硬件，并且根据资源使用情况，它可能需要对检测加以限制。

   a. 添加 Lightweight AXI Protocol Checker (`lapc`)。这些协议检查器是使用 `--debug.protocol` 选项来添加的，如 --debug 选项 中所述。所使用的语法如下：

   ```
   --debug.protocol <compute_unit_name>:<interface_name>
   ```

   一般而言，`<interface_name>` 是可选项。如果未指定该选项，那么将会对 CU 上的所有端口进行分析。`--debug.protocol` 选项用于定义要插入的协议检查器。该选项可接受特殊关键字 `all`，用于 `<compute_unit_name>` 和/或 `<interface_name>`。

   注释： 在单一命令行或配置文件内可指定多个 `--debug.xxx` 选项。

   b. 添加 Performance Monitor (`am, aim, asm`) 可启用详细通信统计数据（计数器）列表。尽管它对于性能分析最为有用，但它在调试暂挂的端口活动中同样可以提供深入见解。这些 Performance Monitor 是使用 `--profile` 选项来添加的，如 --profile 选项 中所述。`--profile` 选项的基本语法是：

   ```
   --profile.data <krnl_name>|all:<cu_name>|all:<intrfc_name>|
   all:<counters>|all
   ```

   需要 3 个字段用于确定性能监控器要连接到的具体接口。但是，如果资源使用不成问题，关键字 `all` 支持您通过单一选项即可对所有现有内核、计算单元和接口应用监控。否则，您可以明确指定 `kernel_name`、`cu_name` 和 `interface_name` 来限制检测。

   最后一个选项 `<counters>|all` 允许您将信息收集操作范围局限于大型设计的 `counters`，而 `all`（默认）则包含收集实际追踪信息。

注释： 在单一命令行或配置文件内可指定多个 `--profile` 选项。

```
[profile]
dataernel1:cu1:m_axi_gmem0
dataernel1:cu1:m_axi_gmem1
dataernel2:cu2:m_axi_gmem
```

2. 重建应用后，使用含已添加的 AIM IP 和 LAPC IP 的 `xclbin` 重新运行主机应用。
3. 当应用挂起时，您可以使用 `xbutil status` 来校验任何错误或异常情况。
4. 检查 AIM 输出：
   - 多次运行 `xbutil status --aim` 来检查是否有任何计数器正在计数。如果计数器正在计数，则内核处于活动状态。

     提示：通过使用命令扩展 `xstatus aim` 进行 GDB 调试还可以支持测试 AIM 输出。

   - 如果计数器停滞，大于零的未完成计数可能意味着某些 AXI 传输事务已挂起。
5. 检查 LAPC 输出：
   - 运行 `xbutil status --lapc` 来检查是否存在任何 AXI 违例。

     提示：通过使用命令扩展 `xstatus lapc` 进行 GDB 调试还可以支持测试 LAPC 输出。

   - 如果存在任何 AXI 违例，则意味着在内核实现中存在问题。

访问存储器时主机应用挂起

应用挂起也可能是由于从主机代码发起的 DMA 传输不完整而造成的。这不一定意味着主机代码错误，也可能是因为内核发出了非法传输事务并锁定了 AXI。

1. 如果平台拥有 AXI 防火墙（如在 Vitis 平台中），则有可能会脱扣。驱动程序会发出 `SIGBUS` 错误、终止该应用并复位器件。您可以通过运行 `xbutil query` 来检查此情况。下图显示了防火墙状态中的此类错误。

```
Firewall Last Error Status:
        0:        0x0     (GOOD)
        1:        0x0     (GOOD)
        2:        0x80000 (RECS_WRITE_TO_BVALID_MAX_WAIT).
                  Error occurred on Tue 2017-12-19 11:39:13 PST

Xclbin ID:    0x5a39da87
```

   提示：如果防火墙未脱扣，则 Linux 工具 `dmesg` 可提供更多洞察。

2. 如果已知防火墙脱扣，则重要的是确定 DMA 超时的原因。该问题可能是由于存在非法 DMA 传输或内核行为不当而导致的。但是，AXI 防火墙脱扣的副作用是驱动程序中的运行状况检查功能会在终止应用之后复位开发板，器件上可能有助于调试根本原因的任何信息都会丢失。要调试该问题，请禁用 `xclmgmt` 内核模块中的运行状况检查线程，以捕获错误。此操作将按照以下顺序使用通用 Unix 内核工具：
   a. `sudo modinfo xclmgmt`：该命令列出模块的当前配置并指示 `health_check` 参数是 ON（开启）还是 OFF（关闭）。它同时返回指向 `xclmgmt` 模块的路径。
   b. `sudo rmmod xclmgmt`：移除并禁用 `xclmgmt` 内核模块。
   c. `sudo insmod <path to module>/xclmgmt.ko health_check=0`：此命令将重新安装 `xclmgmt` 内核模块并禁用运行状况检查。

提示：在 `modinfo` 调用的输出中会报告此模块的路径。

3. 在禁用运行状况检查的情况下，重新运行应用。您可以按前述方法使用内核检测来隔离此问题。

导致应用挂起的典型错误

造成应用挂起的典型用户错误列举如下：

- 在 5.0+ 目标平台中先读取后写入 (read-before-write) 会导致出现 Memory Interface Generator 纠错码 (MIG ECC) 错误。这是一种典型的用户错误。例如，如果内核应在 DDR 中写入 4 KB 数据，但它仅生成 1 KB 数据，随后尝试将整个 4 KB 数据传输到主机，那么就可能产生此错误。如果您向内核提供 1 KB 缓冲器，但内核尝试读取 4 KB 数据，也可能发生该错误。
- 如果从导致 MIG 初始化的最近一次比特流下载开始，未向存储器位置写入任何数据，但对该存储器位置提交读取请求，那么也会产生 ECC 先读取后写入错误。ECC 错误会导致受影响的 MIG 停滞，因为内核通常无法处理此错误。这种情况可能表现为两种不同的方式：
  1. 计算单元可能挂起或停滞，因为它在从受影响的 MIG 中读取或写入的同时无法处理该错误。`xbutil` 查询显示计算单元卡在 `BUSY` 状态，而且没有进展。
  2. 如果向受影响的 MIG 提出了 PCIe® DMA 请求，则 AXI 防火墙可能脱扣，因为 DMA 引擎无法完成该请求。AXI 防火墙脱扣将导致 Linux 内核驱动程序终止利用 `SIGBUS` 信号打开器件节点的所有进程。`xbutil` 查询可显示 AXI 防火墙是否确实已脱扣并包含时间戳。

  如果上述挂起未发生，则主机代码可能未读回正确的数据。此错误数据通常为 0，并且位于数据最后一部分。请务必谨慎复查主机代码，这至关重要。此类状况的常见示例之一是压缩，其中被压缩数据的大小预先未知，而应用可能尝试向主机移植数据，且移植的数据量多于内核产生的数据量。

防御性编程

Vitis 编译器能够创建非常高效的实现。但是在某些情况下，可能会发生实现问题。其中一种情况是，发出了写入请求，但进程中没有足够数据可用于完成写入传输事务。如果此问题影响多个并发内核，并且内核的写入请求要求完成输入读取，那么这就可能会造成死锁状况。

为避免这种情况，在适配器上提供了一种保守模式。原则上，它可延迟写入请求，直至它具备完成写入所需的所有数据为止。可通过在 `v++` 编译器中应用以下 `--advanced.param` 选项，在编译过程中启用该模式。

```
--advanced.param:compiler.axiDeadLockFree=yes
```

由于启用该模式可能会影响性能，您也许更希望将其作为防御性编程技巧来使用，即在开发和测试过程中插入该选项，而后在最优化过程中将其移除。您也可以在加速器反复挂起时添加该选项。

在嵌入式处理器平台上进行调试

在嵌入式处理器平台（例如，`xilinx_zcu104_base_202010_1` 平台）上进行调试需使用 QEMU 仿真环境对器件的 Arm 处理器和操作系统进行建模。如下列章节中所述，运行或调试应用需要执行额外的步骤来启动仿真器或者通过 TCF 代理连接到硬件平台。

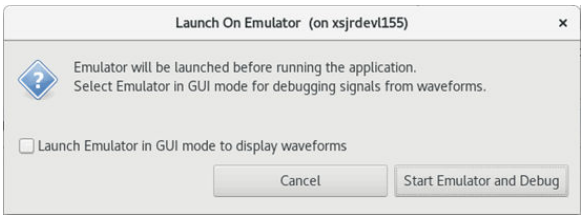
嵌入式处理器的仿真调试

在 Vitis IDE 中，为软件和硬件仿真构建启动调试包含以下步骤：

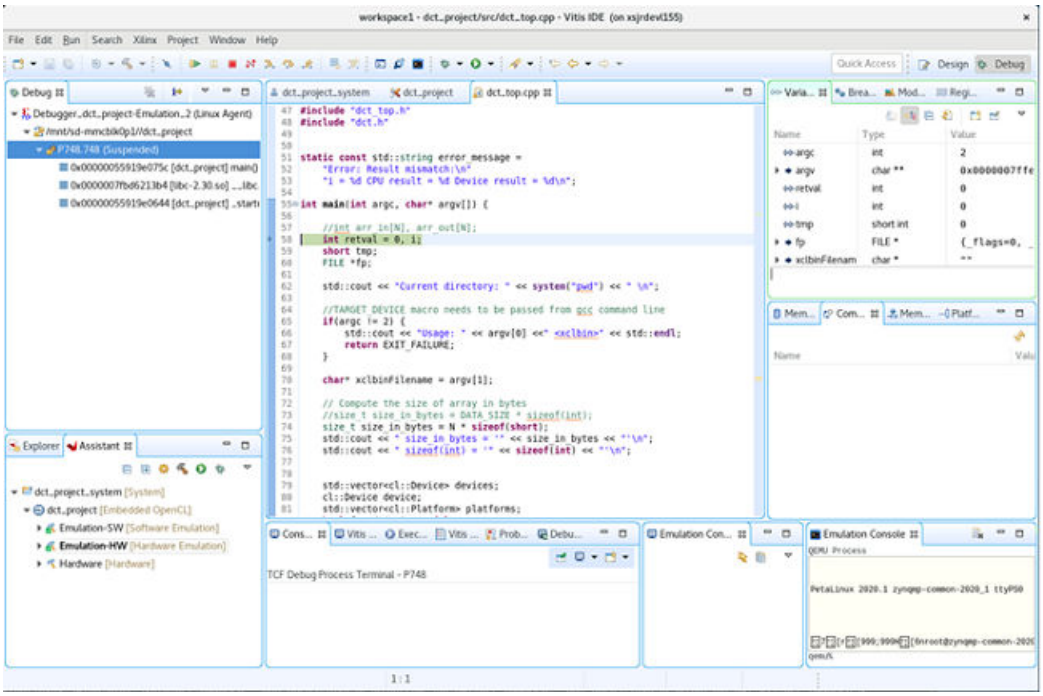
1. 在“Assistant”视图中，右键单击“Emulation-SW”或“Emulation-HW”构建，然后选中“Set Active”以激活构建。

2. 在“Assistant”视图菜单中，选中“Debug”命令 ( )，然后选中“Launch on Emulator”命令以启动调试环境。

   这样将在“Emulator”上打开“Launch”，如下图所示。系统会提示您确认是否要启动仿真环境，并使用 Linux TCF 代理连接到此环境。选择“Start Emulator and Debug”以继续。



   这样就会启动仿真环境 (QEMU)，并加载应用以准备调试。当应用进入 `main()` 函数时，会暂停。“调试 (Debug)”透视图会在 Vitis IDE 中打开，这表示您已准备好开始调试应用。

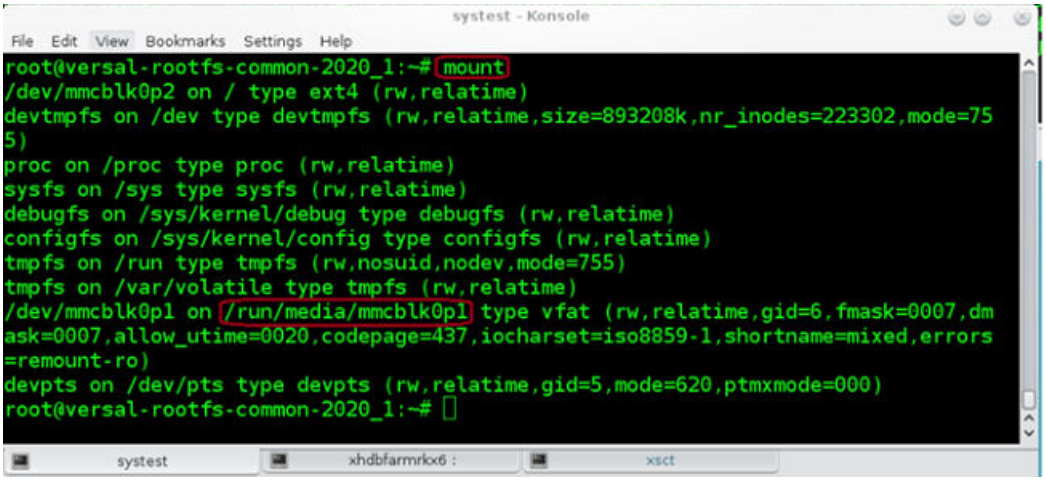


嵌入式处理器的硬件调试

对于硬件构建，设置包含下列步骤：

1. 将 `<project>/Hardware/sd_card/sd_card` 文件夹的内容复制到实体 SD 卡。这样即可为您的目标平台创建一个可启动的介质。
2. 将此 SD 卡插入嵌入式处理器平台的读卡器。
3. 将平台的启动模式设置更改为 SD 启动模式，然后给开发板上电。
4. 器件启动后，在命令提示符处输入 `mount` 命令以获取装载点列表。如下图所示，`mount` 命令显示了系统的装载信息。

提示：根据 `mount` 命令结果，请务必捕获下一步中 `cd` 命令以及后续命令的适当路径。

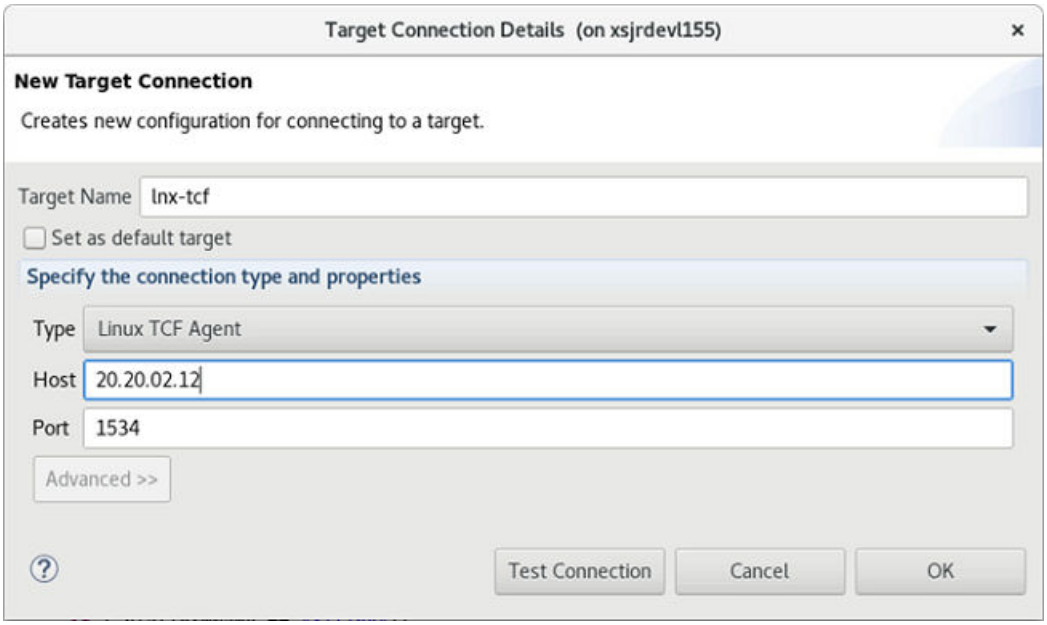


5. 例如，执行下列命令：

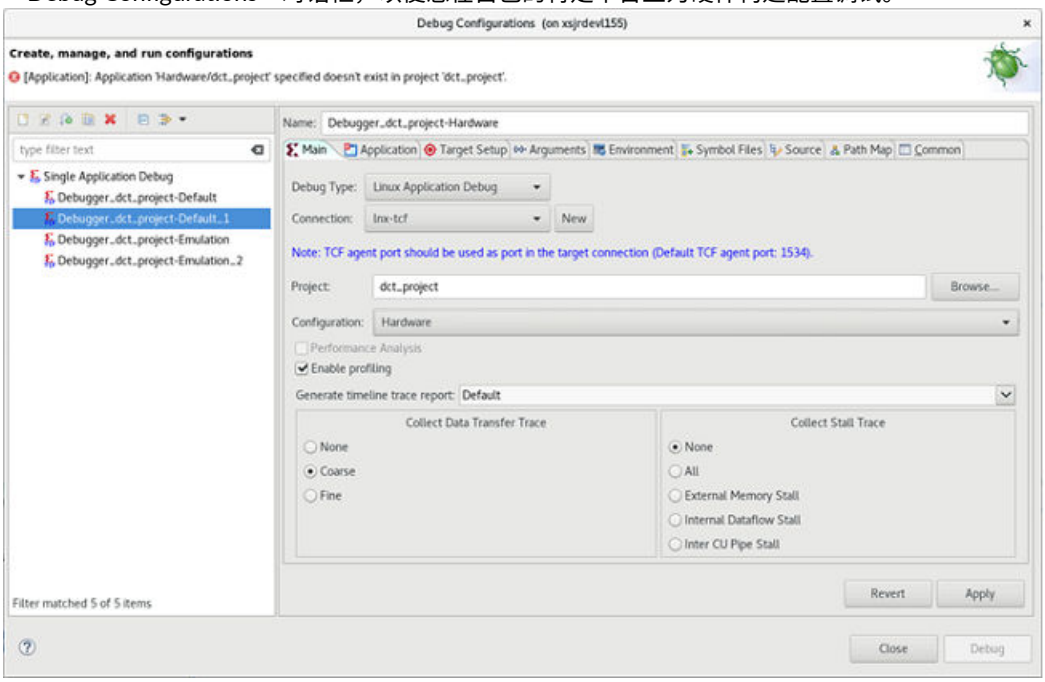
```
cd /run/media/mmcblkop1
source init.sh
cat /etc/xocl.txt
```

`cat` 命令将显示平台名称 `xilinx_vck190_base_202010_1`，以供您确认它与您指定的平台是否相同，以及确认您的设置是否正确。

6. 运行 `ifconfig` 以获取目标卡的 IP 地址。您将使用此 IP 地址来设置 Vitis IDE 中的 TCF 代理连接，以便连接到为嵌入式处理器平台分配的 IP 地址。
7. 创建到远程加速器卡的目标连接。使用“Window” → “Show view” → “Xilinx” → “Target connections”命令即可打开“Target Connections”视图。
8. 在“Target Connections”视图中，右键单击“Linux TCF Agent”并选择“New Target”命令以打开“New Target Connection”对话框。
9. 指定“Target Name”、启用“Set as default target”复选框，并指定上一步中获取的加速器卡的“Host”的 IP 地址。



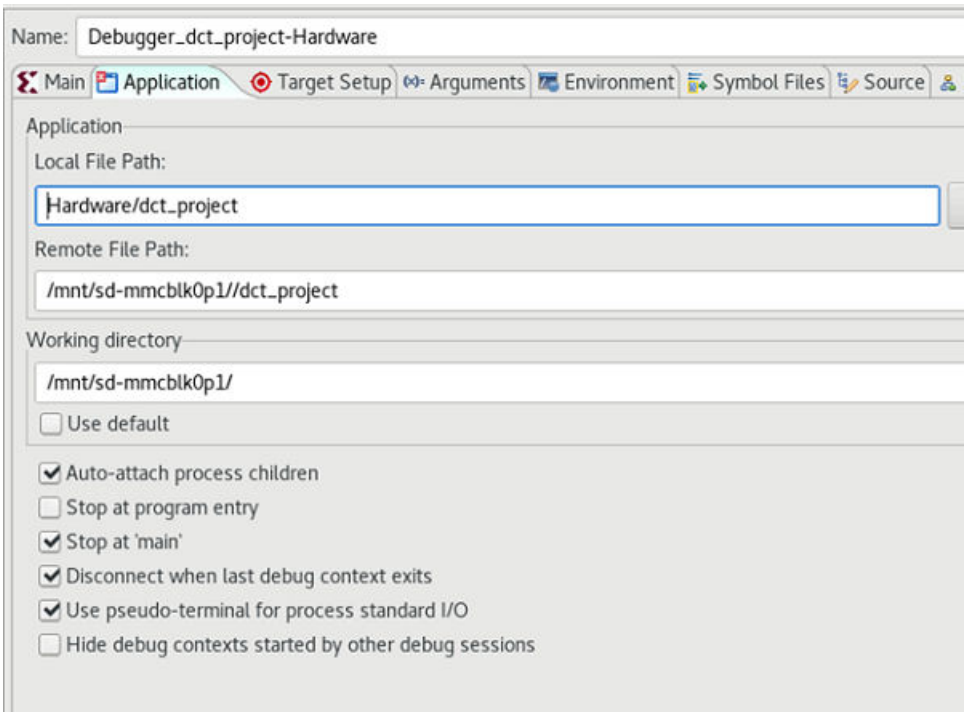
10. 单击“OK”即可关闭此对话框并继续操作。
11. 在“Assistant”视图中，右键单击“Hardware build”并选中“Set Active”以使其成为活动的构建。
12. 在“Assistant”视图菜单中，选中 Debug ( ) 命令，然后选择“Debug Configurations”命令。这样即可打开“Debug Configurations”对话框，以便您在自己的特定平台上为硬件构建配置调试。



在此对话框的“Main”选项卡上设置以下字段：

- “Name”：为您的硬件调试配置指定名称。
- “Linux TCF Agent”：选择您使用指定 IP 地址为加速器卡构建的新代理。
- “Configuration”：确保已选中“Hardware”配置。
- “Enable Profiling”：如果要从事件捕获追踪数据，请执行以下操作。

选择“Debug Configuration”对话框中的“Application”选项卡以查看下列字段：



在“Application”选项卡上设置下列字段：

- “Local File Path”：指定目标平台上创建的文件写回到您的本地磁盘中的位置。
- “Remote File Path”：指定来自先前步骤中确定的加速器卡的远程装载位置。
- “Working directory”：指定目标平台上创建的文件的写入位置。

13. 选择“Apply”以保存更改，然后选择“Debug”以启动此流程。

    这样即可在 Vitis IDE 中打开“Debug”透视图，并连接到硬件平台上的 PS 应用。此应用会在 `main()` 函数处自动中断，以便您设置并配置调试环境。

命令行调试示例

为帮助您熟悉使用命令行进行调试的流程，本示例将为您逐步讲解构建和调试赛灵思 GitHub 上提供的 IDCT 示例的过程。

1. 在终端内，按 设置 Vitis 环境 中所述设置您的环境。
2. 请首先克隆 Vitis 示例 GitHub 仓库，以获取所有 Vitis 示例：

   ```
   git clone https://github.com/Xilinx/Vitis_Accel_Examples.git
   ```

   这样即可创建包含 IDCT 示例的 `Vitis_Examples` 目录。
3. 使用 CD 命令转至 IDCT 示例目录：

   ```
   cd Vitis_Examples/vision/idct/
   ```

   主机代码完全包含在 `src/idct.cpp` 中，内核代码则包含在 `src/krnl_idct.cpp` 中。

4. 构建内核软件用于软件仿真 (software emulation)，如 构建器件二进制文件 中所述。
   a. 编译内核对象文件，以便使用 v++ 编译器进行调试，其中 -g 表示编译的代码用于调试：

```
v++ -t sw_emu --platform <DEVICE> -g -c -k krnl_idct \
-o krnl_idct.xo src/krnl_idct.cpp
```

   b. 链接内核对象文件，并指定 -g：

```
v++ -g -l -t sw_emu --platform <DEVICE> -config config.cfg \
-o krnl_idct.xclbin krnl_idct.xo
```

   --config 选项用于指定配置文件 config.cfg，其中包含构建进程（如 Vitis 编译器配置文件 中所述）的指令。配置文件内容如下：

```
kernel_frequency=250

[connectivity]
nk=krnl_idct:1:krnl_idct_1

sp=krnl_idct_1.m_axi_gmem0:DDR[0]
sp=krnl_idct_1.m_axi_gmem1:DDR[0]
sp=krnl_idct_1.m_axi_gmem2:DDR[1]

[advanced]
prop=solution.hls_pre_tcl='src/hls_config.tcl"
```

5. 编译并链接主机代码，以便使用 GNU 编译器链 g++ 进行调试，如 构建主机程序 中所述：

   注释：对于嵌入式处理器目标平台，请使用 GNU Arm 交叉编译器，如 为 Arm 执行编译和链接 中所述。

   a. 编译主机代码 C++ 文件，以便使用 -g 选项进行调试：

```
g++ -c -I${XILINX_XRT}/include -g -o idct.o src/idct.cpp
```

   b. 链接对象文件，以便使用 -g 进行调试：

```
g++ -g -lOpenCL -lpthread -lrt -lstdc++ -L${XILINX_XRT}/lib/ -o idct
idct.o
```

6. 如 emconfigutil 实用工具 中所述，使用以下命令准备仿真 (emulation) 环境：

```
emconfigutil --platform <device>
```

   随后，需通过 XCL_EMULATION_MODE 环境变量来设置实际仿真模式（sw_emu 或 hw_emu）。在 C-shell 中，操作如下：

```
setenv XCL_EMULATION_MODE sw_emu
```

7. 如 xrt.ini 文件 中所述，您必须设置运行时以供调试。在已编译的主机应用所在目录中，创建包含以下内容的 xrt.ini 文件：

```
[Debug]
app_debug=true
```

8. 在主机和内核代码上运行 GDB。以下步骤用于指引您完成命令行调试进程，此进程需要 3 个独立的命令终端，其设置如 设置 Vitis 环境 中所述。
   a. 在第一个终端内，启动 XRT 调试服务器，用于处理主机与内核代码之间的传输事务：

```
${XILINX_VITIS}/bin/xrt_server --sdx-url
```

b. 在第二个终端中，设置仿真模式：

```
setenv XCL_EMULATION_MODE sw_emu
```

执行以下操作，运行 GDB：

```
xgdb --args idct krnl_idct.xclbin
```

出现 `gdb` 提示时，输入以下内容：

```
run
```

c. 在第三个终端中，将软件仿真模型连接到 GDB 以单步调试整个设计。启动另一个 `xgdb`：

```
xgdb
```

- 对于软件仿真中的调试：
  - 出现 `gdb` 提示时，输入以下内容：

    ```
    file <XILINX_VITIS>/data/emulation/unified/cpu_em/generic_pcie/
    model/genericpciemodel
    ```

    注释：由于 GDB 不会展开环境变量，您必须指定到 Vitis 软件平台安装的路径，此路径以 `<XILINX_VITIS>` 来表示
- 连接到内核进程：

  ```
  target remote :NUM
  ```

  其中 `NUM` 是 `xrt_server` 返回的作为 GDB 监听器端口的端口号。

此时，即可照常使用 GDB 来执行主机与内核代码调试，主机代码和内核代码在两个不同的 GDB 会话内运行。在处理不同进程时，此操作很常见。

重要提示！请注意，应用可能先在某一进程中命中一个断点，然后在另一个进程中命中下一个断点。在此类情况下，其中一个终端内的调试会话会显示为挂起，另一个终端则处于等待输入状态。

第五部分

Vitis 环境参考资料

此处包含的参考资料包括：

- Vitis 编译器命令：编译器选项 (`-c`) 描述、链接选项 (`-l`) 描述、编译和链接共用选项的描述以及有关 `--config` 选项的探讨。
- 赛灵思为 Vitis 工具提供了诸多实用工具，还包括赛灵思的 Xilinx Runtime (XRT)，用于提供有关平台资源的详细信息（如 SLR 和存储器资源可用性等）以帮助您构造 `v++` 命令行，并管理构建和运行进程。
  - emconfigutil 实用工具
  - kernelinfo 实用工具
  - launch_emulator 实用工具
  - manage_ipcache 实用工具
  - package_xo 命令
  - platforminfo 实用工具
  - xbutil 实用工具
  - xbmgmt 实用工具
  - xclbinutil 实用工具

  提示：赛灵思的 Xilinx Runtime (XRT) 架构参考资料可通过 Xilinx Runtime GitHub 仓库获取。
- `xrt.ini` 文件用于初始化 XRT 以生成报告，并在主机与内核之间传输数据时对数据进行调试和剖析。在运行应用时，此文件用于仿真或硬件构建，并且在从命令行运行构建进程时，必须手动创建此文件。
- HLS 编译指示：Vitis HLS 工具在综合 C/C++ 内核过程中所使用的编译指示的描述。

XILINX

第 21 章

Vitis 编译器命令

本章节描述了 Vitis 编译器命令 `v++` 及其支持的各种选项，这些选项可用于编译和链接 FPGA 二进制文件。

Vitis 编译器是一个独立命令行实用工具，可用于将内核加速器函数编译到赛灵思对象 (XO) 文件中，并将其与其它 XO 文件和受支持的平台相链接，以构建 FPGA 二进制文件。

如需了解有关如何使用 `v++` 命令选项执行编译、链接、封装和常规进程的更多信息，请参阅下列章节：

- 使用 Vitis 编译器来编译内核
- 链接内核
- 封装系统

Vitis 编译器常规选项

Vitis 编译器支持许多同时适用于编译进程和链接进程的选项。这些选项可提供诸多功能特性，覆盖范围广泛，部分选项专用于编译或链接，也有部分选项则可同时用于或者必须同时应用于编译和链接。

提示：所有 Vitis 编译器选项均可在搭配 `--config` 选项一起使用的配置文件中指定，如 Vitis 编译器配置文件中所述。例如，通过使用如下语法，即可在不含节头的配置文件中指定 `--platform` 选项：

```
platform=xilinx_u200_xdma_201830_2
```

--advanced

- 适用于：编译和链接

指定参数和属性，以供 `v++` 命令使用。如需了解更多信息，请参阅 --advanced 选项。

--board_connection

- 适用于：编译和链接

```
--board_connection
```

为每个 DIMM 连接器插槽指定一个双列直插式内存模块 (DIMM) 开发板文件。此开发板是使用该 DIMM 卡的 Vendor:Board:Name:Version (vbnv) 属性（显示在开发板存储库中）指定的。

例如：

```
<DIMM_connector>:<vbnv_of_DIMM_board>
```

-c | --compile

· 适用于：编译

```
--compile
```

编译的必需选项，但与 `--link` 和 `--package` 互斥。运行 `v++ -c` 以从内核源文件生成 XO 文件。

--clock

· 适用于：链接

提供一种方法，用于在链接进程中向内核分配时钟。如需了解更多信息，请参阅 --clock 选项。

--config

· 适用于：编译、链接和封装

```
--config <config_file> ...
```

指定包含 `v++` 命令选项的配置文件。此配置文件可用于捕获编译、链接或封装策略，通过查看 `v++` 命令行上的配置文件即可轻松复用这些策略。此外，该配置文件允许将 `v++` 命令行缩短为仅包含配置文件中未指定的选项。如需了解更多信息，请参阅 Vitis 编译器配置文件。

提示：在 `v++` 命令行上可指定多个配置文件。所使用的每个字段都需要 1 个独立的 `--config` 开关。例如：

```
v++ -l --config cfg_connectivity.cfg --config cfg_vivado.cfg ...
```

--connectivity

· 适用于：链接

用于在链接进程中指定重要的器件二进制文件架构详细信息。如需了解更多信息，请参阅 --connectivity 选项。

--custom_script

· 适用于：编译和链接

```
--custom_script <kernel_name>:<file_name>
```

该选项支持您指定在编译或链接期间，构建 (build) 进程中将使用的定制 Tcl 脚本。搭配 `--export_script` 选项一起使用，用于创建、编辑和运行脚本以自定义构建进程。

搭配 `v++ --compile` 命令一起使用时，该选项允许您指定定制 HLS 脚本，以供在编译指定内核时使用。此脚本支持您修改或自定义 Vitis HLS 工具。`--export_script` 选项可用于提取 Vitis HLS 用于编译内核的 Tcl 脚本、按需修改此脚本并使用 `--custom_script` 选项重新提交，以便更好地管理内核构建进程。

实参支持您指定内核名称以及指向要应用于该内核的 Tcl 脚本的路径。例如：

```
v++ -c -k kernel1 -export_script ...
*** Modify the exported script to customize in some way, then resubmit. ****
v++ -c --custom_script kernel1:./kernel1.tcl ...
```

搭配 `v++ --link` 命令一起用于硬件构建目标 (`-t hw`) 时，该选项允许您指定指向已编辑的 `run_script_map.dat` 文件的绝对路径。该文件包含构建进程中的步骤列表以及 Vitis 和 Vivado 工具在执行这些步骤期间运行的 Tcl 脚本。您可编辑 `run_script_map.dat` 以指定在构建进程中执行这些步骤时要运行的定制 Tcl 脚本。要自定义 Tcl 脚本，必须使用以下步骤：

1. 运行构建进程并指定 `--export_script` 选项，如下所示：

   ```
   v++ -t hw -l -k kernel1 -export_script ...
   ```

2. 针对要自定义的任意步骤，复制 `run_script_map.dat` 文件中引用的 Tcl 脚本。例如，复制针对综合运行或实现运行所指定的 Tcl 文件。您必须将此文件复制到位于工程构建结构外部的独立位置。
3. 编译此 Tcl 脚本，添加或修改任意现有命令以创建新的定制 Tcl 脚本。
4. 编译 `run_script_map.dat` 文件，将特定实现步骤指向新的定制脚本。
5. 使用 `--custom_script` 选项重新启动构建进程，指定指向 `run_script_map.dat` 文件的绝对路径，如下所示：

   ```
   v++ -t hw -l -k kernel1 -custom_script /path/to/run_script_map.dat
   ```

重要提示！编译定制综合运行脚本时，必须注释掉与 `dont_touch.xdc` 文件相关的行，或者编辑这些行，使其指向用户指定的新 `dont_touch.xdc` 文件。要注释或编辑的特定行如下所示：

```
read_xdc dont_touch.xdc
set_property used_in_implementation false [get_files dont_touch.xdc]
```

如不执行此操作，则综合运行会返回有关缺少 `dont_touch.xdc` 文件的错误。

--debug

- 适用于：链接

指定在器件二进制文件 (`.xclbin`) 中插入的调试 IP 核。如需了解更多信息，请参阅 --debug 选项。

-D | --define

- 适用于：编译和链接

```
--define <arg>
```

有效的宏名称和定义对：`<name>=<definition>`。

将名称预定义为含定义的宏。该选项将传递给 `v++` 预处理器。

--export_script

- 适用于：编译和链接

```
--export_script
```

该选项可运行构建进程，直至导出脚本文件或者脚本文件列表为止，然后停止执行。构建进程必须使用 `--custom_script` 选项来完成。这样您即可编辑导出的脚本或脚本列表，然后使用您的定制脚本来重新运行构建。

该选项搭配 `v++ --compile` 命令一起使用即可为指定内核导出 Tcl 脚本 `<kernel_name>.tcl`（可用于执行 Vitis HLS），但会在实际启动 HLS 工具前停止构建进程。这样您即可中断构建进程以编辑生成的 Tcl 脚本，然后使用 `--custom_script` 选项重新启动构建进程，如以下示例所示：

```
v++ -c -k kernel1 -export_script ...
```

提示：针对 OpenCL 内核的软件仿真 (`-t sw_emu`) 不支持该选项。

该选项搭配 `v++ --link` 命令一起用于硬件构建目标 (`-t hw`) 时，即可在当前目录中导出 `run_script_map.dat` 文件。该文件包含构建进程中的步骤列表以及 Vitis 和 Vivado 工具在执行这些步骤期间运行的 Tcl 脚本。您可编辑指定的 Tcl 脚本、自定义这些脚本中的构建进程，然后使用 `--custom_script` 选项重新启动构建。使用以下命令导出 `run_script_map.dat` 文件：

```
v++ -t hw -l -k kernel1 -export_script ...
```

--from_step

· 适用于：编译和链接

```
--from_step <arg>
```

为 Vitis 编译器构建进程指定步骤名称，以从该步骤开始启动构建进程。如有中间结果可用，那么链接进程会快进至该指定步骤开始执行（如可能）。这样您即可通过 `--to_step` 运行构建，然后在通过某种方式与工程进行交互之后，于 `--from_step` 处恢复构建进程。您可使用 `--list_step` 选项来判定有效步骤的列表。

重要提示！ `--from_step` 和 `--to_step` 选项属于顺序构建选项，要求您在使用 `--from_step` 启动 Vitis 编译器以恢复构建时，使用的工程目录应与您使用 `--to_step` 启动构建时所指定的工程目录相同。

例如：

```
v++ --link --from_step vpl.update_bd
```

-g

· 适用于：编译和链接

```
-g
```

生成代码用于在软件仿真期间进行内核调试。该选项可用于添加相应的功能特性，以促进内核编辑时的调试。

例如：

```
v++ -g ...
```

-h | --help

```
-h
```

打印 `v++` 命令的帮助内容。例如：

```
v++ -h
```

--hls

· 适用于：编译

指定内核编译期间 Vitis HLS 综合进程的选项。如需了解更多信息，请参阅 --hls 选项。

-I | --include

· 适用于：编译和链接

```
--include <arg>
```

在目录列表中添加指定目录，此列表可供搜索头文件。该选项将传递给 Vitis 编译器预处理器。

<input_file>

· 适用于：编译和链接

```
<input_file1> <input_file2> ...
```

指定 OpenCL 或 C/C++ 内核源文件用于执行 v++ 编译，或者指定赛灵思对象 (XO) 文件用于执行 v++ 链接。

例如：

```
v++ -l kernel1.xo kernelRTL.xo ...
```

--interactive

· 适用于：编译和链接

```
--interactive [ impl ]
```

v++ 能以实现工程来配置所需环境并启动 Vivado 工具。

由于您采用交互式方式来启动 Vivado 工具，因此在完成 vpl 步骤后链接进程即停止，这等同于在 v++ 命令中使用 --to_step vpl 选项。

以交互方式处理完 Vivado 工具并保存设计检查点 (DCP) 后，您可使用 v++ --from_step rtdgen 或者使用 --reuse_impl 或 --reuse_bit 选项来恢复 Vitis 编译器链接进程，以便读入已实现的 DCP 文件或比特流。

例如：

```
v++ --interactive impl
## Interactively use the Vivado tool
v++ --from_step rtdgen
```

-k | --kernel

· 适用于：编译

```
--kernel <arg>
```

仅编译来自输入文件的指定内核。每条 v++ 命令仅允许一个 -k 选项。有效值包括要编译的内核名称（来自输入 .cl 或 .c/.cpp 内核源代码）。

该选项对于 C/C++ 内核而言是必需的，但对于 OpenCL 内核而言则是可选选项。OpenCL 使用 `kernel` 关键字来识别内核。对于 C/C++ 内核，您必须通过 `-k` 或 `--kernel` 来识别内核。

如果编译 OpenCL 源文件时不使用 `-k` 选项，则会编译文件中的所有内核。使用 `-k` 可指定特定内核作为目标。

例如：

```
v++ -c --kernel vadd
```

--kernel_frequency

重要提示！此命令用于具有 `changeable` 时钟的传统平台，对于新平台 shell，可将其替换为 --clock 选项 命令。如需了解更多信息，请参阅 管理时钟频率。

- 适用于：编译和链接

```
--kernel_frequency <freq> | <clockID>:<freq>[<clockID>:<freq>]
```

为内核指定用户定义的时钟频率（以 MHz 为单位），覆盖硬件平台上定义的默认时钟频率。`<freq>` 可为仅有单个时钟的内核指定单一频率，或者也可用于为支持 2 个时钟的内核指定 <clockID> 和 <freq>。

在仅含单个内核时钟的平台上覆盖时钟的语法是只需指定以 MHz 为单位的频率即可：

```
v++ --kernel_frequency 300
```

要在含 2 个时钟的平台上覆盖特定时钟，请指定时钟 ID 和频率：

```
v++ --kernel_frequency 0:300
```

要在多时钟平台上覆盖 2 个时钟，请指定每个时钟 ID 和对应的频率。例如：

```
v++ --kernel_frequency 0:300|1:500
```

-l | --link

```
--link
```

这是编译后的链接进程的必需选项，但该选项与 `--compile` 或 `--package` 互斥。在链接模式下运行 `v++` 即可链接 XO 输入文件并生成 `xclbin` 输出文件。

--linkhook

- 适用于：链接

支持您通过指定将在实现进程的特定步骤中运行的 Tcl 脚本，为器件二进制文件自定义构建进程。如需了解更多信息，请参阅 --linkhook 选项。

--list_steps

- 适用于：编译和链接

```
--list_steps
```

列出给定目标的有效运行步骤。该选项会返回可在 `--from_step` 或 `--to_step` 选项中使用的步骤列表。此命令指定时，必须包含以下选项：

- `-t | --target [sw_emu | hw_emu | hw ]`：
- `[ --compile | --link ]`：指定来自指定构建目标的编译进程或链接进程的步骤列表。

例如：

```
v++ -t hw_emu --link --list_steps
```

--log_dir

- 适用于：编译和链接

```
--log_dir <dir_name>
```

指定用于存储日志文件的目录。如不指定 `--log_dir`，则该工具会将日志文件保存到 `./_x/logs` 中。如需了解更多信息，请参阅 v++ 命令的输出目录。

例如：

```
v++ --log_dir /tmp/myProj_logs ...
```

--message_rules

- 适用于：编译和链接

```
--message-rules <file_name>
```

指定消息规则文件，其中包含用于控制消息的规则。如需了解更多信息，请参阅 使用消息规则文件。

例如：

```
v++ --message_rules ./minimum_out.mrf ...
```

--no_ip_cache

- 适用于：编译和链接

```
--no_ip_cache
```

为 Vivado 综合的非关联 (OOC) 综合禁用 IP 高速缓存。禁用 IP 高速缓存存储库要求该工具为每次构建重新生成 IP 综合结果，可能导致增加构建时间。但它也能生成清洁无错的构建，删除设计中先前的 IP 结果。

例如：

```
v++ --no_ip_cache ...
```

-O | --optimize

- 适用于：编译和链接

```
--optimize <arg>
```

该选项可指定 Vivado 实现结果的最优化级别。有效的最优化值包括：

- `0`：默认最优化缩短编译时间。
- `1`：通过运行 Vivado 实现策略 `Power_DefaultOpt` 进行最优化，以减少功耗。这会导致设计构建时间增加。
- `2`：通过最优化来提升内核速度。该选项会增加构建时间，但也能通过在实现中添加 `PHYS_OPT_DESIGN` 步骤来改善生成的内核的性能。
- `3`：此最优化可在生成的代码中提供最高级别的性能，但是编译时间会显著增加。该选项可指定综合期间的重定时，并在实现期间启用 `PHYS_OPT_DESIGN` 和 `POST_ROUTE_PHYS_OPT_DESIGN`。
- `s`：对设计大小进行最优化。它可通过运行 `Area_Explore` 实现策略来减少内核所使用的器件逻辑资源。
- `quick`：缩短 Vivado 实现时间，但是会降低内核性能，并增加内核所使用的资源。它能为综合和实现启用 `Flow_RuntimeOptimized` 策略。

例如：

```
v++ --link --optimize 2
```

-o | --output

- 适用于：编译和链接

```
-o <output_name>
```

指定 `v++` 命令所生成的输出文件的名称。编译 (`-c`) 进程输出名称必须以赛灵思对象文件的 XO 文件后缀结尾。链接 (`-l`) 进程输出文件必须以赛灵思可执行二进制文件的 `xclbin` 文件后缀结尾。

例如：

```
v++ -o krnl_vadd.xo
```

如不指定 `--o` 或 `--output`，则输出文件名默认设置为：

- `a.o`（对应编译）。
- `a.xclbin`（对应链接）。

-p | --package

- 适用于：封装

指定选项，供 Vitis 编译器用于对设计进行封装，以运行仿真或在硬件上运行。如需了解更多信息，请参阅 --package 选项。

-f | --platform

- 适用于：编译和链接

```
--platform <platform_name>
```

指定受支持的加速平台（由 `$PLATFORM_REPO_PATHS` 环境变量指定）的名称，或者指定指向平台 `.xpfm` 文件的完整路径。要获取该版本支持的平台列表，请参阅 Vitis 软件平台版本说明。

该选项对于编译和链接都属于必需选项，用于定义构建进程的目标赛灵思平台。`--platform` 选项可接受平台名称或者指向平台文件 `xpfm`（使用完整路径或相对路径）的路径。

重要提示！指定的编译和链接的平台和构建目标必须相互匹配。编译生成 XO 文件时所指定的 `--platform` 和 `-t` 选项必须与链接期间所使用的 `--platform` 和 `-t` 相同。如需了解更多信息，请参阅 platforminfo 实用工具。

例如：

```
v++ --platform xilinx_u200_xdma_201830_2 ...
```

提示：所有 Vitis 编译器选项均可在搭配 `--config` 选项一起使用的配置文件中指定。例如，通过使用如下语法，即可在不含节头的配置文件中指定 `platform` 选项：

```
platform=xilinx_u200_xdma_201830_2
```

--profile

· 适用于：编译和链接

指定选项用于配置赛灵思运行时环境以便捕获应用性能信息。如需了解更多信息，请参阅 --profile 选项。

--remote_ip_cache

· 适用于：编译和链接

```
--remote_ip_cache <dir_name>
```

指定远程 IP 高速缓存目录的位置，以供 Vivado 综合在 IP 的非关联 (OOC) 综合期间使用。OOC 综合允许 Vivado 综合工具复用 IP 的综合结果，这些综合结果在设计迭代过程中未进行更改。由于复用综合结果，因此可缩短 `.xclbin` 文件所需的构建时间。

不指定 `--remote_ip_cache` 选项时，IP 高速缓存会写入原先从中启动 `v++` 的当前工作目录。您可使用该选项来指定其它高速缓存位置，此位置可供多个工程使用。

例如：

```
v++ --remote_ip_cache /tmp/IP_cache_dir ...
```

--report_dir

· 适用于：编译和链接

```
--report_dir <dir_name>
```

指定用于存储报告文件的目录。如不指定 `--report_dir`，则该工具会将报告文件保存到 `./_x/reports` 中。如需了解更多信息，请参阅 v++ 命令的输出目录。

例如：

```
v++ --report_dir /tmp/myProj_reports ...
```

-R | --report_level

- 适用于：编译和链接

```
--report_level <arg>
```

有效的报告级别：0、1、2 和 estimate。

这些报告级别的映射保存在 optMap.xml 文件中。您可改写已安装的 optMap.xml 以定义定制报告级别。

- -R0 规格会在 Vivado 实现期间关闭所有中间设计检查点 (DCP) 生成操作。开启布线后时序报告生成。
- -R1 规格包含来自 -R0 的一切以及 report_failfast pre-opt_design 和 report_failfast post-opt_design，还会启用所有中间 DCP 生成操作。
- -R2 规格包含来自 -R1 的一切以及 report_failfast post-route_design。
- -Restimate 规格会强制 Vitis HLS 生成 design.xml 文件（如果此文件不存在），然后生成“系统估算 (System Estimate)”报告，如 系统估算报告 中所述。

提示：如果默认不生成 design.xml，该选项可用于软件仿真构建 (-t sw_emu)。

例如：

```
v++ -R2 ...
```

--reuse_bit

```
--reuse_bit <arg>
```

- 适用于：链接

指定生成的比特流文件 (.bit) 的路径和文件名，生成器件二进制文件 (xclbin) 时将使用此比特流文件。如 使用 -to_step 并以交互方式启动 Vivado 中所述，您可指定 --to_step 选项以中断 Vitis 构建进程并对已综合的设计进行手动布局布线以生成比特流。

重要提示！ --reuse_bit 选项属于顺序构建选项，要求您在使用 --reuse_bit 恢复 Vitis 编译器时，使用的工程目录与您使用 --to_step 启动构建时所指定的工程目录相同。

例如：

```
v++ --link --reuse_bit ./project.bit
```

--reuse_impl

```
--reuse_impl <arg>
```

- 适用于：链接

指定已实现的设计检查点 (DCP) 文件的路径和文件名，生成器件二进制文件 (xclbin) 时将使用此 DCP 文件。链接进程使用已实现的指定 DCP 来提取 FPGA 比特流并生成 xclbin。您可手动编辑由先前已完成的 Vitis 构建所创建的 Vivado 工程，或者也可指定 --to_step 选项以中断 Vitis 构建进程，并对已综合的设计进行手动布局布线。这样您即可以交互方式来使用 Vivado Design Suite，在构建进程中更改设计和使用 DCP。

重要提示！ `--reuse_impl` 选项属于增量构建选项，要求您在使用 `--reuse_impl` 恢复 Vitis 编译器时，使用的工程目录与您使用 `--to_step` 启动构建时所指定的工程目录相同。

例如：

```
v++ --link --reuse_impl ./manual_design.dcp
```

-s | --save-temps

- 适用于：编译和链接

```
--save-temps
```

指示 `v++` 命令保存在编译进程和链接进程期间所创建的中间文件/目录。`--temp_dir` 选项可用于指定中间文件的写入位置。

提示：在构建进程中遇到问题时，该选项可供您用于调试。

例如：

```
v++ --save_temps ...
```

-t | --target

- 适用于：编译和链接

```
-t [ sw_emu | hw_emu | hw ]
```

指定构建目标，如 构建目标 中所述。构建目标会判定编译进程和链接进程的结果。您可以选择构建仿真模型用于调试和测试，或者构建真实系统以供在硬件中运行。如不指定 `-t`，那么构建目标默认设置为 `hw`。

重要提示！指定的编译和链接的平台和构建目标必须相互匹配。编译生成 XO 文件时所指定的 `--platform` 和 `-t` 选项必须与链接期间所使用的 `--platform` 和 `-t` 相同。

有效值包括：

- `sw_emu`：软件仿真
- `hw_emu`：硬件仿真
- `hw`：硬件

例如：

```
v++ --link -t hw_emu
```

--temp_dir

- 适用于：编译和链接

```
--temp_dir <dir_name>
```

该选项允许您管理在构建进程中所创建的临时文件的写入位置。临时结果由 `v++` 编译器写入，然后被移除，除非同时指定 `--save-temps` 选项。

如不指定 `--temp_dir`，那么该工具会将临时文件保存到 `./_x/temp` 中。如需了解更多信息，请参阅 v++ 命令的输出目录。

例如：

```
v++ --temp_dir /tmp/myProj_temp ...
```

--to_step

· 适用于：编译和链接

```
--to_step <arg>
```

为编译进程或链接进程指定步骤名称，以便通过该步骤来运行构建进程。您可使用 `--list_step` 选项来判定有效的编译步骤或链接步骤的列表。

构建进程会在完成指定步骤后终止。此时，您即可与构建结果进行交互。例如，手动访问 HLS 工程或者 Vivado Design Suite 工程以执行特定任务，然后再返回构建流程并以 `--from_step` 选项启动 `v++` 命令。

重要提示！ `--to_step` 和 `--from_step` 选项属于增量构建选项，要求您在使用 `--from_step` 启动 Vitis 编译器以恢复构建时，使用的工程目录应与您使用 `--to_step` 启动构建时所指定的工程目录相同。

使用 `--to_step` 时，必须同时指定 `--save-temps` 以保存 Vivado 工具所需的临时文件。例如：

```
v++ --link --save-temps --to_step vpl.update_bd
```

--trace_memory

· 适用于：链接

```
--trace_memory <arg>
```

`-trace_memory` 选项仅适用于硬件构建目标 (`-t=hw`)，不应用于软件或硬件仿真流程。链接硬件目标时，该选项应与 `--profile.xxx` 选项搭配使用（如 --profile 选项 中所述），以指定用于捕获追踪数据的存储器类型和量。

`<FIFO>:<size>|<MEMORY>[<n>]` 可指定要用于进行剖析的追踪缓冲器存储器类型。

· FIFO:<size>：以 KB 为单位来指定。默认值为 FIFO:8K。最大值为 4G。
· Memory[<N>]：指定平台上的存储器资源类型和数量。可通过 `platforminfo` 命令来识别目标平台的存储器资源。受支持的存储器类型包括 HBM、DDR、PLRAM、HP、ACP、MIG 和 MC_NOC。例如，DDR[1]。

重要提示！ 在链接步骤中使用 `--trace_memory` 时，应在 `xrt.ini` 文件中同时使用 `[Debug] trace_buffer_size`，如 xrt.ini 文件 中所述。

-v | --version

```
-v
```

打印 `v++` 命令的版本和构建信息。例如：

```
v++ -v
```

--vivado

- 适用于：链接

指定属性和参数，以在构建器件二进制文件之前配置 Vivado 综合和实现环境。如需了解更多信息，请参阅 --vivado 选项。

--user_board_repo_paths

- 适用于：编译和链接

```
--user_board_repo_paths
```

为 DIMM 开发板文件指定现有用户开发板存储库。该值将追加到 Vivado 工程的 `board_part_repo_paths` 属性之前。

--user_ip_repo_paths

- 适用于：编译和链接

```
--user_ip_repo_paths <repo_dir>
```

指定一个或多个用户 IP 存储库路径的目录位置，以供首先在其中搜索内核设计中所使用的 IP。该值将附加到 Vivado 工具所用的 `ip_repo_paths` 开头位置之后，用于定位 IP 核。优先使用来自这些指定路径的 IP 定义，然后再使用来自硬件平台 (`.xsa`) 或来自赛灵思 IP 目录的 IP 存储库的 IP 定义。

提示：在 `v++` 命令行上可指定多个 `--user_ip_repo_paths`。

以下列表显示了构建进程中查找 IP 定义的优先顺序（从高到低）。请注意，所有这些条目均可包含多个目录。

- 对于系统硬件构建 (`-t hw`)：
  1. 来自 `--user_ip_repo_paths` 的 IP 定义。
  2. 内核 IP 定义（`vpl --iprepo` 开关值）。
  3. 来自与平台关联的 IP 存储库的 IP 定义。
  4. 来自安装区域的 IP 高速缓存（例如，`<Install_Dir>/Vitis/2019.2/data/cache/`）。
  5. 来自安装区域的赛灵思 IP 目录（例如，`<Install_Dir>/Vitis/2019.2/data/ip/`）
- 对于硬件仿真构建 (`-t hw_emu`)：
  1. 来自 `--user_ip_repo_paths` 的 IP 定义和用户仿真 IP 存储库。
  2. 内核 IP 定义（`vpl --iprepo` 开关值）。
  3. 来自与平台关联的 IP 存储库的 IP 定义。
  4. 来自安装区域的 IP 高速缓存（例如，`<Install_Dir>/Vitis/2019.2/data/cache/`）。
  5. `$::env(XILINX_VITIS)/data/emulation/hw_em/ip_repo`
  6. `$::env(XILINX_VIVADO)/data/emulation/hw_em/ip_repo`
  7. 来自安装区域的赛灵思 IP 目录（例如，`<Install_Dir>/Vitis/2019.2/data/ip/`）

例如：

```
v++ --user_ip_repo_paths ./myIP_repo ...
```

--advanced 选项

`--advanced.param` 和 `--advanced.prop` 选项用于指定供 `v++` 命令使用的参数和属性。执行编译或链接时，这些选项可以为 Vitis 核开发套件所生成的硬件以及硬件仿真 (emulation) 进程提供细化的控制。

`--advanced.xxx` 选项的实参按如下方式来指定：`<param_name>=<param_value>`。例如：

```
v++ --link --advanced.param compiler.enableXSAIntegrityCheck=true
--advanced.prop kernel.foo.kernel_flags="-std=c++0x"
```

提示：所有 Vitis 编译器选项均可在搭配 `--config` 选项一起使用的配置文件中指定，如 Vitis 编译器配置文件中所述。例如，通过使用如下语法，即可在不含节头的配置文件中指定 `--platform` 选项：

```
platform=xilinx_u200_xdma_201830_2
```

--advanced.param

```
--advanced.param <param_name>=<param_value>
```

按下表所述指定高级参数。

表 36：**参数选项**

参数名称	有效值	描述
`compiler.acceleratorBinaryContent`	类型：字符串 默认值：`<empty>`	表示要在 `xclbin` 中插入的内容。有效选项包括 `bitstream`、`pdi`、`dcp`、`dcp, bitstream` 和 `dcp, pdi`。 在构建硬件目标时使用该选项，它适用于： • `v++ --link` • `vpl.impl` • `xclbinutil` `bitstream` 与 `pdi` 互斥。`pdi` 适用于 Versal 平台，`bitstream` 则适用于非 Versal 平台。 **提示：**当此参数设为 `dcp`、`bitstream` 或 `pdi` 时，`v++` 会生成 2 个 `xclbin` 文件：其中一个包含 DCP 文件，另一个则包含比特流或 PDI 文件。
`compiler.addOutputTypes`	类型：字符串 默认值：`<empty>`	表示由 Vitis 编译器生成的其它输出类型。有效值包括：`xclbin` 和 `hw_export`。`hw_export` 可用于创建固定 XSA（源自动态硬件平台）以供在嵌入式软件开发流程中使用。 适用于： • `v++ --link` • `vpl.impl` • XSA 生成

表 36：**参数选项** (续)

参数名称	有效值	描述
compiler.axiDeadLockFree	类型：布尔值 默认值：TRUE	用于避免死锁。Vitis HLS 默认启用该选项。
compiler.deadlockDetection	类型：布尔值 默认值：FALSE	支持在硬件仿真 (emulation) 过程中，运行仿真 (simulation) 期间检测内核死锁。当应用发生死锁时，该工具会向控制台和日志文件发布 1 条错误消息： `// ERROR!!! DEADLOCK DETECTED at 42979000 ns! SIMULATION WILL BE STOPPED! //` 此消息将重复直至死锁终止为止。您必须手动终止应用以结束死锁状况。 **提示：**仿真期间遇到死锁时，您可在 Vitis HLS 中打开内核代码（如 使用 Vitis HLS 编译内核 中所述）以使用其它死锁检测和调试功能。 适用于： • v++ --compile • Vitis HLS • config_export
compiler.enableIncrHwEmu	类型：布尔值 默认值：FALSE	需对平台进行某些次要更改时，该选项用于支持对硬件仿真 xclbin 进行增量编译。当平台完成更新后，该选项支持对硬件仿真的器件二进制文件进行快速重构。 适用于： • v++ --link • vpl.impl
compiler.errorOnHoldViolation	类型：布尔值 默认值：TRUE	完成 Vivado 实现的最后一步之后，在时序分析检查期间，如需进行时钟缩放，则可使用该选项。如果发现保持时间违例，v++ 默认会退出并返回错误，且不生成 xclbin。此参数允许您覆盖默认行为。 适用于： • v++ --link • vpl.impl
compiler.fsanitize	类型：字符串 默认值：<empty>	支持对 OpenCL 内核执行额外的存储器访问，如 调试 OpenCL 内核 中所述。有效值包括：address 和 memory。 适用于软件仿真和调试 (Software Emulation and Debug)。
compiler.interfaceRdBurstLen	类型：整数范围 默认值：0	用于指定内核 AXI 接口上期望的 AXI 读取突发长度。该选项搭配 compiler.interfaceRdOutstanding 选项一起使用可判定硬件缓冲器大小。有效值为 1 到 256。 适用于： • v++ --compile • Vitis HLS • config_interface

表 36：**参数选项** (续)

参数名称	有效值	描述
compiler.interfaceWrBurstLen	类型：整数范围 默认值：0	用于指定内核 AXI 接口上期望的 AXI 写入突发长度。该选项搭配 compiler.interfaceWrOutstanding 选项一起使用可判定硬件缓冲器大小。有效值为 1 到 256。 适用于： • v++ --compile • Vitis HLS • config_interface
compiler.interfaceRdOutstanding	类型：整数范围 默认值：0	指定内核 AXI 接口上未完成的缓冲器读取数。有效值为 1 到 256。 适用于： • v++ --compile • Vitis HLS • config_interface
compiler.interfaceWrOutstanding	类型：整数范围 默认值：0	指定内核 AXI 接口上未完成的缓冲器写入数。有效值为 1 到 256。 适用于： • v++ --compile • Vitis HLS • config_interface
compiler.maxComputeUnits	类型：整数 默认值：-1	系统内允许的最大计算单元数。默认值为 60 个计算单元，或者在硬件平台 (.xsa) 内通过 numComputeUnits 属性来指定。 指定的值将覆盖硬件平台的默认值。默认值 -1 表示保留默认值。 适用于 v++ --link。
compiler.skipTimingCheckAndFrequencyScaling	类型：布尔值 默认值：FALSE	此参数可导致 Vivado 工具在完成实现进程的最后一步（route_design 或布线后 phys_opt_design）之后，跳过时序检查和可选时钟频率缩放。 适用于： • v++ --link • vpl.impl
compiler.userPreCreateProjectTcl	类型：字符串 默认值：<empty>	指定在 Vitis 构建进程中创建 Vivado 工程之前要运行的 Tcl 脚本。 适用于： • v++ --link • vpl.create_project
compiler.userPreSysLinkOverlayTcl	类型：字符串 默认值：<empty>	指定在 Vitis 构建进程中，打开 Vivado IP integrator 块设计后，在运行编译器生成的 dr.bd.tcl 脚本之前要运行的 Tcl 脚本。 适用于： • v++ --link • vpl.create_bd
compiler.userPostSysLinkOverlayTcl	类型：字符串 默认值：<empty>	指定运行编译器生成的 dr.bd.tcl 脚本之后要运行的 Tcl 脚本。 适用于： • v++ --link • vpl.update_bd

表 36：**参数选项** (续)

参数名称	有效值	描述
compiler.userPostDebugProfileOverlayTcl	类型：字符串 默认值：<empty>	指定在 vpl.update_bd 步骤中，在 Vivado IP integrator 块设计中调试剖析数据覆盖插入之后要运行的 Tcl 脚本。 适用于： • v++ --link • vpl.updated_bd
compiler.worstNegativeSlack	类型：浮点 默认值：0	在执行时序分析检查期间，该选项可指定设计可接受的最差负时序裕量（以纳秒 (ns) 为单位）。当负时序裕量超过指定的值时，该工具可能会尝试通过缩放时钟频率来达成时序结果。这样会指定可接受的负时序裕量值，以代替零 (0) 裕量。 适用于： • v++ --link • vpl.impl
compiler.xclDataflowFifoDepth	类型：整数 默认值：-1	指定内核数据流区域内使用的 FIFO 深度。 适用于： • v++ --compile • Vitis HLS • config_dateflow
hw_emu.aie_shim_sol_path	类型：字符串 默认值：<empty>	该选项供 Versal 平台使用，用于指定到 AI 引擎 SHIM 解决方案约束文件的路径，此文件由 aiecompiler 生成。 此文件在仿真 (simulation)、编译和细化期间使用，可提供到物理接口的逻辑映射。此映射对于第三方仿真器（如 Mentor Graphics Questa Advanced Simulator 或 Cadence Xcelium Logic Simulation）而言是必需的。
hw_emu.compiledLibs	类型：字符串 默认值：<empty>	将所提及的 clibs 用于指定的仿真器 (simulator)。 适用于硬件仿真和调试 (Hardware Emulation and Debug)。
hw_emu.debugMode	wdb 默认值：wdb	默认值为 WDB，以波形模式运行仿真 (simulation)。 该选项只能搭配 -g 或 --debug 选项一起使用。 适用于硬件仿真和调试 (Hardware Emulation and Debug)。
hw_emu.enableProtocolChecker	类型：布尔值 默认值：FALSE	支持在硬件仿真期间运行 Lightweight AXI Protocol Checker (LAPC)。该选项用于确认设计中任意 AXI 接口的精度。 适用于硬件仿真和调试 (Hardware Emulation and Debug)。
hw_emu.json_device_file_path	类型：字符串 默认值：<empty>	该选项供 Versal 平台使用，可指定到 AI 引擎 JSON 器件文件的路径，此文件位于 Vitis 软件安装区域内。 此文件在仿真 (simulation)、编译和细化期间使用，可指定 AI 引擎阵列的大小。此映射对于第三方仿真器（如 Mentor Graphics Questa Advanced Simulator 或 Cadence Xcelium Logic Simulation）而言是必需的。

表 36：**参数选项** (续)

参数名称	有效值	描述
hw_emu.platformPath	类型：字符串 默认值：<empty>	指定到定制平台目录的路径。 <platformPath> 目录应满足以下要求才能在平台创建时使用： · 此目录应包含名为 ip_repo 的子目录。 · 此目录应包含名为 scripts 的子目录，并且该 scripts 目录应包含 hw_em_util.tcl 文件。hw_em_util.tcl 文件中应定义以下 2 个过程： ○ hw_em_util::add_base_platform ○ hw_em_util::generate_simulation_scripts_and_compile 适用于硬件仿真和调试 (Hardware Emulation and Debug)。
hw_emu.reduceHwEmuCompileTime	类型：布尔值 默认值：FALSE	将顶层块设计移入 v++ --link 的“生成目标”步骤。 适用于硬件仿真和调试 (Hardware Emulation and Debug)。
hw_emu.post_sim_settings	类型：字符串	指定到 Tcl 脚本的路径，此脚本用于在运行硬件仿真之前配置 Vivado 仿真器的设置。完成默认工具配置后，启动仿真之前运行此脚本。您可使用此 Tcl 脚本来覆盖特定设置或者按需对仿真器 (simulator) 进行定制配置。 适用于硬件仿真和调试 (Hardware Emulation and Debug)。
hw_emu.scDebugLevel	none | waveform | log | waveform_and_log 默认值：waveform_and_log	设置 Vivado 逻辑仿真器 (xsim) 的 TLM 传输事务调试级别。 · 设为 NONE 表示禁用 TLM 调试 · 设为 LOG 表示将 TLM 传输事务日志转储到报告文件中 · 设为 WAVEFORM 即可启用 TLM 传输事务波形视图 · 设为 WAVEFORM_AND_LOG 即可启用“日志消息 (Log Messages)”和“波形 (Waveform)”视图 适用于硬件仿真和调试 (Hardware Emulation and Debug)。
hw_emu.simulator	XSIM | QUESTA 默认值：XSIM	使用指定仿真器 (simulator) 来运行硬件仿真 (hardware emulation)。 适用于硬件仿真和调试 (Hardware Emulation and Debug)。

例如：

```
--advanced.param compiler.addOutputTypes="hw_export"
```

提示：在配置文件中的 [advanced] 节头下可使用如下格式指定该选项：

```
[advanced]
param=compiler.addOutputTypes="hw_export"
```

--advanced.prop

```
--advanced.prop <arg>
```

为内核编译指定高级内核或解决方案属性，其中 <arg> 取下表中所述值之一。

表 37：**属性选项**

属性名称	有效值	描述
kernel.<kernel_name>.kernel_flags	类型：字符串 默认值：<empty>	在 <kernel_name> 内核上设置特定编译标记。
solution.device_repo_path	类型：字符串 默认值：<empty>	指定到硬件平台的存储库的路径。应改为使用 --platform 选项（含指向 .xpfm 平台文件的完整路径）。
solution.kernel_compiler_margin	类型：浮点 默认值：内核时钟周期的 12.5%。	内核的时钟裕度（以 ns 为单位）。在综合前，将内核时钟周期减去该值即可为布局布线延迟提供一些裕度。

--advanced.misc

```
--advanced.misc <arg>
```

指定用于内核编译的高级工具指令。

--clock 选项

重要提示！在嵌入式处理器平台上和适用于数据中心加速器卡的新型平台上均支持此处描述的 --clock 选项，如 管理时钟频率 中所述。

您可使用来自平台 shell 脚本的时钟 ID 或者通过指定内核时钟频率来指定 --clock 选项。指定时钟 ID 时，内核频率由平台上该时钟 ID 的频率来定义。指定内核频率时，平台会尝试通过按比例缩放某一可用 fixed 平台时钟来创建指定频率。在某些情况下，只能实现近似时钟频率，您可指定 --clock.tolerance 或 --clock.default_tolerance 来标示可接受的范围。如果可用的固定时钟无法在可接受容限范围内进行缩放，则会发出一条警告，并且内核会连接到默认时钟。

--clock.XXX 选项提供了一种方法，可用于在链接进程中从 v++ 命令行向内核分配时钟，并定位所需的内核时钟频率源。有多种选项可供使用，且具体适用范围各不相同。优先级顺序是根据时钟选项的专用性来判定的。此处所列规则是根据从泛用到专用的顺序罗列的，专用规则优先于泛用规则：

- 如果不指定 --clock.XXX 选项，则会对每个计算单元 (CU) 应用平台默认时钟。对于含 2 个时钟的内核，来自平台的时钟 ID 0 将分配给 ap_clk，时钟 ID 1 则分配给 ap_clk_2。
- 指定 --clock.defaultId=<id> 可为所有内核定义专用时钟 ID，从而覆盖平台默认时钟分配。
- 指定 --clock.defaultFreqHz=<Hz> 会为所有内核定义专用时钟频率，覆盖用户指定的默认时钟 ID 和平台默认时钟。
- 指定 --clock.id=<id>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]] 会向关联 CU 列表分配 1 个时钟 ID，并（可选）为 CU 分配时钟管脚。
- 指定 --clock.freqHz=<Hz>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]] 会向关联 CU 列表分配指定时钟频率，并（可选）为 CU 分配时钟管脚。

--clock.defaultFreqHz

```
--clock.defaultFreqHz <arg>
```

指定默认时钟频率（以 Hz 为单位）以供所有内核使用。这样即可覆盖默认平台时钟并分配指定时钟频率作为默认值。<arg> 则被指定为时钟频率（以 Hz 为单位）。

例如：

```
v++ --link --clock.defaultFreqHz 300000000
```

提示：在配置文件中的 [clock] 节头下可使用如下格式指定该选项：

```
[clock]
defaultFreqHz=300000000
```

--clock.defaultId

```
--clock.defaultId <arg>
```

指定 --clock.defaultId=<id> 会为所有内核定义专用时钟 ID，以覆盖平台默认时钟。<arg> 则被指定为来自目标平台上定义的某一时钟的时钟 ID，而不是默认时钟 ID。

提示：您可使用 platforminfo -v 命令为目标平台决定可用的时钟 ID 和时钟状态，如 platforminfo 实用工具 中所述。

例如：

```
v++ --link --clock.defaultId 1
```

提示：在配置文件中的 [clock] 节头下可使用如下格式指定该选项：

```
[clock]
defaultId=1
```

--clock.defaultTolerance

```
--clock.defaultTolerance <arg>
```

指定默认时钟容限，可采用值的形式或者默认时钟频率的百分比的形式。指定 clock.defaultFreqHz 时，您还可指定容限（值或百分比）。这样即可更新时序约束以反映接受的容限。

容限值 <arg> 可指定为整数（表示 clock.defaultFreqHz ± 指定容限）；或者指定为默认时钟频率（指定为十进制值）的百分比。

重要提示！不指定该选项时，默认时钟容限为 5%。

例如：

```
v++ --link --clock.defaultFreqHz 300000000 --clock.defaultTolerance 0.10
```

提示：在配置文件中的 `[clock]` 节头下可使用如下格式指定该选项：

```
[clock]
defaultTolerance=0.10
```

--clock.freqHz

```
--clock.freqHz <arg>
```

指定时钟频率（以 Hz 为单位）并将其分配到关联计算单元 (CU) 的列表以及（可选）CU 上的专用时钟管脚。`<arg>` 指定为 `<frequency_in_Hz>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`：

- `<frequency_in_Hz>`：定义指定的时钟频率（以 Hz 为单位）。
- `<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`：将定义的频率应用于指定 CU 和（可选）CU 上的指定时钟管脚。

例如：

```
v++ --link --clock.freqHz 300000000:vadd_1,vadd_3
```

提示：在配置文件中的 `[clock]` 节头下可使用如下格式指定该选项：

```
[clock]
freqHz=300000000:vadd_1,vadd_3
```

--clock.id

```
--clock.id <arg>
```

指定来自目标平台的可用时钟 ID 并将其分配到关联计算单元 (CU) 的列表以及（可选）CU 上的专用时钟管脚。`<arg>` 指定为 `<reference_ID>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`：

- `<reference_ID>`：定义要使用的时钟 ID（来自目标平台）。

  提示：您可使用 `platforminfo` 实用工具来为目标平台决定可用时钟 ID，如 platforminfo 实用工具 中所述。

- `<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`：将定义的频率应用于指定 CU 和（可选）CU 上的指定时钟管脚。

例如：

```
v++ --link --clock.id 1:vadd_1,vadd_3
```

提示：在配置文件中的 `[clock]` 节头下可使用如下格式指定该选项：

```
[clock]
id=1:vadd_1,vadd_3
```

--clock.tolerance

```
--clock.tolerance <arg>
```

指定时钟容限，可采用值的形式或者时钟频率的百分比的形式。指定 `--clock.freqHz` 时，您还可指定容限（值或百分比）。这样即可更新时序约束以反映接受的容限。<arg> 指定为 `<tolerance>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`

- `<tolerance>`：可指定为整数（表示 `clock.freqHz` ± 指定容限值）；或者指定时钟频率（指定为十进制值）的百分比。
- `<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`：将定义的时钟容限应用于指定 CU 和（可选）CU 上的指定时钟管脚。

重要提示！不指定该选项时，默认时钟容限为 `clock.FreqHz` 的 5%。

例如：

```
v++ --link --clock.tolerance 0.10:vadd_1,vadd_3
```

提示：在配置文件中的 `[clock]` 节头下可使用如下格式指定该选项：

```
[clock]
tolerance=0.10:vadd_1,vadd_3
```

--connectivity 选项

如 链接内核 中所述，有多个 `--connectivity.XXX` 选项可支持您定义 FPGA 二进制文件的拓扑结构、指定 CU 数量、将其分配给 SLR、将内核端口连接到全局存储器并建立数据流传输端口连接。这些命令是构建进程中不可或缺的一部分，对于应用的定义和构造都至关重要。

--connectivity.nk

```
--connectivity.nk <arg>
```

其中 `<arg>` 按如下方式指定：`<kernel_name>:#:<cu_name1>.<cu_name2>...<cu_name#>`。

它可在链接进程中生成的 FPGA 二进制文件 (`.xclbin`) 内，为指定内核 (`kernel_name`) 例化指定数量的 CU (#)。`cu_name` 为可选。如不指定 `cu_name`，则直接对内核实例进行编号：`kernel_name_1`、`kernel_name_2` 以此类推。默认情况下，Vitis 编译器会为每个内核例化 1 个计算单元。

例如：

```
v++ --link --connectivity.nk vadd:3:vadd_A.vadd_B.vadd_C
```

提示：在配置文件中的 `[connectivity]` 节头下可使用如下格式指定该选项：

```
[connectivity]
nk=vadd:3:vadd_A.vadd_B.vadd_C
```

--connectivity.sc

```
--connectivity.sc <arg>
```

在 2 个计算单元之间通过其 AXI4-Stream 接口创建数据流传输连接。针对每个数据流传输接口连接单独使用 1 个 `--connectivity.sc` 选项。连接顺序必须为从首个内核的数据流传输输出端口到第二个内核的数据流传输输入端口。有效值包括：

```
<cu_name>.<streaming_output_port>:<cu_name>.<streaming_input_port>[:<fifo_depth>]
```

其中：

- `<cu_name>` 是 `--connectivity.nk` 选项中指定的计算单元名称。通常，该值为 `<kernel_name>_1`，除非指定其它名称。
- `<streaming_output_port>/<streaming_input_port>` 是声明为 AXI4-Stream 的计算单元端口的函数实参。
- `[:<fifo_depth>]` 会在 2 个数据流传输端口之间插入指定深度的 FIFO 以防止发生停滞。请指定整数值。

例如，要将计算单元 `mem_read_1` 的 AXI4-Stream 端口 `s_out` 连接到计算单元 `increment_1` 的 AXI4-Stream 端口 `s_in`，请使用：

```
--connectivity.sc mem_read_1.s_out:increment_1.s_in
```

提示：在配置文件中的 `[connectivity]` 节头下可使用如下格式指定该选项：

```
[connectivity]
sc=mem_read_1.s_out:increment_1.s_in
```

包含可选 <fifo_depth> 值即可支持 `v++` 连接器在 2 个内核之间添加 FIFO 以帮助防止停滞。它使用来自器件的 BRAM 资源（如果已指定），但无需再更新 HLS 内核以包含 FIFO。如果连接具有不同时钟或不同总线宽度，该工具还会例化 Clock Converter (CDC) 或 Datawidth Converter (DWC) IP。

--connectivity.slr

```
--connectivity.slr <arg>
```

该选项可用于将 CU 分配给器件上的特定 SLR。针对分配给 SLR 的每个内核或 CU 都必须重复指定该选项。

重要提示！如果使用 `--connectivity.slr` 来分配内核布局，那么还必须使用 `--connectivity.sp` 来为内核分配存储器访问。

有效值包括：

```
<cu_name>:<SLR_NUM>
```

其中：

- `<cu_name>` 是 `--connectivity.nk` 选项中指定的计算单元的名称。通常，该值为 `<kernel_name>_1`，除非指定其它名称。
- `<SLR_NUM>` 是 CU 分配到的 SLR 编号。例如，SLR0 和 SLR1。

例如，要将 CU `vadd_2` 分配给 SLR2，并将 CU `fft_1` 分配给 SLR1，请使用：

```
v++ --link --connectivity.slr vadd_2:SLR2 --connectivity.slr fft_1:SLR1
```

提示：在配置文件中的 `[connectivity]` 节头下可使用如下格式指定该选项：

```
[connectivity]
slr=vadd_2:SLR2
slr=fft_1:SLR1
```

--connectivity.sp

```
--connectivity.sp <arg>
```

该选项可用于指定在平台内向系统端口分配内核实参的方式。该选项的主要用例是将内核实参连接到特定存储器资源。要将内核的每个实参映射到特定存储器资源，都需要单独使用 1 个 `--connectivity.sp` 选项。在构建进程中，未通过 `--connectivity.sp` 选项显式映射到存储器资源的任一实参都会自动连接到可用的存储器资源。

建议：赛灵思建议使用 `--connectivity.sp` 选项时指定实参名称，因为这样可以提供最大的连接灵活性。但您也可以通过该选项来指定内核接口端口。

有效值包括：

```
<cu_name>.<kernel_argument_name>:<sptag[min:max]>
```

其中：

- `<cu_name>` 是 `--connectivity.nk` 选项中指定的计算单元的名称。通常，该值为 `<kernel_name>_1`，除非指定其它名称。
- `<kernel_argument_name>` 是内核的函数实参的名称，或计算单元接口端口的名称。
- `<sptag>` 表示系统端口标签，例如，来自目标平台的存储器控制器接口名称。有效的 `<sptag>` 名称包括 DDR、PLRAM 和 HBM。
- `[min:max]` 支持使用存储器范围，例如，DDR[0:2]。也支持单索引：DDR[2]。

提示：目标平台支持的 `<sptag>` 和存储器资源范围可使用 `platforminfo` 命令来获取。如需了解更多信息，请参阅 platforminfo 实用工具。

以下示例演示的是将 VADD 内核的指定 CU 的输入实参 (A) 映射到 DDR[0:3]、将输入实参 (B) 映射到 HBM[0:31]，并将输出实参 (C) 写入 PLRAM[2]：

```
v++ --link --connectivity.sp vadd_1.A:DDR[0:3] --connectivity.sp
vadd_1.B:HBM[0:31] \
--connectivity.sp vadd_1.C:PLRAM[2]
```

提示：在配置文件中的 `[connectivity]` 节头下可使用如下格式指定该选项：

```
[connectivity]
sp=vadd_1.A:DDR[0:3]
sp=vadd_1.B:HBM[0:31]
sp=vadd_1.C:PLRAM[2]
```

--connectivity.connect

```
--connectivity.connect <X:Y>
```

该选项可用于通过 Vivado IP integrator 建立连接，但 v++ 不会对指定的连接执行任何错误检查。该选项可用于指定内核与目标平台的非 AXI 元素之间的常规连接，例如，与 GT 端口的连接。

X 和 Y 连接必须指定为与 IP integrator connect_bd_net 或 connect_bd_intf_net 命令兼容的实参。<X:Y> 的具体格式为：

```
src/hierarchy_name/cell_name/pin_name:dst/hierarchy_name/cell_name/pin_name
```

这些连接不包括 AXI4-Stream 接口之间的连接（需使用 --conectivity.sc）或 M_AXI 接口之间的连接（需使用 --connectivity.sp），如上所述。

提示：在配置文件中的 [connectivity] 节头下可使用如下格式指定该选项：

```
[connectivity]
connect=<X:Y>
```

--debug 选项

该选项支持在器件二进制文件 (.xclbin) 中插入调试 IP 核用于进行硬件调试。该选项允许您指定要添加的调试核类型以及要通过 ChipScope™ 监控的计算单元和接口。--debug.xxx 选项允许您将接口上的 AXI Protocol Checker 和 System ILA 核连接到内核或特定计算单元 (CU) 用于进行调试和性能监控：

- System Integrated Logic Analyzer (ILA) 可提供对硬件上运行的加速内核或函数的传输事务级可视性。还可使用 System ILA 核来捕获和查看感兴趣的 AXI 流量。
- AXI Protocol Checker 调试核旨在监控加速内核上的 AXI 接口。将其连接到 CU 的接口时，它会主动检查协议违例并指示发生了哪些违例。

在配置文件中的 [debug] 节头下可使用如下格式示例来指定 --debug.xxx 命令：

```
[debug]
protocol=all:all            # Protocol analyzers on all CUs
protocol=cu2:port3          # Protocol analyzer on port3 of cu2
chipscope=cu2               # ILA on cu2
```

--debug 的各选项包括：

--debug.chipscope

```
--debug.chipscope <cu_name>[:<interface_name>]
```

将 System Integrated Logic Analyzer 调试核添加到设计中的指定 CU。

重要提示！ --debug.chipscope 选项要求指定 <cu_name>，并且不接受关键字 all。（可选）您可以指定 <interface_name>。

例如，以下命令可将 ILA 核添加到 vadd_1 CU 中：

```
v++ --link --debug.chipscope vadd_1
```

--debug.list_ports

显示当前设计中的有效计算单元和端口组合列表。该选项供参考，可帮助您为 `--debug` 命令制作命令行或配置文件。

该选项需在链接期间指定，但不运行链接进程。命令行的必需选项如以下示例所示，此示例在将指定内核与列出的平台相链接时会返回可用端口。

```
v++ --platform <platform> --link --debug.list_ports <kernel.xo>
```

--debug.protocol

```
--debug.protocol all|<cu_name>[:<interface_name>]
```

将 AXI Protocol Checker 调试核添加到设计中。对于该选项，可为其指定关键字 `all` 或 `<cu_name>` 以及可选的 `<interface_name>`，以便向指定 CU 和接口添加协议检查器。

例如：

```
v++ --link --debug.protocol all
```

--hls 选项

以下所述 `--hls.XXX` 选项用于为内核编译期间调用的 Vitis HLS 综合进程指定相应的选项。

--hls.clock

```
--hls.clock <arg>
```

指定频率 (Hz)，Vitis HLS 应按此频率编译所列示的一个或多个内核。

其中，`<arg>` 指定方式为：`<frequency_in_Hz>:<cu_name1>,<cu_name2>,...,<cu_nameN>`

- `<frequency_in_Hz>`：定义内核频率（以 Hz 为单位来指定）。
- `<cu_name1>,<cu_name2>,...`：定义要按指定目标频率编译的内核或内核实例 (CU) 的列表。

例如：

```
v++ -c --hls.clock 300000000:mmult,mmadd --hls.clock 100000000:fifo_1
```

提示：在配置文件中的 `[hls]` 节头下可使用如下格式指定该选项：

```
[hls]
clock=300000000:mmult,mmadd
clock=100000000:fifo_1
```

--hls.export_mode

```
--hls.export_mode <file_type>:<file_path>
```

指定 Vitis HLS 的 RTL 导出模式，以及导出的文件的路径和名称。作为 `v++` 编译选项，唯一受支持的 `<file_type>` 为 XO。

例如：

```
v++ --hls.export_mode xo:./kernel.xo
```

提示：在配置文件中的 `[hls]` 节头下可使用如下格式指定该选项：

```
[hls]
export_mode=xo:./kernel.xo
```

--hls.export_project

```
--hls.export_project <arg>
```

指定 Vitis HLS 工程设置脚本导出到的目录。

例如：

```
v++ --hls.export_project ./hls_export
```

提示：在配置文件中的 `[hls]` 节头下可使用如下格式指定该选项：

```
[hls]
export_project=./hls_export
```

--hls.jobs

```
--hls.jobs <arg>
```

指定用于启动 HLS 运行的作业数。

该选项用于指定并行作业数量，Vitis HLS 使用这些作业来对 RTL 内核代码进行综合。增加作业数量即可允许工具生成更多并行进程并更快完成作业。

例如：

```
v++ --hls.jobs 4
```

提示：在配置文件中的 `[hls]` 节头下可使用如下格式指定该选项：

```
[hls]
jobs=4
```

--hls.lsf

```
--hls.lsf <arg>
```

指定 `bsub` 用于向 LSF 提交作业以供 HLS 运行。

指定 bsub 命令行作为传递给 LSF 集群的字符串。该选项是使用 IBM Platform Load Sharing Facility (LSF) 进行 Vitis HLS 综合所必需的选项。

例如：

```
v++ --compile --hls.lsf '{bsub -R \"select[type=X86_64]\" -N -q medium}'
```

提示：在配置文件中的 [hls] 节头下可使用如下格式指定该选项：

```
[hls]
lsf='{bsub...
```

--hls.post_tcl

```
--hls.post_tcl <arg>
```

指定包含 Tcl 命令的 Tcl 文件，供 vitis_hls 在 csynth_design 之后用作为源文件。

例如：

```
v++ --hls.post_tcl ./runPost.tcl
```

提示：在配置文件中的 [hls] 节头下可使用如下格式指定该选项：

```
[hls]
post_tcl=./runPost.tcl
```

--hls.pre_tcl

```
--hls.pre_tcl <arg>
```

指定包含 Tcl 命令的 Tcl 文件，供 vitis_hls 在 csynth_design 之前用作为源文件。

例如：

```
v++ --hls.pre_tcl ./runPre.tcl
```

其中，runPre.tcl 包含以下命令，用于在 Vitis HLS 中配置 m_axi 接口：

```
config_interface -m_axi_auto_max_ports=1
config_interface -m_axi_max_bitwidth 512
```

提示：在配置文件中的 [hls] 节头下也可使用如下格式来指定该选项：

```
[hls]
pre_tcl=./runPre.tcl
```

--linkhook 选项

以下描述的 `--linkhook.XXX` 选项用于指定 Tcl 脚本，这些脚本在执行 Vitis 链接进程中的特定步骤时运行。通过使用 `--linkhook.list_steps` 命令即可判定有效的步骤，如下所述。

--linkhook.custom

```
--linkhook.custom <step name, path to script file>
```

指定要在构建进程中的预定义时间点执行的 Tcl 脚本。用于指定脚本的路径可以是绝对路径或者与构建目录相关的部分路径。

例如，以下命令会在构建中的 SysLink 步骤前运行指定的 Tcl 脚本：

```
v++ -l --linkhook.custom preSysLink,./runScript.tcl
```

-linkhook.do_first

```
--linkhook.do_first <step name, path to script file>
```

指定 Tcl 脚本在给定步骤名称之前执行。用于指定脚本的路径可以是绝对路径或者与构建目录相关的部分路径。

例如，以下命令会在构建中的 `place_design` 步骤之前运行指定的 Tcl 脚本：

```
v++ -l --linkhook.do_first vpl.impl.place_design,runScript.tcl
```

-linkhook.do_last

```
--linkhook.do_last <step name, path to script file>
```

指定 Tcl 脚本在给定步骤完成后立即执行。用于指定脚本的路径可以是绝对路径或者与构建目录相关的部分路径。

例如，以下命令会在构建中的 `place_design` 步骤之后运行指定的 Tcl 脚本：

```
v++ -l --linkhook.do_last vpl.impl.place_design,runScript.tcl
```

-linkhook.list_steps

```
--linkhook.list_steps
```

列出针对指定目标支持脚本挂钩的默认和可选构建步骤。此命令需同时指定 `--target` 和 `--link` 选项。

例如：

```
v++ --target hw -l --linkhook.list_steps
```

此命令会返回构建进程中始终启用的默认步骤以及您可按需启用的可选步骤。请参阅 管理 Vivado 综合与实现结果 以获取有关启用可选步骤的指示信息。

--package 选项

简介

`v++ --package` 或 `-p` 步骤用于在 `v++` 编译和链接构建进程结束时封装最终产品。这是所有嵌入式平台（包括 Versal 器件、AI 引擎和 Zynq 器件）的必要步骤。

`--package` 的各选项包括：

--package.aie_debug_port

```
--package.aie_debug_port <arg>
```

其中 `<arg>` 用于指定 TCP 端口，此端口中的仿真器 (emulator) 会监听从调试器到 Versal AI 引擎调试核的传入连接。

例如：

```
v++ -l --package.aie_debug_port 1440
```

--package.bl31_elf

```
--package.bl31_elf <arg>
```

其中 `<arg>` 用于指定到 Arm 可信 FW ELF（在 A72 #0 核上执行）的绝对路径或相对路径。

例如：

```
v++ -l --package.bl31_elf ./arm_trusted.elf
```

--package.boot_mode

```
--package.boot_mode <arg>
```

其中 `<arg>` 用于指定 `<ospi | qspi | sd>` 启动模式，此模式用于在仿真 (emulation) 中或在硬件上运行应用。

提示： `ospi` 仅适用于 Versal 数据中心平台。

例如：

```
v++ -l --package.boot_mode sd
```

--package.defer_aie_run

```
--package.defer_aie_run
```

该选项可指定嵌入式处理器应用 (PS) 启用 Versal AI 引擎核。如不指定该选项，则此工具会生成 CDO 命令以改为在 PDI 加载中启用 AI 引擎核。

例如：

```
v++ -l --package.defer_aie_run
```

--package.domain

```
--package.domain <arg>
```

其中 `<arg>` 用于指定域名。

例如：

```
v++ -l --package.domain xrt
```

--package.dtb

```
--package.dtb <arg>
```

其中 `<arg>` 用于指定到设备树二进制文件 (DTB) 的绝对路径或相对路径，此二进制文件用于在 APU 上加载 Linux。

例如：

```
v++ -l --package.dtb ./device_tree.image
```

--package.enable_aie_debug

```
--package.enable_aie_debug
```

如果启用该选项，那么此工具会在 PDI 加载期间生成 CDO 命令以停止 AI 引擎核。

例如：

```
v++ -l --package.enable_aie_debug
```

--package.image_format

```
--package.image_format <arg>
```

其中 `<arg>` 用于指定 SD 卡上使用的 `<ext4 | fat32>` 输出镜像文件格式。

- ext4：Linux 文件系统
- fat32：Windows 文件系统

重要提示！ EXT4 格式在 Windows 上不受支持。

例如：

```
v++ -l --package.image_format fat32
```

--package.kernel_image

```
--package.kernel_image <arg>
```

其中 <arg> 用于指定到 Linux 内核镜像文件的绝对路径或相对路径。覆盖平台中可用的现有镜像。此平台镜像文件可从 xilinx.com 下载。如需了解更多信息，请参阅 Vitis 软件平台安装。

例如：

```
v++ -l --package.kernel_image ./kernel_image
```

--package.no_image

```
--package.no_image
```

绕过 SD 卡镜像创建操作。针对 --package.boot_mode sd 有效。

--package.out_dir

```
--package.out_dir <arg>
```

其中 <arg> 用于指定到 --package 命令的输出目录的绝对路径或相对路径。

例如：

```
v++ -l --package.out_dir ./out_dir
```

--package.ps_debug_port

```
--package.ps_debug_port <arg>
```

其中 <arg> 用于指定 TCP 端口，此端口中的仿真器 (emulator) 会监听从调试器到 PS 调试核的传入连接。

例如：

```
v++ -l --package.debug_port 3200
```

--package.ps_elf

```
--package.ps_elf <arg>
```

其中 <arg> 用于指定 <path_to_elf_file,core>。

- path_to_elf_file：指定 PS 核的 ELF 文件。
- core：指示应在其中运行此文件的 PS 核。

当裸机 ELF 文件在器件处理器核上运行时，使用该选项。该选项用于指定要包含在启动镜像中的 ELF 文件和处理器核对。以下列出了受支持的器件的可用处理器：

- Versal 处理器核值包括：a72-0、a72-1、a72-2 和 a72-3。
- Zynq UltraScale+ MPSoC 处理器核值包括：a53-0、a53-1、a53-2、a53-3、r5-0 和 r5-1。
- Zynq-7000 处理器核值包括：a9-0 和 a9-1。

提示：为每个 ELF/核对单独指定选项。

例如：

```
v++ -l --package.ps_elf a53_0.elf,a53-0 --package.ps_elf r5_0.elf,r5-0
```

--package.rootfs

```
--package.rootfs <arg>
```

其中 `<arg>` 用于指定到已处理的 Linux 根文件系统文件的绝对路径或相对路径。此平台 RootFS 文件可从 Xilinx.com 下载。如需了解更多信息，请参阅 Vitis 软件平台安装。

例如：

```
v++ -l --package.rootfs ./rootfs.ext4
```

--package.sd_dir

```
--package.sd_dir <arg>
```

其中 `<arg>` 用于指定要封装到 `sd_card` 目录/镜像中的文件夹。目录内容将被复制到 `sd_card` 文件夹的子文件夹中。

例如：

```
v++ -l --package.sd_dir ./test_data
```

--package.sd_file

```
--package.sd_file <arg>
```

其中 `<arg>` 用于指定要封装到 `sd_card` 目录/镜像中的 ELF 或其它数据文件。可重复使用该选项以指定将多个文件添加到 `sd_card` 中。

例如：

```
v++ -l --package.sd_file ./arm_trusted.elf
```

--package.uboot

```
--package.uboot <arg>
```

其中 `<arg>` 用于指定到 U-boot ELF 文件的路径，此文件用于覆盖平台 U-boot。

例如：

```
v++ -l --package.uboot ./uboot.elf
```

--profile 选项

如 在应用中启用剖析 中所述，有多个 `--profile` 选项支持您在运行时执行期间启用应用与内核事件剖析。该选项支持捕获内核与主机之间、内核停滞时、内核与计算单元 (CU) 执行时间以及 Versal AI 引擎中的监控活动的数据流量的剖析数据。

重要提示！ 在 `v++` 中使用 `--profile` 选项还需要将 `profile=true` 语句添加到 `xrt.ini` 文件中。请参阅 xrt.ini 文件。

在配置文件中的 `[profile]` 节头下可使用如下格式来指定 `--profile` 命令：

```
[profile]
data=all:all:all            # Monitor data on all kernels and CUs
data=k1:all:all             # Monitor data on all instances of kernel k1
data=k1:cu2:port3           # Specific CU master
data=k1:cu2:port3:counters  # Specific CU master (counters only, no trace)
stall=all:all               # Monitor stalls for all CUs of all kernels
stall=k1:cu2                # Stalls only for cu2
exec=all:all                # Monitor execution times for all CUs
exec=k1:cu2                 # Execution tims only for cu2
aie=all                     # Monitor all AIE streams
aie=DataIn1                 # Monitor the specific input stream in the SDF
graph
aie=M02_AXIS                # Monitor specific stream interface
```

此命令的各选项如下所述：

--profile.aie <arg>

支持在自适应数据流 (ADF) 应用中对 AI 引擎数据流传输进行剖析，其中 `<arg>` 为：

```
<ADF_graph_argument|pin name|all>
```

- `<ADF_graph_argument>`：指定来自 ADF Graph 应用的实参名称。
- `<pin_name>`：表示 AI 引擎内核上的端口。
- `<all>`：指示监控 ADF 应用中的所有数据流传输连接。

例如，要监控 `DataIn1` 输入数据流传输，请使用以下命令：

```
v++ --link --profile.aie:DataIn1
```

--profile.data <arg>

通过监控 API 启用数据端口监控。此选项只需在链接期间指定即可。

其中 `<arg>` 为：

```
[<kernel_name>|all]:[<cu_name>|all]:[<interface_name>|all](:[counters|all])
```

- `[<kernel_name>|all]` 用于定义此命令要应用到的特定内核。但是，您也可以通过单一选项，指定关键字 `all` 来将该监控应用于所有现存内核、计算单元和接口。

- [<cu_name>|all]：已指定 <kernel_name> 的情况下，您还可指定将此命令应用于特定 CU 或者指明应将此命令应用于内核的所有 CU。
- [<interface_name>|all] 用于定义内核或 CU 上的特定接口以监控其中的数据活动，或者监控所有接口。
- [<counters|all] 为可选实参，如不指定，则默认为 all。它允许您将较大型设计的信息收集操作范围局限于仅限 counters，如设为 all 则将包含收集实际追踪信息。

例如，要将数据剖析分配到内核 k1 的所有 CU 和接口，请使用以下命令：

```
v++ --link --profile.data:k1:all:all
```

--profile.exec <arg>

该选项会记录内核的执行时间，并在系统运行期间提供最小范围的端口数据收集。此选项只需在链接期间指定即可。

提示：默认情况下，指定 --profile.data 或 --profile.stall 时，会收集内核的执行时间。您可为 data 或 stall 未涵盖的任意 CU 定 --profile.exec。

exec 剖析的语法是：

```
[<kernel_name>|all]:[<cu_name>|all](:[counters|all])
```

例如，要对内核 k1 的 cu2 的执行进行剖析，请使用以下语法：

```
v++ --link --profile.exec:k1:cu2
```

--profile.stall

将停滞监控日志添加到器件二进制文件 (.xclbin) 中，这需要在内核接口上添加停滞端口。为便于执行此操作，必须在编译和链接期间指定 stall 选项。

stall 剖析的语法是：

```
[<kernel_name>|all]:[<cu_name>|all](:[counters|all])
```

例如，要监控内核 k1 的 cu2 的停滞，请使用以下命令：

```
v++ --compile -k k1 --profile.stall ...
v++ --link --profile.stall:k1:cu2 ...
```

--profile.trace_memory

构建硬件目标 (-t=hw) 时，请使用该选项来指定用于捕获追踪数据的存储器类型和量。您可按如下方式来指定实参：

```
<FIFO>:<size>|<MEMORY>[<n>]
```

该实参可指定要用于进行剖析的追踪缓冲器存储器类型。

- FIFO:<size>：以 KB 为单位来指定。默认值为 FIFO:8K。最大值为 4G。
- Memory[<N>]：指定平台上的存储器资源类型和数量。可通过 platforminfo 命令来识别目标平台的存储器资源。受支持的存储器类型包括 HBM、DDR、PLRAM、HP、ACP、MIG 和 MC_NOC。例如，DDR[1]。

重要提示！ 在 `xrt.ini` 文件中搭配 `[Debug] trace_buffer_size` 一起使用，如 xrt.ini 文件 中所述。

--vivado 选项

`--vivado.XXX` 选项用于配置 Vivado 工具，以对器件二进制文件 (`.xclbin`) 执行综合与实现。例如，您可以指定要生成的作业数量、用于实现运行的 LSF 命令或者要使用的特定实现策略。您还可配置最优化、布局、时序或者指定要输出的报告。

重要提示！ 为了最有效地利用这些选项，您需要熟悉 Vivado Design Suite。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：实现》(UG904)。

--vivado.impl.jobs

```
--vivado.impl.jobs <arg>
```

指定 Vivado Design Suite 用于实现器件二进制文件的并行作业数量。增加作业数量即可允许 Vivado 实现步骤生成更多并行进程并更快完成作业。

例如：

```
v++ --link --vivado.impl.jobs 4
```

--vivado.impl.lsf

```
--vivado.impl.lsf <arg>
```

指定 `bsub` 命令行作为传递给 LSF 集群的字符串。该选项是使用 IBM Platform Load Sharing Facility (LSF) 进行 Vivado 实现所必需的选项。

例如：

```
v++ --link --vivado.impl.lsf '{bsub -R \"select[type=X86_64]\" -N -q
medium}'
```

--vivado.impl.strategies

```
--vivado.impl.strategies <arg>
```

指定 Vivado 实现运行的策略名称列表（以逗号分隔）。`ALL` 可用于运行所有可用的实现策略。这样您即可在构建进程中同时运行各种实现策略，使您能够更快速地解决设计的时序和布线问题。

--vivado.param

```
--vivado.param <arg>
```

指定要在 FPGA 二进制文件 (`xclbin`) 的综合与实现期间使用的 Vivado Design Suite 参数。

--vivado.prop

```
--vivado.prop <arg>
```

指定要在 FPGA 二进制文件 (xclbin) 的综合与实现期间使用的 Vivado Design Suite 属性。

表 38：**属性选项**

属性名称	有效值	描述
vivado.prop <object_type>.<object_name>.<prop_name>	类型：各种类型	支持您指定 Vivado 硬件编译流程中使用的任何属性。 <object_type> 为 run|fileset|file|project。 如需了解有关 <object_name> 和 <prop_name> 值的信息，请参阅《Vivado Design Suite 属性参考指南》(UG912)。 示例： vivado.prop run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-no_bufg_opt} vivado.prop fileset.current.top=foo 如果 <object_type> 设置为 file，则不支持 current。 如果 <object type> 设置为 run，则可使用特殊值 __KERNEL__ 来指定所有内核的运行最优化设置，而无需逐一指定其设置。

例如，在命令行中输入：

```
v++ --link --vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
--vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
--vivado.prop run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/.../xxx.tcl
```

以上示例支持在 Vivado 实现过程中使用可选 PHYS_OPT_DESIGN 步骤、为该步骤设置 Explore 指令，并指定要在 PLACE_DESIGN 步骤之前运行的 Tcl 脚本。

提示：如 管理 Vivado 综合与实现结果 中所述，Vivado 综合与实现进程中的每个步骤都可包含 1 个 Tcl PRE 脚本（于该步骤之前运行）和 1 个 Tcl POST 脚本（于该步骤之后运行）。这样您即可通过围绕不同步骤插入预处理和后处理 Tcl 命令来自定义构建进程。您可按上述示例所示来指定这些脚本。

在配置文件中的 [vivado] 节头下也可使用如下格式指定这些选项：

```
[vivado]
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
prop=run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/.../xxx.tcl
```

重要提示！部分 Vivado 属性的名称中包含空格（如 MORE OPTIONS），Tcl 语法要求使用括号 {} 将这些属性括起。但在 `--vivado` 选项中，括号的放置方式至关重要。您必须使用括号将整个属性名称括起，而不只是将名称中某一部分括起。例如，正确的放置方式如下：

```
--vivado_prop run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-
no_bufg_opt}
```

而以下示例会在构建进程中导致错误：

```
--vivado_prop run.impl_1.STEPS.PLACE_DESIGN.ARGS.{MORE OPTIONS}={-
no_bufg_opt}
```

--vivado.synth.jobs

```
--vivado.synth.jobs <arg>
```

指定 Vivado Design Suite 用于综合器件二进制文件的并行作业数量。增加作业数量即可允许 Vivado 综合生成更多并行进程并更快完成作业。

例如：

```
v++ --link --vivado.synth.jobs 4
```

--vivado.synth.lsf

```
--vivado.synth.lsf <arg>
```

指定 `bsub` 命令行作为传递给 LSF 集群的字符串。该选项是使用 IBM Platform Load Sharing Facility (LSF) 进行 Vivado 综合所必需的选项。

例如：

```
v++ --link --vivado.synth.lsf '{bsub -R \"select[type=X86_64]\" -N -q
medium}'
```

Vitis 编译器配置文件

配置文件也可用于指定 Vitis 编译器选项。配置文件可以提供一种有条理的方式，通过将相似开关组合在一起来向编译器传递选项，并最大程度缩短 `v++` 命令行的长度。可通过配置文件条目来控制的部分功能特性包括：

- 用于配置内核编译的 HLS 选项
- 适用于系统链接的连接指令，例如，要例化的内核数或者将内核端口分配到全局存储器的方式
- 供 Vivado Design Suite 用于管理硬件综合与实现的指令。

总之，在配置文件内可指定任何 `v++` 命令选项。但是，配置文件支持对包含各组相关命令的各段进行定义，以帮助管理构建选项和策略。下表中列出了定义的各段。

表 39：**配置文件的段标签**

段名	编译器/连接器	描述
[advanced]	任一	--advanced 选项： · misc · param · prop
[clock]	编译器	--clock 选项： · defaultFreqHz · defaultID · defaultTolerance · freqHz · id · tolerance
[connectivity]	连接器	--connectivity 选项： · nk · sc · slr · sp · connect
[debug]	连接器	--debug 选项 · chipscope · list_ports · protocol
[hls]	编译器	HLS 指令 --hls 选项： · clock · export_mode · export_project · jobs · lsf · post_tcl · pre_tcl
[linkhook]	连接器	--linkhook 选项 · custom · do_first · do_last · list_steps

表 39：**配置文件的段标签** (续)

段名	编译器/连接器	描述
[package]	封装器	--package 选项 · aie_debug_port · bl31_elf · boot_mode · defer_aie_run · domain · dtb · enable_aie_debug · image_format · kernel_image · no_image · out_dir · ps_debug_port · ps_elf · rootfs · sd_dir · sd_file · uboot
[profile]	连接器	--profile 选项 · aie · data · exec · stall · trace_memory
[vivado]	连接器	--vivado 选项: · impl.jobs · impl.lsf · impl.strategies · param · prop · synth.jobs · synth.lsf

提示：可通过在行开头处使用“#”来为配置文件添加注释。段末位置是以该段结束位置的空行来指定的。

由于 `v++` 命令在每条 `v++` 命令行上支持多个配置文件，因此您可将自己的配置文件拆分为相关选项，用于定义编译和链接策略或者 Vivado 实现策略，并在构建进程中应用多个配置文件。

配置文件为可选。对于这些文件并没有命名限制，且配置文件的数量可为 0 或多个文件。如果愿意，可将所有 `v++` 选项置于单个配置文件内。但将相关开关组合到独立的文件内可帮助您组织自己的构建策略。例如，将 `[connectivity]` 相关开关组合到一个文件内，将 `[Vivado]` 选项组合到另一个文件内。

配置文件是通过使用 `v++ --config` 选项来指定的，如 Vitis 编译器常规选项 中所述。以下是 `--config` 选项示例：

```
v++ --config ../src/connectivity.cfg
```

开关按遇到的顺序来读取。如果重复遇到相同开关，并且其中所含信息有冲突，则使用第一个读取的开关。开关优先顺序如下所示，第一项优先级最高。

1. 命令行开关。
2. 命令行上的配置文件，从左到右。
3. 在配置文件内，优先顺序为从上到下。

使用消息规则文件

v++ 命令可在内核编译和链接期间执行各种赛灵思工具。这些工具会生成大量消息以便为您提供构建状态。这些消息可能与您关注的对象和设计阶段有关，也可能无关。消息规则文件 (.mrf) 可用于更好地管理这些消息。它可提供各种命令以便将重要消息推送到终端或者禁止推送不重要的消息。这有助于您更好地了解内核构建结果，并探索各种内核最优化方法。

消息规则文件是由注释和受支持的命令组成的文本文件。每行仅允许一条命令。

注解

以“#”开头作为第一个非空格字符的任何行均为注释。

受支持的命令

默认情况下，v++ 会以递归方式扫描整个工作目录，并将所有错误消息推送到 v++ 输出。以下 promote 和 suppress 命令可用于进一步控制 v++ 输出。

- promote：此命令表示应将匹配的消息推送到 v++ 输出。
- suppress：此命令表示应禁止将匹配的消息推送到 v++ 输出或者从输出中过滤掉此类消息。请注意，无法禁止推送错误消息。

每行仅限输入一条命令。

命令选项

消息规则文件可包含多条 promote 和 suppress 命令。每条命令都包含且仅含下列其中一个选项。这些选项区分大小写。

- -id [<message_id>]：推送或者禁止推送匹配指定消息 ID 的所有消息。消息 ID 格式为 nnn-mmm。以下是来自 HLS 的警告消息示例。在此例中，消息 ID 为 204-68。

  ```
  WARNING: [V++ 204-68] Unable to enforce a carried dependence constraint
  (II = 1, distance = 1, offset = 1)
  between bus request on port 'gmem'
  (/matrix_multiply_cl_kernel/mmult1.cl:57) and bus request on port 'gmem'-
  severity [severity_level]
  ```

  例如，要禁止含消息 ID 204-68 的消息，请指定：suppress -id 204-68。

- -severity [<severity_level>]：以下为严重性级别的有效值。匹配指定严重性级别的所有消息都将被推送或禁止。

  - info

- warning
- critical_warning

  例如，要推送严重性 (severity) 为“critical-warning”的消息，请指定：promote -severity critical_warning。

消息规则的优先顺序

suppress 规则优先于 promote 规则。如果将相同消息 ID 或严重性级别同时传递到消息规则文件中的 promote 和 suppress 命令，则匹配的消息将被禁止推送，不予显示。

消息规则文件示例

以下是有效的消息规则文件示例：

```
# promote all warning, critical warning
promote -severity warning
promote -severity critical_warning
# suppress the critical warning message with id 19-2342
suppress -id 19-2342
```

emconfigutil 实用工具

在命令行流程中运行软件或硬件仿真时，需要创建仿真配置文件 `emconfig.json` 供运行时库在仿真期间使用。此仿真配置文件可定义要为指定平台进行仿真的器件的类型和数量。针对软件仿真和硬件仿真可使用同一个 `emconfig.json` 文件。

注释：在真实硬件上运行时，运行时和驱动程序会查询已安装的硬件以判定安装的器件类型和数量以及器件特性。

要使用 `emconfigutil` 实用工具来自动创建仿真文件，请在 `emconfigutil` 命令行中指定目标平台和其它选项：

```
emconfigutil --platform <platform_name> [options]
```

至少必须通过 `-f` 或 `--platform` 选项来指定目标平台，以生成所需的 `emconfig.json` 文件。此处指定的平台必须与主机和硬件构建期间指定的平台相同。

下表中提供了 `emconfigutil` 选项。

表 40：**emconfigutil 选项**

选项	有效值	描述
`-f` 或 `--platform`	目标器件	必需。用于定义来自指定平台的目标器件。 要获取受支持的器件列表，请参阅 受支持的平台。
`--nd`	任何正整数值	可选。用于指定器件数量。默认值为 1。
`--od`	有效目录	可选。指定输出目录。运行仿真时，`emconfig.json` 文件必须与主机可执行文件位于相同目录中。默认操作将输出写入当前目录。
`-s` 或 `--save-temps`	不适用	可选。指定执行此命令后不删除中间文件，而是将其保留。默认操作是移除临时文件。
`--xp`	有效的赛灵思参数和属性。	可选。指定其它参数和属性。例如： `--xp prop:solution.platform_repo_paths=<xsa_path>` 此示例用于为目标平台设置搜索路径。
`-h` 或 `--help`	不适用	打印命令帮助。

`emconfigutil` 命令会在输出目录或当前工作目录内生成 `emconfig.json` 配置文件。

提示：运行仿真时，`emconfig.json` 文件必须与主机可执行文件位于相同目录中。

以下示例会创建以 2 个 xilinx_u200_qdma_201910_1 器件为目标的配置文件。

```
$emconfigutil --xilinx_u200_qdma_201910_1 --nd 2
```

kernelinfo 实用工具

kernelinfo 实用工具可以提取并显示来自赛灵思对象 (XO) 文件的信息，这些信息可在主机代码开发期间使用。此信息包含硬件函数名称、实参、偏移和端口数据。

其中包含如下命令选项：

表 41：**kernelinfo 命令**

选项	描述
-h [--help]	打印帮助消息。
-x [--xo_path] <arg>	到文件的绝对路径，包括文件名和 .xo 扩展名。
-l [--log] <arg>	默认情况下，信息显示在屏幕上。或者，您可以使用 --log 选项将信息输出为文件。
-j [--json]	以 JSON 格式输出文件。
[input_file]	XO 文件。指定 XO 文件位置，或者使用 --xo_path 选项。
[output_file]	赛灵思 OpenCL 编译器的输出。按位置指定输出文件，或者使用 --log 选项。

要运行 kernelinfo 实用工具，请在 Linux 终端中输入：

```
kernelinfo <filename.o>
```

输出分为 3 个部分：

- 内核定义
- 实参
- 端口

通过复审以下命令所生成的报告，有助于更好地了解报告内容。请注意，此报告拆分为多个特定部分以便于理解。

```
kernelinfo krnl_vadd.xo
```

其中 krnl_vadd.xo 是已编译的内核。

内核定义

报告高层次内核定义信息。在 `name` 字段中提供的内核名称至关重要，它可用于主机代码开发。在本例中，内核名称为 `krnl_vadd`。

```
=== Kernel Definition ===
name: krnl_vadd
language: c
vlnv: xilinx.com:hls:krnl_vadd:1.0
preferredWorkGroupSizeMultiple: 1
workGroupSize: 1
debug: true
containsDebugDir: 1
sourceFile: krnl_vadd/cpu_sources/krnl_vadd.cpp
```

实参

报告内核函数实参。

在以下示例中有 4 个实参：`a`、`b`、`c` 和 `n_elements`。

```
=== Arg ===
name: a
addressQualifier: 1
id: 0
port: M_AXI_GMEM
size: 0x8
offset: 0x10
hostOffset: 0x0
hostSize: 0x8
type: int*

=== Arg ===
name: b
addressQualifier: 1
id: 1
port: M_AXI_GMEM
size: 0x8
offset: 0x1C
hostOffset: 0x0
hostSize: 0x8
type: int*

=== Arg ===
name: c
addressQualifier: 1
id: 2
port: M_AXI_GMEM1
size: 0x8
offset: 0x28
hostOffset: 0x0
hostSize: 0x8
type: int*

=== Arg ===
```

```
name: n_elements
addressQualifier: 0
id: 3
port: S_AXI_CONTROL
size: 0x4
offset: 0x34
hostOffset: 0x0
hostSize: 0x4
type: int const
```

端口

报告内核所使用的存储器和控制端口。

```
=== Port ===
name: M_AXI_GMEM
mode: master
range: 0xFFFFFFFF
dataWidth: 32
portType: addressable
base: 0x0

=== Port ===
name: M_AXI_GMEM1
mode: master
range: 0xFFFFFFFF
dataWidth: 32
portType: addressable
base: 0x0

=== Port ===
name: S_AXI_CONTROL
mode: slave
range: 0x1000
dataWidth: 32
portType: addressable
base: 0x0
```

XILINX

第 24 章

launch_emulator 实用工具

对于具有 Arm 子系统的嵌入式平台，Vitis 工具使用 QEMU 来对 PS 子系统进行仿真。QEMU 进程必须随 RTL 仿真器 (simulator) 进程一起运行，才能对硬件仿真中的整个系统进行仿真。`launch_emulator.py` 实用工具用于启动 QEMU 并管理 PL 仿真器进程的同步。它会以所提供的实参来启动 QEMU 和仿真进程。Vitis IDE 在启动和停止仿真器 (emulator) 时也会调用此命令。

提示：如需获取 QEMU 内的帮助信息，请在仿真器 (emulator) shell 内按 “Ctrl + a h”。要终止 QEMU 命令，请在仿真器 (emulator) shell 内按 “Ctrl + a x”。

对于嵌入式平台，--package 选项 命令可生成 `launch_hw_emu.sh` 或 `launch_sw_emu.sh` 脚本以根据平台和目标应用通过所需实参来调用 `launch_emulator.py` 命令。

使用 `launch_hw_emu.sh` 或 `launch_sw_emu.sh` 封装器脚本时，您可通过命令行向 `launch_emulator` 实用工具传递其它实参。只需在运行脚本时向命令行追加相应选项即可。这样您即可按需自定义 `launch_emulator` 实用工具以支持自己的特定平台或应用。

下表显示了可用选项列表。

表 42：**launch_emulator.py 实用工具选项**

选项	接受的值	描述
`-aie-sim-config`	不适用	指向 AI 引擎仿真配置文件，此文件可提供 AI 引擎的 SystemC 模型所需的各种 AI 引擎文件。 该选项由 `v++` 包自动传递。 对于 AI 引擎设计，该选项是必需的。
`-aie-sim-options`	不适用	指向 AI 引擎仿真选项文件，其中包含 AI 引擎 SystemC 模块调试所需的各种 AI 引擎调试标记。 指定的选项文件应包含与 `package.hw_emu/sim/behav_waveform/xsim/` 相关的相对路径。 **提示：**该选项为可选，仅适用于 AI 引擎设计。
`-config-file`	配置文件（ini 格式）	指定用于命令的选项的配置文件。
`-device-family`	7Series | UltraScale | Versal	该选项为必需，用于指定平台的器件系列。 该选项由 `v++` 包生成的 `launch_hw_emu.sh` 或 `launch_sw_emu.sh` 脚本根据所选目标来自动传递。 要直接使用 `launch_emulator` 命令，需手动传递该选项。
`-disable-host-completion-check`	不适用	跳过主机/测试完成检查。通常在使用 python 脚本来检查测试完成状态 (PASS/FAIL) 的应用中使用该选项。 默认情况下，使用 `-run-app` 开关时搜索 “TEST PASSED”。
`-emu-data`	不适用	<其它仿真数据文件>：AI 引擎内核运行 QEMU 时需 shim 解决方案文件。 对于 AI 引擎设计，该选项是必需的。

表 42：**launch_emulator.py 实用工具选项** (续)

选项	接受的值	描述
`-enable-debug`	不适用	调试模式，为 QEMU 和 PL 打开 2 个不同 XTERM。 **重要提示！**该选项对于批处理模式用户了解 QEMU 与 PL 进程之间的流程和握手很有用。
`-forward-port`	<目标> <主机>	将 TCP 端口从目标转发到主机。
`-g | -graphic-xsim | -sim_gui`	不适用	启动可编程逻辑仿真器 (Programmable Logic Simulator) GUI。
`-gdb-port`	端口号	QEMU 会等待 <端口> 上的 GDB 连接。
`-help`	不适用	打印帮助消息。
`-kill`	<pid>	结束指定的仿真器 (emulator) 进程。
`-no-reboot`	不适用	退出 QEMU，而不是重启。用于通过在嵌入式 Linux 提示符处执行 `reboot -f` 命令来正常退出 QEMU。
`-noc-memory-config <path/to/noc_memory_config.txt>`	不适用	默认情况下，`v++ --package` 会基于设计配置创建 NoC 存储器配置，您可在看到仿真二进制文件的同时看到此文件。您可通过覆盖仿真二进制文件夹中指定的文件来覆盖此文件。`-user-pre-sim-script` 选项可用于将您的 `noc_memory_config.txt` 文件复制到仿真二进制文件区域，并应用配置。
`-pid-file`	文件名	将进程 ID 写入 <文件>，以便稍后用于 `-kill`。在仿真成功后，用于结束 Vitis 软件平台。
`-pl-sim-args`	仿真器 (simulator) 的实参	这些实参将被追加到仿真器 (simulator) 命令行中。可替代 `pm-sim-args-file`。
`-pl-sim-args-file`	仿真实参文件名	在此文件中可提供仿真器 (simulator) 工具的任意选项。
`-pl-sim-dir`	仿真 (Simulation) 目录	通过从该目录启动脚本来启动可编程逻辑仿真器 (Programmable Logic Simulator)。该选项在 `v++` 包生成的脚本中自动传递。该工具预期在指定目录中存在名为 `simulate.sh` 的文件，并且将执行该文件以启动 PL 仿真器（例如，XSIM）。
`-pl-sim-script`	仿真脚本位置	高级用户可使用单个直接脚本来启动仿真（例如，Vivado 用户）。 指定该选项后，运行脚本时其它选项无价值。
`-pmc-args`	PMC 实参	PMC/PMU 由 `qemu-system-microblazeel` 进行仿真。在 `pmc_args.txt` 中可捕获大部分常用的 PMC 命令行切换。 您可直接提供需追加到 PMC 命令行的所有实参，而无需写入名为 `pmc_args.txt` 的文件。该选项可替代 `-pmc-args-file`。 在 适用于 QEMU 的 Versal PS 和 PMC 实参 以及后续有关 Zynq 器件的章节中可找到特定器件的 PMC/PMU 实参。
`-pmc-args-file`	PMC QEMU 实参文件名	在此文件中可提供要传递给 PMU/PMC 的任意选项。具体格式由所选平台上的库文件来确定。 该选项在 `v++` 包生成的脚本中自动传递。
`-pmc-dtb`	<path_to_DTB_file>	`v++ --package` 会基于设计中的寻址来自动创建设备树二进制 (DTB) 文件，并将此文件传递给 `launch_emulator` 命令。该选项允许您指定用于覆盖默认设置的 DTB 文件。 **注释：**请确保此 DTB 与所使用的 `noc_memory_config.txt` 文件兼容。
`-ps-only`	不适用	仅限 PS 仿真 (emulation)。无 PL 仿真 (emulation)

表 42：**launch_emulator.py 实用工具选项** (续)

选项	接受的值	描述
-qemu-args	QEMU 实参	PS 由 qemu-system-aarch64 进行仿真。在 qemu_args.txt 中可捕获大部分常用的 PS 命令行切换。 您可直接提供需追加到 QEMU 命令行的所有实参，而无需写入名为 qemu_args.txt 的文件。该选项可替代 qemu-args-file。 在 适用于 QEMU 的 Versal PS 和 PMC 实参 以及后续有关 Zynq 器件的章节中可找到特定器件的 PS 实参。
-qemu-args-file	PS QEMU 实参文件名	在此文件中可提供要传递给 QEMU 的任意选项。您可通过此特定格式从所选平台提取库文件。该选项在 v++ 包生成的脚本中自动传递。
-qemu-dtb	<path_to_DTB_file>	v++ --package 会基于设计中的寻址来自动创建 DTB 文件，并将此文件传递给 launch_emulator 命令。该选项允许您指定用于覆盖默认设置的 DTB 文件。 **注释：**请确保此 DTB 与所使用的 noc_memory_config.txt 文件兼容。
-qspi-high-image	指定 QSPI 高位镜像文件	此镜像文件作为 QEMU 实参（以启动模式的形式）来传递。该选项在 V++ 包生成的脚本中自动传递。 仅当使用 DUAL QSPI 模式时，才需要该选项。
-qspi-image	指定 qspi.bin	此镜像文件作为 QEMU 实参（以启动模式的形式）来传递。该选项在 V++ 包生成的脚本中自动传递。 仅当选择采用 QSPI 模式时，才需要该选项。
-qspi-low-image	指定 QSPI 低位镜像文件	此镜像文件作为 QEMU 实参（以启动模式的形式）来传递。该选项在 V++ 包生成的脚本中自动传递。 仅当使用 DUAL QSPI 模式时，才需要该选项。
-result-string	不适用	Result 字符串用于搜索测试完成的状态。默认值 = TEST PASSED。
-run-app	<application_script_name>	确保在封装步骤中，使用 --package.sd_file 选项来封装应用脚本。仅当应用脚本已封装在 sd_card 中之后，在 QEMU 装载、启动并正常运行后，才能运行此应用脚本。
-run-sim-in-gdb	不适用	在 GDB 中运行仿真器 (Simulator)。
-runtime	c++/ocl	指定 C++ 或 OCL 的运行时流程。
-sd-card-image	指定 sd_card.img	此镜像文件作为 QEMU 实参（以启动模式的形式）来传递。该选项在 V++ 包生成的脚本中自动传递。 仅当使用 SD 模式时，才需要该选项。
-t | -target	sw_emu 或 hw_emu	指定运行 sw_emu 或 hw_emu。 根据 v++ 中所选的目标，v++ 包会生成相应的脚本。 对于 sw_emu 目标，会生成 launch_sw_emu.sh，对于 hw_emu 目标，则会生成 launch_hw_emu.sh。
-timeout	<n>	<n> 秒后终止仿真 (emulation)。
-user-post-sim-script	该选项表示仿真后，在退出前需执行的 Tcl 脚本的路径	为转发到 Tcl 文件中的任意操作创建 Tcl 并将 Tcl 脚本传递到此开关。
-user-pre-sim-script	Tcl 脚本路径	对于首次运行，在 GUI 模式下执行 launch_emulator.py，并添加要观测的信号。 将命令从 Tcl 控制台复制并保存到 Tcl 脚本中。 从下一次运行开始，以批处理模式传递此 Tcl 脚本，launch_emulator.py -user-pre-sim-script <path_to_saved_tcl_script>。 仅支持 Vivado 仿真器 (xsim)。

表 42：**launch_emulator.py 实用工具选项** (续)

选项	接受的值	描述
-vivado	$XILINX_VIVADO	设置 VIVADO_LOC，供 simulate.sh 用于加载 simulation/c-model libraries
-wcfg-file-path	不适用	指定 XSIM 创建的 wcfg 文件，指定的文件将在 GUI 仿真期间打开
-xtlm-aximm-log	不适用	此开关可生成 xTLM AXI4 传输事务日志用于 2 个 SystemC 模型之间的接口连接（包含诸如地址、数据、大小等信息）。 运行此开关时，可从以下位置获取仿真 (emulation) 日志（目录结构因使用的 v++ 选项和仿真器 (simulator) 而异）： package.hw_emu/sim/behav_waveform/xsim/xsc_report.log
-xtlm-axis-log	不适用	此开关可生成 xTLM AXI4-Stream 传输事务日志，用于 2 个 SystemC 模型之间的接口连接。 运行此开关时，可从以下位置获取仿真 (emulation) 日志（目录结构因使用 v++ 选项和仿真器 (simulator) 而异）： package.hw_emu/sim/behav_waveform/xsim/xsc_report.log

适用于 QEMU 的 Versal PS 和 PMC 实参

在 Versal™ 器件中，PS(a72) 由 qemu-system-aarch64 进行仿真，PMC 由 qemu-system-microblazeel 进行仿真。在 qemu_args.txt 中捕获大部分常用的 PS 命令行开关，在 pmc_args.txt 中捕获 PMC 命令行开关。

表 43：**适用于 qemu_args.txt 的 Versal 选项**

实参名称	值	描述	源	如何提取信息
-boot	mode=<boot-number> 例如，对于 sd1：-boot mode=5	指定平台上的启动模式： · qspi24 = 1 · qspi32 = 2 · sd0 = 3 · sd1 = 5 · emmc0 = 6 · ospi = 8	v++ -p	在 CIPS 启用的启动模式与 v++ -p 选择之间需 DRC
-display	none	默认情况下，QEMU 会为用户 I/O 创建 display。该选项用于禁用 display	静态	指定 none 即可禁用 display
-hw-dtb	<ps-dtb-file>	描述 PS (a72) 的设备树文件。Vitis 编译器 --package 命令用于生成 dtb 文件，并将其追加到 qemu-args.txt。	v++ -p	
-M	arm-generic-fdt	用于指定要创建的 QEMU 机器。arm-generic-fdt 机器选项可告知 QEMU 解析 dtb 用于机器生成，由 -hw-dtb user.dtb 进行传递。	与器件相关	针对 Versal 进行硬编码

表 43：**适用于 qemu_args.txt 的 Versal 选项** (续)

实参名称	值	描述	源	如何提取信息

-serial	-serial null -serial null -serial mon:stdio	默认情况下，QEMU 会将调用终端连接到 UART0 以执行用户 I/O 操作。您可以通过指定该选项来改写此行为。Versal 平台使用定位实参指定了 4 个 UART：前 2 个用于调试，后 2 个分别为 UART0 和 UART1。
		要将 UART0 连接到终端，请指定 `-serial null -serial null -serial mon:stdio`，这样即可忽略调试相关的 UART 并将 UART0 连接至终端。
		同样，要仅将 UART1 连接到终端，请指定 `-serial null -serial null -serial null -serial mon:stdio`
			基于 CIPS 配置上启用的 UART。	Versal 器件具有 4 个 UART。前 2 个为调试相关 UART。
				启用 UART0 时：
				`CONFIG.PS_UART0_PERIPHERAL_ENABLE = 1`
				`CONFIG.PS_UART1_PERIPHERAL_ENABLE =0 or 1`
				则指定 `-serial null -serial null -serial mon:stdio`
				仅启用 UART1 时：
				`CONFIG.PS_UART0_PERIPHERAL_ENABLE = 0`
				`CONFIG.PS_UART1_PERIPHERAL_ENABLE = 1`
				则指定：`-serial null -serial null -serial null -serial mon:stdio`

-sync-quantum	时间（以毫秒为单位）	指定 QEMU 与 RTL 仿真器 (simulator) 同步的频率。对此项进行修改可能影响仿真 (simulation) 速度。	静态	针对 Versal 器件进行硬编码（需用户更改）

Versal CIPS 具有 2 个以太网接口。大部分赛灵思 Versal CIPS 开发板均已启用 eth0。如不指定 `-net` 或 `-netdev`，那么默认情况下 QEMU 启用 eth0 并映射到用户模式后端。

表 44：**适用于 pmc_args.txt 的 Versal 选项**

开关名称	值	描述	配置源	如何提取信息
-M	microblaze-fdt	指定要创建的 QEMU 机器。QEMU 以来自 `dtb` 的节点创建 MicroBlaze。	静态	针对 Versal 器件进行硬编码
-display	none	默认情况下，QEMU 会为用户 I/O 创建 display。该选项指示 QEMU 无需显示 (display)。	静态	针对 Versal 器件进行硬编码

表 44：**适用于 pmc_args.txt 的 Versal 选项** (续)

开关名称	值	描述	配置源	如何提取信息
-device	loader,file=<BOOT_bh.bin>,addr=0xf201e000,force-raw	指定启动头文件，其中加载地址指定为 0xF201E000（BOOT_bh.bin 加载地址为 0xF201E000）。这是 pmc_args.txt 中的固定实参，由 v++ -p 进行处理并用作为最终实参，其中包含 BOOT_bh.bin 文件的绝对路径。BOOT_bh.bin 是由 v++ -p 从最终 PDI 生成的。BOOT_bh.bin 直接加载到 QEMU 上，因为 QEMU 不具有从 PDI 加载启动头文件的 BOOT ROM 访问权限。	v++ -pack	v++ pack 可提取 BOOT_bh.bin 并生成此开关。
-device	loader,file=<pmc_cdo.bin>,addr=0xF2000000,force-raw	指定 pmc_cdo.bin，加载地址为 0xF2000000。这是 pmc_args.txt 中的固定实参。它由 v++ -p 进行处理并用作为最终实参，其中包含 pmc_cdo.bin 文件的绝对路径。	v++ -pack	v++ pack 可提取 pmc_cdo.bin 并生成此开关。
-device	loader,file=<plm.bin>,addr=0xF0200000,force-raw	指定 plm 二进制文件固件，加载地址为 0xF0200000。这是 pmc_args.txt 中的固定实参，由 v++ -p 进行处理并用作为最终实参。其中包含 plm.bin 文件的绝对路径。此 plm 由 PPU1 在脱离复位状态后执行。	v++ -pack	v++ pack 可提取 plm.bin 并生成此开关。
-device	loader,addr=0xf0000000,data=0xba020004,data-len=4 -device loader,addr=0xf0000004,data=0xb800fffc,data-len=4	指定 PPU0 进程进入启动循环。由于 QEMU 不具有 BOOTROM 访问权限，PPU0 将置于启动循环内，这样通常会将 BOOT 头文件从 PDI 加载到存储器中。	静态	针对 Versal 器件进行硬编码
-device	loader,addr=0xF1110624,data=0x0,data-len=4 -device loader,addr=0xF1110620,data=0x1,data-len=4	使 PPU1 脱离复位状态并置于执行模式下。	静态	针对 Versal 器件进行硬编码
-hw-dtb	<ps-dtb-file>	描述 aout PS(a72) 的 dtb 文件。v++ pack 会生成此 dtb 文件，并将其追加到 qemu-args.txt。	v++ pack	v++ pack 会基于器件上的 DDR 配置来生成 dtb 文件。

适用于 QEMU 的 Zynq UltraScale+ MPSoC PS 和 PMU 实参

Zynq UltraScale+ MPSoC PS(a53) 由 qemu-system-aarch64 进行仿真，PMU 由 qemu-system-microblazeel 进行仿真。在 qemu_args.txt 中捕获大部分常用的 PS 命令行开关，在 pmu_args.txt 中捕获 PMC 命令行开关。

表 45：**适用于 qemu_args.txt 的 Zynq UltraScale+ MPSoC 选项**

开关名称	值	描述	配置源	如何提取信息
-M	arm-generic-fdt	用于指定要创建的 QEMU 机器。`arm-generic-fdt` 机器选项可告知 QEMU 解析 `dtb` 用于机器生成，由 `-hw-dtb user.dtb` 进行传递。	静态	针对 Zynq 器件进行硬编码

-serial	mon:stdio	-serial 为定位实参。将串行端口重定向到指定的 char dev（即，stdio、tcp port、file 等）。	基于 Zynq UltraScale+ MPSoC 上的 UART 配置	Zynq 具有 2 个 UART。
				启用 UART0 时：
				`CONFIG.PSU__UART0__PERIPHERAL__ENABLE = 1`
				`CONFIG.PSU__UART1__PERIPHERAL__ENABLE = 0 or 1`
				则指定：`-serial mon:stdio`
				仅启用 UART1 时：
				`CONFIG.PSU__UART0__PERIPHERAL__ENABLE = 0`
				`CONFIG.PSU__UART1__PERIPHERAL__ENABLE = 1`
				则指定：`-serial null -serial mon:stdio`

-global	xlnx,zynqmp-boot.cpu-num=0	使指定 CPU 脱离复位。	静态	针对 Zynq 器件进行硬编码

表 45：**适用于 qemu_args.txt 的 Zynq UltraScale+ MPSoC 选项** (续)

开关名称	值	描述	配置源	如何提取信息

-net	-net nic -net nic -net nic -net nic -net user	`-net` 为定位实参。初始化网络接口 gem3。将指定网络适配器连接到用户模式网络。
		提示： `-net none` 将禁用所有以太网接口。
			静态	基于以太网配置：
				如果启用 gem0(eth0)：
				`CONFIG.PSU__ENET0__PERIPHERAL__ENABLE =1`
				则指定 `-net nic -net user`
				如果启用 gem1：
				`CONFIG.PSU__ENET1__PERIPHERAL__ENABLE = 1`
				则指定 `-net nic -net nic -net user`
				如果启用 gem2：
				`CONFIG.PSU__ENET2__PERIPHERAL__ENABLE = 1`
				则指定 `-net nic -net nic -net nic -net user`
				如果启用 gem 3：
				`CONFIG.PSU__ENET3__PERIPHERAL__ENABLE = 1`
				则指定 `-net nic -net nic -net nic -net nic -net user`
				提示：如果未提及 `-net`（和/或 `-netdev`），则默认情况下 QEMU 将启用第一个以太网 (gem0)，并将其映射到用户模式后端。

-m	4G	在 Zynq UltraScale+ MPSoC 上启用 4 GB DDR。	静态	在 Zynq UltraScale+ MPSoC 上仿真整个 DDR
-device	loader,file=<bl31.elf>,cpu-num=0	在 A53 核 0 上加载 `bl31.elf` 文件。	静态	`v++ --package` 应将 `bl31.elf` 替换为 `bl31.elf` 的绝对路径
-device	loader,file=<u-boot.elf>	加载 `u-boot.elf`.	静态	`v++ --package` 应将 `bl31.elf` 替换为 `u-boot.elf` 的绝对路径

表 45：**适用于 qemu_args.txt 的 Zynq UltraScale+ MPSoC 选项** (续)

开关名称	值	描述	配置源	如何提取信息
-hw-dtb	<ps-dtb-file>	描述由 QEMU 仿真的 PS 的 dtb 文件，可使用 -hw-dtb 来指定。	静态	针对 Zynq 器件进行硬编码： <ps-dtb-file>=/proj/xbuilds/HEAD_daily_latest/installs/lin64/Vitis/HEAD//data/emulation/dtbs/zynqmp/zynqmp-arm-cosim.dtb

表 46：**适用于 pmu_args.txt 的 Zynq UltraScale+ MPSoC 选项**

开关名称	值	描述	配置源	如何提取信息
-M	microblaze-fdt	用于指定要创建的 QEMU 机器。microblaze-fdt 会告知 QEMU 解析 dtb 用于机器生成，由 -hw-dtb user.dtb 进行传递。	静态	针对 Zynq 器件进行硬编码
-device	loader,file=<pmufw.elf>	在 PMU RAM 上加载 pmufw.elf 文件。	静态	针对 Zynq 器件进行硬编码
-machine-path	<path-to-xsim-dir>	将 -machine-path 指向文件夹，以创建共享 RAM 和远程端口插槽。	静态	launch_emulator 命令将用于设置此机器路径
-display	none	默认情况下，QEMU 会为用户 I/O 创建 display。该选项用于禁用 display。	静态	针对 Zynq 器件进行硬编码
-hw-dtb	<pmu-dtb-file>	描述由 QEMU 仿真的 PMU 的 dtb 文件，可使用 -hw-dtb 来指定。	静态	<pmu-dtb-file>=/proj/xbuilds/HEAD_daily_latest/installs/lin64/Vitis/HEAD//data/emulation/dtbs/zynqmp/zynqmp-pmu.dtb

提示：虽然此文件此处称为 pmu_args.txt，但此文件是使用 -pmc-args-file 命令为 launch_emulator 指定的。

适用于 QEMU 的 Zynq-7000 PS 实参

Zynq-7000 PS(a9) 由 qemu-system-aarch64 QEMU 二进制文件进行仿真。在 qemu_args.txt 内会捕获 PS 的大部分常用命令行开关。

表 47：**适用于 qemu_args.txt 的 Zynq-7000 选项**

开关名称	值	描述	配置源	如何提取信息
-M	arm-generic-fdt-7series	指示要创建的 QEMU 机器。arm-generic-fdt-7series 会告知 QEMU 解析 `dtb` 用于机器生成，由 `-hw-dtb user.dtb` 进行传递。	静态	针对 Zynq 器件进行硬编码

-serial	-serial /dev/null -serial mon:stdio	将串行端口重定向到指定的 char dev（即，stdio、tcp port、file 等）	基于 Zynq IP 的 UART 配置。	Zynq 具有 2 个 UART。
				启用 UART0 时：
				`CONFIG.PCW_UART0_PERIPHERAL_ENABLE = 1`
				`CONFIG.PCW_UART1_PERIPHERAL_ENABLE = 0 or 1`
				则指定：`-serial mon:stdio`
				仅启用 UART1 时：
				`CONFIG.PCW_UART1_PERIPHERAL_ENABLE = 1`
				则指定：`-serial null -serial mon:stdio`

-device	loader,addr=0xf8000008,data=0xDF0D,data-len=4 -device loader,addr=0xf8000140,data=0x00500801,data-len=4 -device loader,addr=0xf800012c,data=0x1ed044d,data-len=4 -device loader,addr=0xf8000108,data=0x0001e008,data-len=4 -device loader,addr=0xF800025C,data=0x00000005,data-len=4 -device loader,addr=0xF8000240,data=0x00000000,data-len=4	寄存器写入 SLCR 块，哦那个与设置 PLL 和 CLK_CTRL 寄存器（针对 Linux 为必需）。	静态	针对 Zynq 器件进行硬编码
-boot	mode=5	启动模式 5 适用于 SD 启动。	`v++ -p`	
-kernel	<u-boot.elf>	要在启动期间加载的访客软件。	静态	<u-boot.elf> 替换为来自目标平台的 `u-boot.elf` 的绝对路径
-machine	linux=on	将 QEMU 本身用作为 Linux 镜像的加载器。	静态	针对 Zynq 器件进行硬编码

manage_ipcache 实用工具

为了在应用设计中进行内核综合期间提供更好的性能，Vitis 编译器使用 IP 高速缓存来存储和复用综合结果。这样即可使 `.xclbin` 文件的构建进程免于为未曾发生更改的内核与 CU 重复执行综合。IP 高速缓存可以存储综合结果，并将其应用于设计中未更改的内核。

默认情况下，IP 高速缓存存储在工程的 Vitis IDE 工作空间内，或者从命令行运行 `v++` 时，IP 高速缓存存储在构建级别。您可使用 `--remote_ip_cache` 指定新位置以便自定义 IP 高速缓存的位置，或者也可以使用 `--no_ip_cache` 来禁用 IP 高速缓存的使用。如需了解有关这些选项的更多信息，请参阅 Vitis 编译器常规选项。

`manage_ipcache` 实用工具是一个独立实用工具，可帮助您管理自己的 IP 高速缓存存储库内容。它允许您报告 IP 高速缓存存储库的统计数据，并基于各种条件来移除其中条目。

表 48：**manage_ipcache 选项**

选项	描述
`-c | --cache`	必需。指定要使用的 IP 高速缓存目录。
`-d | --disk_space <size>`	删除除最近使用的条目以外的所有条目，保留的条目应与指定的磁盘空间大小（以 MB 为单位）匹配。
`-h | --help`	打印 `manage_ipcache` 命令的帮助。
`-k | --keep_top <N>`	删除除最近使用的前 N 个条目（N 为整数）外的所有条目。
`-o | --outfile <file>`	报告指定文件的 IP 高速缓存统计数据。
`-p | --purge`	删除所有高速缓存条目。
`-r | --report`	报告 stdout 的 IP 高速缓存的统计数据。
`-u | --unused`	删除从未曾使用（无缓存命中）的高速缓存条目。

以下示例报告了指定 IP 高速缓存的条目：

```
manage_ipcache --cache ./ip_cache --report
```

`manage_ipcache` 命令如果成功，则返回 0，如果发生错误，则返回 -1。

package_xo 命令

语法

```
package_xo  -kernel_name <arg> [-force] [-kernel_xml <arg>]
            [-output_kernel_xml <arg>] [-design_xml <arg>]
            [-ip_directory <arg>] [-parent_ip_directory <arg>]
            [-kernel_files <args>] [-kernel_xml_args <args>]
            [-kernel_xml_pipes <args>] [-kernel_xml_connections <args>]
            [-ctrl_protocol <arg>] -xo_path <arg> [-quiet] [-verbose]
```

描述

`package_xo` 命令是 Vivado Design Suite 中的 Tcl 命令。以 RTL 编写的内核在 Vivado 工具内使用 `package_xo` 命令行实用工具来编译，以生成赛灵思对象 (XO) 文件，随后供 `v++` 命令在链接阶段中使用。

表 49：**实参**

实参	描述
`-kernel_name <arg>`	（必需）指定 RTL 内核的名称。
`-force`	（可选）覆盖现有 XO 文件（如果存在）。
`-kernel_xml <arg>`	（可选）指定到现有内核 XML 文件的路径。Vivado 工具将为 XO 文件创建 `kernel.xml` 文件。
`-output_kernel_xml`	（可选）指定写入内核 XML 文件的路径。Vivado 工具将创建 `kernel.xml` 文件，将其包含在 XO 文件内，并写入指定的输出文件。 **提示：**您可以使用该选项来生成 `kernel.xml` 文件，以供您对其进行编辑并在 `package_xo` 命令中用作为输入。
`-design_xml <arg>`	（可选）指定到现有设计 XML 文件的路径
`-ip_directory <arg>`	（可选）指定到已封装的 IP 目录的路径。
`-parent_ip_directory`	（可选）如果指定的内核 IP 目录包含多个 IP，则指定到父 IP 的目录路径，该内核 IP 目录的 `component.xml` 即直接位于目录路径下。
`-kernel_files`	（可选）内核文件名。可用于将 C 语言模型添加到您的内核 XO 以便为您的内核启用软件仿真。
`-kernel_xml_args <args>`	（可选）生成含指定函数实参的 `kernel.xml`。每个实参值都应使用以下格式： `{name:addressQualifier:id:port:size:offset:type:memSize}` **注释：** `memSize` 为可选。
`-kernel_xml_pipes <args>`	（可选）生成含指定的一个或多个流水线的 `kernel.xml`。每个流水线值均使用以下格式： `{name:width:depth}`

表 49：**实参** (续)

实参	描述
`-kernel_xml_connections <args>`	（可选）生成含指定连接的 `kernel.xml` 文件。每个连接值都应使用以下格式。 `{srcInst:srcPort:dstInst:dstPort}`
`-ctrl_protocol`	内核控制协议，如 内核属性 中所述。有效值包括：`ap_ctrl_hs`、`ap_ctrl_chain`、`ap_ctrl_none` 和 `user_managed`。 **提示：**如果未指定 `-ctrl_protocol`，那么默认 `ap_ctrl_hs` 将写入 `kernel.xml` 文件。
`-xo_path <arg>`	（必需）指定已编译的对象 (XO) 文件的路径和文件名。
`-quiet`	（可选）以静默方式执行命令，不返回来自该命令的任何消息。此命令还会返回 TCL_OK，忽略执行期间遇到的所有错误。 **注释：**启动该命令时，将返回命令行上遇到的任何错误。仅捕获该命令内部发生的错误。
`-verbose`	（可选）暂时改写任何消息限制，并返回来自该命令的所有消息。 **注释：**可使用 `set_msg_config` 命令定义消息限制。

示例

以下示例使用 `ap_ctrl_chain` 控制协议创建指定 XO 文件，其中包含指定名称的 RTL 内核，并创建 `kernel.xml` 文件，因为尚未指定此文件。

```
package_xo -xo_path Vadd_A_B.xo -kernel_name Vadd_A_B -ctrl_protocol
ap_ctrl_chain -ip_directory ./ip
```

以下示例使用指定 `kernel.xml` 文件创建 XO 文件：

```
package_xo -xo_path Vadd_A_B.xo -kernel_name Vadd_A_B -kernel_xml
kernel.xml -ip_directory ./ip
```

提示：控制协议将在指定的 `kernel.xml` 文件内定义。

RTL 内核 XML 文件

提示：`package_xo` 命令将从已封装的 IP 的 `component.xml` 创建 `kernel.xml` 文件，因此您无需手动提供此文件或使用 RTL Kernel Wizard 来生成此文件。

针对每个 RTL 内核必须创建 1 个名为 `kernel.xml` 的 XML 内核描述文件，以供在 Vitis 应用加速开发流程中使用。此 `kernel.xml` 文件可指定内核属性，如运行时和 Vitis 工具流程所需的寄存器映射和端口。以下代码显示的是 `kernel.xml` 文件的示例。

```
<?xml version="1.0" encoding="UTF-8"?>
<root versionMajor="1" versionMinor="6">
  <kernel name="vitis_kernel_wizard_0" language="ip_c"
      vlnv="mycompany.com:kernel:vitis_kernel_wizard_0:1.0"
      attributes="" preferredWorkGroupSizeMultiple="0" workGroupSize="1"
interrupt="true">
    <ports>
      <port name="s_axi_control" mode="slave" range="0x1000" dataWidth="32"
portType="addressable" base="0x0"/>
      <port name="m00_axi" mode="master" range="0xFFFFFFFFFFFFFFFF"
dataWidth="512" portType="addressable"
          base="0x0"/>
    </ports>
    <args>
      <arg name="axi00_ptr0" addressQualifier="1" id="0" port="m00_axi"
size="0x8" offset="0x010" type="int*"
          hostOffset="0x0" hostSize="0x8"/>
    </args>
  </kernel>
</root>
```

注释： `kernel.xml` 文件可使用 RTL Kernel Wizard 来自动创建，以指定 RTL 内核的接口规范。如需了解更多信息，请参阅 RTL Kernel Wizard。

下表详细描述了 `kernel.xml` 的格式。

表 50：**内核 XML 文件内容**

标签	属性	描述
`<root>`	`versionMajor`	对于最新版本的 Vitis 软件平台，该属性设置为 1。
	`versionMinor`	对于最新版本的 Vitis 软件平台，该属性设置为 6。

表 50：**内核 XML 文件内容** (续)

标签	属性	描述
<kernel>	name	内核名称
	language	对于 RTL 内核，该属性始终设置为 ip_c。
	vlnv	必须与 IP 的 component.xml 中的供应商、库、名称和版本属性匹配。例如，如果 component.xml 包含以下标签： <spirit:vendor>xilinx.com</spirit:vendor> <spirit:library>hls</spirit:library> <spirit:name>test_sincos</spirit:name> <spirit:version>1.0</spirit:version> 内核 XML 中的 vlnv 属性必须设置为： xilinx.com:hls:test_sincos:1.0
	attributes	保留。将其设置为空字符串：""
	preferredWorkGroupSizeMultiple	保留。将其设置为 0。
	workGroupSize	保留。将其设置为 1。
	interrupt	如果 RTL 内核具有中断，该属性设置为“true”(interrupt="true")，否则省略。
	hwControlProtocol	指定 RTL 内核的控制协议。 · ap_ctrl_hs：RTL 内核的默认控制协议。 · ap_ctrl_chain：支持数据流的链式内核的控制协议。在控制寄存器中添加 ap_continue 以启用 ap_done/ap_continue 完成确认。 · ap_ctrl_none：为数据驱动的内核应用的控制协议 (none)。 · user_managed：指定满足接口要求的内核，Vitis 编译器遵循此接口要求即可将此内核与其它内核进行链接，但不遵循 XRT 的执行管理要求。如需了解更多信息，请参阅 创建用户管理的 RTL 内核。
<port>	name	指定端口名称。 **重要提示！** AXI4-Lite 接口必须命名为 S_AXI_CONTROL。
	mode	至少需要一个 AXI4 主端口和一个 AXI4-Lite 从端口。 可指定 AXI4-Stream 端口以在内核之间进行数据流传输。 · 对于 AXI4 主端口，该属性设置为“master”。 · 对于 AXI4 从端口，该属性设置为“slave”。 · 对于 AXI4-Stream 主端口，将其设置为“write_only”。 · 对于 AXI4-Stream 从端口，将其设置为“read_only”。
	range	端口的地址空间范围。
	dataWidth	通过此端口的数据宽度，默认为 32 位。
	portType	指示端口是否可寻址或进行数据流传输。 · 对于 AXI4 主端口和从端口，将其设置为“addressable”。 · 对于 AXI4-Stream 端口，将其设置为“stream”。
	base	对于 AXI4 主端口和从端口，将其设置为 0x0。此标签不适用于 AXI4-Stream 端口。

表 50：**内核 XML 文件内容** (续)

标签	属性	描述
<arg>	name	指定内核软件实参名称。
	addressQualifier	有效值： 0：标量内核输入实参 1：全局存储器 2：本地存储器 3：常数存储器 4:：管道
	id	仅适用于 AXI4 主端口和从端口。ID 应是有顺序的。它用于确定内核实参的顺序。 不适用于 AXI4-Stream 端口。
	port	指定与 arg 连接的 <port> 名称。
	size	实参大小（以字节为单位）。默认值为 4 个字节。
	offset	表示寄存器存储器地址。
	type	实参的 C 数据类型。例如，uint*、int* 或 float*。
	hostOffset	保留。设置为 0x0。
	hostSize	实参的大小。默认值为 4 个字节。
	memSize	对于 AXI4-Stream 端口，memSize 设置创建的 FIFO 的深度。 **提示：**不适用于 AXI4 端口。
以下标签用于为 AXI4-Stream 端口指定其它标签。这些标签不适用于 AXI4 端口。		
<connection>	connection 标签用于描述硬件中的实际连接，从内核到针对 PIPE 插入的 FIFO 的连接或从 FIFO 到内核的连接。	
	srcInst	指定连接的源实例。
	srcPort	指定连接的源实例上的端口。
	dstInst	指定连接的目标实例。
	dstPort	指定连接的目标实例上的端口。

platforminfo 实用工具

`platforminfo` 命令行实用工具可用于以结构化格式来报告平台元数据，包括有关接口、时钟、有效的 SLR 和已分配的资源的信息以及有关存储器的信息。举例来说，在向 SLR 或存储器资源分配内核时，可参考此信息。

以下命令选项可搭配 `platforminfo` 一起使用：

表 51：**platforminfo 命令**

选项	描述
`-f [ --force ]`	覆盖现有输出文件。
`-h [ --help ]`	打印帮助消息并退出。
`-k [ --keys ]`	获取给定平台的密钥。返回 JSON 路径的列表。
`-l [ --list ]`	列出平台。搜索用户存储库路径 $PLATFORM_REPO_PATHS，然后搜索安装位置，以查找 `.xpfm` 文件。
`-e [ --extended ]`	列出含扩展信息的平台。搭配“--list”一起使用。
`-d [ --hw ] <arg>`	硬件平台的定义 (`*.dsa`)，用户在此平台上执行操作。该值必须为完整路径，包括文件名和 `.dsa` 扩展名。
`-s [ --sw ] <arg>`	软件平台定义 (`*.spfm`)，用户在此平台上执行操作。该值必须为完整路径，包括文件名和 `.spfm` 扩展名。

`-p [ --platform ] <arg>`	赛灵思平台定义 (`*.xpfm`)，用户在此平台上执行操作。`--platform` 的值可以是包含文件名和 `.xpfm` 扩展名的完整路径，如以下示例 1 所示。如果仅提供文件名和 `.xpfm` 扩展名但不提供路径，则此实用工具将仅搜索当前工作目录。您还可以仅指定平台的基本名称。如果提供的值为基本名称，那么该实用工具将搜索 $PLATFORM_REPO_PATHS 和安装位置，以查找对应的 `.xpfm` 文件，如以下示例 2 所示。
	`Example 1: --platform /opt/xilinx/platforms/xilinx_u50_gen3x16_xdma_201920_3.xpfm`
	`Example 2: --platform xilinx_u200_xdma_201830_1`

`-o [ --output ] <arg>`	指定结果要写入的输出文件。默认情况下，输出将返回至终端 (stdout)。

`-j [ --json ] <arg>`	指定生成的输出的 JSON 格式。如果使用时不提供值，那么 `platforminfo` 实用工具会以 JSON 格式打印整个平台。该选项还接受指定 JSON 路径的实参，此路径为 `-k` 选项返回的路径。此 JSON 路径如果有效，则用于提取 JSON 子树、列表或值。
	`Example 1: platforminfo --json="hardwarePlatform" --platform <platform base name>`
	`Example 2: Specify the index when referring to an item in a list. platforminfo --json="hardwarePlatform.devices[0].name" --platform <platform base name>`
	`Example 3: When using the short option form (-j), the value must follow immediately. platforminfo -j"hardwarePlatform.systemClocks[]" -p <platform base name>`

表 51：**platforminfo 命令** (续)

选项	描述
`-v [ --verbose ]`	指定含更多详细信息的输出。默认行为是生成人类可读的报告，其中包含指定平台的最重要的特性。

注释：除非使用 `--help` 或 `--list` 选项，否则必须指定平台。您可以使用 `--platform` 选项或者使用 `--hw` 或 `--sw` 来指定平台。您也可以直接在命令行中的具体位置插入平台名称或完整路径。

要了解生成的报告，请基于以下命令复查简明输出日志。请注意，此报告拆分为多个特定部分以便于理解。

```
platforminfo -p $PLATFORM_REPO_PATHS/xilinx_u200_xdma_201830_1.xpfm
```

提示：请参阅 xilinx_zcu104_base_202010_1 的平台信息 以获取嵌入式处理器平台的示例。

基本平台信息

以下报告了平台信息及高层次描述。

```
Platform:           xdma
File:               /proj/xbuilds/2020.2_daily_latest/internal_platforms/
xilinx_u200_xdma_201830_3/xilinx_u200_xdma_201830_3.xpfm
Description:
```

硬件平台信息

报告有关硬件平台的常规信息。对于“软件仿真和硬件仿真 (Software Emulation and Hardware Emulation)”字段，值为“1”表示此平台适合这些配置。

```
Vendor:                           xilinx
Board:                            U200 (xdma)
Name:                             xdma
Version:                          201830.3
Generated Version:                2018.3
Hardware:                         1
Software Emulation:               1
Hardware Emulation:               1
Hardware Emulation Platform:      0
FPGA Family:                      virtexuplus
FPGA Device:                      xcu200
Board Vendor:                     xilinx.com
Board Name:                       xilinx.com:au200:1.0
Board Part:                       xcu200-fsgd2104-2-e
```

接口信息

以下显示了报告的 PCIe 接口信息。

```
Interface Name: PCIe
Interface Type: gen3x16
PCIe Vendor Id:    0x10EE
PCIe Device Id:    0x5000
PCIe Subsystem Id: 0x000E
```

时钟信息

报告可用的最大内核时钟频率。“时钟索引 (Clock Index)”是覆盖默认值时，`--kernel_frequency v++` 指令中所使用的参考。

```
Default Clock Index: 0
Clock Index:         1
  Frequency:         500.000000
Clock Index:         0
  Frequency:         300.000000
```

有效的 SLR

报告平台中有效的 SLR。

```
SLR0, SLR1, SLR2
```

资源可用性

在报告中不仅包含可用资源总量，还包含每个 SLR 的可用资源量。此信息可用于评估平台是否适用于设计，并且有助于指导用户在可用 SLR 上分配计算单元。

```
=====
Total
=====
  LUTs:  1047139
  FFs:   2186064
  BRAMs: 1896
  DSPs:  6833

=======
Per SLR
=======
  SLR0:
    LUTs:  354690
```

```
      FFs:    723308
      BRAMs:  638
      DSPs:   2265
    SLR1:
      LUTs:   159739
      FFs:    331654
      BRAMs:  326
      DSPs:   1317
    SLR2:
      LUTs:   354839
      FFs:    723294
      BRAMs:  638
      DSPs:   2265
```

存储器信息

报告每个 SLR 可用的 DDR 和 PLRAM 存储器连接，如以下输出示例所示。

```
  Type: ddr4
  Bus SP Tag: DDR
    Segment Index: 0
      Consumption: automatic
      SP Tag:      bank0
      SLR:         SLR0
      Max Masters: 15
    Segment Index: 1
      Consumption: default
      SP Tag:      bank1
      SLR:         SLR1
      Max Masters: 15
    Segment Index: 2
      Consumption: automatic
      SP Tag:      bank2
      SLR:         SLR1
      Max Masters: 15
    Segment Index: 3
      Consumption: automatic
      SP Tag:      bank3
      SLR:         SLR2
      Max Masters: 15
  Bus SP Tag: PLRAM
    Segment Index: 0
      Consumption: explicit
      SLR:         SLR0
      Max Masters: 15
    Segment Index: 1
      Consumption: explicit
      SLR:         SLR1
      Max Masters: 15
    Segment Index: 2
      Consumption: explicit
      SLR:         SLR2
      Max Masters: 15
```

`Bus SP Tag` 头可以是 DDR 或 PLRAM，并可提供以下关联信息。

Segment Index 字段与 SP Tag 搭配使用，以生成关联的存储器资源索引，如下所示。

```
Bus SP Tag[Segment Index]
```

例如，如果 Segment Index 为 0，则关联 DDR 资源索引为 DDR[0]。

此存储器索引在 v++ 命令中指定存储器资源时使用，如下所示：

```
v++ ... --connectivity.sp vadd.m_axi_gmem:DDR[3]
```

每个 SLR 都可能存在多个关联的 Segment Index。例如，在以上输出中，SLR1 具有 Segment Index 1 和 2。

Consumption 字段指示构建设计时所用的存储器资源量。

- default：如不指定 --connectivity.sp 指令，那么默认它会在 v++ 构建期间使用此存储器资源。例如，在以下报告中，默认使用含 Segment Index 1 的 DDR。
- automatic：如果在 Consumption: default 下已使用的存储器接口数量已达最大值，那么将使用 automatic 下的接口。在“Max Masters”字段下，提供了每个存储器资源的最大接口数。
- explicit：对于 PLRAM，“consumption”设为 explicit，表示当通过 v++ 命令行显式指示仅限使用此存储器资源时，才会使用此资源。

功能特性 ROM 信息

功能特性 ROM 信息可提供 ROM 平台上的构建相关信息，赛灵思技术支持人员在调试系统问题时可能要求提供此信息。

```
  ROM Major Version:        10
  ROM Minor Version:        1
  ROM Vivado Build ID:      2388429
  ROM DDR Channel Count:    5
  ROM DDR Channel Size:     16
  ROM Feature Bit Map:      655885
  ROM UUID:                 00194bb3-707b-49c4-911e-a66899000b6b
  ROM CDMA Base Address 0: 620756992
  ROM CDMA Base Address 1: 0
  ROM CDMA Base Address 2: 0
  ROM CDMA Base Address 3: 0
```

软件平台信息

虽然报告包含软件平台信息，但此信息仅适用于在器件上运行操作系统的用户，而不适用于使用主机的用户。

```
Number of Runtimes:             1
Default System Configuration:   config0_0
System Configurations:
  System Config Name:                          config0_0
  System Config Description:                   config0_0 Linux OS on x86_0
  System Config Default Processor Group:       x86_0
  System Config Default Boot Image:
```

```
  System Config Is QEMU Supported:          0
  System Config Processor Groups:
    Processor Group Name:       x86_0
    Processor Group CPU Type:   x86
    Processor Group OS Name:    Linux OS
  System Config Boot Images:
Supported Runtimes:
  Runtime: OpenCL
```

xilinx_zcu104_base_202010_1 的平台信息

以下命令可用于返回 xilinx_zcu104_base_202010_1 平台的平台信息 (platforminfo)：

```
platforminfo -p xilinx_zcu104_base_202010_1
```

返回的结果如下所示：

```
==========================
Basic Platform Information
==========================
Platform:           xilinx_zcu104_base_202010_1
File:               /platforms/xilinx_zcu104_base_202010_1/
xilinx_zcu104_base_202010_1.xpfm
Description:
A basic static platform targeting the ZCU104 evaluation board, which
includes 2GB DDR4, GEM, USB, SDIO interface and UART of the Processing
System. It reserves most of the PL resources for user to add acceleration
kernels


====================================
Hardware Platform (Shell) Information
====================================
Vendor:                             xilinx
Board:                              zcu104_base
Name:                               zcu104_base
Version:                            1.0
Generated Version:                  2020.1
Software Emulation:                 1
Hardware Emulation:                 0
FPGA Family:                        zynquplus
FPGA Device:                        xczu7ev
Board Vendor:                       xilinx.com
Board Name:                         xilinx.com:zcu104:1.1
Board Part:                         xczu7ev-ffvc1156-2-e
Maximum Number of Compute Units:    60

=================
Clock Information
=================
  Default Clock Index: 0
  Clock Index:         0
    Frequency:         150.000000
  Clock Index:         1
    Frequency:         300.000000
  Clock Index:         2
```

```
    Frequency:          75.000000
  Clock Index:          3
    Frequency:          100.000000
  Clock Index:          4
    Frequency:          200.000000
  Clock Index:          5
    Frequency:          400.000000
  Clock Index:          6
    Frequency:          600.000000

==================
Memory Information
==================
  Bus SP Tag: HP0
  Bus SP Tag: HP1
  Bus SP Tag: HP2
  Bus SP Tag: HP3
  Bus SP Tag: HPC0
  Bus SP Tag: HPC1
=======================
Feature ROM Information
=======================

============================
Software Platform Information
============================
Number of Runtimes:             1
Default System Configuration:   xilinx_zcu104_base_202010_1
System Configurations:
  System Config Name:                       xilinx_zcu104_base_202010_1
  System Config Description:                xilinx_zcu104_base_202010_1
  System Config Default Processor Group:    xrt
  System Config Default Boot Image:         standard
  System Config Is QEMU Supported:          1
  System Config Processor Groups:
    Processor Group Name:       xrt
    Processor Group CPU Type:   cortex-a53
    Processor Group OS Name:    linux
  System Config Boot Images:
    Boot Image Name:            standard
    Boot Image Type:
    Boot Image BIF:             xilinx_zcu104_base_202010_1/boot/linux.bif
    Boot Image Data:            xilinx_zcu104_base_202010_1/xrt/image
    Boot Image Boot Mode:       sd
    Boot Image RootFileSystem:
    Boot Image Mount Path:      /mnt
    Boot Image Read Me:         xilinx_zcu104_base_202010_1/boot/
generic.readme
    Boot Image QEMU Args:       xilinx_zcu104_base_202010_1/qemu/
pmu_args.txt:xilinx_zcu104_base_202010_1/qemu/qemu_args.txt
    Boot Image QEMU Boot:
    Boot Image QEMU Dev Tree:
Supported Runtimes:
  Runtime: OpenCL
```

xbutil 实用工具

本赛灵思开发板实用工具 (`xbutil`) 是一种独立的命令行实用工具，包含在赛灵思的 Xilinx Runtime (XRT) 安装包内。在卡上需安装并识别 XRT。如需了解 `xbutil` 命令的详细信息，请参阅 https://xilinx.github.io/XRT/2021.1/html/xbutil2.html。

`xbutil` 包含多项命令，用于确认和识别已安装的加速器卡，以及有关这些卡的其它详细信息，包括卡存储器、主机接口、目标平台名称和系统信息等。这些信息可用于卡管理和应用调试。加速器卡划分为用户函数和管理函数，以提供不同的卡访问级别。用户函数允许您加载并运行应用，而管理函数则供系统管理员用于对卡进行管理。`xbutil` 实用工具能够与用户函数进行交互。`xbmgmt` 实用工具则需要根用户权限，用于与管理函数进行交互。将函数访问拆分为两种实用工具的原因是为了给该工具的管理功能特性提供一定程度的安全性。

您可使用 `help` 命令来列出可用的 `xbutil` 命令和选项：

```
xbutil help
```

在 XRT 安装过程中，使用以下脚本设置 `xbutil` 命令：

- 对于 csh shell：

  ```
  $ source /opt/xilinx/xrt/setup.csh
  ```

- 对于 bash shell：

  ```
  $ source /opt/xilinx/xrt/setup.sh
  ```

提示：如需获取与旧 `xbutil` 命令相关的参考资料，请参阅 xbutil 实用工具 - 旧版本。

xbmgmt 实用工具

赛灵思开发板管理 (xbmgmt) 实用工具是一个独立命令行工具，随附于赛灵思的 Xilinx Runtime (XRT) 安装包内。在卡上需安装并识别 XRT。如需了解 xbmgmt 命令的详细信息，请参阅 https://xilinx.github.io/XRT/2021.1/html/xbmgmt2.html。

加速器卡划分为用户函数和管理函数，以提供不同的卡访问级别。用户函数允许您加载并运行应用，而管理函数则供系统管理员用于对卡进行管理。xbutil 实用工具能够与用户函数进行交互。xbmgmt 实用工具则需要根用户权限，用于与管理函数进行交互。将函数访问拆分为两种实用工具的原因是为了给该工具的管理功能特性提供一定程度的安全性。

xbmgmt 命令用于卡的安装和管理，并且运行时需要 sudo 权限。xbmgmt 支持的任务包括烧写卡固件和扫描当前器件配置。

您可以使用 help 命令来列出可用的 xbmgmt 命令和选项，并使用以下命令来访问各项命令的帮助信息：

```
xbmgmt help <command>
```

如需获取各条子命令的详细帮助信息，请使用以下命令：

```
xbmgmt help <subcommand>
```

在 XRT 安装过程中，使用以下脚本设置 xbmgmt 命令：

- 对于 csh shell：

  ```
  $ source /opt/xilinx/xrt/setup.csh
  ```

- 对于 bash shell：

  ```
  $ source /opt/xilinx/xrt/setup.sh
  ```

提示：如需获取与旧 xbutil 命令相关的参考资料，请参阅 xbmgmt 实用工具 - 旧版本。

xclbinutil 实用工具

xclbinutil 实用工具可以创建、修改和报告 xclbin 内容信息。

可用命令如下图所示。

表 52：**xclbinutil 命令**

选项	描述
-h [--help]	打印帮助消息。
-i [--input]<arg>	输入文件名。将 xclbin 读入存储器。
-o [--output]<arg>	输出文件名。将存储器 xclbin 镜像写入文件。
--target <arg>	此镜像的目标流程。有效值包括：hw、hw_emu 和 sw_emu。
----private-key <arg>	签署 xclbin 镜像时使用的专用密钥。
--ceritifcate <arg>	用于签署和确认 xclbin 镜像的证书。
--digest-algorithm <arg>	摘要算法。默认值：sha512
--validate-signature	确认给定 xclbin 存档的特征符。
-v [--verbose]	显示详细的调试信息
-q [--quiet]	最大程度减少报告信息。
--migrate-forward	将 xclbin 存档迁移至新的二进制格式。
--add-section <arg>	要添加到 xclbin 镜像中的节的名称。格式：<section>:<format>:<file>
--add-replace-section <arg>	替换现有的节，或者添加 xclbin 镜像的节（如果不存在）。格式：<section>:<format>:<file>
--add-merge-section <arg>	添加节（如果不存在）或者将内容与现有的节合并。格式：<section>:<format>:<file>
--remove-section<arg>	要从 xclbin 镜像中移除的节的名称。
--dump-section<arg>	要转储的节。格式：<section>:<format>:<file>
--replace-section<arg>	要替换的节。
--key-value<arg>	键值对。格式：[USER|SYS]:<key>:<value>
--remove-key<arg>	从 xclbin 存档移除给定的用户密钥。
--add-signature<arg>	将用户定义的特征符添加到给定的 xclbin 镜像。
--remove-signature	从 xclbin 镜像中移除特征符。
--get-signature	返回 xclbin 镜像中的用户定义的特征符（如已设置）。
--info	报告加速器二进制内容。包括：生成和封装数据、内核特征符、连接、时钟、节等。
--list-sections	列出所有可能的节名称（独立选项）。
--version	此可执行文件的版本。
--force	强制执行文件覆盖。

以下为该工具的各种用例。

- 报告 xclbin 信息：`xclbinutil --info --input binary_container_1.xclbin`
- 抽取比特流镜像：`xclbinutil --dump-section BITSTREAM:RAW:bitstream.bit --input binary_container_1.xclbin`
- 抽取构建元数据：`xclbinutil --dump-section BUILD_METADATA:HTML:buildMetadata.json --input binary_container_1.xclbin`
- 移除节：`xclbinutil --remove-section BITSTREAM --input binary_container_1.xclbin --output binary_container_modified.xclbin`

对于大部分用户，都需要有关内容以及创建 `xclbin` 的方式的信息。此信息可通过 `--info` 选项获取，可用于报告有关 `xclbin`、硬件平台、时钟、存储器配置、内核与 `xclbin` 生成方式的信息。

以下分多个部分显示了使用 `--info` 选项时 `xclbinutil` 命令的输出。

```
xclbinutil -i binary_container_1.xclbin --info
```

xclbin 信息

```
   Generated by:            v++ (2020.1) on Mon Apr 13 20:19:40 MDT 2020
   Version:                 2.6.436
   Kernels:                 CopyKernel
   Signature:
   Content:                 Bitstream
   UUID (xclbin):           d081de98-3fd3-4e9b-bab3-108b42c73101
   UUID (IINTF):            862c7020a250293e32036f19956669e5
   Sections:                DEBUG_IP_LAYOUT, BITSTREAM, MEM_TOPOLOGY,
IP_LAYOUT,
                            CONNECTIVITY, CLOCK_FREQ_TOPOLOGY,
BUILD_METADATA,
                            EMBEDDED_METADATA, SYSTEM_METADATA,
PARTITION_METADATA
```

硬件平台信息

```
Vendor:                 xilinx
Board:                  u200
Name:                   xdma
Version:                201830.1
Generated Version:      Vivado 2018.3 (SW Build: 2388429)
Created:                Wed Nov 14 20:06:10 2018
FPGA Device:            xcu200
Board Vendor:           xilinx.com
Board Name:             xilinx.com:au200:1.0
Board Part:             xilinx.com:au200:part0:1.0
Platform VBNV:          xilinx_u200_xdma_201830_1
Static UUID:            00194bb3-707b-49c4-911e-a66899000b6b
Feature ROM TimeStamp:  1542252769
```

时钟

报告可用的最大内核时钟频率。其中提供了时钟名称和时钟索引。此处时钟索引与 platforminfo 实用工具 中报告的时钟索引相同。

```
Name:       DATA_CLK
Index:      0
Type:       DATA
Frequency:  300 MHz

Name:       KERNEL_CLK
Index:      1
Type:       KERNEL
Frequency:  500 MHz
```

存储器配置

```
Name:          bank0
Index:         0
Type:          MEM_DDR4
Base Address:  0x0
Address Size:  0x400000000
Bank Used:     No

Name:          bank1
Index:         1
Type:          MEM_DDR4
Base Address:  0x400000000
Address Size:  0x400000000
Bank Used:     Yes

Name:          bank2
Index:         2
Type:          MEM_DDR4
Base Address:  0x800000000
Address Size:  0x400000000
Bank Used:     No

Name:          bank3
Index:         3
Type:          MEM_DDR4
Base Address:  0xc00000000
Address Size:  0x400000000
Bank Used:     No

Name:          PLRAM[0]
Index:         4
Type:          MEM_DDR4
Base Address:  0x1000000000
Address Size:  0x20000
Bank Used:     No

Name:          PLRAM[1]
Index:         5
Type:          MEM_DRAM
```

```
Base Address: 0x1000020000
Address Size: 0x20000
Bank Used:    No

Name:         PLRAM[2]
Index:        6
Type:         MEM_DRAM
Base Address: 0x1000040000
Address Size: 0x20000
Bank Used:    No
```

内核信息

对于 `xclbin` 内的每个内核，函数定义、端口和实例信息都将包含在报告内。

以下提供了报告的函数定义示例。

```
Definition
----------
   Signature: krnl_vadd (int* a, int* b, int* c,
              int const  n_elements)
```

以下提供了报告的端口示例。

```
Ports
-----
   Port:          M_AXI_GMEM
   Mode:          master
   Range (bytes): 0xFFFFFFFF
   Data Width:    32 bits
   Port Type:     addressable

   Port:          M_AXI_GMEM1
   Mode:          master
   Range (bytes): 0xFFFFFFFF
   Data Width:    32 bits
   Port Type:     addressable

   Port:          S_AXI_CONTROL
   Mode:          slave
   Range (bytes): 0x1000
   Data Width:    32 bits
   Port Type:     addressable
```

以下提供了报告的实例示例。

```
Instance:        krnl_vadd_1
   Base Address: 0x0

   Argument:          a
   Register Offset:   0x10
   Port:              M_AXI_GMEM
   Memory:            bank1 (MEM_DDR4)

   Argument:          b
   Register Offset:   0x1C
```

```
    Port:               M_AXI_GMEM
    Memory:             bank1 (MEM_DDR4)

    Argument:           c
    Register Offset:    0x28
    Port:               M_AXI_GMEM1
    Memory:             bank1 (MEM_DDR4)

    Argument:           n_elements
    Register Offset:    0x34
    Port:               S_AXI_CONTROL
    Memory:             <not applicable>
```

工具生成信息

该实用工具还可报告用于生成 `xclbin` 的 `v++` 命令行。“命令行 (Command Line)”部分提供了实际使用的 `v++` 命令行，而“选项 (Options)”部分则以更便于阅读的格式显示了命令行中所使用的每个选项，即每行显示一个选项。

```
Generated By
------------
Command:        v++
Version:        2018.3 - Tue Nov 20 19:42:42 MST 2018 (SW BUILD: 2394611)
Command Line:   v++ -t hw_emu --platform /opt/xilinx/platforms/
xilinx_u200_xdma_201830_1/xilinx_
                u200_xdma_201830_1.xpfm --save-temps -l --connectivity.nk
krnl_vadd:1
                -g --messageDb binary_container_1.mdb
                --temp_dir binary_container_1
                --report_dir binary_container_1/reports --log_dir
binary_container_1/logs
                   --remote_ip_cache /wrk/tutorials/ip_cache
                   -obinary_container_1.xclbin binary_container_1/
krnl_vadd.o
Options:        -t hw_emu
                --platform /opt/xilinx/platforms/xilinx_u200_xdma_201830_1/
xilinx_u200_xdma_201830_1.xpfm
                --save-temps
                -l
                --connectivity.nk krnl_vadd:1
                -g
                --messageDb binary_container_1.mdb
                --temp_dir binary_container_1
                --report_dir binary_container_1/reports
                --log_dir binary_container_1/logs
                --remote_ip_cache /wrk/tutorials/ip_cache
             -obinary_container_1.xclbin binary_container_1/krnl_vadd.o
================================================================================
==
User Added Key Value Pairs
---------------------------
   <empty>
================================================================================
==
```

xrt.ini 文件

赛灵思的 Xilinx Runtime (XRT) 库在运行主机应用和内核执行时，使用各种控制参数来指定调试、剖析和消息记录。这些控制参数在运行时初始化文件 `xrt.ini` 中指定，并用于在启动时配置 XRT 的功能。

如果您是命令行用户，则需手动创建 `xrt.ini` 文件并将其保存至主机可执行文件所在的目录。该运行时库会检查主机可执行文件中是否存在 `xrt.ini` 并自动读取该文件来配置运行时。

提示：Vitis IDE 会根据运行配置自动创建 `xrt.ini` 文件，并将其与主机可执行文件保存在一起。

运行时初始化文件格式

`xrt.ini` 文件是简单的文本文件，其中包含多组密钥和密钥值。以分号 (;) 或井号 (#) 开头的任何行都是注释。组名、密钥和密钥值都区分大小写。

以下是 `xrt.ini` 文件示例，此文件支持时间线轨迹功能，并且指示运行时日志消息显示在“Console”视图上。

```
#Start of Debug group
[Debug]
opencl_trace = true

#Start of Runtime group
[Runtime]
runtime_log = console
```

其中有 3 组初始化密钥：

- 运行时
- 调试
- 仿真

下表列出了每个组的所有受支持的密钥、每个密钥支持的值以及密钥用途的简短描述。

表 53：**运行时组**

密钥	有效值	描述
`api_checks`	`[true|false]`	启用或禁用 OpenCL API 检查。 · `true`：启用。这是默认值。 · `false`：禁用。
`cpu_affinity`	`{N,N,...}`	将所有运行时线程固定到指定的 CPU。示例： `cpu_affinity = {4,5,6}`
`exclusive_cu_context`	`[true|false]`	这样即可允许主机应用指示 OpenCL 获取专属 CU 访问，以便低层次 AXI 读写（xclRegRead 和 xclRegWrite）可供常规内核使用。

表 53：**运行时组** (续)

密钥	有效值	描述
runtime_log	[null | console | syslog | <filename>]	指定运行时日志的打印位置 • null：不打印任何日志。这是默认值。 • console：将日志打印至 stdout • syslog：将日志打印至 Linux syslog。 • <filename>：将日志打印至指定文件。例如，runtime_log=my_run.log。
verbosity	[0 | 1 | 2 | 3]	日志文件的详细程度。默认值为 0。

表 54：**调试组**

密钥	有效值	描述
aie_profile	[true|false]	启用 AI 引擎硬件性能计数器的运行时配置和轮询。仅在运行 VCK190 硬件时才可用。 • true：启用。 • false：禁用。这是默认值。
aie_profile_core_metrics	[heat_map|stalls|execution]	控制从 AI 引擎核性能计数器读取的统计数据的配置。默认值为 heat_map。 **注释：**仅当 aie_profile = true 时，此开关才有效。
aie_profile_interval_us	<int>	控制读取 AI 引擎计数器值的时间间隔（以微秒 (µs) 为单位）。默认时间间隔为 1000 µs。 **注释：**仅当 aie_profile = true 时，此开关才有效。
aie_profile_memory_metrics	[dma_locks|conflicts]	控制从 AI 引擎存储器性能计数器读取的统计数据的配置。默认值为 dma_locks。 **注释：**仅当 aie_profile = true 时，此开关才有效。
aie_trace	[true|false]	启用 AI 引擎事件追踪的运行时配置和收集。仅在运行 VCK190 硬件时才可用。 • true：启用。 • false：禁用。这是默认值。
aie_trace_buffer_size	<string>	控制为 AI 引擎事件追踪分配的缓冲器的总大小。此大小最终将根据 AI 引擎输出的不同追踪数据流的数量进行分区。默认值为 8M。 **注释：**仅当 aie_trace = true 时，此开关才有效。
aie_trace_metrics	[functions| functions_partial_stalls| function_all_stalls|all]	控制 AI 引擎寄存器的配置，以生成指定级别的事件追踪。 **注释：**仅当 aie_trace = true 时，此开关才有效。
app_debug	[true|false]	启用 OpenCL 应用调试。 • true：启用。 • false：禁用。这是默认值。

表 54：**调试组** (续)

密钥	有效值	描述
continuous_trace	[true|false]	启用将器件数据的追踪和连续读取对应的文件连续转储到主机中。 • true：启用。 • false：禁用。这是默认值。 **注释：**仅当 data_transfer_trace = off 时，此开关才有效。
data_transfer_trace	[off|fine|coarse|accel]	启用从 PL 上插入的监控器收集数据，以添加至汇总数据和轨迹。 • accel：追踪计算单元启动/停止。 • coarse：在计算单元的每次执行下归并所有读写。 • fine：追踪发生的一切。 • off：关闭运行时期间器件级别追踪的读取和报告。这是默认值。
debug	[true|false]	为软件仿真 (software emulation) 启用内核调试功能。 • true：启用。 • false：禁用。这是默认值。
lop_trace	[true|false]	启用低开销 OpenCL API 主机追踪的生成。不应配合其它 OpenCL 选项一起使用。 • true：启用。 • false：禁用。这是默认值。
native_xrt_trace	[true|false]	启用原生 C/C++ API 追踪的生成。 • true：启用。 • false：禁用。这是默认值。
opencl_device_counter	[true|false]	将硬件计数器信息添加到生成的 OpenCL 汇总文件。如果指定 data_transfer_trace，则不需要该选项。 • true：启用。 • false：禁用。这是默认值。
opencl_summary	[true|false]	启用 OpenCL API 汇总报告的生成。如果这是指定的唯一标记，则仅包含主机信息。 • true：启用。 • false：禁用。这是默认值。
opencl_trace	[true|false]	启用 OpenCL API 主机追踪的生成。 • true：启用。 • false：禁用。这是默认值。
power_profile	[true|false]	在应用执行期间，启用功耗数据的轮询。 • true：启用。 • false：禁用。这是默认值。
power_profile_interval_ms	<int>	控制读取功耗计数器的时间间隔（以毫秒为单位）。默认时间间隔为 20 ms。 **注释：**仅当 power_profile = true 时，此开关才有效。

表 54：**调试组** (续)

密钥	有效值	描述
stall_trace	[on|off]	启用硬件在计算单元中生成停滞。默认值为 off。 **注释：** 仅当 data_transfer_trace = off 时，此开关才有效。
trace_buffer_offload_interval_ms	<int>	控制从器件到主机的器件数据读取（以毫秒 (ms) 为单位）。默认值为 10 ms。 **注释：** 仅当 data_transfer_trace = off 时，此开关才有效。
trace_buffer_size	<string>	如果创建的 .xclbin 中已指定追踪的存储器卸载，则此开关可确定在存储器中要为追踪分配的缓冲器大小。默认值为 1M。 **注释：** 仅当 data_transfer_trace = off 时，此开关才有效。
trace_file_dump_interval_s	<int>	控制追踪文件转储的时间间隔（以秒 (s) 为单位）。默认值为 5s。 **注释：** 仅当 data_transfer_trace = off 时，此开关才有效。
vitis_ai_profile	[true|false]	剖析汇总和其它文件来自 Vitis AI 应用层。 • true：启用。 • false：禁用。这是默认值。

表 55：**仿真組**

密钥	有效值	描述
aliveness_message_interval	任何整数	指定需要打印活动消息的时间间隔（以秒为单位）。默认值为 300。
debug_mode	[off|batch|gui]	指定在仿真 (emulation) 期间如何保存和显示波形。 • off：不启动仿真器波形 GUI，且不保存 wdb 文件。这是默认值。 • batch：不启动仿真器波形 GUI，但保存 wdb 文件。 • gui：启动仿真器波形 GUI，并保存 wdb 文件 **注释：** 内核需要在调试启用 (v++ -g) 的情况下进行编译，以便保存波形并在仿真器 GUI 中显示。
enable_memory_persistence	[true|false]	为平台中的 DDR 存储器启用存储器持久性或自动刷新支持。在挂起/暖复位期间，保留存储器的内容。 • true：启用存储器持久性 • false：禁用存储器持久性（默认设置）

表 55：**仿真組** (续)

密钥	有效值	描述
print_infos_in_console	[true|false]	控制将仿真 (emulation) 参考消息打印到用户控制台的行为。仿真参考消息始终记录到名为 emulation_debug.log 的文件中 · true：在用户控制台中打印。这是默认值。 · false：不在用户控制台中打印。
print_warnings_in_console	[true|false]	控制将仿真 (emulation) 警告消息打印到用户控制台的行为。仿真警告消息始终记录到名为 emulation_debug.log 的文件中。 · true：在用户控制台中打印。这是默认值。 · false：不在用户控制台中打印。
print_errors_in_console	[true|false]	控制在用户控制台中打印仿真错误消息的行为。仿真错误消息始终记录到 emulation_debug.log 文件中。 · true：在用户控制台中打印。这是默认值。 · false：不在用户控制台中打印。
user_pre_sim_script	Tcl 文件路径	首次运行时，以 GUI 模式运行仿真 (simulation)。添加要添加的信号。将命令从 Tcl 控制台复制并保存到 Tcl 脚本中。 对于下一次运行，以批处理模式传递此 Tcl 脚本。
user_post_sim_script	Tcl 文件路径	任何转发操作均可在 Tcl 中指定并传递给开关。Tcl 中提供的所有命令都在仿真 (simulation) 完成后执行。
xtlm_aximm_log	[true|false]	在运行时启用 XTLM AXI4 存储器映射传输事务日志记录，您可在 xsc_report.log 文件中看到所有传输事务。
xtlm_axis_log	[true|false]	在运行时启用 XTLM AXI4-Stream 传输事务日志记录，您可在 xsc_report.log 文件中看到所有传输事务。
timeout_scale	na/ms/sec/min	针对仿真 (emulation) 中的 clPollStream API 的超时支持。为 clPollStream API 中指定的超时提供刻度。代码中指定的超时是以毫秒 (ms) 为单位来指定的，可能对于仿真 (emulation) 无效。因此，请使用 timeout_scale 将 ms 映射到其它刻度，以满足仿真 (emulation) 所需。 **重要提示！**默认情况下，在仿真 (emulation) 中不启用超时。该选项可用于启用 clPollStream 超时。

HLS 编译指示

Vitis HLS 中的最优化

在 Vitis 软件平台中，以 C/C++ 语言或 OpenCL™ C 语言定义的内核必须编译到寄存器传输级 (RTL)，并且此 RTL 必须可实现到赛灵思器件的可编程逻辑内。`v++` 编译器通过调用 Vitis 高层次综合 (HLS) 工具来实现来自内核源代码的 RTL 代码。

HLS 工具旨在以无交互方式处理 Vitis IDE 工程。但是，HLS 工具还提供各种编译指示，用于最优化设计、降低时延、提升吞吐量性能，以及减少生成的 RTL 代码的面积和器件资源利用率。这些编译指示可直接添加到内核源代码中。

HLS 编译指示可包含下表中指定的最优化类型。

如需了解有关编译指示的详细信息，请参阅 Vitis HLS 流程。

表 56：**Vitis HLS 编译指示（按类型）**

类型	属性
内核最优化	· pragma HLS aggregate · pragma HLS bind_op · pragma HLS bind_storage · pragma HLS expression_balance · pragma HLS latency · pragma HLS reset · pragma HLS top
函数内联	· pragma HLS inline
接口综合	· pragma HLS interface
任务级流水线	· pragma HLS dataflow · pragma HLS shared · pragma HLS stream
流水线	· pragma HLS pipeline · pragma HLS occurence
循环展开	· pragma HLS unroll · pragma HLS dependence
循环最优化	· pragma HLS loop_flatten · pragma HLS loop_merge · pragma HLS loop_tripcount
阵列最优化	· pragma HLS array_partition · pragma HLS array_reshape
结构封装	· pragma HLS aggregate · pragma HLS dataflow

第六部分

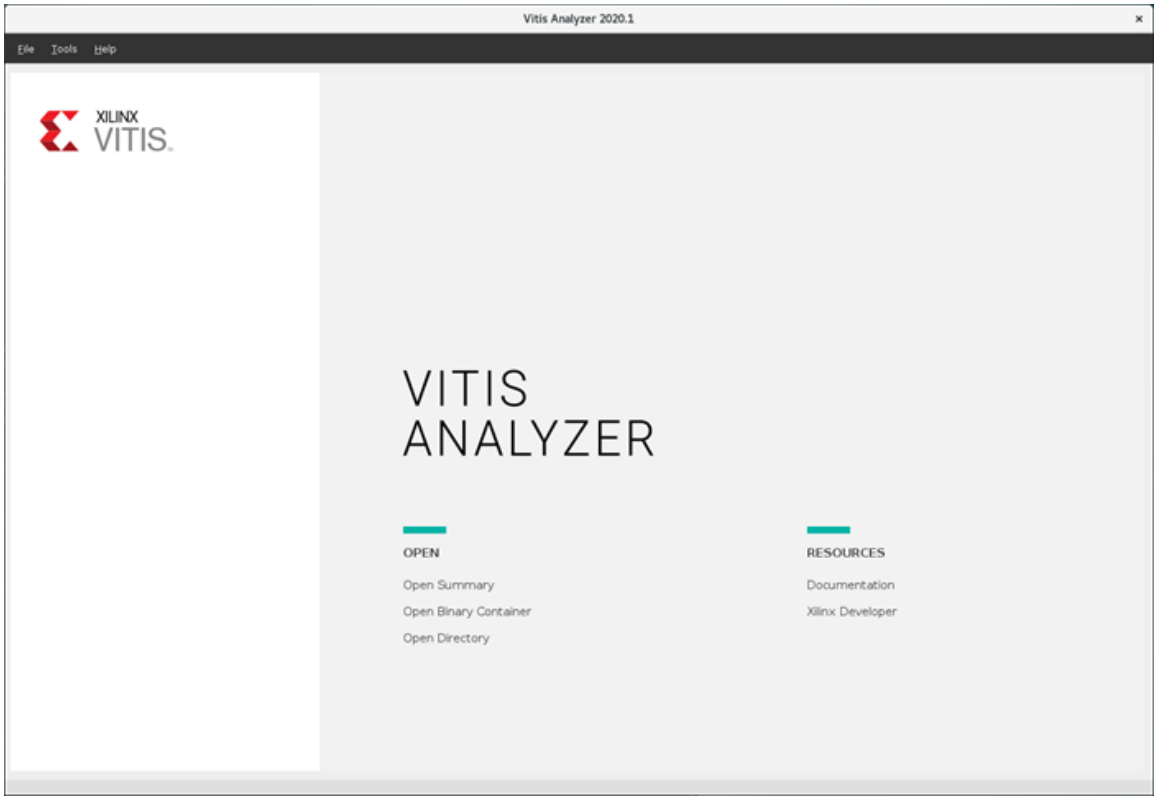
使用 Vitis 分析器

本部分包含以下章节：

Vitis™ 分析器实用工具支持您在构建和运行应用的同时查看并分析报告。它旨在允许您查看在构建应用时由 Vitis 编译器生成的报告以及在运行应用时由赛灵思的 Xilinx Runtime (XRT) 库生成的报告。Vitis 分析器可用于查看来自 `v++` 命令行的报告和来自 Vitis 集成设计环境 (IDE) 的报告。您将使用 `vitis_analyzer` 命令启动该工具（请参阅 设置 Vitis 环境）。

首次启动时，Vitis 分析器会打开主页屏幕，您可在其中打开汇总文件、二进制容器或目录。单击其中任一链接即可打开文件浏览器以供您选择所描述的特定类型的文件。

图 78：**Vitis 分析器 - 主页屏幕**



- “Open Summary”：报告汇总是与 Vitis 工具中特定应用开发阶段相关的报告的集合。其中包含为构建进程的 2 个步骤（编译和链接）创建的报告汇总以及来自执行应用时的运行进程的汇总信息。选择“Open Summary”即可打开：
  - “Compile Summary”：“编译汇总 (Compile Summary)”报告是由 `v++` 命令在编译期间生成的，可提供内核编译进程的状态。查看“Compile Summary”报告时，该工具还会引用编译期间生成的下列报告：“内核估算 (Kernel Estimate)”、“内核指南 (Kernel Guidance)”、“HLS 综合 (HLS Synthesis)”和编译日志。
  - “Link Summary”：“链接汇总 (Link Summary)”报告是由 `v++` 命令在链接和创建 `.xclbin` 文件期间创建的。查看“Link Summary”报告时，该工具还会引用链接期间生成的以下报告：“系统估算 (System Estimate)”、“系统指南 (System Guidance)”、“时序汇总 (Timing Summary)”、“利用率 (Utilization)”、“操作追踪 (Operation Trace)”、“平台和系统框图 (Platform and System Diagrams)”以及链接日志。打开“Link Summary”时，Vitis 分析器将自动打开已链接到 `.xclbin` 文件中的内核的关联“Compile Summaries”。

    提示：仅在构建目标硬件（而不是仿真）时，才会生成“Timing Summary”和“Utilization”。

  - “Package Summary”：“封装汇总 (Package Summary)”报告是由 `v++ --package` 命令生成的，可提供与仿真脚本和 SD 卡输出的生成相关的信息。查看“Package Summary”报告时，该工具还会引用命令行上使用的配置文件和生成的日志文件。
  - “Run Summary”：“运行汇总 (Run Summary)”报告是由 XRT 库在应用执行期间创建的，可提供运行进程的汇总信息。查看“Run Summary”报告时，该工具还会引用运行应用期间生成的以下报告：“指南 (Guidance)”、“剖析汇总 (Profile Summary)”、“应用时间线 (Application Timeline)”、“平台和系统框图 (Platform and System Diagrams)”以及“仿真波形 (Simulation Waveforms)”（如果已启用）。

重要提示！应用的运行时执行期间生成的报告通常需要提前设置，如 剖析应用 中所述。

在 v++ 构建进程后运行应用时，来自“Link Summary”的 ID 将分配给“Run Summary”。打开“Run Summary”和“Link Summary”时，Vitis 分析器会基于共享 ID 来链接这些报告。

- “AI Engine Compile and Run Summaries”：Vitis 分析器还支持您查看来自 Versal AI 引擎编译和运行进程的报告。这些报告是由 aiecompiler 和 aiesimulator 生成的，如《Versal ACAP AI 引擎编程环境用户指南》(UG1076) 中所述。

提示：在 Vitis IDE 中，工程层级包含顶层系统工程，其中子工程包含设计元素：处理器应用、硬件内核、硬件链接工程和 AI 引擎工程。上述各种汇总报告均可在相应的子工程中找到：“Compile Summary”位于特定硬件内核工程中、“Link Summary”位于硬件链接工程中、“Package Summary”位于系统工程中、“Run Summary”位于处理器应用中，AI 引擎汇总信息位于 ./Work 文件夹或 ./aiesimulator 输出中。

- “Open Binary Container”：打开所选 .xclbin 文件即可显示构建的平台框图和系统框图。
- “Open Directory”：指定要打开的目录。该工具会以递归方式检验目录内容，并显示对话框以供您选择要打开的文件类型以及要打开的个别文件。

提示：主页屏幕上的“打开最近打开的报告 (Open Recent)”部分提供了最近打开的汇总和报告的列表，以便您快速重新打开。

vitis_analyzer 命令允许您打开工具并显示主页屏幕（如上所述），或者指定打开工具时要加载的文件。您可通过指定要打开的文件的名称来打开此文件。您可通过 Vitis 分析器支持的任意文件来打开此工具，如 使用报告 中所述。例如：

```
vitis_analyzer project1.run_summary
```

您可通过输入以下命令来访问 vitis_analyzer 命令的命令帮助：

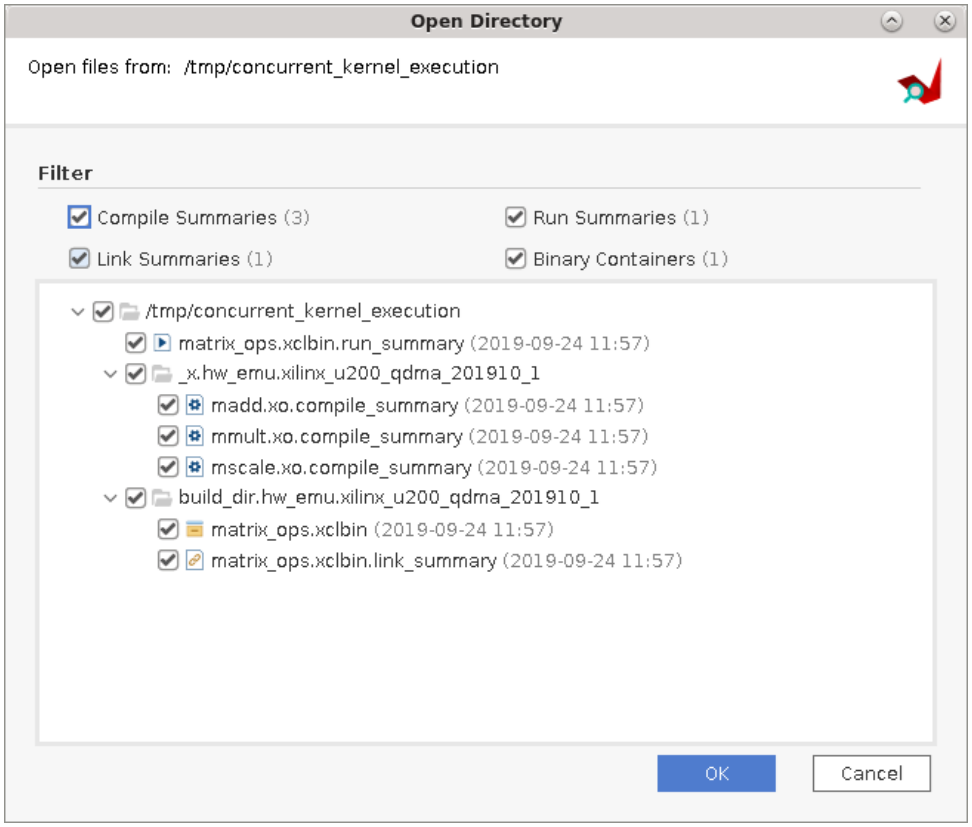
```
vitis_analyzer -help
```

使用报告

通常，汇总报告为您提供了应用构建和剖析过程中特定步骤的综述，以便您了解应用在性能和最优化方面的表现。要查看个别内核的汇总信息，请参阅“编译汇总 (Compile Summary)”。对于器件二进制文件 (`xclbin`)，请从“链接汇总 (Link Summary)”开始，此报告还会加载已链接的内核的“Compile Summary”。对于嵌入式处理器和 Versal™ AI 引擎系统，请复查“封装汇总 (Package Summary)”。要了解与应用执行有关的剖析数据，请从“运行汇总 (Run Summary)”开始。

此外，“File”菜单可提供命令以供您打开各报告和报告目录。

图 79：**打开目录**



- “Open Directory”：指定要打开的目录。该工具会以递归方式检验目录内容，并显示对话框以供您选择要打开的文件类型以及要打开的个别文件。
- “Open Binary Container”：如需了解有关编译和链接进程所创建的 FPGA 二进制文件 `<name>.xclbin` 的描述，请参阅 第三部分：构建和运行应用。
- “Open Report”：打开 Vitis™ 核开发套件在编译、链接或运行应用期间生成的任一报告文件。可打开的报告包括：

- “Application Timeline”：请参阅 时间线轨迹。
- “Profile Summary”：请参阅 剖析汇总报告。
- “Waveform”：如需了解有关波形数据库和波形配置文件的描述，请参阅 波形视图和实时波形查看器。
- “Utilization”：资源使用情况报告，由 Vivado® 工具在构建系统硬件 (HW) 目标时生成。

Vitis 分析器还可打开“内核估算 (Kernel Estimate)”、“操作追踪 (Operation Trace)”、“AI 引擎追踪 (AI Engine Trace)”、“时间线轨迹 (Timeline Trace)”、“系统估算 (System Estimate)”、“日志 (Log)”和“时序汇总 (Timing Summary)”报告。请参阅 剖析应用 以获取有关构建和运行进程生成的各报告的更多信息。

Vitis 分析器所显示的日志文件，通过特殊呈现方式来改善可读性。查看日志文件时，该工具可提供的部分关键新增功能包括：换行、消息严重性标签（Error、Warning、Info）、添加参考文件的超链接、搜索功能以及实时日志监控。最后一项功能允许您打开正在进行中的日志文件，并实时查看呈现的文件。

查看报告内容

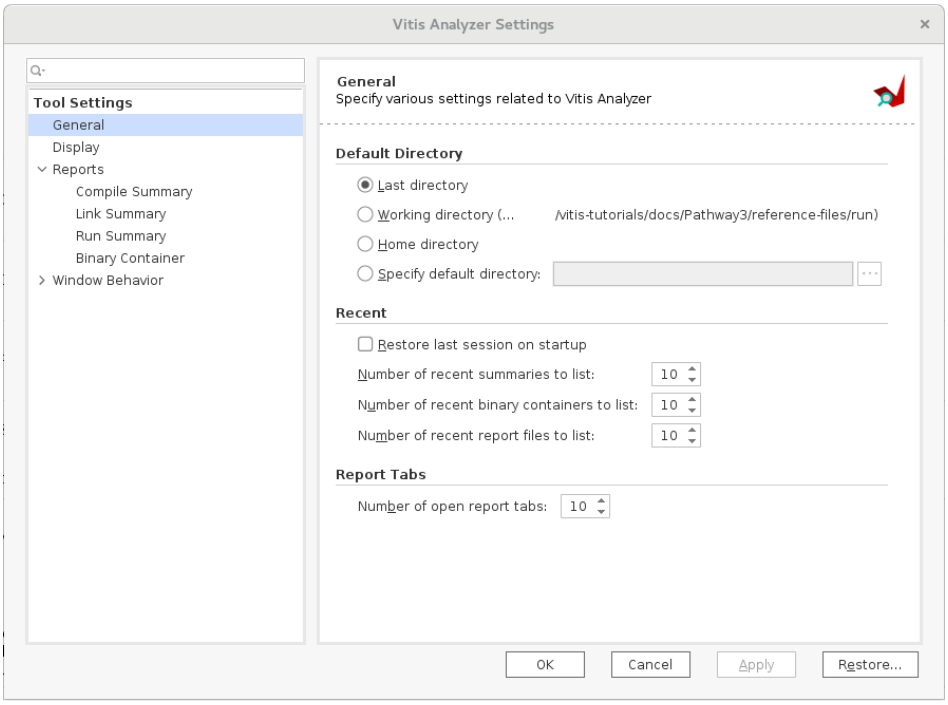
Vitis 分析器的功能特性取决于您正在查看的具体报告。当报告结构类似电子数据表时，您能够像处理电子数据表一样来与报告交互，如选择数据的行或单元格，以及单击列标题对列进行排序。当报告以图形方式呈现时，您可以通过方法报告查看细节、缩小以查看更多信息等方式来与报告进行交互。Vitis 分析器支持以下鼠标手势，允许您快速缩放图形报告。

- “Zoom In”：按住鼠标左键，将鼠标从左上角拖动到右下角以定义放大区域。
- “Zoom Out”：按住鼠标左键，绘制从左下角到右上角的对角线。这样即可缩小窗口，缩小的量可变。绘制的线长度可决定应用的缩放因子。或者，按住 “Ctrl” 键并向下滚动鼠标滚轮以缩小。
- “Zoom Fit”：按住鼠标左键，绘制从右下角到左上角的对角线。这样窗口即可缩小以显示整个器件。
- “Horizontal scrolling”：在“应用时间线 (Application Timeline)”之类的报告中，您可以按下 Shift 键并滚动鼠标滚轮以滚动时间线。
- “Panning”：按住鼠标滚轮并拖动以进行平移。

配置 Vitis 分析器

“Tools” → “Settings”命令可打开“Vitis Analyzer Settings”对话框，如下所示。

图 80：**设置对话框**



在“General”设置中，可配置以下选项：

- “Default Directory”：指定 Vitis™ 分析器打开时所使用的默认目录。
- “Recent”：配置该工具，以在重新打开 Vitis 分析器时复原工作空间，并指定要为“File” → “Open Recent”命令显示的条目数。
- “Report Tabs”：定义可在“报告 (Reports)”主窗口中打开的报告和视图数量。

在“显示 (Display)”设置中，您可配置以下显示功能特性：

- “Scaling”：设置字体缩放，使显示的内容在高分辨率显示器上更便于阅读。“使用操作系统字体缩放 (Use OS font scaling)”选项使用操作系统为您的主显示器所设置的值。“用户定义的缩放 (User-defined scaling)”允许您指定特定于 Vitis 分析器的值。
- “Spacing”：设置 Vitis IDE 所使用的空间量。默认设置为“舒适 (Comfortable)”。“紧凑 (Compact)”会减少元素间的空间量，以便在更小的空间内显示更多元素。

“Reports”部分还可将 Vitis 分析器配置为在打开“Compile Summary”、“Link Summary”、“Run Summary”或“Binary Container”报告时打开指定的报告：

- “Compile Summary”：选择在使用“Compile Summary”时，要在“报告导航器 (Report Navigator)”视图中列示并打开的报告。
- “Link Summary”：选择在使用“Link Summary”时，要列示并打开的报告。
- “Run Summary”：选择在使用“Run Summary”时，要列示并打开的报告。此外，该选项还可用于选择在运行应用时，要动态更新的报告。捕获连续追踪数据时，“Application Timeline”报告和“Timeline Trace”报告均可自动重新加载，如 连续追踪捕获 中所述。这样您即可在 Vitis 分析器中观测从运行的应用中捕获的追踪数据。
- “Binary Container”：选择在使用“Binary Container”时，要列示并打开的报告。

- “AI Engine Compile Summary”：选择在使用“AI Engine Compile Summary”时，要列示并打开的报告。
- “AI Engine Run Summary”：选择在使用“AI Engine Run Summary”时，要列示并打开的报告。

对于“Window Behavior”设置，可配置以下选项：

- “Warnings”：在退出该工具时或者仅退出 Vitis 分析器时，显示警告。
- “Alerts”：在不受支持的操作系统上运行该工具时，发出警报。

配置该工具后，请单击“OK”、“Apply”或“Cancel”。您也可以使用“Restore”命令来复原该工具的默认设置。

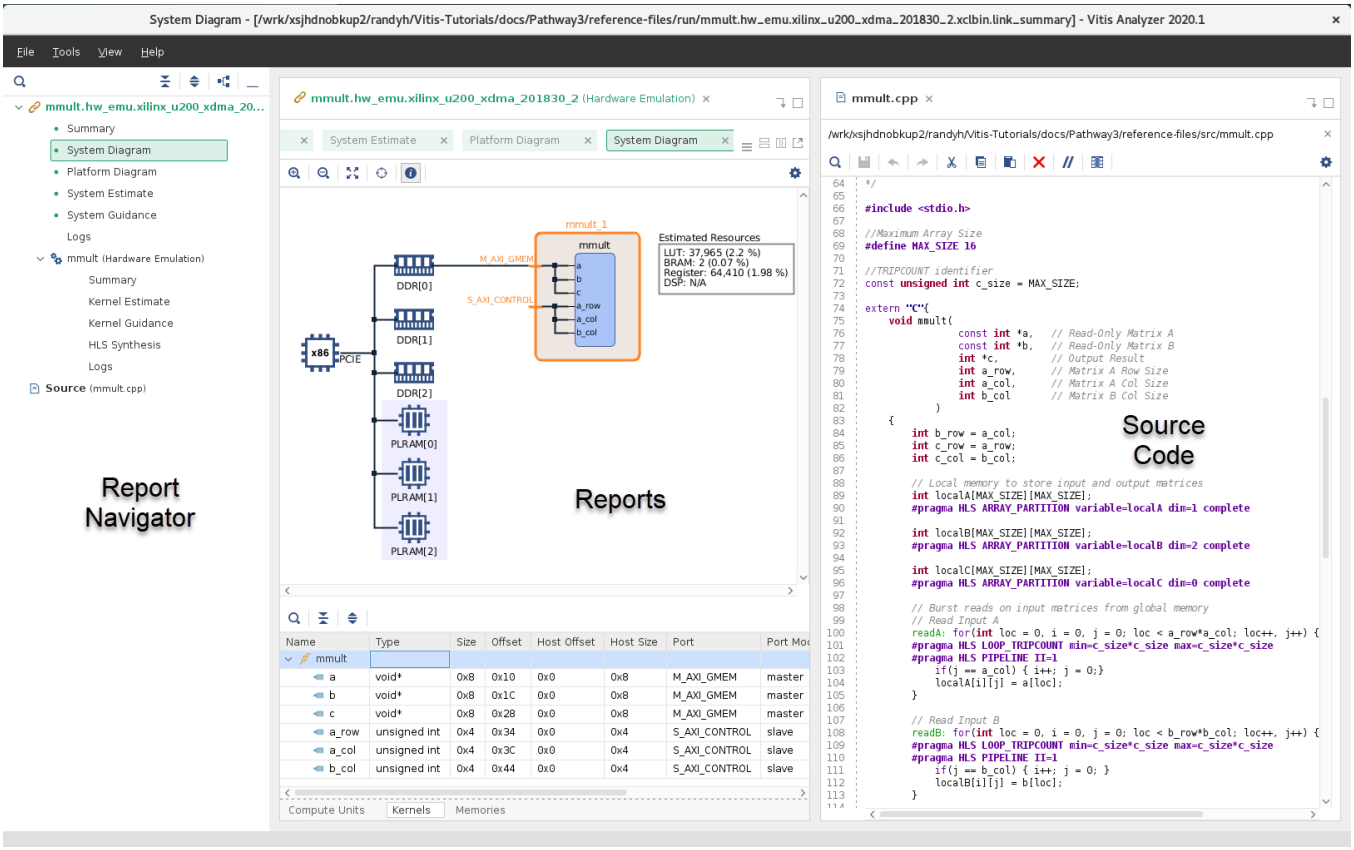
布局

Vitis 分析器还支持您以多种不同布局来重新排列报告，并保存这些布局以供复用。这样您即可安排比对两份报告，或者基于一份报告来检查另一份报告。主菜单中的“Layout”菜单允许您在现有布局之间进行切换，或者使用“Save Layout As”命令来保存定制报告布局。您可以使用“Reset Layout”将布局还原为保存的配置，或者使用“Remove Layout”来删除保存的布局。

Vitis 分析器 GUI 和窗口管理器

如下所示，Vitis™ 分析器工作空间排列为 3 个视图，包括“Report Navigator”、“Reports”和“Source Code”视图。不同视图可按需打开、关闭和重排。

图 81：**Vitis 分析器工作空间**



- Report Navigator：此视图位于左侧，可列出所有已打开的汇总文件和关联的报告。您可使用此视图来快速查找并打开报告。在上图中，您可看到“Link Summary”处于打开状态，其中包含“Compile Summary”，其所有关联报告都列在“Report Navigator”中。

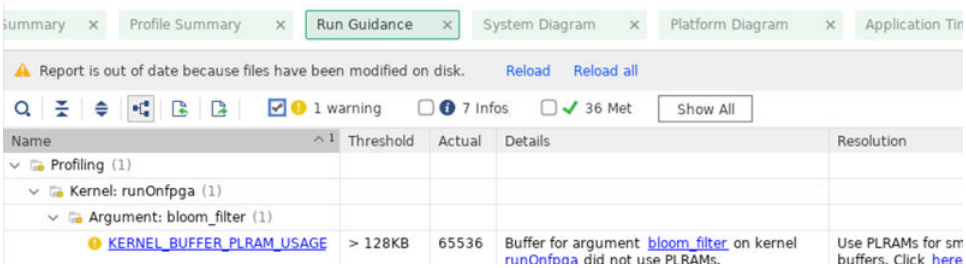
  单击“Report Navigator”中的任意文件时，它会在“报告 (Report)”视图中打开新选项卡。打开文件会在“Report Navigator”中的报告名称旁添加一个绿点，以便您快速确定报告在工具中是否已打开。

> **提示：**您可右键单击“Compile Summary”文件选择“Open HLS Project”以打开 HLS 工程，或者右键单击“Link Summary”或“Run Summary”选择“Open Vivado Project”以打开 Vivado 工程（如果需要）。无法打开 Vivado 工程以执行软件仿真 (Software Emulation) 构建。

- Reports：中心区域显示的是汇总文件和打开的报告的内容。在“Reports”视图中可打开多个报告，并且可通过选择视图顶部的窗口选项卡以快速切换报告。

  与“Compile Summary”、“Link Summary”或“Run Summary”相关的所有报告都组合在一起，包含在单个容器内。您可通过多种不同方式来排列任一容器内的各报告，包括使用“New Horizontal Group”、“New Vertical Group”或“Float”命令。“Multiple Summary”报告可作为集合打开，其中内容也可作为集合来进行管理。

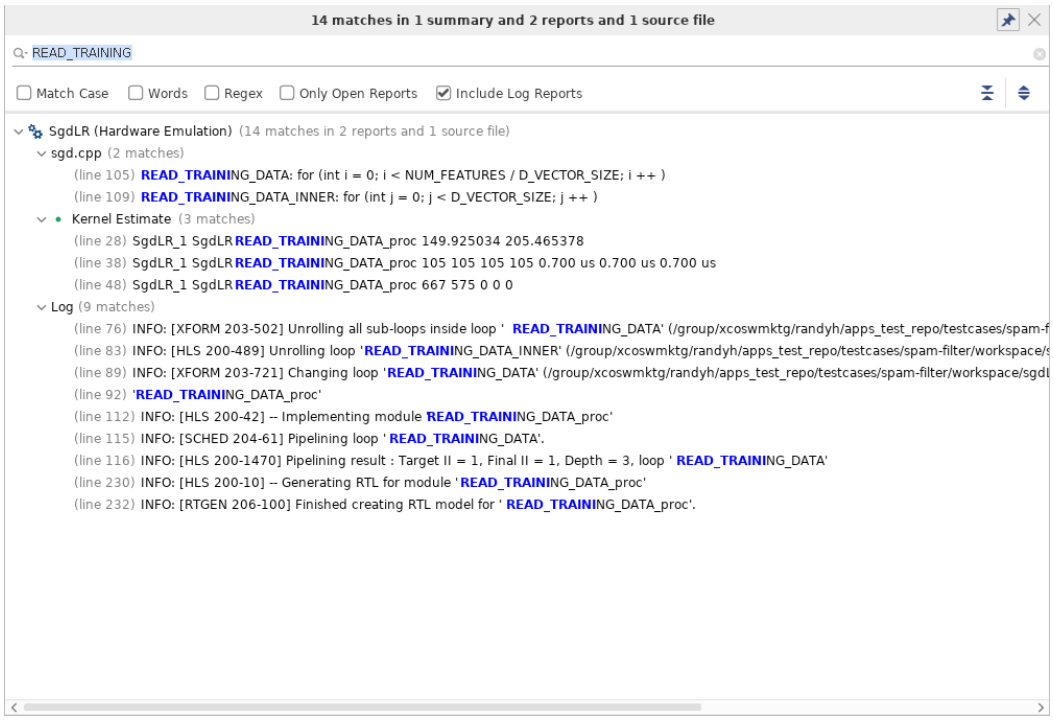
  如果磁盘上的汇总文件或报告均已更新（由于重新编译或重新运行应用等原因），那么 Vitis 分析器中当前打开的报告将显示“out-of-date”条幅。您可在当前打开的报告中继续工作，或者也可以重新加载更新后的文件。



- Source Code：可选的“Source Code”视图在工作空间右侧打开。此视图允许您根据“System Guidance”报告的反馈等信息来查看和编辑内核源代码。您可通过如下方式打开“Source Code”窗口：选中“Guidance”报告中的链接，或者通过右键单击“Report Navigator”中的“Compile Summary”并单击“Open Source”。

- Global Search：Vitis 分析器在显示的窗口顶部主菜单中的“Help”命令旁提供了一个搜索字段。您可使用该字段来搜索加载的“汇总 (Summary)”信息、关联的报告和源文件。此搜索对话框提供了各种选项，用于限制搜索项的搜索范围，如下图所示。

图 82：**Global Search**



“Report Navigator”和“Source Code”视图可通过单击工具栏中的“Minimize”按钮来折叠，也可以通过单击折叠的视图的选项卡来复原。

要关闭所有已打开的源代码视图，请选择“File” → “Close All Sources”命令。

要关闭与“Summary”报告关联的所有已打开的报告（如“Link Summary”），请右键单击“Report Navigator”视图中的“Summary”，然后选择“Close Tabs”。这样即可关闭与“Reports”视图中的“Summary”关联的所有已打开的报告。

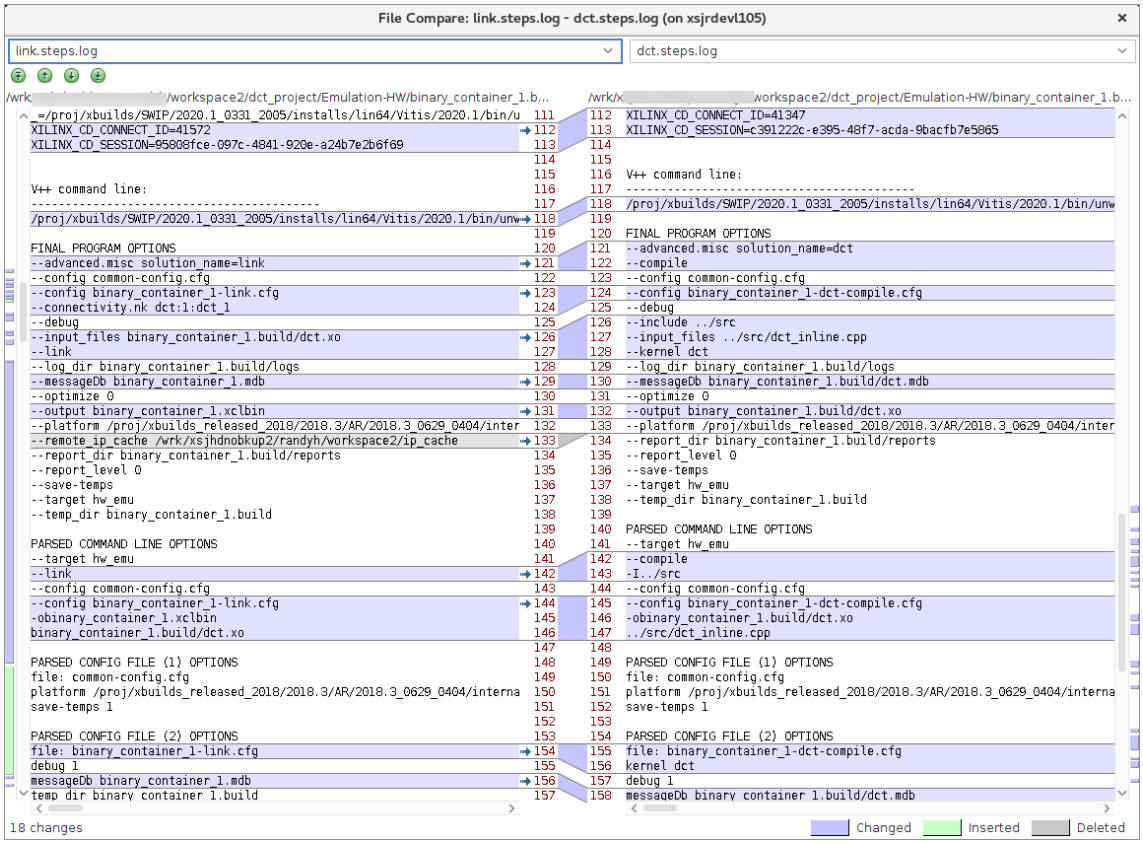
要关闭“Summary”文件（如“Link Summary”），请右键单击“Report Navigator”区域中的文件，然后选择“Close File”。关闭“Summary”文件会关闭所有关联的报告和文件。因此，关闭“Link Summary”也会结束“Compile Summary”的构建。

要关闭“Report Navigator”中显示的所有文件，请选择“File” → “Close All Files”命令。这样即可使 Vitis 分析器返回至主屏幕。

比较两个文本文件的差异

Vitis 分析器允许您对两份同类型报告进行比较。它可打开如下窗口。

图 83：**比较文本文件**



基于文本的比较报告包括：

- 内核估算
- 系统估算

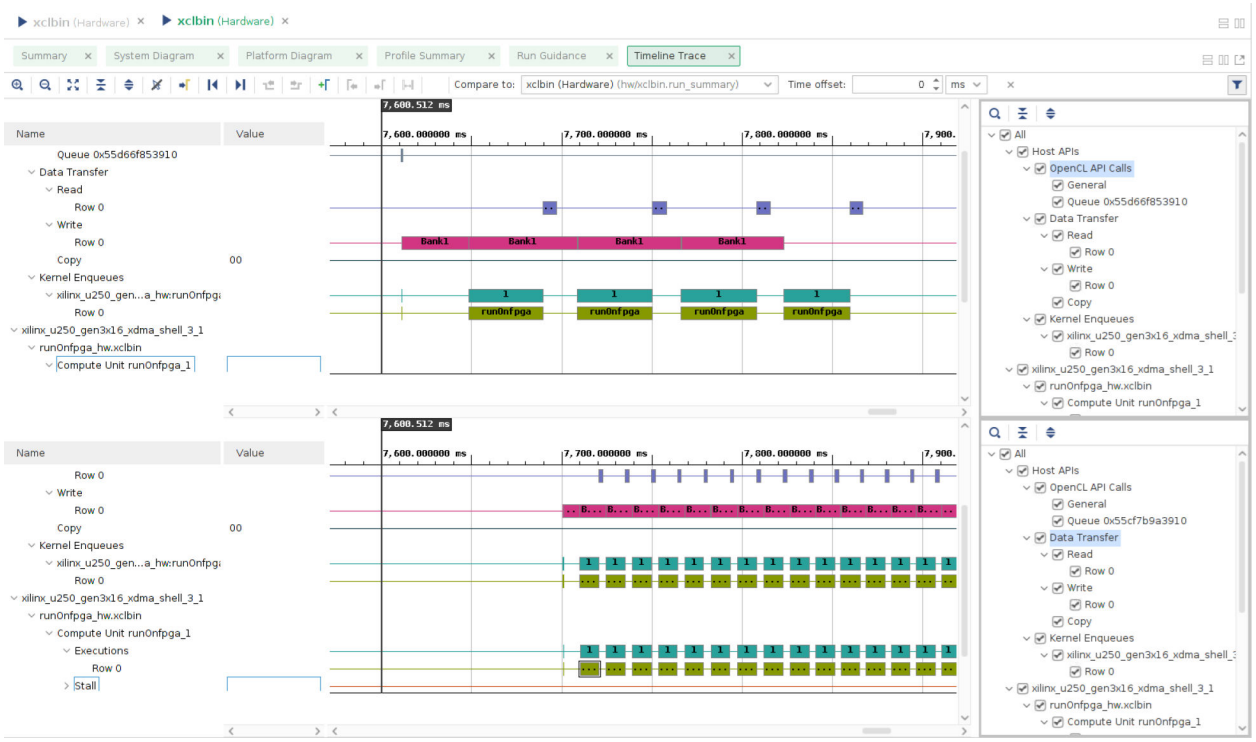
- 时序汇总
- 日志文件

要比较报告，必须在“报告导航器 (Report Navigator)”窗口中列出两份同类型的报告，或者在“报告 (Reports)”视图中打开这些报告。右键单击“Report Navigator”中或者“Report”视图中受支持的报告，然后选择“Diff with >”命令。此命令允许您指定另一份相同类型的报告，以便与当前选中的报告进行比较。

比对两份时间线轨迹报告

当有多份“Timeline Trace”报告可用时，Vitis 分析器提供了将两份“Timeline Trace”报告进行比对的功能。在打开的“Timeline Trace”报告中，单击报告工具栏菜单中的“Compare”链接，以显示可与当前报告进行比较的其它“Timeline Trace”报告的下拉列表。Vitis 分析器将同步报告间的时间线（如下图所示），当您在任一轨迹窗口内水平滚动时，两份轨迹报告会一起滚动。

提示：您也可以右键单击“Report Navigator”窗格中左侧列出的“Timeline Trace”报告，并选择“Compare to”命令，当有多条时间线可供比较时即启用此命令。



在两次运行应用之间的不同时间点，可触发“Timeline”报告中的各种事件。您可以使用“Time offset”选项报告工具栏菜单（如上图所示）来同步两个窗口内的光标，以便执行可视化比较。缩放顶部轨迹窗口时，底部窗口也会自动进行缩放。

正如图中所示，在同一时间段内，下方时间线轨迹内的内核队列 (Kernel Enqueues) 数比上方时间线内的更多。在下方时间线轨迹中，针对读写 (Read/Write) 操作的数据传输 (Data Transfer) 缓冲数也比上方时间线内的更多。这种可视化比较可帮助您比较两条轨迹的传输密度，例如，底部时间线轨迹内的每个队列之间的距离比顶部轨迹内的短得多，每个队列的计算时间也短得多。

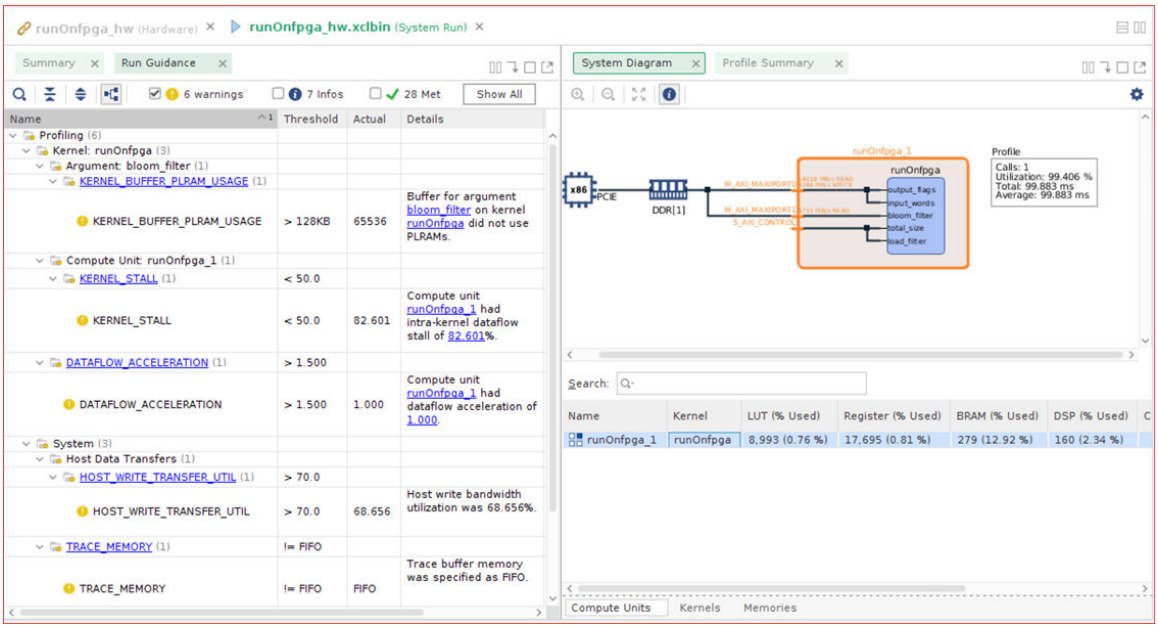
您可以单击右上角的“Filter”命令以指定要在每条时间线上显示的元素。这样您即可限制“时间线轨迹 (Timeline Trace)”报告中显示的信息量，从而集中显示您感兴趣的信息以供比较。

不同报告之间的交叉探测

Vitis 分析器支持在不同报告和视图内包含各种可选对象：

- “Compute Units (CU)”：在“系统框图 (System Diagram)”和关联的“计算单元 (Compute Units)”表中可选。选中内核会同时选中关联的计算单元，反之亦然。CU 可在“利用率 (Utilization)”报告、“剖析汇总 (Profile Summary)”、“应用时间线 (Application Timeline)”和“波形 (Waveform)”视图中找到。
- “CU ports”：可在“System Diagram”中选中。
- “Kernels”：可在“System Diagram”和关联的“内核 (Kernels)”表中选中。请注意，选中内核会同时选中 CU，反之亦然。“Kernels”可在“Link Summary”、“Utilization”报告、“Accelerators”下的“System Guidance”、“Profile Summary”以及“Waveform”视图下找到。
- “Kernel ports”：可在“System Diagram”中选中。
- “Function arguments”：可在“System Diagram”和“Kernels”表中选中。
- “AXI interconnects”：可在“System Diagram”中选中。选择此对象会选中到存储体的所有连接。
- “AXI ports”：可在“System Diagram”中选中。这些对象已“扁平化”，例如，这些对象对于所有内核都相同。显示在“Profile Summary”中和“Waveform”视图中（数据传输）。
- “Memory resources”：可在“平台框图 (Platform Diagram)”和“系统框图 (System Diagrams)”以及关联的“存储器 (Memories)”表中选中。显示在“Profile Summary”中（数据传输：内核到全局存储器）。
- “Host CPU”：可在“Platform Diagram”和“System Diagrams”中选中。

图 84：**不同报告之间的交叉探测**



Vitis 分析器支持在不同报告之间进行交叉探测，例如，在“System Diagram”内，以及从“Guidance”视图到其它视图之间进行交叉探测。“Guidance”视图将为报告的违例提供切实可行的解决方案，您可使用来自违例的交叉探测快速导航至其它报告和视图。

根据报告，可执行双向或单向交叉探测。“Guidance”报告允许您选择其它报告中的对象，但不支持从其它报告或视图进行交叉探测。

- “System Diagram”与“Profile Summary”报告之间的双向交叉探测。在其中任一报告中选中内核、计算单元或计算单元端口就会在另一个报告中同时选中该内核、计算单元或计算单元端口。选中内核还会同时选中报告中关联的 CU。
- 从“Guidance”到“System Diagram”和“Profile Summary”报告的单向交叉探测。“Guidance”报告的“详细信息 (Details)”列会显示对应于设计对象（例如，内核、CU 等）的超链接。
  - 单击“Guidance”中的任一内核、计算单元或计算单元端口就会在“System Diagram”和“Profile Summary”中同时选中该内核、计算单元或计算单元端口。
  - 单击存储器或内核实参超链接会在“System Diagram”中将其选中，但不会在“Profile Summary”中将其选中。
  - 在“Guidance”中单击内核端口超链接就会在“System Diagram”中选中 CU 端口。
  - 在某些情况下，“Details”列会显示值的超链接，例如，82.601%。
    - 单击值超链接会选中对应的设计对象，并导航至“Profile Summary”报告中的关联部分。
    - 如果报告已打开，但隐藏在另一个选项卡背后，则它将转至前台。
    - 如果报告未打开，那么单击值超链接将会打开此报告。
  - “Guidance”超链接还包含工具提示，用于解释单击操作的作用。
  - 此外，选择“System Diagram”中的其它对象（例如，主机、存储器、AXI Interconnect 和内核实参）并不会交叉探测“Profile Summary”，因为此报告并不会将这些对象显示为可选对象。

平台框图和系统框图

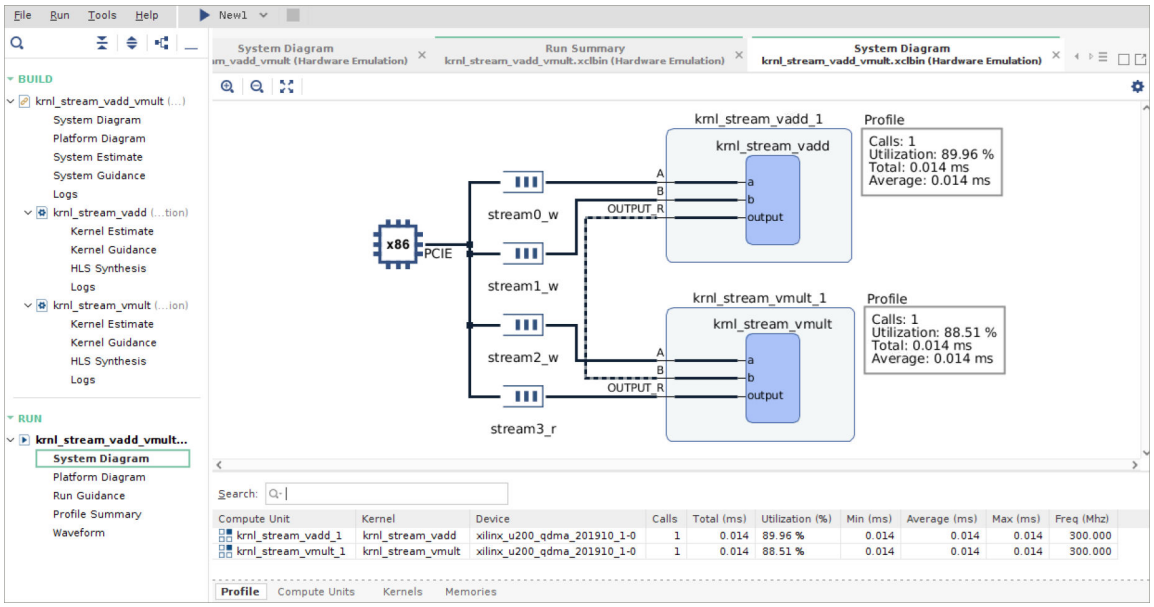
“平台框图(Platform Diagram)”和“系统框图 (System Diagrams)”显示了平台资源以及集成到平台的内核代码。用户可在 Vitis™ 分析器的“Link Summary”、“Run Summary”或工程的 `.xclbin` 中查看这些资源和代码。

“Platform Diagram”是加载 `.xclbin` 之前，目标平台的模块框图。此框图显示了可用的所有 DDR 存储体和 PLRAM 及其可用连接。其底部表格显示了各存储体的名称以及存储器类型、其大小和可用存储器所在的 SLR 区域的详细信息。

“System Diagram”显示了 `.xclbin` 所使用的存储体或 PLRAM。您还可查看计算单元的函数实参连接到 AXI4 接口的方式。“System Diagram”底部的表格显示了每个计算单元、内核以及存储器的信息。对于包含 AI 引擎内核的设计，“System Diagram”还显示了这些内核的相关信息。“System Diagram”的功能特性包括：

- 内核名称以及有关可用内核所在 SLR 的指示信息。
- LUT%
- 寄存器 %
- 已用 BRAM %
- 已用 URAM %
- 已用 DSP %

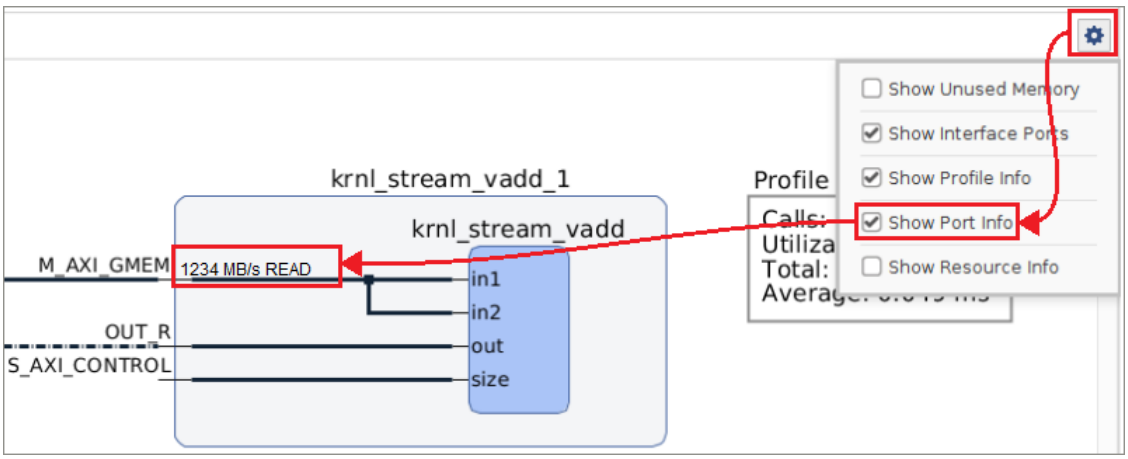
图 85：**含剖析数据的系统框图**



加载“Run Summary”时，“System Diagram”包含来自运行的剖析数据。打开包含 `opencl_summary.csv` 以及（可选）用于硬件仿真的 `profile_kernels.csv` 的“Run Summary”时，Vitis 分析器会自动运行 `perf_analyze`。剖析数据将加载到“System Diagram”底部表格中，并且可在框图中查看，如下图所示。

来自该表的资源信息还会显示在“System Diagram”中的每个内核或 CU 旁的框中。“Settings”命令 ( ) 支持您显示或隐藏未使用的存储器、接口端口、剖析信息和资源信息。

图 86：**显示端口信息**



计算单元上的端口可在系统框图上显示传输速率以及 CU 利用率百分比。CU 端口传输速率取自“Profile Summary”报告的“内核传输 (Kernel Transfer)”部分。CU 利用率统计数据则取自“Profile Summary”的“计算单元利用率 (Compute Unit Utilization)”部分。只要在硬件和硬件仿真运行期间使用 Vitis 编译器 `--profile` 选项启用了“剖析”功能（如 --profile 选项 中所述），就会显示性能数据。

器件映射

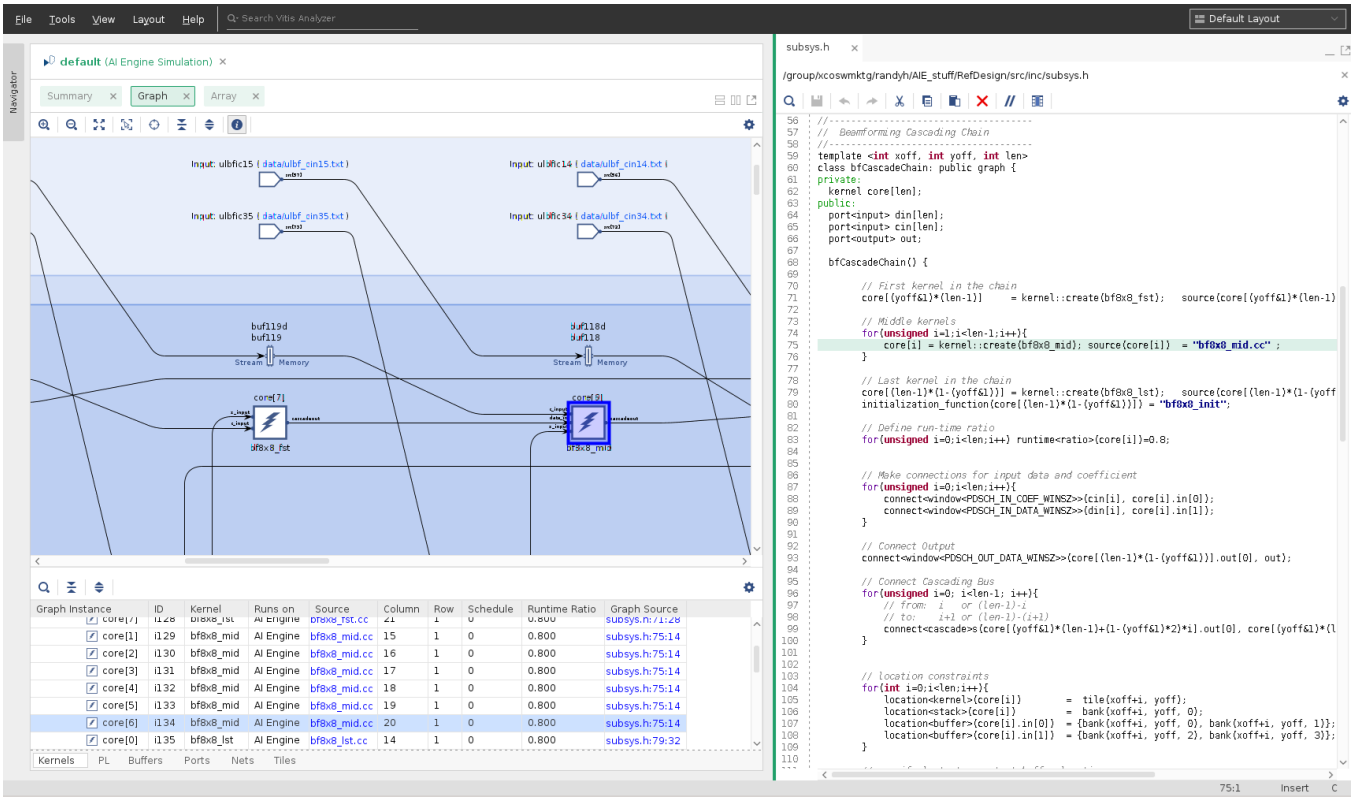
Vitis™ 分析器还在工程的“Link Summary”中提供了“器件映射 (Device Map)”，此工程可从“Report Navigator”窗格打开。“Device Map”可显示目标平台的静态和动态区域的抽象视图，并显示器件和 SLR 上的内核布局。您还可查看动态区域内部每个计算单元的布局位置。

“Device Map”底部的表格所示信息与“System Diagram”中的表格所示信息类似。您可在该表中选中各行，这样即可在“Device Map”中高亮显示对应的部分。表中任意对象的高亮颜色均可通过在该表中右键单击对象并选择“Highlight”颜色来更改。更新后的颜色用于高亮“Device Map”上选定的对象。此外，“Device Map”和“System Diagram”对象支持跨多个视图高亮对象。例如，您也可以在“System Diagram”或“Device Map”表中选中对象，那么这 2 个视图就都会高亮表中选中的对象。

AI 引擎 Graph 和阵列

AI 引擎 Graph 和阵列图可提供 AI 引擎拼块 (tile) 中实现的 ADF Graph 应用结构概览。

图 87：**AI 引擎 Graph 应用**

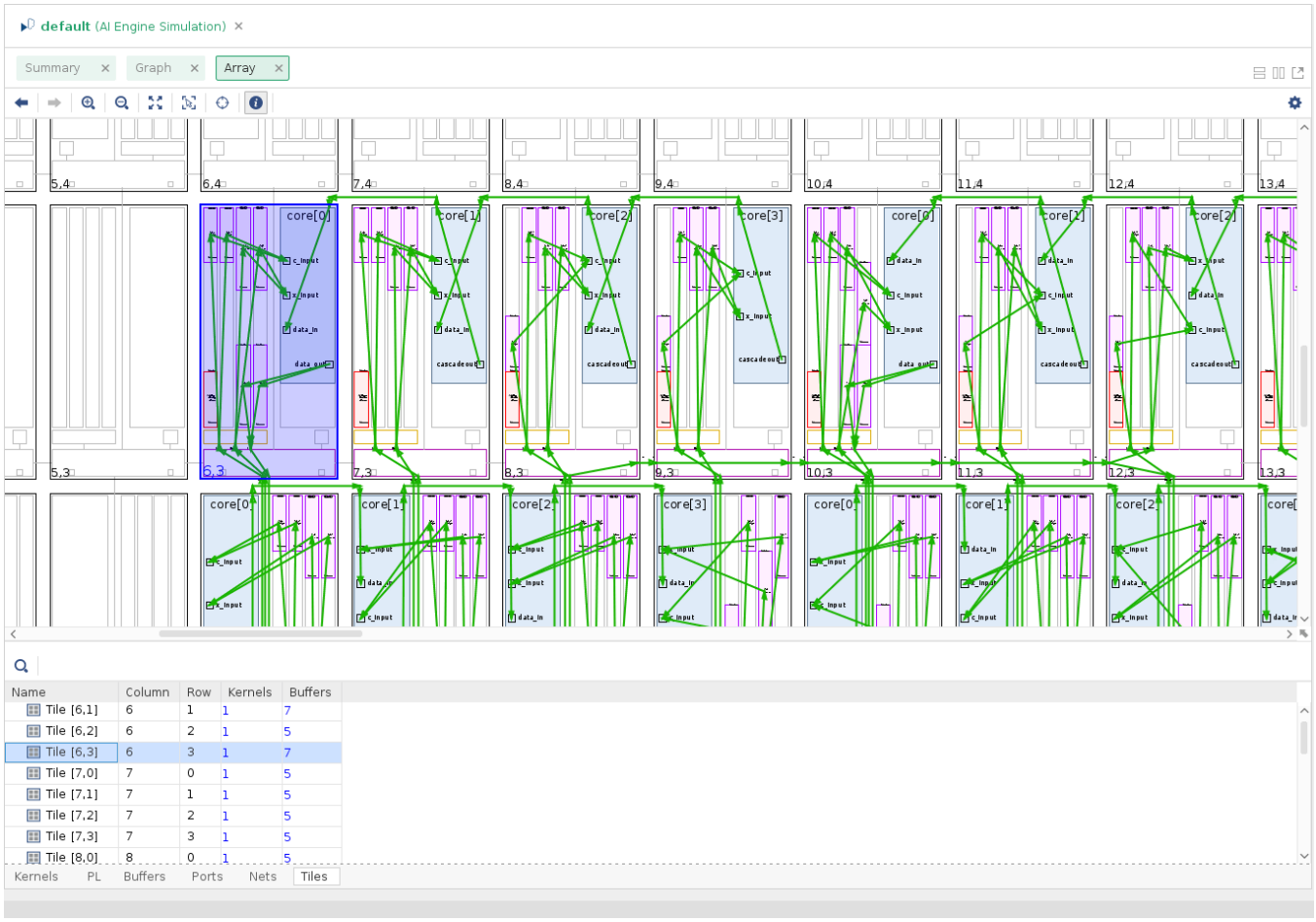


AI 引擎 Graph 视图显示了 AI 引擎编译器所见的 ADF Graph 的连接。画布显示了表示内核、内核实参、存储缓冲器和主输入输出的各节点，边缘绘制为表示这些元素之间的连接关系。

在画布右侧显示的“Settings”面板可用于自定义 Graph 图的视图，您可在其中显示或隐藏所示 Graph 的各种元素。

在图形化视图下有一组表格，其中包含图形化视图中所绘制的所有对象：内核、连接、路径、存储器等。选中这些表中的元素即可与图形化视图进行双向交叉探测。

图 88：**AI 引擎阵列图**



“阵列 (Array)”视图显示了 AI 引擎拼块阵列上 ADF 图的空间分布方式。阵列画布包含阵列的所有核与存储器组件。在核中，内核按其编程位置进行绘制，核和/或存储器之间的连接则以连接线来显示。PL 区域通过抽象化表示法来按需显示，以便在可编程逻辑区域内对 PL 核和接口进行建模。

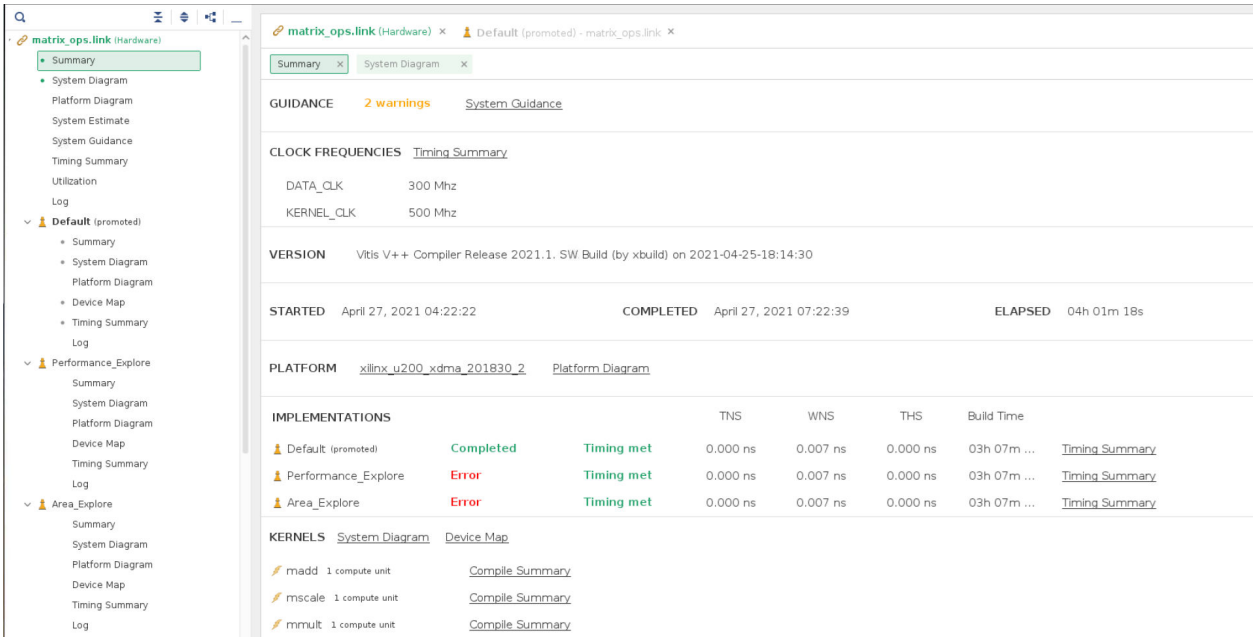
在画布右侧显示的“Settings”面板可用于自定义阵列视图，您可在其中显示或隐藏资源和/或拼块阵列的元素。

链接汇总：多种策略和时序报告

如 运行多项实现策略来实现时序收敛 中所述，`v++` 链接步骤可使用不同策略对 Vivado 实现运行多次迭代以达成时序收敛。Vitis 分析器则可将多次实现尝试所产生的结果可视化并进行比较。如前所述，Vitis 编译器会自动挑选首个成功完成并且满足时序的运行来继续构建并生成器件二进制文件 (`xclbin`)。这其中就包括仅为这轮运行记录报告文件。但您可让 Vitis 编译器保持等待，直至所有实现运行全部完成后，再继续生成 `xclbin` 文件。在此情况下，在 `link_summary` 中会提供每次实现运行的结果以供您在 Vitis 分析器中进行复查和比对。

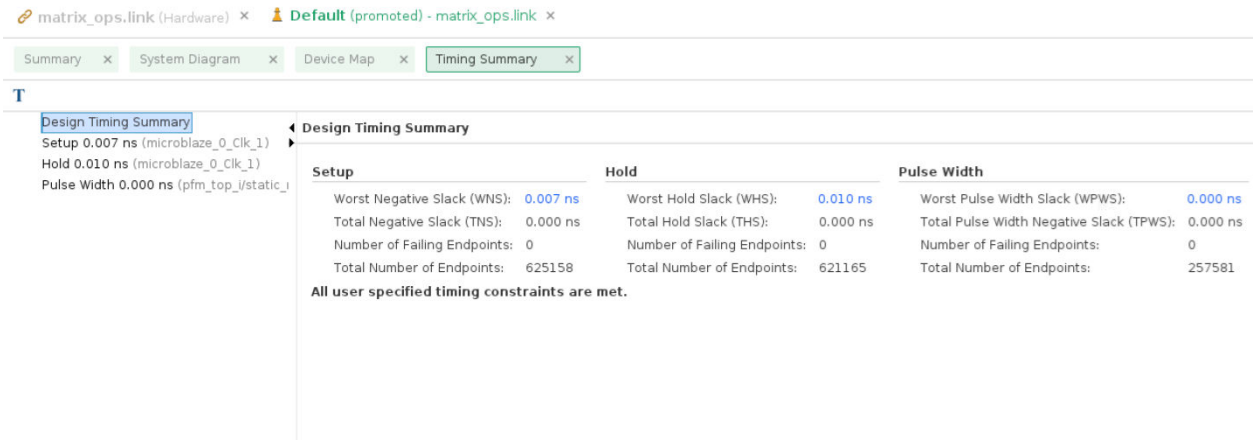
含多种实现策略的“链接汇总 (Link Summaries)”可在“Report Navigator”窗格内显示每一项策略，如下图所示。其中每项策略都包含一份“Timing Summary”报告。即使仅为一项策略生成 `xclbin`，所有实现运行仍可通过“Link Summary”报告提取实用的资源和时序信息。

图 89：**链接汇总**



Vitis 分析器中的“Timing Summary”报告是 Vivado 布局布线所生成的完整“Timing Summary”报告的简化版本。简化的报告仅显示该设计的最差建立时间、保持时间和脉冲宽度路径。您无需打开 Vivado 工程或网表即可查看此报告，因此您可以快速浏览收敛失败的时序路径。报告查看器中显示了详细信息，如下所示。此处显示了“Timing Summary”以及设计中观测到的最差建立时间 (Setup)、保持时间 (Hold) 和脉冲宽度 (Pulse Width) 的详细路径报告。要查看完整的时序报告，请单击工具栏上的“T”按钮，这样即可显示此报告的文本版本。

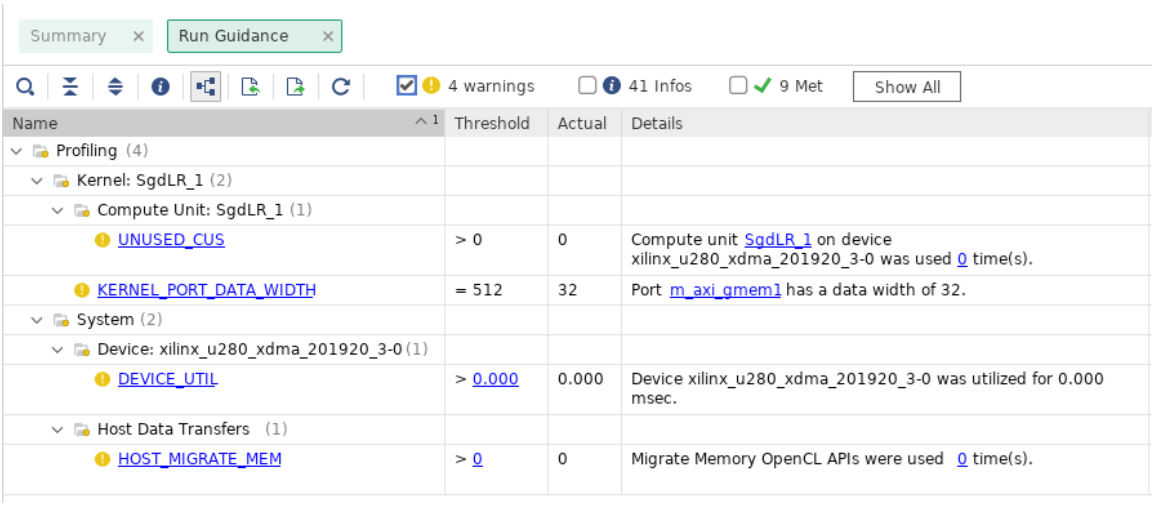
图 90：**时序汇总**


matrix_ops.link (Hardware)
Default (promoted) - matrix_ops.link
Summary
System Diagram
Device Map
Timing Summary
Design Timing Summary
Setup 0.007 ns (microblaze_0_Clk_1)
Hold 0.010 ns (microblaze_0_Clk_1)
Pulse Width 0.000 ns (pfm_top_i/static_
Design Timing Summary
Setup
Worst Negative Slack (WNS): 0.007 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 625158
Hold
Worst Hold Slack (WHS): 0.010 ns
Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 621165
Pulse Width
Worst Pulse Width Slack (WPWS): 0.000 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 257581
All user specified timing constraints are met.

设置指南阈值

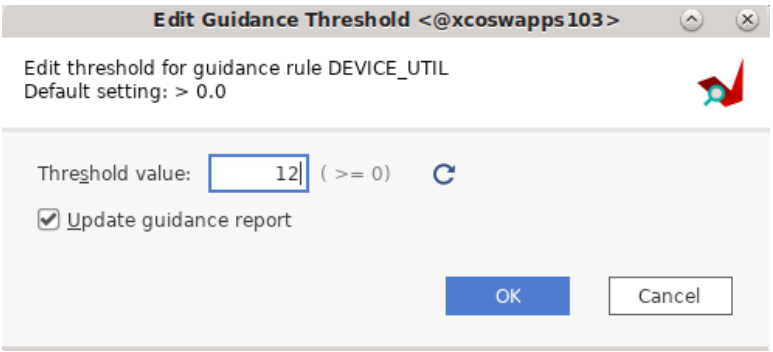
“Run Guidance”报告中显示的指南消息是由工具中定义的特定规则以及阈值触发的。用户可在 Vitis™ 分析器中对其中部分规则和阈值进行修改。打开“Run Guidance”报告时，在报告的“Threshold”列中，可修改的值将显示为链接，如下图所示。

图 91：**运行指南阈值**



单击“Threshold”列中的链接将显示“Edit Guidance Threshold”对话框，如下所示。此对话框顶部显示的是选定规则的当前运算符和阈值，下半部分则允许您对值进行编辑。

图 92：**编辑指南阈值**



上图显示的是以用户指定的值来重新定义“Threshold value”的过程。在某些案例中，会显示接受的值的范围，该工具会检查提供的值是否在此范围内。在以上示例中，指定的值必须大于或等于 0。

“Edit Guidance Threshold”对话框还会提供“Reset threshold”命令，以允许您将用户定义的值复位为工具提供的硬编码值。这对应于上图所示的  。

提示： “运行指南 (Run Guidance)”报告的菜单栏还提供了“Reset Guidance Thresholds”命令，供您用于清除报告中的所有用户修改的阈值。

“Update guidance report”复选框允许您指定值更新后是否重新运行指南报告。如果您不勾选此框，那么阈值将按指定方式更改，且由于用户指定了新的阈值，此报告将被标记为过期 (out-of-date)。您将需要使用“Reload”来重新加载报告，以查看用户提供的值的影响。

单击“OK”以更改该值，或者单击“Cancel”以关闭此对话框，不执行更改。

导入/导出规则阈值

有关哪些阈值可修改的规则以及所有规则的默认值都存储在工具安装中。您可使用定制规则文件来改写这些值，在 Vitis 分析器中，您可将此类定制规则文件导出和导入不同工程。

在“Run Guidance”报告中有一条或多条自定义规则时，在报告的工具栏菜单中，“Export User Guidance Thresholds”命令将变为活动状态。您可在上图中看到显示的此命令。此命令允许您导出自定义的阈值，以供在其它设计中复用。

以下显示了一个样本文件：

```
profile_rules =
(
  {
    id = "HOST_MIGRATE_MEM";
    value = "1";
  },
  {
    id = "DEVICE_UTIL";
    value = "3";
  }
);
version = "1";
```

用户阈值文件无需包含所有指南规则，只需包含涉及值更改的规则以及规则 ID 和值字段即可：

- `id` 是规则的名称，与“Run Guidance”报告的“Name”列中显示的名称相同。
- `value` 是用户指定的值，与“Threshold”列中显示的值相同。

Vitis 分析器 IDE 还允许您通过“Import User Guidance Thresholds”导入用户指南阈值，以便在其它工厂内复用定制阈值。您可以根据需要导出 `user-thresholds.cfg` 文件、编辑值并导入文件。导入用户阈值文件会重新生成“Run Guidance”报告，以使用导入的值。

重要提示！ 导入用户指南阈值文件则将改写任何现有的用户修改的阈值。

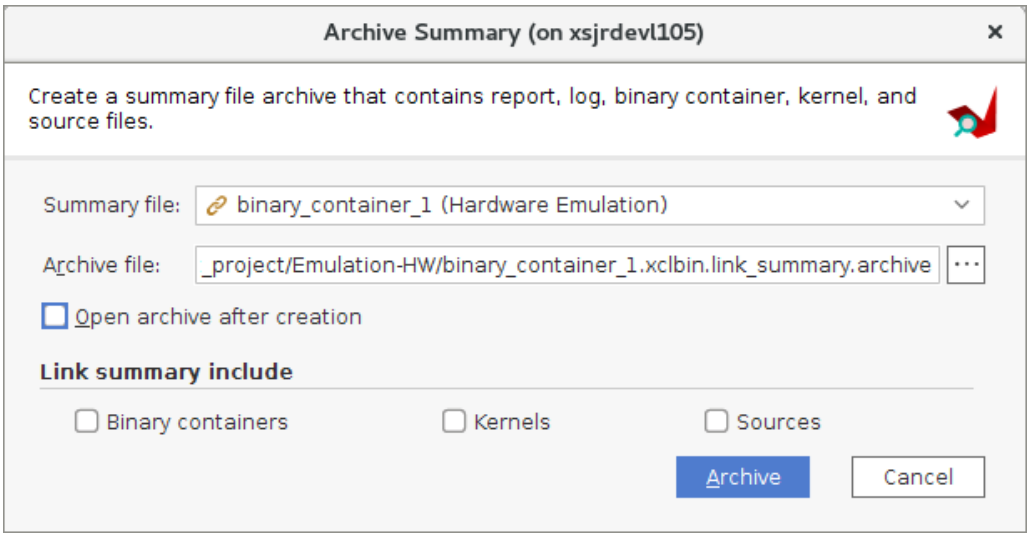
创建存档文件

执行工程开发时，您对主机或内核代码所执行的某些更改可能完全改变该工具所生成的各种构建和运行汇总信息的内容或质量。除非您手动保存相关文件，否则现有报告和分析数据将被覆盖。“Archive Summary”命令可用于保存处于打开状态的“Link Summary”或“Run Summary”的所有相关文件。

提示：该功能允许您通过共享存档汇总信息来与其它团队成员共享设计报告。

选择“File”→“Archive Summary”菜单命令，或者右键单击“Report Navigator”中的任一汇总信息，然后选择“Archive Summary”。这样即可打开“Archive Summary”对话框，如下所示。

图 93：**“存档汇总”对话框**



存档文件名必须包含扩展名 `link_summary.archive` 或 `run_summary.archive` 才可供 Vitis 分析器识别。存档的内容取决于要存档的汇总报告。

“Link Summary”的内容包含以下文件扩展名：

- 二进制容器：.xclbin
- 内核：.xo
- 系统/平台框图：.json
- 系统估算：.xtxt
- 指南：.html 或 .pb
- 时序汇总：.rpt 或 .rpv

- 利用率：.xutil
- HLS 综合报告：报告

“Run Summary”的内容包含以下文件扩展名：

- 系统框图：.run_summary
- 平台框图：.run_summary
- 指南：非必需，通过使用 perf_analyze 和 opencl_summary.csv 来生成
- 剖析汇总：opencl_summary.csv
- 应用时间线：opencl_trace.csv
- 波形报告：.wdb

重要提示！对于 AI 引擎设计，当 `run_summary` 包含追踪报告，但不包含对应 AI 引擎工作目录的条目时，Vitis 分析器无法运行 `hwanalyze` 来显示此追踪报告。在此情况下，该工具会提示您提供 AI 引擎 `compile_summary` 的位置，并更新磁盘上的 `run_summary` 文件。如果您在提供此位置前对 `run_summary` 进行存档，则无法查看来自已存档的 `run_summary` 的追踪报告。

您也可以选择保存 `.xclbin` 文件扩展名、任何已编译的内核对象文件扩展名（`.o` 和 `.xo`）以及原始源文件扩展名（`.cpp`、`.c` 和 `.cl`），用于生成汇总报告。

提示：指南信息不予保存，因为它是由 Vitis™ 分析器从 `opencl_summary.csv` 以及（可选）`profile_kernels.csv` 动态生成的。

要打开现有存档文件，请使用“File” → “Open Summary”命令，并浏览所需归档文件。您也可以在启动 Vitis 分析器时使用以下命令打开存档文件：

```
vitis_analyzer design.archive
```

“存档汇总 (Archive Summary)”还支持命令行格式，欲知详情，请使用以下命令：

```
archive_summary -help
```

注释：Vitis 分析器存档是压缩存档，可通过解压来访问其中个别存档文件，包括存档中包含的 `xclbin` 和 `xo` 文件。

第七部分

使用 Vitis IDE

本部分包含以下章节：

XILINX

第 40 章

Vitis 命令选项

在 Vitis™ 集成设计环境 (IDE) 中，您可以创建新应用工程或平台开发工程。

`vitis` 命令可按您定义的选项来启动 Vitis IDE。它可提供相应选项，用于指定工程的工作空间和选项。以下小节描述了 `vitis` 命令选项。

显示选项

请复查下列选项所显示的指定信息。

- `-help`：显示 Vitis 核开发套件命令选项的帮助信息。
- `-debug`：启动 Vitis IDE 以在命令行工程上运行调试。

  提示：要查看 `vitis -debug` 命令的帮助，请使用 `-debug -help`。

- `-version`：显示 Vitis 核开发套件发行版本。

命令选项

以下命令选项可指定针对当前工作空间和工程配置的 `vitis` 命令。

- `-workspace <workspace location>`：指定 Vitis IDE 工程的工作空间目录。
- `{-lp <repository_path>}`：将 `<repository_path>` 添加到“驱动程序/操作系统/库 (Driver/OS/Library)”搜索目录列表中。
- `-eclipseargs <eclipse arguments>`：Eclipse 专用的实参将传递到 Eclipse。
- `-vmargs <java vm arguments>`：表示将传递到 Java VM 的其它实参。

第 41 章

创建 Vitis IDE 工程

在 Vitis™ IDE 中，您可以创建新应用工程或平台开发工程。以下章节为您演示了如何设置工作空间、创建 Vitis IDE 工程以及如何使用 IDE 的新功能。

启动 Vitis IDE 工作空间

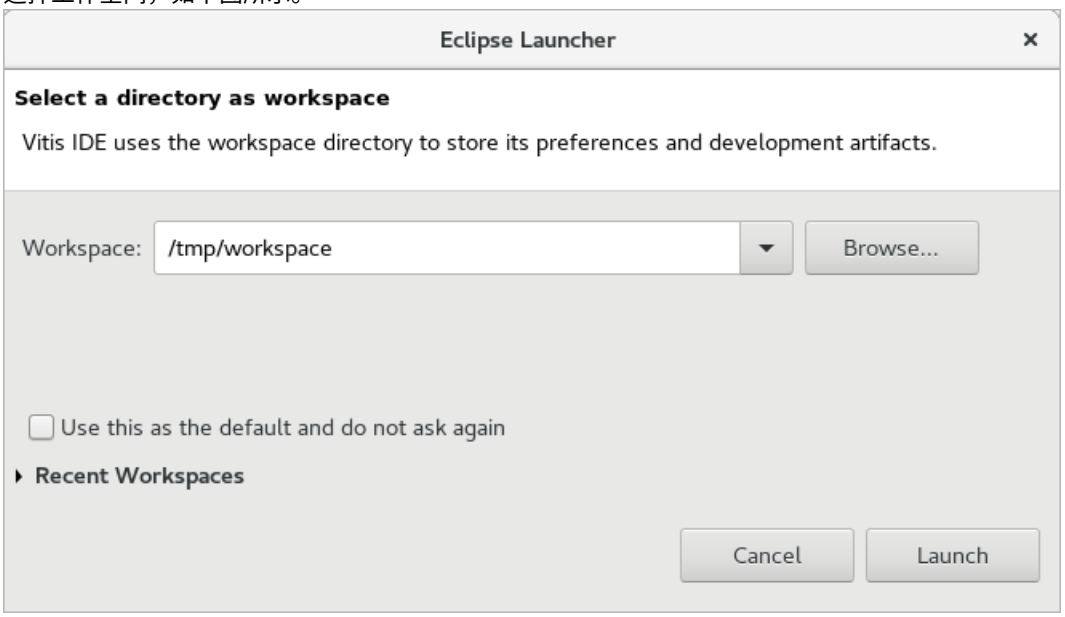
1. 直接从以下命令行启动 Vitis IDE。

```
$vitis
```

> **重要提示！**打开新目标平台时，要输入 Vitis 核开发套件命令，请确保按 设置 Vitis 环境 中所述完成相应设置。

这样会打开 Vitis IDE。

2. 选择工作空间，如下图所示。



工作空间是您在使用 IDE 时，用于存储您的工程、源文件和结果的文件夹。您可以为每个工程定义独立的工作空间，或者使用单一工作空间来包含多个工程和不同类型。以下指示信息为您演示了为 Vitis IDE 工程定义工作空间的方式。

3. 单击“Browse”以浏览并指定工作空间，或者在“Workspace”字段中输入相应的路径。
4. （可选）启用“Use this as the default and do not ask again”以将指定的工作空间设置为默认选项，这样即可在后续使用 IDE 时不再显示此对话框。

注释：要将此对话框复原，请浏览至 “Window” → “Preference” → “Additional” → “General” → “Startup and Shutdown” → “Workspaces”，然后选中“Prompt for workspace on startup”。

5. 单击“Launch”。

提示：要在 Vitis IDE 中更改当前工作空间，请依次选中“File” → “Switch Workspace”。

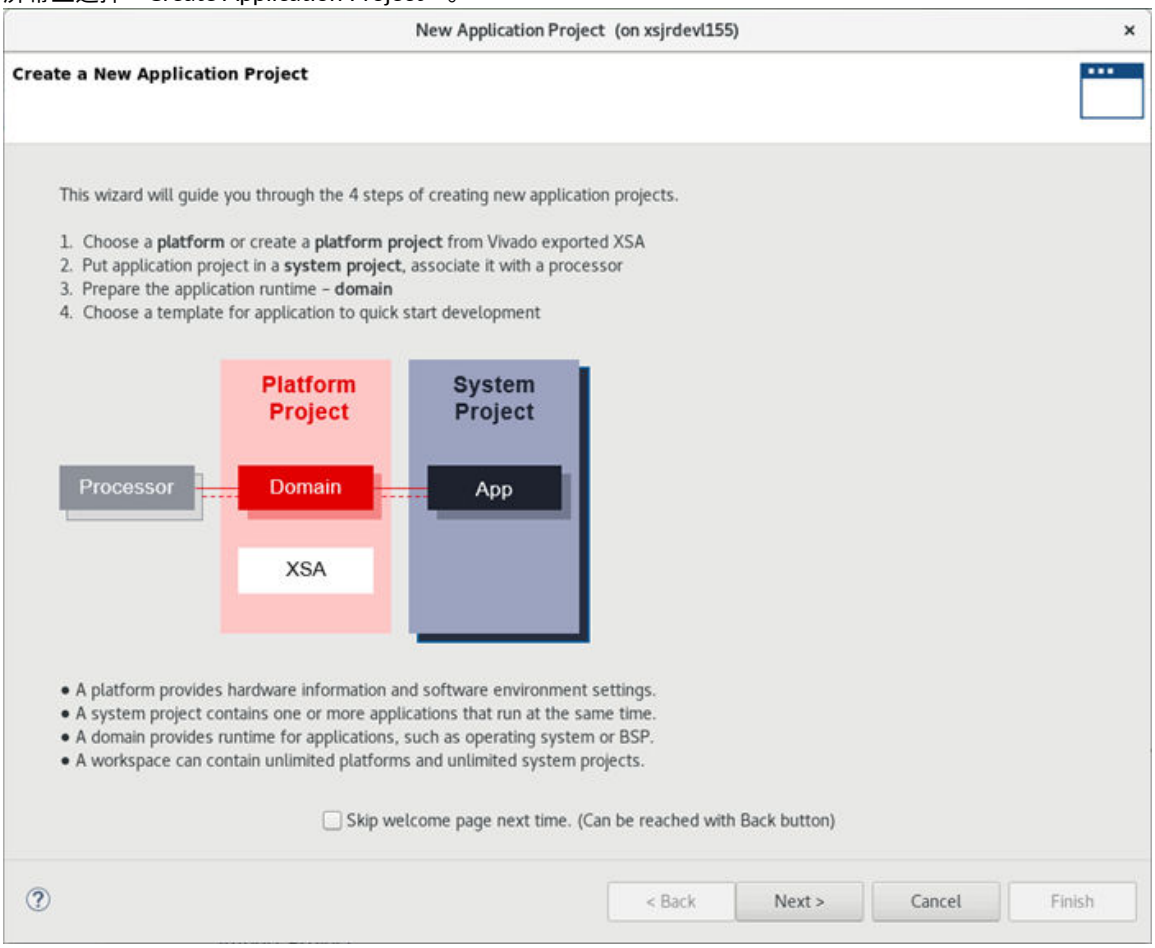
现在，您已经创建了一个工作空间，下一步就可以在其中填充工程了。

创建应用工程

提示：设计示例随 Vitis 核开发套件安装一起提供，在赛灵思 Vitis 示例 GitHub 仓库上也可以找到。如需了解更多信息，请参阅 入门示例。

启动 Vitis IDE 后，您可创建新的应用工程。

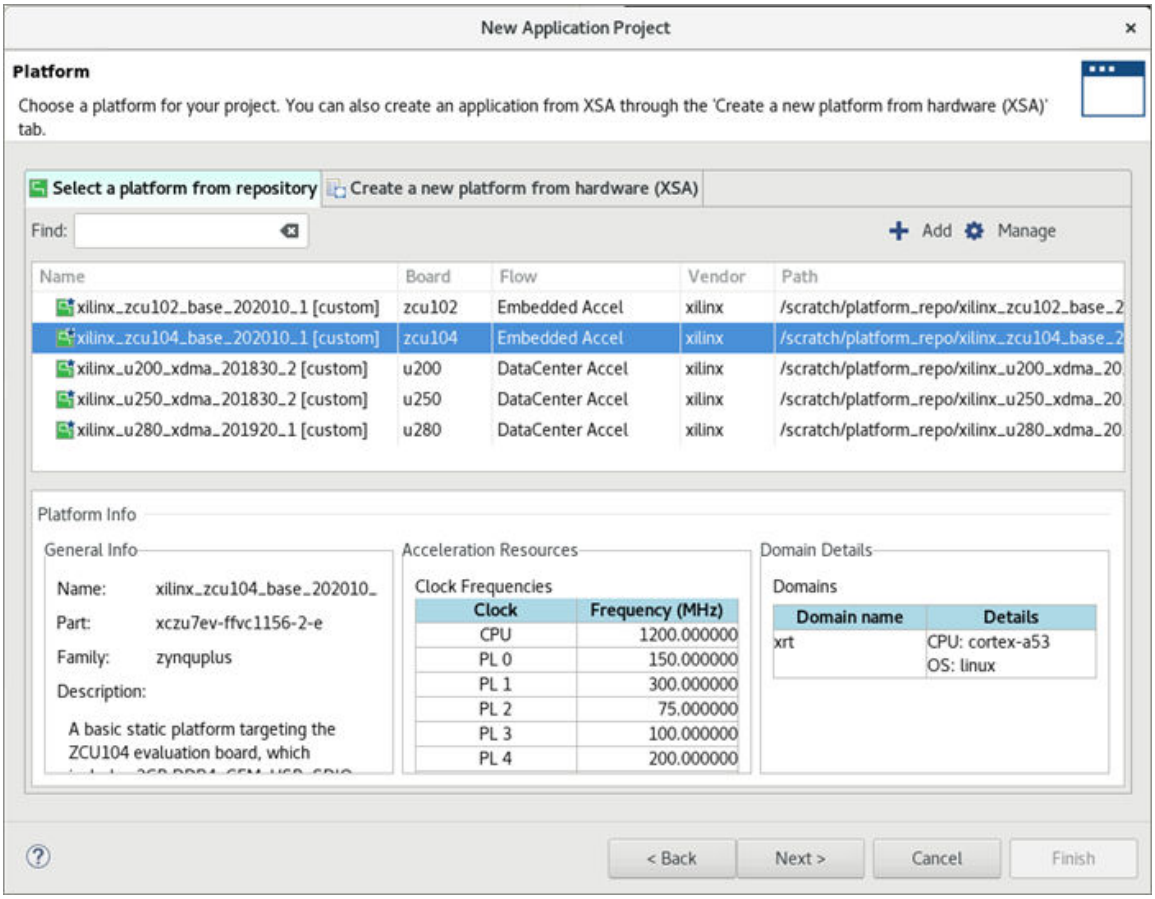
1. 选择“File” → “New” → “Vitis Application Project”或者如果这是首次启动 Vitis IDE，则可在“Welcome”屏幕上选择“Create Application Project”。



这样会打开“New Application Project”Wizard 并显示“Welcome”页面，为新用户解释整个过程。您可通过启用“Skip welcome page next time”以避免再次显示此页面。

2. 单击“Next”打开“New Application Project”Wizard 的“Platform”页面以指定目标平台。

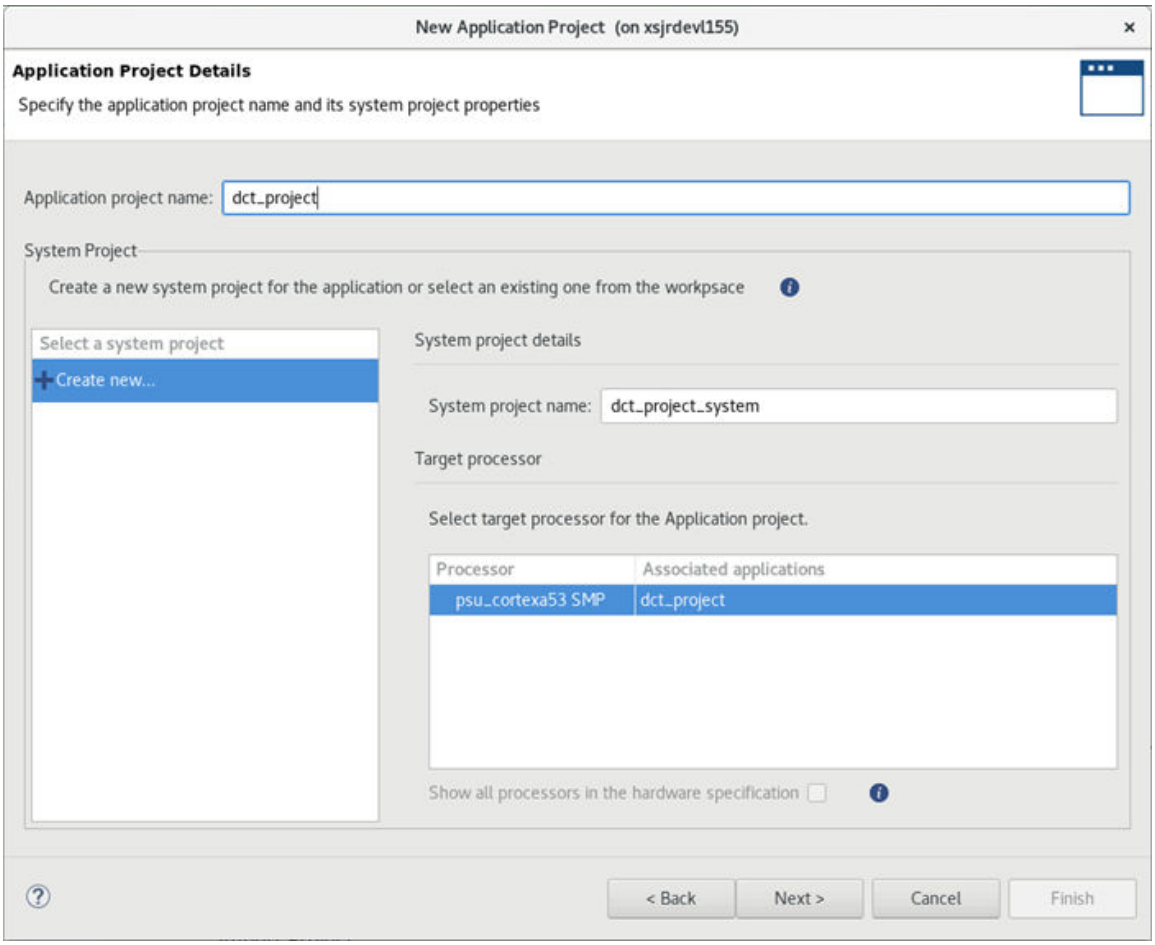
   目标平台由基本硬件设计与元数据组成，此元数据即为将加速器连接到声明的接口过程中所使用的元数据。使用“Select a platform from repository”选项卡即可为工程选择平台。您可在“Find”字段中输入值来限制显示的选项，以便找到所需的平台。底部会显示有关当前所选平台的信息，如下图所示。



注释：要了解特定版本支持的平台，请参阅 第一部分：Vitis 入门 中的版本说明。

您也可以将定制平台或第三方平台添加至存储库。如需了解更多信息，请参阅 管理平台和存储库。

3. 在“Application Project Details”页面中的“Application project name”字段中指定名称，如下图所示。

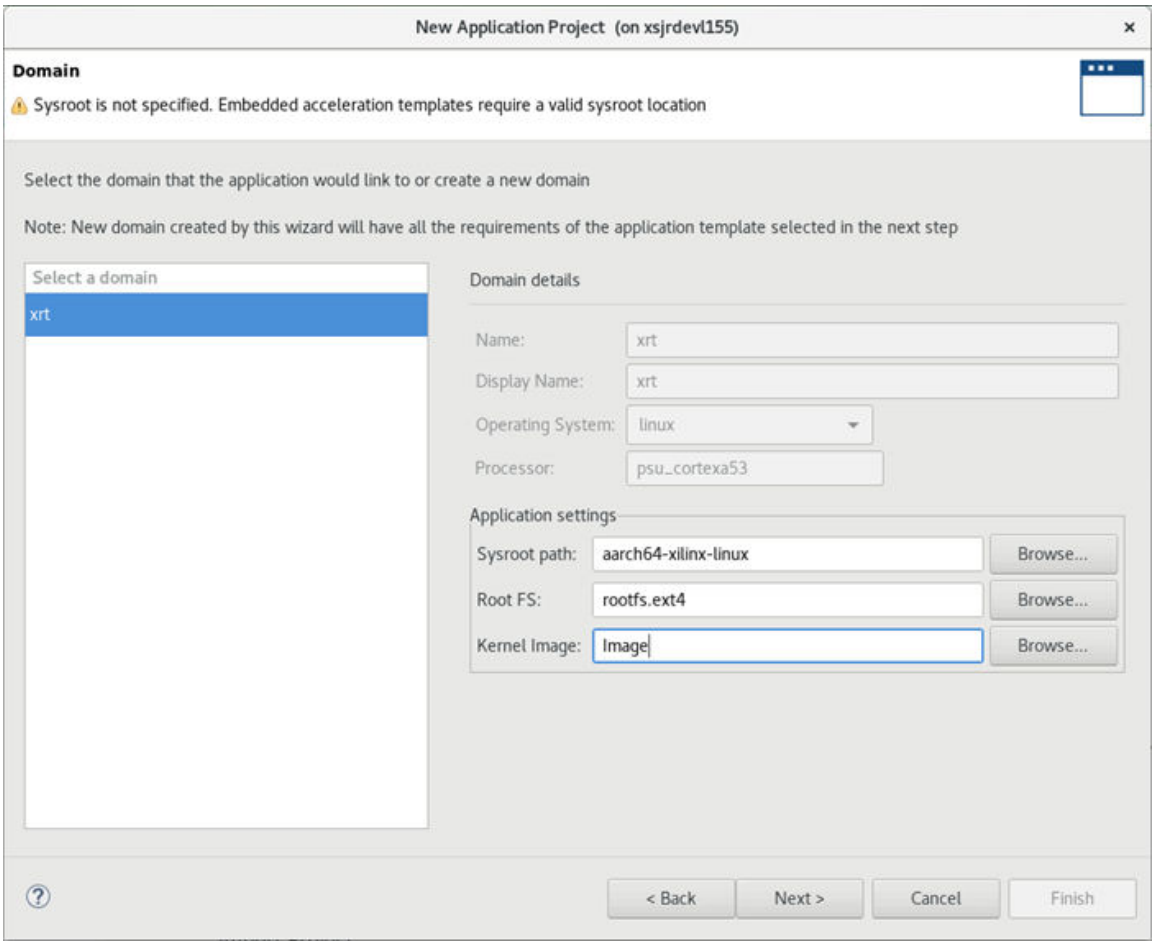


默认情况下，该工具会为您的应用工程创建新的系统工程。但是，您还可以将自己的应用工程添加到现有系统工程（如果存在）中。系统工程是不同工程的顶层管理器，通过组合这些不同工程即可创建系统视图。

4. 单击“Next”以继续。

   注释：如果您在步骤 2 中选择数据中心加速器卡作为工程平台，则不会显示以下页面，并且您可跳至步骤 6。

5. 如果您在“Platform”页面上选择“Embedded Acceleration”目标平台（如“Flow”列所示），则下一步会显示“Domain”页面，如下图所示。

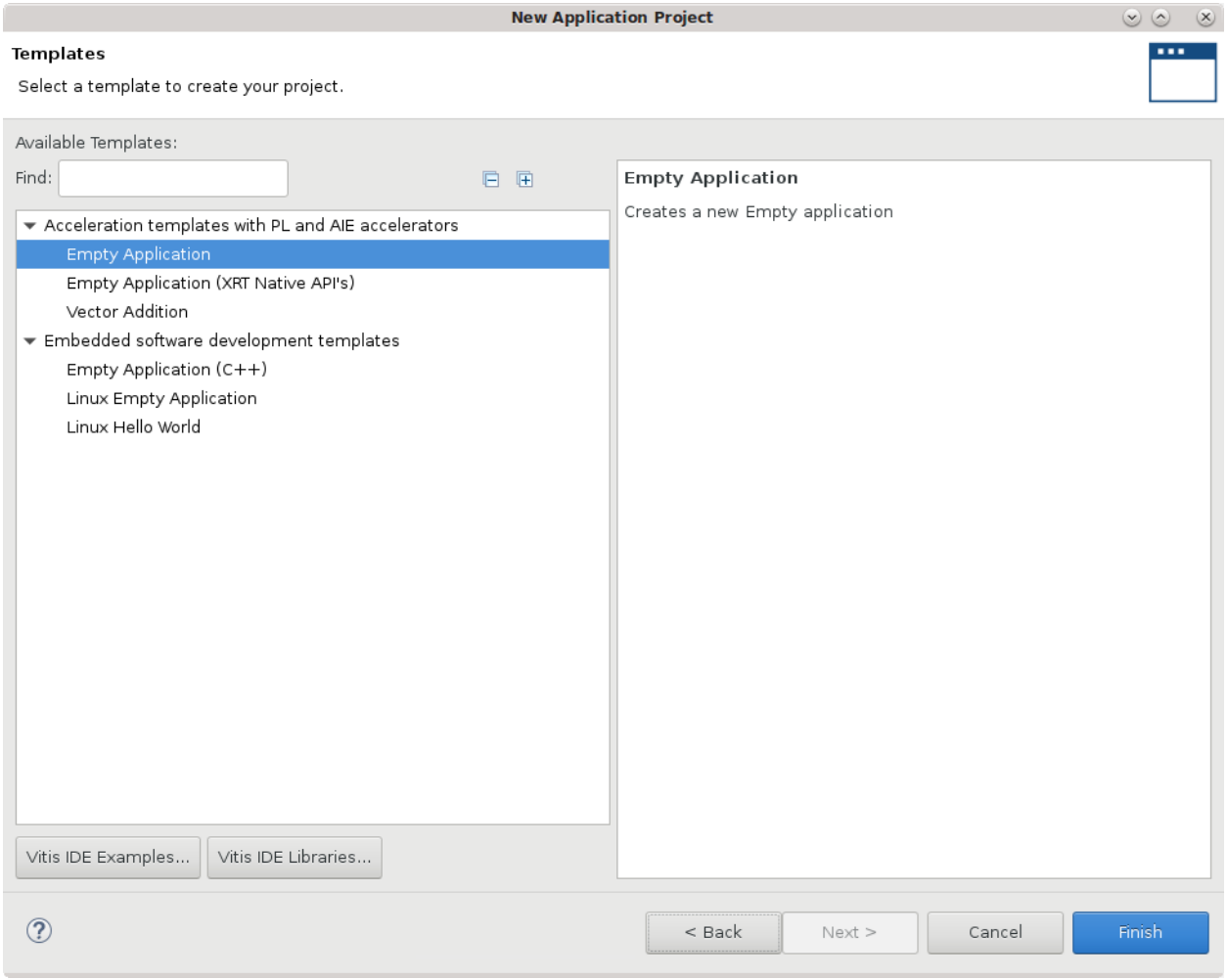


从平台上的现有域列表中选择“Domain”，这样即可根据您的选择来填充“Domain”详细信息。域可定义用于在目标平台上运行主机程序的处理器和操作系统。您还必须设置以下“Application Settings”以便在嵌入式平台上正确构建工程。

- “Sysroot path”：`sysroot` 是平台上用于定义基本信息根文件结构的部分。Sysroot 路径允许您为自己的应用定义新的 `sysroot`。
- “Root FS”：指定根文件系统的位置。
- “Kernel Image”：指定操作系统内核的位置。

从“Project Editor”窗口的“System Project Settings”创建工程后，可更改这些选项。

6. 单击“Next”可打开“Templates”页面， 以供您为自己的新工程选择应用加速模板。



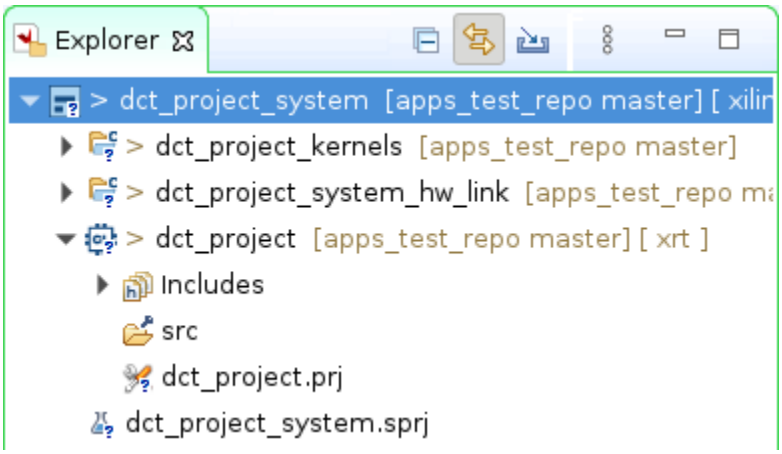
选择“Empty Application (XRT Native API)”即可创建空白工程以导入 XRT API 源文件（如 主机编程 中所述）并从头开始构建工程。并且您也可以使用提供的任一模板工程作为基础来创建新应用工程，这些模板工程可帮助您开始创建自己的工程，或者也可以帮助您学习该工具。

提示：单击“Vitis IDE Examples”按钮或者“Vitis IDE Libraries”按钮以安装其它示例，如 入门示例 中所述。

7. 单击“Finish”以关闭“New Application Project”Wizard 并在 IDE 中打开工程。

提示：Vitis IDE 会在“Design”透视图中打开，如 了解 Vitis IDE 中所述。如果您不熟悉其中显示的内容，请复查此信息。

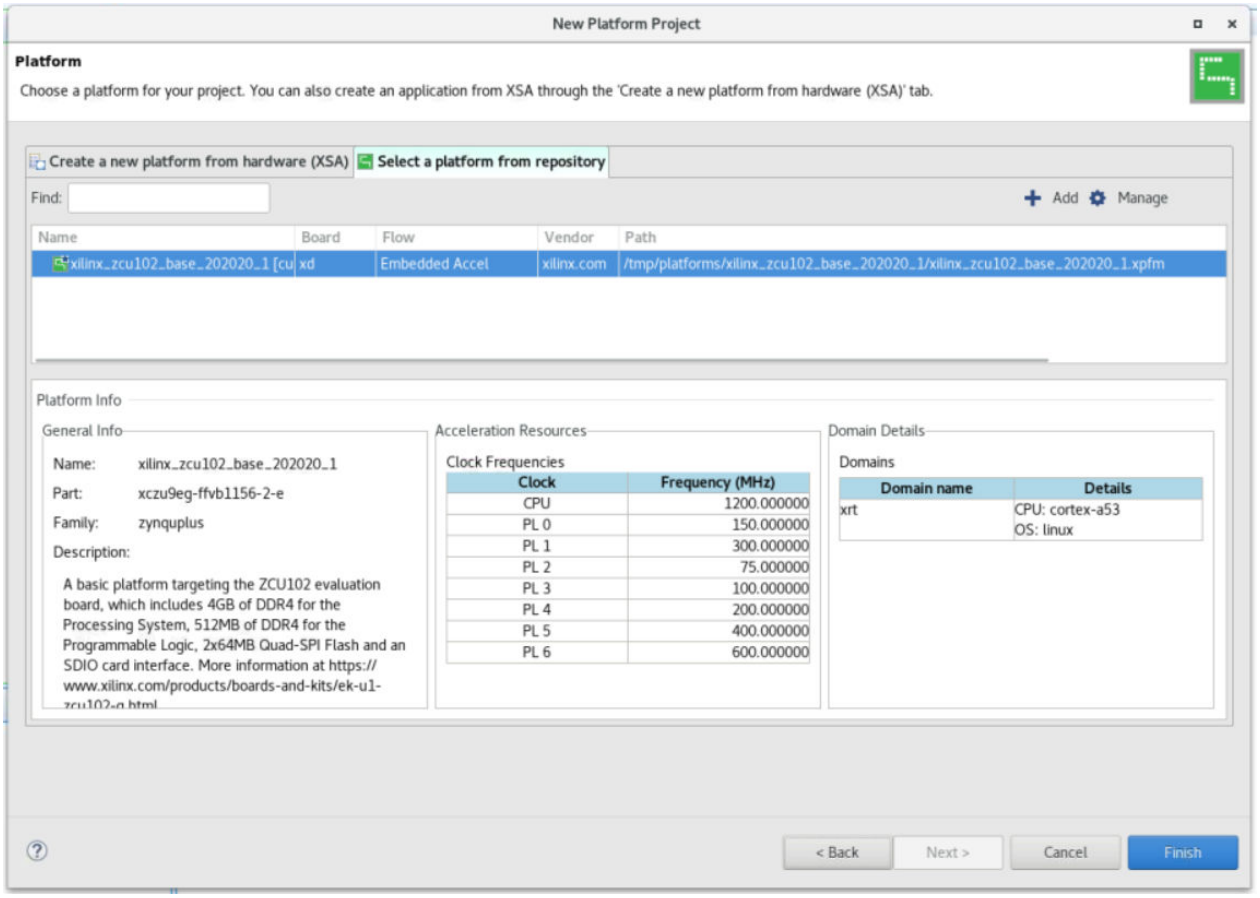
在 Vitis IDE 中创建新的应用加速工程时，它包含顶层系统工程（嵌套在应用工程内用于主机代码）、硬件内核工程（用于编译内核工程）和 `hw_link` 工程（用于将硬件内核链接到目标平台以及用于各硬件内核彼此间的链接）。这些工程显示在“Explorer”视图内，如下图所示。



管理平台和存储库

您可通过以下方式来管理可在 Vitis IDE 工程中使用的平台：在已打开的工程的主菜单中，使用“Xilinx” → “Add Custom Platform”，或者使用“New Application”Wizard 和“New Platform”Wizard 上的“Platform”页面。

图 94：**新建平台工程**



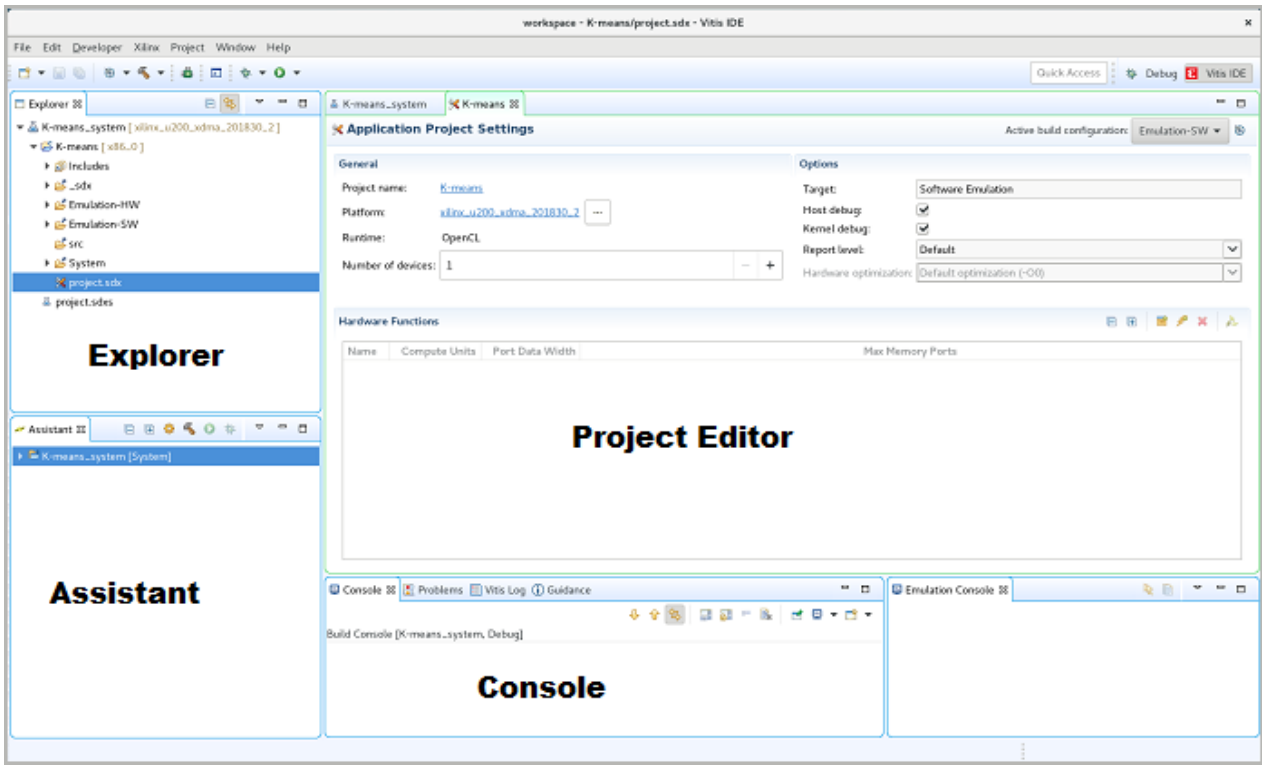
在“Platform”页面中，可使用以下任一选项来管理可用平台和平台存储库：

- Add ( )：将您自己的平台添加到可用平台列表中。要添加新平台，请导航到定制平台的顶层目录，选中该平台然后单击“OK”。这样即可从可用平台列表中选择定制平台。
- Manage ( )：添加或删除标准平台和定制平台。如果添加了定制平台，则新平台的路径将自动添加到存储库中。从存储库列表中移除平台后，在可用平台列表中将不再显示此平台。

了解 Vitis IDE

在 Vitis IDE 中打开工程时，工作空间中将排列显示一系列不同的视图和编辑器，这在基于 Eclipse 的 IDE 中也称为透视图。打开该工具时，将显示“设计 (Design)”透视图，如下图所示。

图 95：**Vitis IDE - 默认透视图**



默认透视图中的一些关键视图和编辑器包括：

- “Explorer”视图：显示工程文件夹及其关联源文件的面向文件的树视图，以及构建文件和工具生成的报告。您可使用该视图来浏览工程文件层级。
- “Assistant”视图：提供集中位置，用于查看和管理工作空间的工程，以及构建和运行工程配置。您可与不同配置的各种工程设置以及报告进行交互。在此视图中，您可以构建并运行自己的 Vitis IDE 应用工程，并启动 Vitis 分析器以查看报告和性能数据，，如 第六部分：使用 Vitis 分析器 中所述。
- “Project Editor”视图：显示当前工程、目标平台和处于活动状态的构建配置，并指定硬件函数；允许您直接编辑工程设置。
- “Console”视图：提供多个视图，包括命令控制台、设计指南、工程属性、日志和终端视图。

Vitis IDE 包含多个预定义的透视图，例如，Vitis IDE 透视图、“Debug”透视图和“Performance Analysis”透视图。要在不同透视图之间快速切换，请单击 Vitis IDE 右上角的透视图名称。

您可以通过将视图拖放到 IDE 中的新位置来排列视图以满足您的需求，视图排列方式会保存在当前透视图中。您可通过选择“View”选项卡上的“Close”按钮 (“X”) 来关闭窗口。您可使用“Window” → “Show View”命令并选择特定视图来打开新窗口。

要将透视图复原为默认视图排列方式，请将该透视图激活，并选择“Window” → “Reset Perspective”。

要打开其它透视图，请选择“Window” → “Open Perspective”。

添加源文件

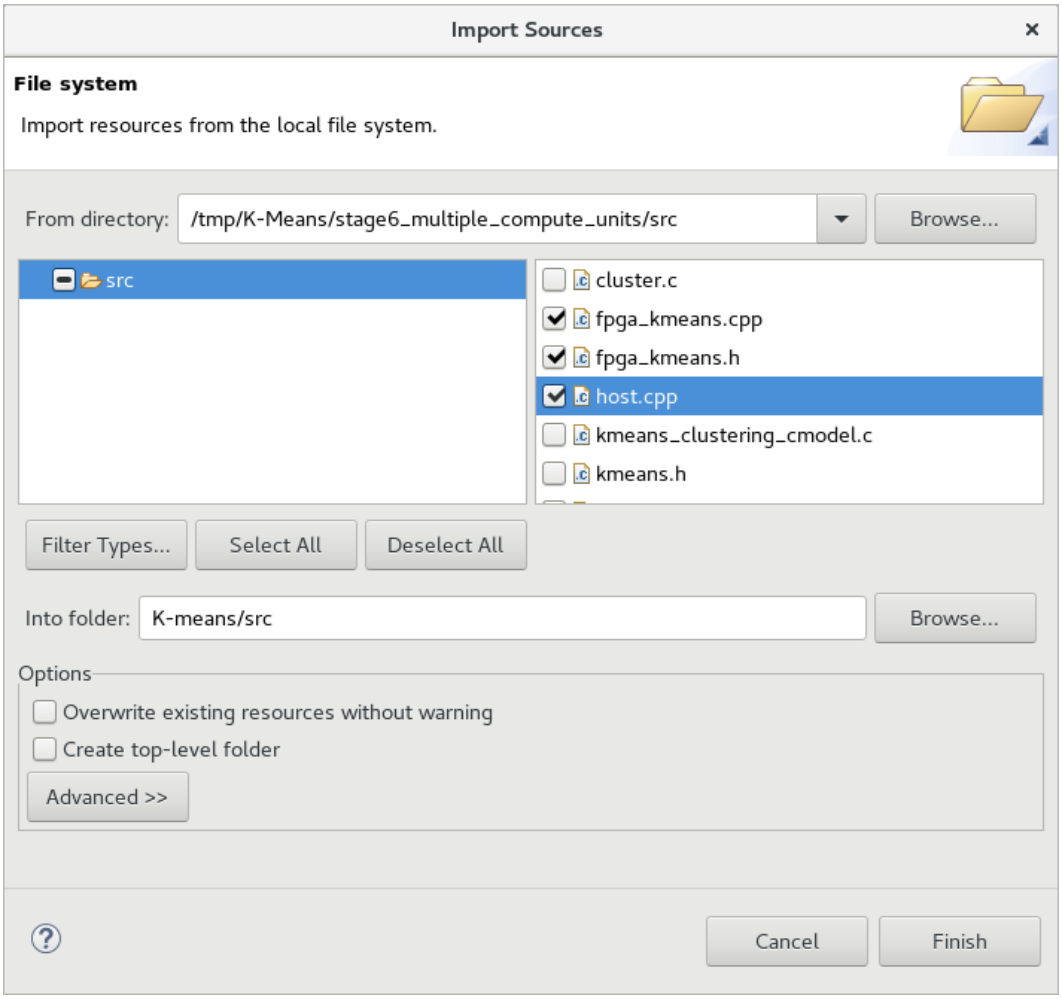
工程包含诸多不同的源文件，包括 主机编程 的 C/C++ 文件和头文件、C/C++ 内核、已编译的赛灵思对象 (XO) 文件（其中包含 RTL 内核，如 RTL 内核 中所述）或者 HLS 内核（如 使用 Vitis HLS 编译内核 中所述）。您可按需添加这些源文件以支持自己的应用。

系统工程内每个工程都需要其自己的源文件和数据文件。主机应用代码 (`host.cpp`) 添加到处理器应用工程下的 `./src` 文件夹内，内核代码 (`kernel.cpp`) 或已编译的 `kernel.xo` 文件则添加到内核应用工程的 `./src` 文件夹内。您可按下一节中所述方式，使用 “Import Sources” 命令添加这些文件。

添加源文件

1. 在 Vitis IDE 中打开工程后，要添加源文件，请右键单击“Project Explorer”中的 `src` 文件夹，然后单击“Import Sources”。

   这样即可显示“Import Sources”对话框，如下图所示。



2. 在此对话框中，对于“From directory”字段，请单击“Browse”按钮以选择含要导入的源文件的目录。
3. 在“Into folder”字段中，请确保指定的文件夹为工程的 `src` 文件夹。
4. 启用相应文件名旁的复选框以选中所需源文件，然后单击“Finish”。

> **重要提示！**将源文件导入工作空间时，会将文件复制到工作空间中。如果删除工作空间，对文件的任何更改都将丢失。

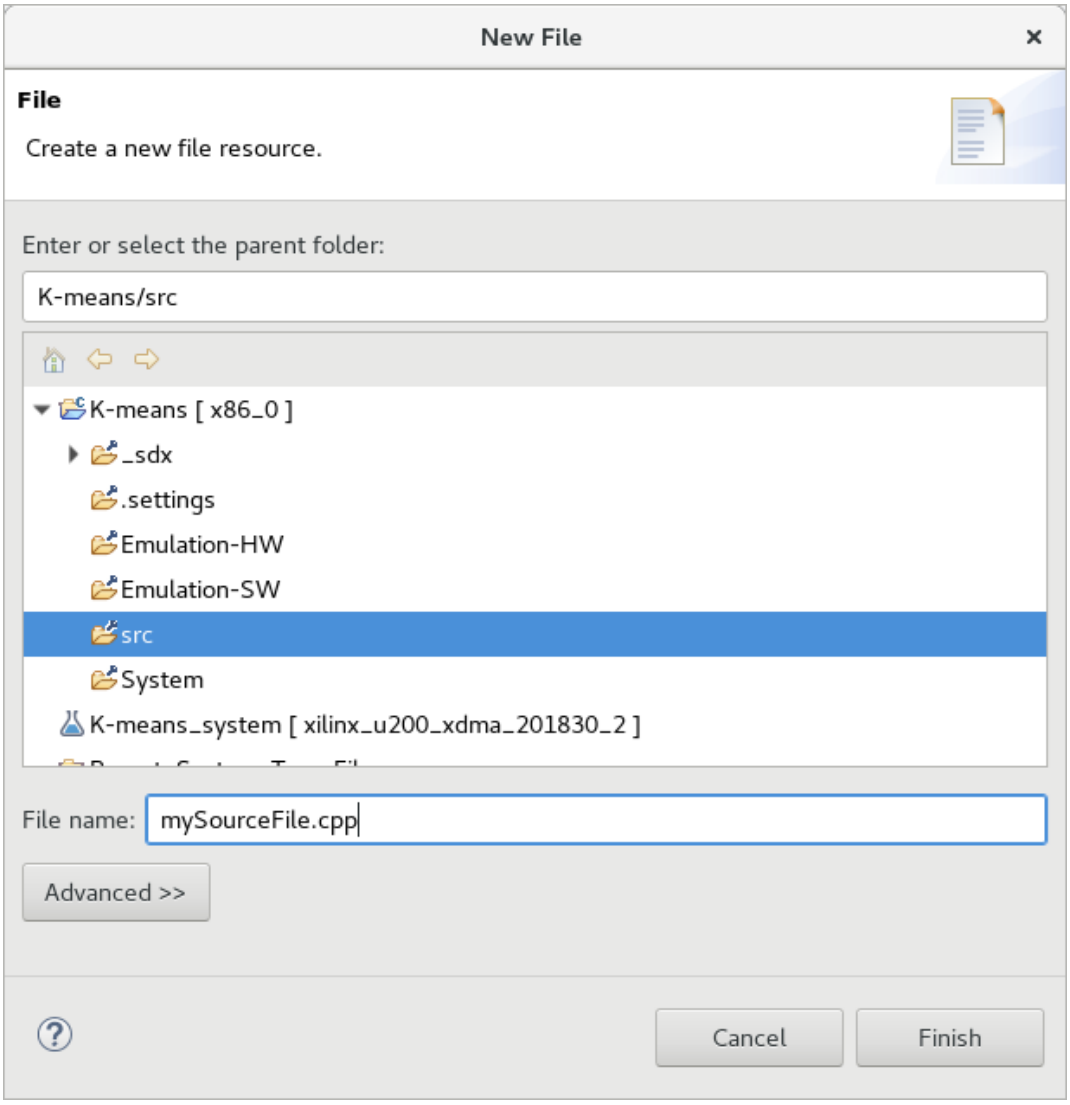
将源文件添加到工程后，您即可开始配置、构建和运行应用。要在内置文本编辑器中打开源文件，请展开“Project Explorer”中的 `src` 文件夹，然后双击特定文件。

创建和编辑新的源文件

您也可以在 Vitis IDE 中直接创建和编辑新的源文件。

1. 在已打开的工程中，右键单击 `src` 文件夹，然后依次选中“New” → “File”。

   这样会显示“New File”对话框，如下图所示。



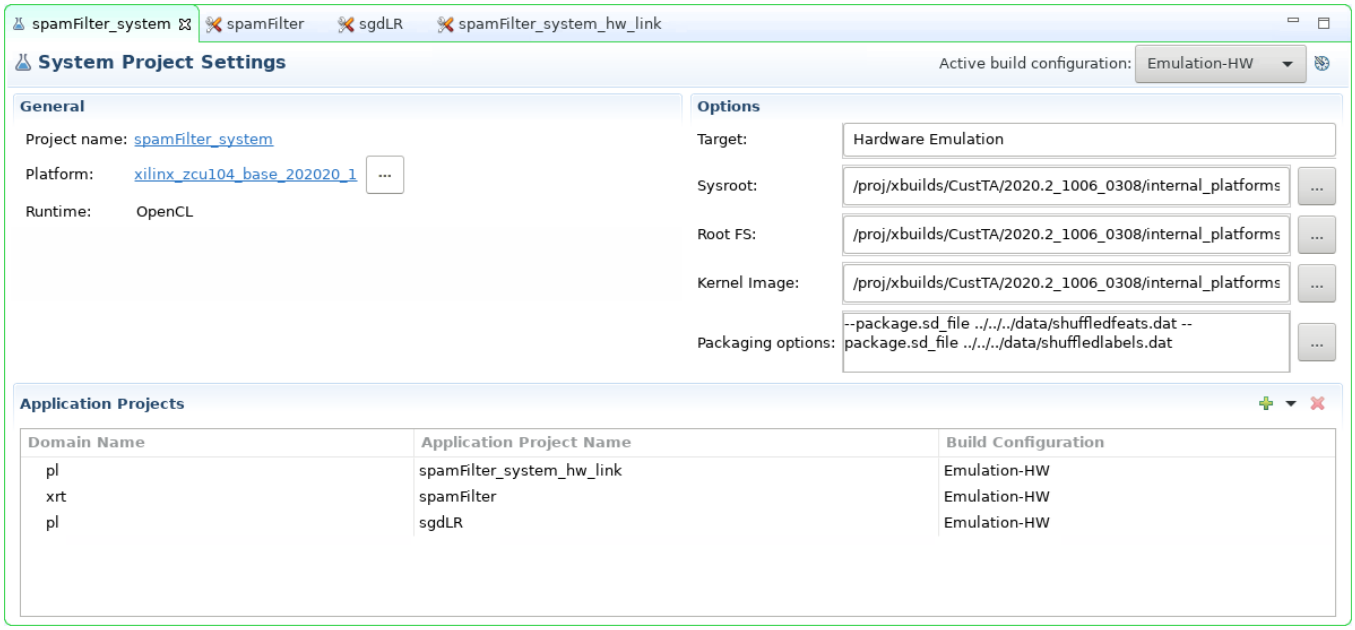
2. 选择要在其中创建新文件的文件夹，然后输入文件名。
3. 单击“Finish”，将文件添加到工程。

将源文件添加到工程后，您即可开始配置、构建和运行应用。要在内置文本编辑器中打开源文件，请展开“Project Explorer”中的 `src` 文件夹，然后双击特定文件。

使用“Project Editor”视图

构建系统需要编译主机程序和 FPGA 二进制文件 (`xclbin`) 并将两者链接在一起。您定义的应用工程包含顶层系统工程、主机处理器工程、硬件内核工程和 `hw_link` 工程。主机工程和内核工程包含源代码，这些源代码是在工程中导入或创建的，位于 `src` 文件夹内。这些工程均可在“Project Editor”视图中打开，如下图所示，该视图提供了工程及其各种构建配置的顶层视图。

图 96：**“Project Editor”视图**



根据您当前查看的工程类型（系统工程、主机、内核或链接），“Project Editor”视图可提供下列详细信息：

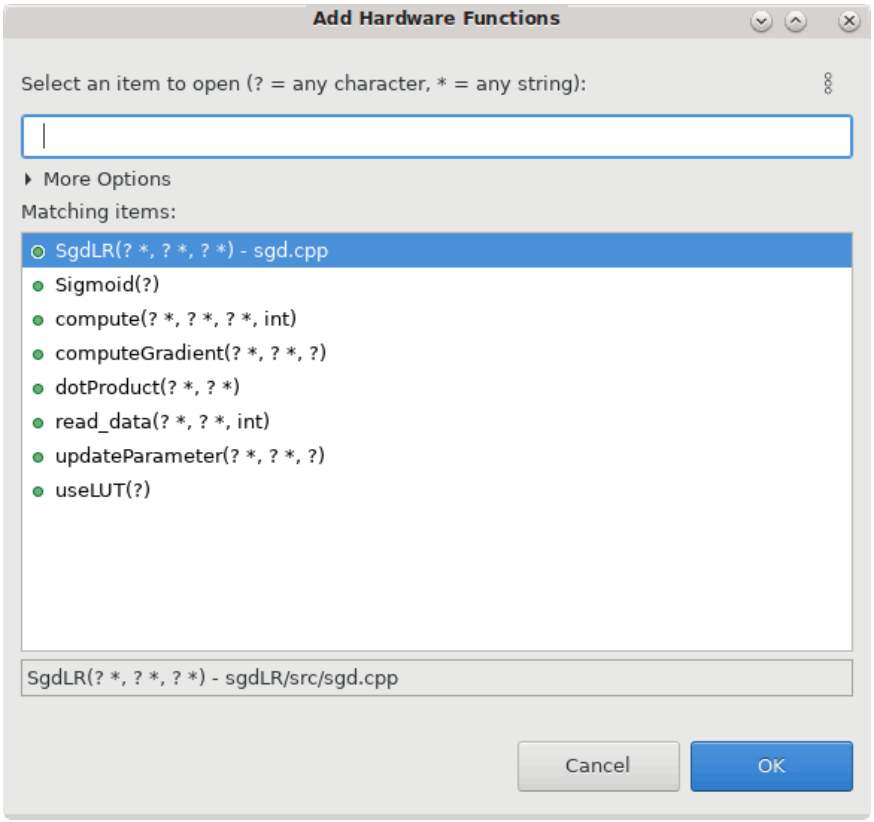
- 有关工程名称的常规信息
- 目标平台
- 当前处于活动状态的构建配置
- 与所选工程相关的多项配置选项

这些信息包括系统工程的启动文件、主机或内核工程的调试选项以及用于选择硬件内核工程的报告级别的菜单（如 控制报告生成 中所述）。

“Project Editor”视图底部显示了顶层系统工程中包含的“Application Projects”（如上图所示），或者显示“Hardware Functions”，这些硬件函数将在硬件内核工程中进行编译，或者分配到 `hw_link` 工程中的二进制容器以供构建到 `xclbin` 内。

请单击位于“Hardware Functions”窗格右上角的“Add Hardware Function”按钮（  ），以指定要在硬件内核工程内编译的函数。这样即可打开“Add Hardware Functions”对话框，以显示当前工程的源代码中定义的函数列表，如下所示。

图 97：**将硬件函数添加至二进制容器**

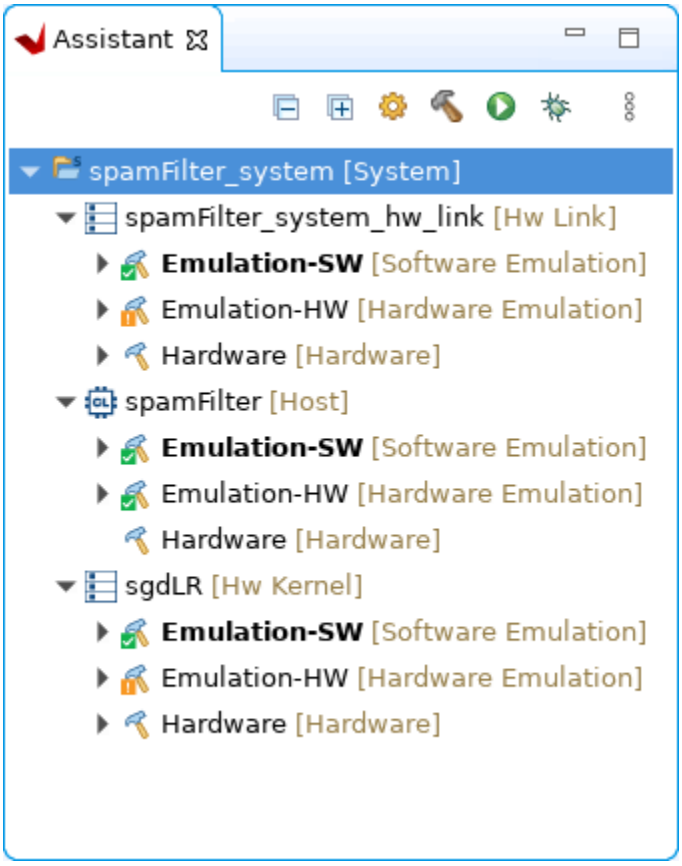


从列表中选择一项函数以指定硬件函数，然后单击“OK”。选定的函数将成为硬件内核工程的构建进程目标，并添加到 `hw_link` 工程中的器件二进制文件中。

使用助手视图

“Assistant”视图可提供工程树，用于管理构建配置、运行配置并设置这些配置的属性。该视图与“Explorer”视图搭配，在默认 Vitis IDE 透视图下，“Assistant”视图可直接显示在“Explorer”视图下方。下图显示了“Assistant”视图及其树结构的示例。

图 98：**“Assistant”视图**



“Assistant”视图层级中显示的对象包含顶层系统工程、主机工程、硬件内核工程和 `hw_link` 工程。对于其中每个工程，还会显示不同的构建配置：软件仿真和硬件仿真构建配置以及硬件构建配置。构建配置用于按 构建目标 中所述来定义构建目标，并指定编译和链接进程的选项。

提示：通过上图显示的图标，即可快速确定构建配置的状态：
- 完成的“Emulation-SW”构建以绿色复选框来表示
- 对于硬件内核和 `hw_link` 工程，需要更新的“Emulation-HW”构建以黄色感叹号 (!) 来表示
- “Hardware”构建未进行构建

选择构建配置（例如，“Emulation-HW”）并单击“Settings”图标 ( ) 后，会打开 Vitis 构建配置设置 对话框。您将使用此“Settings”对话框来为特定仿真 (emulation) 或硬件目标配置构建进程。

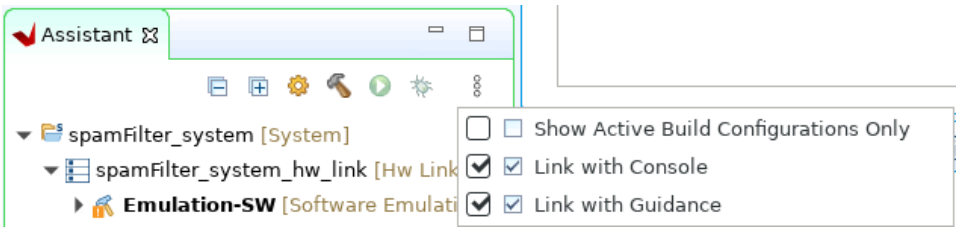
在这些构建配置的层级内包含二进制容器（或 `.xclbin`）、硬件函数、运行配置以及通过构建或运行进程生成的所有报告或汇总信息。为特定构建配置选中硬件函数并单击“Settings”图标后，会显示 Vitis 硬件函数设置 对话框。您将使用此对话框来指定每个内核的计算单元数量、将计算单元分配给 SLR 并向全局存储器分配内核端口。

构建完成特定构建配置后，启动配置即变为可供此工程使用。启动配置分为两种类型：

- 运行配置可指定用于运行已编译和已链接的应用的剖析数据，它可定义用于运行主机应用和内核代码的环境和选项。使用工具栏菜单上的 Run 命令 ( ) 可访问运行配置。这样可打开 Vitis 运行和调试配置设置 对话框，以供您在其中先配置运行然后再将其启动。

- 调试配置可指定用于调试应用的剖析数据。它会启动对主机和内核代码进行交互式调试所需的环境。您可通过工具栏菜单上的 Debug 命令 ( ) 来访问调试配置。如需了解更多信息，请参阅 Vitis IDE 调试流程。

图 99： **“Assistant”视图菜单**

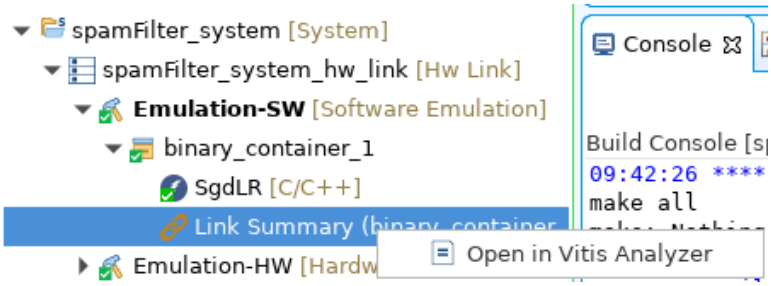


在“Assistant”视图内，“View”菜单包含影响“Assistant”视图显示内容的选项或者影响“Assistant”视图与其它视图的交互方式的选项。左键单击菜单命令打开“View”菜单即可显示以下选项：

- “Show Active Build Configurations Only”：启用该选项时，“Assistant”视图将仅显示每个工程的处于活动状态的构建配置。该选项可用于使“Assistant”视图保持简洁。要更改活动配置，请选中“工程编辑器 (Project Editor)”视图中的“Active build configuration”。
- “Link with Console”：启用该选项时，“Console”视图中的构建控制台会自动切换以匹配“Assistant”视图中当前选中的构建配置。如果不启用该选项，则构建控制台不会自动更改以匹配“Assistant”视图。
- “Link with Guidance”：启用该选项时，“Console”视图的“Guidance”选项卡会自动切换以匹配“Assistant”视图中当前所选配置。

对于每个构建配置，构建和运行进程期间都会生成报告，并且这些报告会显示在“Assistant”视图中。不同报告会分组为“Compile Summary”、“Link Summary”、“Package Summary”和“Run Summary”，可在 Vitis 分析器工具中查看这些报告，如 第六部分：使用 Vitis 分析器 中所述。您可以右键单击“Assistant”视图中的上述任一汇总报告，并选择“Open in Vitis Analyzer”，如下所示。

图 100：**在 Vitis 分析器中打开**



Vitis IDE 的输出目录

以下示例演示了由 Vitis IDE 为应用加速工程样本自动生成的目录结构，其中包括：

- 顶层系统工程
- 主机应用工程
- 硬件内核编译工程

· 硬件内核链接工程

Vitis IDE 的默认目录结构与命令行流程创建的目录结构相似，但不完全相同。

```
### Vitis IDE Directory Structure
workspace
    >pathwayKernel - a hardware kernel project in the Vitis application
acceleration development flow. The top-level project can contain multiple
kernel projects.
        >Emulation-HW - the hardware emulation build folder.
            >build
                > The build folder for the compiled hardware kernel (.XO)
            >guidance.html
            >guidance.json
            >guidance.pb
            >makefile
            >mmult-compile.cfg
            >pathwayKernel_Emulation-HW.build.ui.log
            >xcd.log
        >Emulation-SW
            >Files related to the Software Emulation Build
        >pathwayKernel.prj - Hardware kernel project
        >src - Hardware kernel source files
            >kernel.cpp
    >pathwayTest - The host application project
        >Emulation-HW
            >au250_image.jpg
            >bd_d216_ddr4_mem00_0_microblaze_mcs_bd.tcl
            >bd_d216_ddr4_mem00_0_microblaze_mcs.hwh
            >bd_d216_ddr4_mem01_0_microblaze_mcs_bd.tcl
            >bd_d216_ddr4_mem01_0_microblaze_mcs.hwh
            >bd_d216_ddr4_mem02_0_microblaze_mcs_bd.tcl
            >bd_d216_ddr4_mem02_0_microblaze_mcs.hwh
            >bd_d216_ddr4_mem03_0_microblaze_mcs_bd.tcl
            >bd_d216_ddr4_mem03_0_microblaze_mcs.hwh
            >bd_d216_interconnect_DDR4_MEM00_0_bd.tcl
            >bd_d216_interconnect_DDR4_MEM00_0.hwh
            >bd_d216_interconnect_DDR4_MEM01_0_bd.tcl
            >bd_d216_interconnect_DDR4_MEM01_0.hwh
            >bd_d216_interconnect_DDR4_MEM02_0_bd.tcl
            >bd_d216_interconnect_DDR4_MEM02_0.hwh
            >bd_d216_interconnect_DDR4_MEM03_0_bd.tcl
            >bd_d216_interconnect_DDR4_MEM03_0.hwh
            >bd_d216_interconnect_PLRAM_MEM00_0_bd.tcl
            >bd_d216_interconnect_PLRAM_MEM00_0.hwh
            >bd_d216_interconnect_PLRAM_MEM01_0_bd.tcl
            >bd_d216_interconnect_PLRAM_MEM01_0.hwh
            >bd_d216_interconnect_PLRAM_MEM02_0_bd.tcl
            >bd_d216_interconnect_PLRAM_MEM02_0.hwh
            >bd_d216_interconnect_PLRAM_MEM03_0_bd.tcl
            >bd_d216_interconnect_PLRAM_MEM03_0.hwh
            >bd_d216_interconnect_S00_AXI_0_bd.tcl
            >bd_d216_interconnect_S00_AXI_0.hwh
            >bd_d216_interconnect_S01_AXI_0_bd.tcl
            >bd_d216_interconnect_S01_AXI_0.hwh
            >bd_d216_interconnect_S02_AXI_0_bd.tcl
            >bd_d216_interconnect_S02_AXI_0.hwh
            >bd_d216_interconnect_S03_AXI_0_bd.tcl
            >bd_d216_interconnect_S03_AXI_0.hwh
            >binary_container_1.xclbin
            >board
                >board files for the selected hardware platform
            >dsa.xml
```

```
            >emconfig.json
            >emu
                >dynamic_post_sys_link.tcl
                >dynamic_pre_sys_link.tcl
                >emu.bd
                >emu.bxml
                >emu.xml
                >hdl
                    >emu_wrapper.v
                >ip
                    >IP files from the selected platform
                >ipshared
                    >
                >prop.json
                >sim
                    >
                >synth
                    >
            >emulation_debug.log
            >ext_metadata.json
            >firmware
                >xilinx_u250_xdma_201830_3.mcs
                >xilinx_u250_xdma_201830_3.prm
            >guidance.html
            >guidance.json
            >guidance.pb
            >launch_options.cfg
            >makeemconfig.mk
            >makefile
            >pathwayTest
            >pathwayTest_Emulation-HW.build.ui.log
            >pfm_dynamic_bd.tcl
            >pfm_dynamic_debug_bridge_xsdbm_0_bd.tcl
            >pfm_dynamic_debug_bridge_xsdbm_0.hwh
            >pfm_dynamic.hwh
            >pfm_dynamic_memory_subsystem_0_bd.tcl
            >pfm_dynamic_memory_subsystem_0.hwh
            >pfm_top_bd.tcl
            >pfm_top.hwh
            >pfm_top_jtag_fallback_0_bd.tcl
            >pfm_top_jtag_fallback_0.hwh
            >pfm_top_mgmt_debug_bridge_0_bd.tcl
            >pfm_top_mgmt_debug_bridge_0.hwh
            >pfm_top_mgmt_debug_hub_0_bd.tcl
            >pfm_top_mgmt_debug_hub_0.hwh
            >pfm_top_user_debug_bridge_0_bd.tcl
            >pfm_top_user_debug_bridge_0.hwh
            >pfm_top_user_debug_hub_0_bd.tcl
            >pfm_top_user_debug_hub_0.hwh
            >src
                >host.cpp
            >sysdef.xml
            >SystemDebugger_pathwayTest_system_pathwayTest
                >binary_container_1.xclbin.run_summary
                >guidance.html
                >guidance.json
                >guidance.pb
                >profile_guidance.json
                >profile_guidance.pb
                >profile_kernels.csv
                >opencl_summary.csv
                >profile_summary.xprf
                >timeline_kernels.csv
```

```
                >opencl_trace.csv
                >xilinx_u250_xdma_201830_3-0-binary_container_1.protoinst
                >xilinx_u250_xdma_201830_3-0-binary_container_1_simulate.log
                >xilinx_u250_xdma_201830_3-0-binary_container_1.wcfg
                >xilinx_u250_xdma_201830_3-0-binary_container_1.wdb
                >xilinx_u250_xdma_201830_3-0-
binary_container_1_xsc_report.log
                >xrc.log
                >xrt.ini
            >SystemDebugger_pathwayTest_system_pathwayTest.launch.log
            >SystemDebugger_pathwayTest_system_pathwayTest.launch.ui.log
            >tcl_hooks
                >dynamic_postlink.tcl
                >dynamic_postopt.tcl
                >dynamic_prelink.tcl
                >xilinx_u250_xdma_201830_3_dynamic_impl.xdc
            >update_dsa.log
            >xilinx_u250_xdma_201830_3.hpfm
        >Emulation-SW
        >pathwayTest.prj
        >src
            >kernel.cpp
    >pathwayTest_system
        >binary_container_1.xclbin
        >binary_container_1.xclbin.package_summary
        >makefile
        >package.build
            >logs
            >package
                >system diagram and package project
            >reports
                >package
                    >v++_package_binary_container_1_guidance.html
            >v++_package_binary_container_1_guidance.json
            >v++_package_binary_container_1_guidance.pb
        >package.cfg
        >pathwayTest_system_Emulation-HW.build.ui.log
        >v++_binary_container_1.log
        >xcd.log
        >xrc.log
    >pathwaytTest_system_hw_link
        >binary_container_1.build
            >link
                >activetask.json
                >int
                    >address_map.xml
                    >appendSection.rtd
                    >behav_waveform
                        >xsim
                    >behav.xse
                    >binary_container_1_build.rtd
                    >binary_container_1.gpp_so.log
                    >binary_container_1.rtd
                    >binary_container_1.so
                    >binary_container_1.xml
                    >binary_container_1_xml.rtd
                    >cf2sw_full.rtd
                    >cf2sw.rtd
                    >consolidated.cf
                    >debug_ip_layout.rtd
                    >dr.bd.tcl
                    >kernel_info.dat
                    >_kernel_inst_paths.dat
```

```
                    >kernel_service.json
                    >mmult
                        >
                    >_new_clk_freq
                    >sdsl.dat
                    >syslinkConfig.ini
                    >systemDiagramModel.json
                    >systemDiagramModelSlrBaseAddress.json
                    >vplConfig.ini
                    >vplsettings.json
                    >xclbin_orig.1.xml
                    >xclbin_orig.xml
                    >xclbin_orig.xml.tmp
                    >xo
                        >ip_repo
                        >mmult
                            >cpu_sources
                            >debug
                            >kernel.xml
                            >mmult.design.xml
                >link.spr
                >link.steps.log
                >run_link

                >sys_link

                >vivado

            >logs
                >link
            >reports
                >link
        >binary_container_1-link.cfg
        >binary_container_1.mdb
        >binary_container_1.xclbin
        >binary_container_1.xclbin.info
        >binary_container_1.xclbin.link_summary
        >binary_container_1.xclbin.sh
        >guidance.html
        >guidance.json
        >guidance.pb
        >makefile
        >pathwayTest_system_hw_link_Emulation-HW.build.ui.log
        >xcd.log
```

构建系统

构建系统时，最好使用 构建目标 中所述的 3 个可用构建目标。每个构建目标在“Assistant”视图中都呈现为一个独立的构建配置。请按下列顺序处理这些构建配置：

- “Emulation-SW”：针对软件仿真 (`sw_emu`) 进行构建，以确认主机程序与内核代码的算法功能能够协同工作。
- “Emulation-HW”：针对硬件仿真 (`hw_emu`) 进行构建，以将内核编译为硬件描述语言 (HDL)、确认生成的逻辑正确与否，并评估其仿真性能。
- “Hardware”：执行系统硬件构建 (`hw`) 以实现在目标平台上运行的应用。

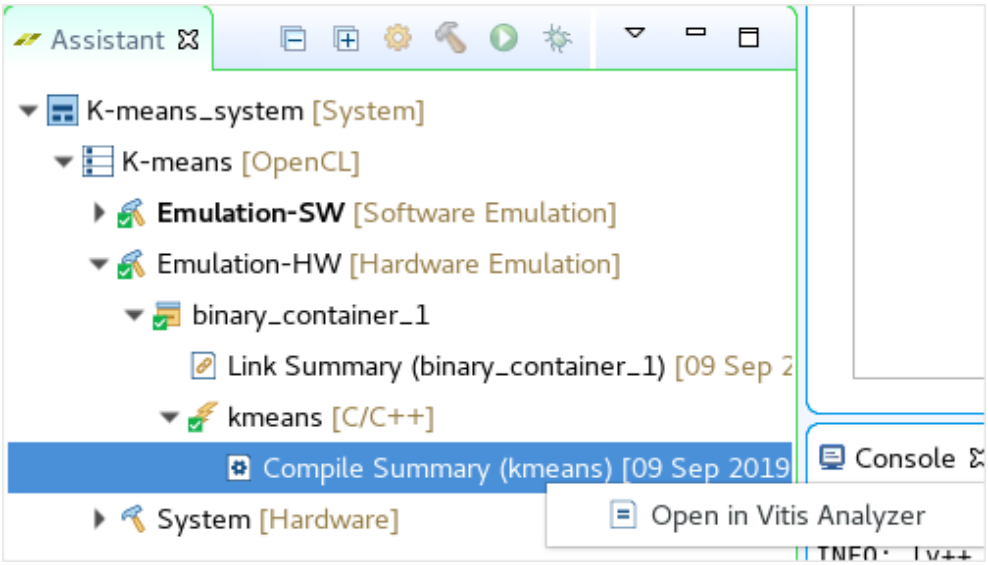
在启动构建命令前，请配置每个构建配置，以确保它满足您的需求。选中特定构建配置，单击“Settings”图标以打开“Build Configuration Settings”对话框。如需了解有关使用此对话框的更多信息，请参阅 Vitis 构建配置设置。

除构建配置设置外，在通过“Vitis Hardware Function Settings”对话框访问的“硬件函数 (Hardware Function)”中，包含了将影响应用的诸多设置。建议最好复查每个“Settings”对话框，如 配置 Vitis IDE 中所述。

在“Assistant”视图中，您可利用指定构建配置的各选项来启动构建进程，具体方法是选中构建配置，然后单击“Build”( ) 按钮。Vitis 核开发套件使用两部分组成的构建进程，通过 Vitis™ 编译器 `v++` 命令，为硬件内核生成 FPGA 二进制文件 (`.xclbin`)，并使用 `g++` 编译器来编译和链接主机程序代码。

构建进程完成后，“Assistant”视图会显示特定构建配置，此配置包含绿色复选标记，用于表示它已成功完成构建，如下图所示。您可以打开任一构建报告，例如，“Hardware Function”中的“编译汇总 (Compile Summary)”报告，或二进制容器中的“链接汇总 (Link Summary)”报告。右键单击“Assistant”视图中的报告，然后单击“Open in Vitis Analyzer”。

图 101：**助手视图 - 成功构建**



构建完成后，您即可在特定构建配置所提供的环境内运行应用。例如，在 Emulation-SW 构建中实践主机程序与 FPGA 二进制文件协同工作的 C 语言模型，或者在 Emulation-HW 构建的仿真中复查主机程序和 RTL 内核代码，或者在 Hardware 构建中于目标平台上运行应用。

要在 Vitis IDE 内运行应用，请选中构建配置，然后单击“Run”按钮 ( ) 以启动默认运行配置。您也可以右键单击构建配置并使用“Run”菜单来选择特定运行配置，或者按 Vitis 运行和调试配置设置 中所述方式编辑运行配置。

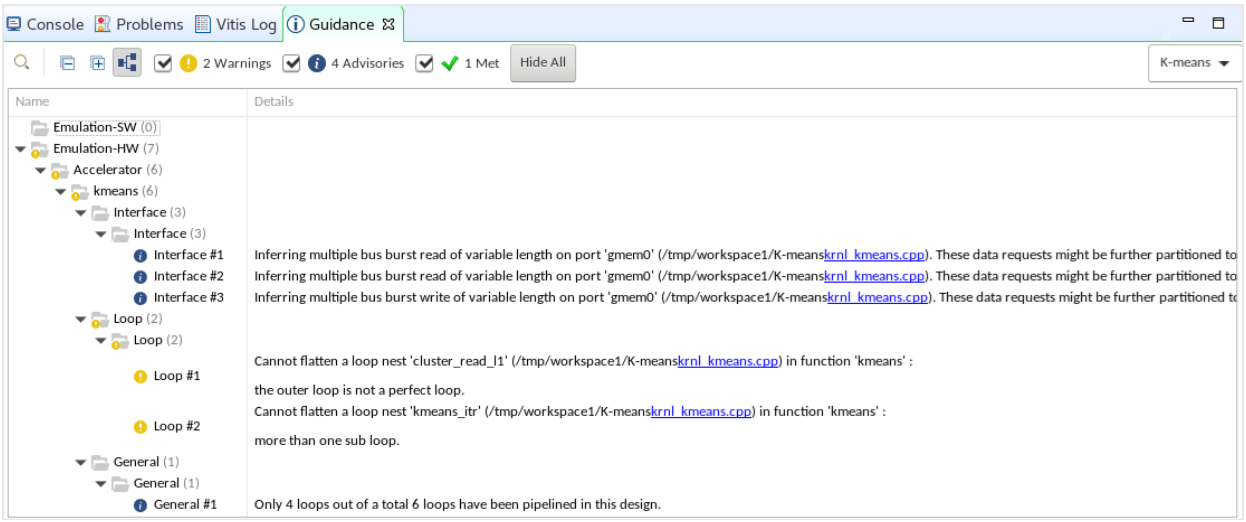
提示：Vitis IDE 会创建一个文件夹，此文件夹根据当前运行的特定构建配置中的运行配置来命名。例如，`./project/Emulation-HW/run_config`。应用运行的输出文件和日志将写入此文件夹。传递到主机程序的所有实参都应按与该文件夹的相对关系来写入。

Vitis IDE 的 Guidance 视图

构建或运行特定构建配置后，“Console”视图的“Guidance”选项卡就会显示与构建和运行进程相关的错误、警告和建议列表。在“Console”视图的选项卡中会自动填充并显示“Guidance”视图。您可复查其中的指南消息，并在代码或构建进程中进行任何必要的更改。

运行硬件仿真后，“Guidance”视图如下所示。

图 102：**构建指南**



提示：在 Vitis 分析器中也可以查看“Guidance”报告，如 第六部分：使用 Vitis 分析器 中所述。

为了简化“Guidance”视图信息排序，Vitis IDE 允许您搜索并筛选“Guidance”视图以查找特定指南规则条目。您可以折叠或展开树视图，甚至可以禁止显示分层树图并可视化指南规则的简明视图。最后，您可以选择“Guidance”视图中显示的内容，例如，可启用（显示）或禁用（不显示）警告和已满足的规则，并基于消息来源（例如，构建和仿真）来限制特定内容。

默认情况下，“Guidance”视图会显示下拉列表中选中的工程的所有指南信息。要将所示内容限制为单个构建或运行步骤，请执行以下操作：

1. 依次选中“Window” → “Preferences”

2. 选中“Guidance”类别。
3. 取消选中“Group guidance rule checks by project”。

在 Vitis IDE 中使用 Vivado 工具

Vitis 核开发套件在链接进程中会调用 Vivado Design Suite，以便在生成 FPGA 二进制文件 (`.xclbin`) 时自动运行 RTL 综合与实现。您也可以选择直接在 Vitis IDE 内启动 Vivado 工具，与工程进行交互以便对 FPGA 二进制文件执行综合与实现。有 3 条命令可用于支持通过 Vitis IDE 与 Vivado 工具进行交互，这些命令可通过“Xilinx” → “Vivado Integration”菜单来访问：

提示：在 IDE 中必须打开 `hw_link` 工程，并且必须将其作为当前工程才能使用这些选项：

- “Open Vivado Project”：这样即可自动打开与硬件构建配置关联的 Vivado 工程 (.xpr)。为了正常使用该功能，您必须先前已完成硬件构建，这样 Vivado 工程才会存在并且可用于构建。

  打开 Vivado 工程会启动 Vivado IDE 并打开实现设计检查点 (DCP) 文件以编辑该工程，并允许您更为直接地管理综合与实现的结果。随后，您即可在此基础上，通过选择“Import Design Checkpoint”来生成 FPGA 二进制文件。
- “Import Design Checkpoint”：允许您指定 Vivado DCP 文件，用作为硬件构建的基础以及用于生成 FPGA 二进制文件。
- “Import Vivado Settings”：允许您指定 Vivado 工具所使用的配置文件（如 Vitis 编译器配置文件 中所述）以供在链接进程中使用。

在独立模式下使用 Vivado IDE 可以利用各种综合和实现选项来进一步最优化内核的性能和区域。还有其它选项可供您用于与 FPGA 构建进程进行交互。如需了解更多信息，请参阅 管理 Vivado 综合与实现结果。

重要提示！在独立工程中所应用的最优化开关并不会被自动整合回 Vitis IDE 构建配置中。您需要使用 `v++ --config` 文件选项来确保为构建指定各种综合与实现属性。如需了解更多信息，请参阅 Vitis 编译器命令。

Vitis IDE 调试流程

Vitis™ IDE 支持轻松访问调试功能。手动执行时，设置用于调试的可执行文件需经历多个步骤。使用调试流程时，这些步骤由 Vitis IDE 自动处理。

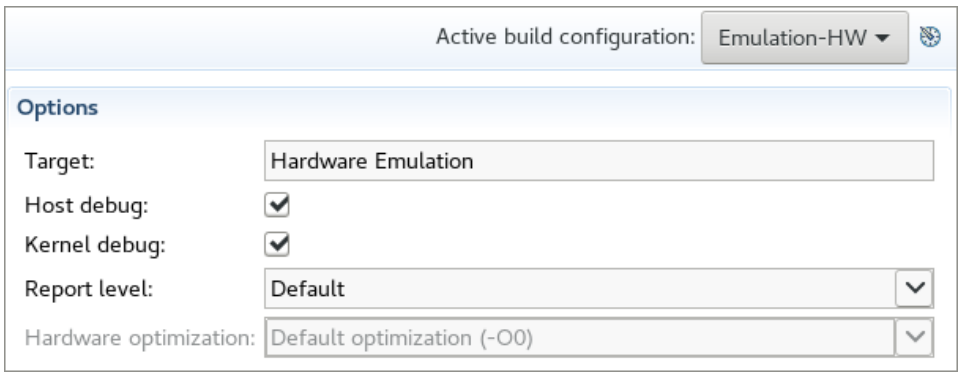
注释： Vitis IDE 中的调试流程在调试期间依赖于 shell 脚本。这就要求设置文件（例如，`.bashrc` 或 `.cshrc`）不干预环境设置，例如，`LD_LIBRARY_PATH`。

为了准备用于调试的可执行文件，您必须更改构建配置以启用“Host debug”和“Kernel debug”。请在 Vitis IDE 的“Project Editor”视图中设置这些选项，如下图所示。在“活动的构建配置 (Active build configuration)”下的“Options”部分中提供了 2 个复选框：

- “主机调试 (Host debug)”支持在主机编译中调试构造，可用于所有构建类型。
- “内核调试 (Kernel debug)”则可启用内核调试，但仅在软件和硬件仿真构建中可用。要在硬件构建中启用调试，请使用“Chipscope Debug”设置，如 Vitis 硬件函数设置 中所述。

这些复选框可在 `g++` 和 Vitis 编译器中启用 `-g` 或 `--debug` 选项。

图 103：**工程编辑器视图的调试选项**



您也可以从“Build Configuration Settings”对话框启用调试功能，方法是选中“Assistant”视图中的构建配置并单击“Settings”按钮，如 Vitis 构建配置设置 中所示。或者，您可双击构建配置。这样同样可以显示以上两个复选框。尽管您可以在所有目标上启用主机调试，但只有软件仿真和硬件仿真构建目标才支持内核调试。

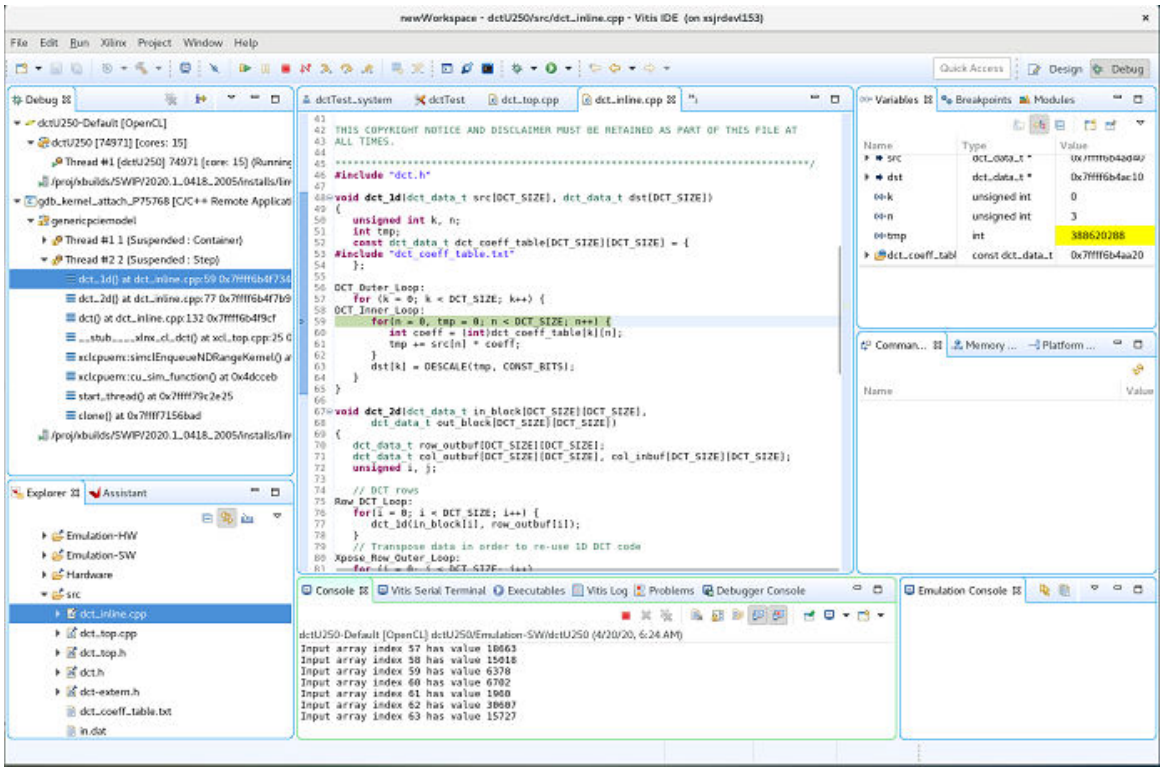
从 Vitis IDE 运行 GDB 会话即可处理所有必要的设置。它会自动为软件仿真管理环境设置。它通过配置 XRT 来确保运行应用时支持调试（如 xrt.ini 文件 中所述），并可管理执行主机代码、内核代码和调试服务器所需的不同控制台。

在嵌入式平台上运行时，Vitis IDE 还会配置并启动 QEMU 系统模式、用于 PL 内核的逻辑仿真器，并管理其同步。如需了解更多信息，请参阅 launch_emulator 实用工具。

设置用于调试的构建配置后，请清除构建目录，并重构应用以确保工程准备好在 GDB 调试环境下运行。

要启动调试会话，请在“Assistant”视图中选中构建配置，然后单击“Debug”( )。在 Vitis IDE 中启动调试会话时，透视图会切换至“Debug”透视图，后者配置为显示另一个窗口，用于管理不同的调试视图和源代码窗口。下图显示了“Debug”透视图。

图 104：**调试透视图**



启动调试环境后，默认情况下，在主机代码中 `main` 函数体开始时，应用会停止。正同任何 GDB 图形前端一样，此时您可以设置断点并检查主机代码中的变量。Vitis IDE 会以用户不可知方式为已加速的内核实现启用相同的功能。如需了解更多信息，请参阅 调试应用与内核。

使用独立调试流程

Vitis IDE 支持您使用命令行流程打开调试工具以处理已构建的工程。

为嵌入式平台启动独立调试

独立调试流程支持嵌入式处理器应用加速流程 (`embedded_accel`) 和嵌入式处理器软件开发流程 (`embedded`)。对于嵌入式平台，应用在器件的 Arm 处理器上运行，启动系统以及加载应用和内核所需的文件位于远程系统上，但调试工具在本地系统上运行，生成的数据和报告需从嵌入式系统移入到本地系统。在此环境下进行调试的过程需进一步的设置和配置。

在 Vitis IDE 中为 `embedded_accel` 进程运行独立调试的过程分为 2 个步骤。

1. 您必须使用 `launch_sw_emu.sh` 或 `launch_hw_emu.sh` 脚本首先启动 QEMU 仿真器 (emulator) 环境，这两个脚本是在 `--package` 进程中生成的。

2. 随后，您必须在独立调试模式下使用 `-debug` 选项启动 Vitis IDE。

要在 Vitis IDE 中为 `embedded` 流程运行独立调试，您必须使用 `launch_hw_emu.sh` 脚本首先启动 QEMU 仿真器环境，此脚本是在 `--package` 进程中生成的。

系统仿真 (emulation) 所需的文件同样由 `--package` 命令来定义。这意味着为嵌入式平台启动独立调试进程依赖于封装进程的输出，包括仿真 (emulation) 脚本。用于启动仿真 (emulation) 环境的命令示例如下。

```
launch_hw_emu.sh -pid-file emulation.pid -no-reboot -forward-port 1440 1534
\
-enable-debug
```

其中：

- `-enable-debug`：打开 2 个不同的命令 shell 以启动 QEMU 和 XSIM，并启用到 QEMU shell 的 GDB 连接。
- `-forward-port`：将 TCP 端口从目标转发至主机，以供连接到 QEMU shell。QEMU 端口默认为 1440。如果需要，您可将此端口更改为 1446，但必须为 `launch_emulation` 命令或脚本指定此端口，并在 `vitis -debug` 命令行中指定此端口。此外，还支持启用多个转发端口。例如，`launch_sw_emu.sh -forward-port` 1440 1534 `-forward-port` 9455 1560。
- `-no-reboot`：完成后退出 QEMU 环境。
- `-pid-file`：将进程 ID 写入指定文件，用于按需结束进程。

对于硬件仿真，这样可启动 2 个运行 QEMU 系统模式的终端端口，并启动 Vivado 仿真器 (simulator) 以对 PL 内核进行仿真。

当终端和仿真 (emulation) 正常启动并运行后，您就可在单独的命令 shell 中以独立调试模式启动 Vitis IDE：

```
vitis -debug -flow embedded_accel -target hw_emu -exe vadd.elf \
-program-args vadd.xclbin -kernels vadd
```

其中：

- `vitis -debug`：以独立调试模式启动 Vitis IDE。
- `-flow embedded_accel`：在嵌入式处理器平台上指定应用加速流程。
- `-target hw_emu`：指示要调试的目标构建。
- `-exe vadd.elf`：指示要运行和调试的可执行应用。
- `-program-args vadd.xclbin`：指定要作为实参加载到可执行文件中的 `.xclbin` 文件。

根据 vitis -debug 命令行 中所述，还有其它选项可供指定，根据应用和构建环境的配置，可能需要使用这些选项。

默认嵌入式系统会在仿真 (emulation) 环境的 `/mnt` 文件夹或嵌入式系统上搜索可执行文件和 `.xclbin` 文件以及任何其它必要的输入文件。启动工具时，可通过指定 `-target-work-dir` 来更改此行为。这样即可启动 Vitis IDE 并启用“Debug”透视图、并为指定的可执行应用和内核代码运行调试配置。此时，您即可在基于 GUI 的调试环境内执行所有调试活动，例如，单步进入、单步跳过、查看变量或添加断点。

为数据中心平台启动独立调试

为数据中心应用启动独立调试较为简单。在此情况下，您需要识别构建目标以及要运行并调试的可执行文件。数据中心平台无需仿真 (emulation) 环境。

以下示例演示如何为以软件仿真 (emulation) 构建为目标的 `data_center` 流程启动 Vitis 独立调试。它指定了可执行文件 `host.exe`（在当前目录中查找此文件）并指定要调试的内核。

```
vitis -debug -flow data_center -target sw_emu -exe host.exe -kernels
krnl_vadd
```

默认情况下，独立调试流程会在当前目录中查找指定文件并将结果写入其中。您可指定 `-work-dir` 选项以从默认目录更改为其它工作目录。在不同目录中构建 `.xclbin` 文件时，可能需要执行此操作。

这样会启动 Vitis IDE 并启用“Debug”透视图，以便您在基于 GUI 的调试环境内执行调试活动，例如，单步进入、单步跳过、查看变量或添加断点。

vitis -debug 命令行

命令行用法

Vitis 软件平台独立调试功能允许您启动 Vitis IDE 以对现有命令行工程进行调试。在以下章节中，对每个命令行选项进行了解释，并提供了为不同平台和目标构建启动独立调试环境的示例。

-debug

```
vitis -debug
```

以独立调试模式启动 Vitis IDE。

-flow

```
-flow [ data_center | embedded_accel | embedded ]
```

指定要调试的应用工程的类型。该选项会配置 Vitis IDE，以便对 Alveo 卡上运行的数据中心应用进行调试；例如，嵌入式平台（如 `zcu104_base` 平台）上运行的应用加速工程或嵌入式软件工程。

> **重要提示！**对于 `embedded` 流程和 `embedded_accel` 流程，您必须使用 `launch_hw_emu.sh` 或 `launch_sw_emu.sh` 脚本来启动 QEMU 系统仿真器 (emulator)，这 2 个脚本是在 `--package` 步骤中（如 嵌入式平台封装 所述）或者使用 `launch_emulator.py` 命令生成的。

-workspace

```
-workspace <workspace>
```

指定在调试模式下打开应用工程时要使用的 Vitis IDE 工作空间。如果不指定该选项，此工具将在当前工作目录中创建名为 `workspace` 的目录。如果已存在名为 `workspace` 的目录，那么该工具将使用此目录作为工作空间。

-exe

```
-exe <path_to_executable>
```

指定应用（主机）可执行文件的文件名和到此文件的路径。

例如：

```
vitis -debug -exe ./host.elf
```

-target

```
-target [ sw_emu | hw_emu | hw ]
```

指定用于调试的构建目标。

提示：该选项仅适用于 data_center 流程和 embedded_accel 流程。

例如：

```
vitis -debug -target hw_emu
```

-program-args

```
-program-args <program arguments>
```

指定要在运行时传递给主机应用的命令行实参。如果不指定该选项，那么选中 data_center 流程或 embedded_accel 流程时，此工具将传递 .xclbin 作为程序实参。

例如：

```
vitis -debug -program-args ./xclbin in.dat
```

-kernels

```
-kernels <list of kernels>
```

指定要调试的内核列表。可指定多个内核名称，以逗号分隔。列出的内核定义为函数级别的断点，因此当内核执行开始时，调试器会停止运行。如果不指定内核，则不提供函数级别调试。

该选项仅对 data_center 流程有效，对于 embedded 流程或 embedded_accel 流程则不受支持。

例如：

```
vitis -debug -kernels mmult madd
```

-work-dir

```
-work-dir <path_to_working_directory>
```

指定用于保存生成的输出文件和报告的工作目录。该选项对 data_center 流程和 embedded_accel 流程有效。

对于 data_center 流程，将在此目录中启动指定的 .exe 文件。对于 embedded_accel 流程，启动目录将由 -target-work-dir 来定义。

提示：如果不指定该选项，那么将使用当前工作目录作为工作目录。

-target-work-dir

```
-target-work-dir <Target working directory>
```

这是目标开发板操作系统和 QEMU 环境上的目录，可执行文件将在此目录中启动。该选项对于 `embedded_accel` 流程和使用 Linux 操作系统的 `embedded` 流程有效。

提示：如果不指定该选项，那么目标工作目录为 `/mnt`。

-xrt-ini

```
-xrt-ini <path_to_xrt.ini>
```

指定 `xrt.ini` 文件的位置。该选项对于 `data_center` 流程和 `embedded_accel` 流程有效。

如果不指定位置，它将在应用的 .exe 所在目录中或者在工作目录中查找此位置。

-os

```
-os [ linux | baremetal ]
```

指定目标开发板上运行的操作系统。该选项对于 `embedded` 流程有效。

-host

```
-host <host_name or ip_address>
```

指定运行 TCF 代理或 `hw_server` 的主机系统的名称或 IP 地址。该选项对于 `embedded_accel` 流程和 `embedded` 流程有效。如果不指定该选项，那么对于裸机，默认主机名为 `localhost`，对于 Linux 目标操作系统，默认 IP 地址为 192.168.0.1。

-port

```
-port <port number>
```

目标 Linux 上运行的 TCF 代理的端口，或裸机目标的本地主机上运行的 `hw_server` 的端口。如果不指定该选项，那么对于 tcf-agent，此端口为 1534，对于 `hw_server`，此端口为 3121。

-launch-script

```
-launch-script <path_to_tcl_script>
```

指定将应用连接到调试器之前要通过 source 命令定位的 Tcl 脚本。该选项仅对含裸机操作系统的 `embedded` 流程有效。Tcl 脚本可包含用于初始化开发板、下载应用、添加断点和使目标做好准备以连接调试器的命令。

第 44 章

配置 Vitis IDE

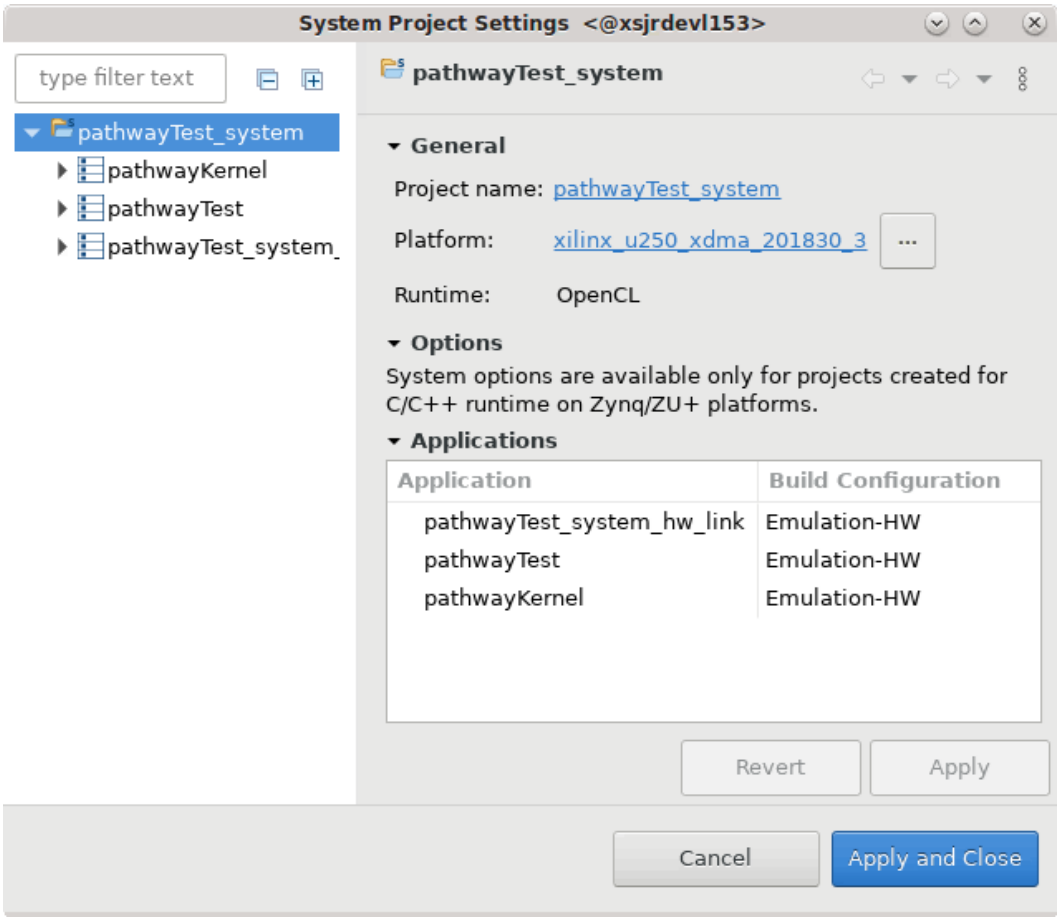
在“Assistant”视图中，使用“Settings”按钮 ( ) 可配置选定的工程或配置对象。如需了解更多信息，请参阅下列主题：

- Vitis 工程设置
- Vitis 构建配置设置
- Vitis 硬件函数设置
- Vitis 二进制容器设置
- Vitis 工具链设置
- Vitis 运行和调试配置设置

Vitis 工程设置

要编辑 Vitis™ 工程设置，请选中“Assistant”视图中的顶层系统工程，然后单击“Settings”按钮 ( )，这样会显示“Project Settings”对话框。此对话框支持您为 Vitis 编译器 `v++` 命令指定链接和编译选项，以便您自定义工程构建进程。

图 105：**工程设置**

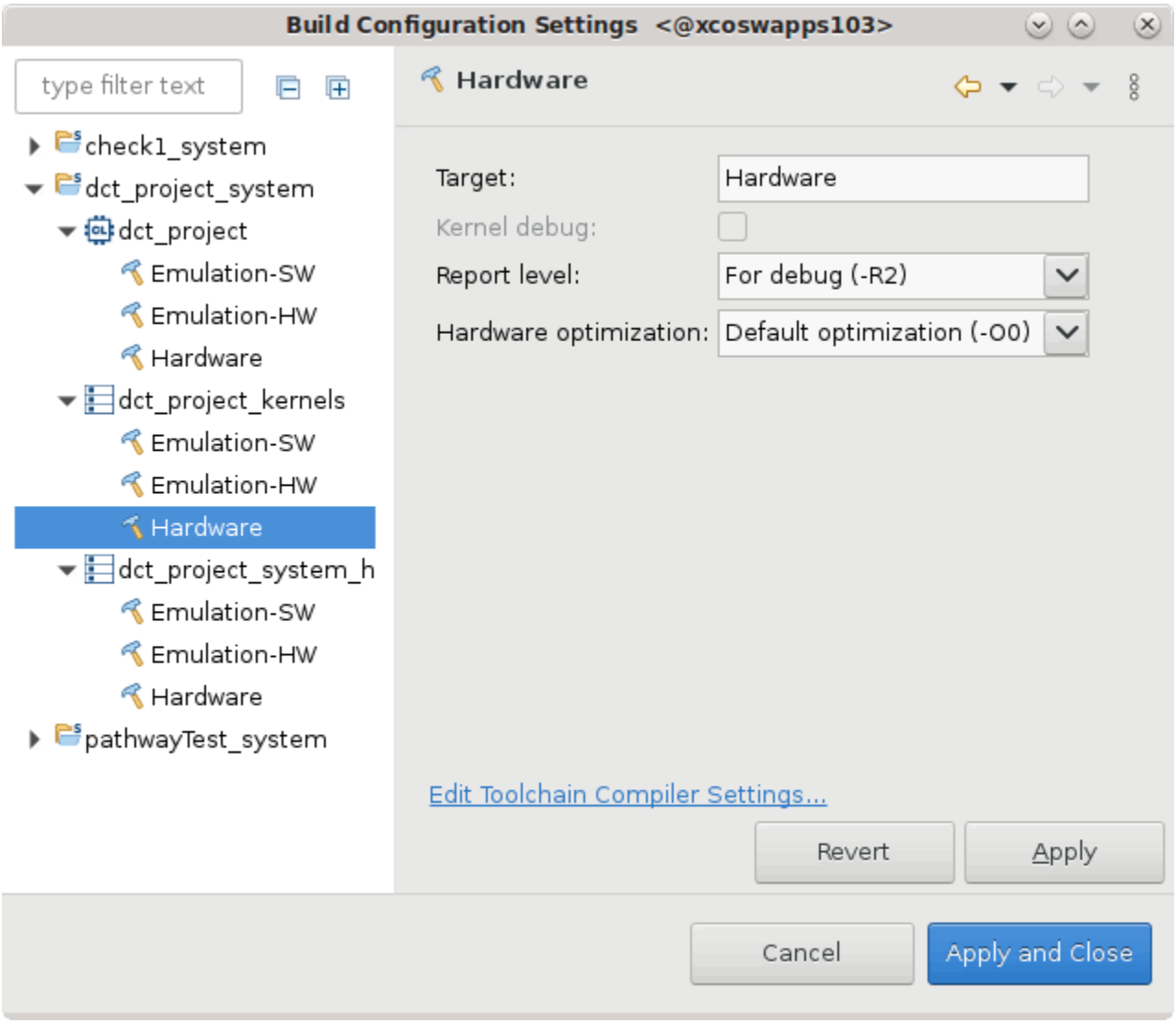


- “Project name”：工程名称。单击此链接即可打开工程的“Properties”对话框。
- “Platform”：此工程的目标平台。单击此链接即可打开“Platform Description”对话框。单击“Browse”以更改平台。
- “Runtime”：显示此工程中使用的运行时。
- “Options”：如果主机应用是使用 XRT 本机 C++ API 编写的，则会显示选项。
- “Applications”：列出系统工程内包含的各子工程，并显示当前构建目标。

Vitis 构建配置设置

要为工程下的任意构建配置编辑设置，请在“Assistant”视图中选中构建配置，然后单击“Settings”按钮 ( ) 以打开“Build Configuration Settings”对话框。此对话框的具体功能因工程类型以及您所选的构建目标而异。在此对话框中，您可以启用主机与内核调试、指定要在构建进程中报告的信息级别以及为硬件构建指定最优化级别。

图 106：**构建配置设置**



此对话框上显示的各选项包括：

- “Target”：构建配置目标，如 构建目标 中所述。
- “Host debug”：选择此设置以启用主机代码调试。
- “Kernel debug”：选择此设置以启用内核代码调试。
- “Report level”：指定要生成的报告级别，如 控制报告生成 中所述。
- “Hardware optimization”：指定用于硬件最优化的工作量。硬件最优化是计算密集型任务。最优化级别越高，可生成的硬件优化程度更高，但构建时间更长。该选项仅在“构建配置系统 (Build Configuration System)”中可用。

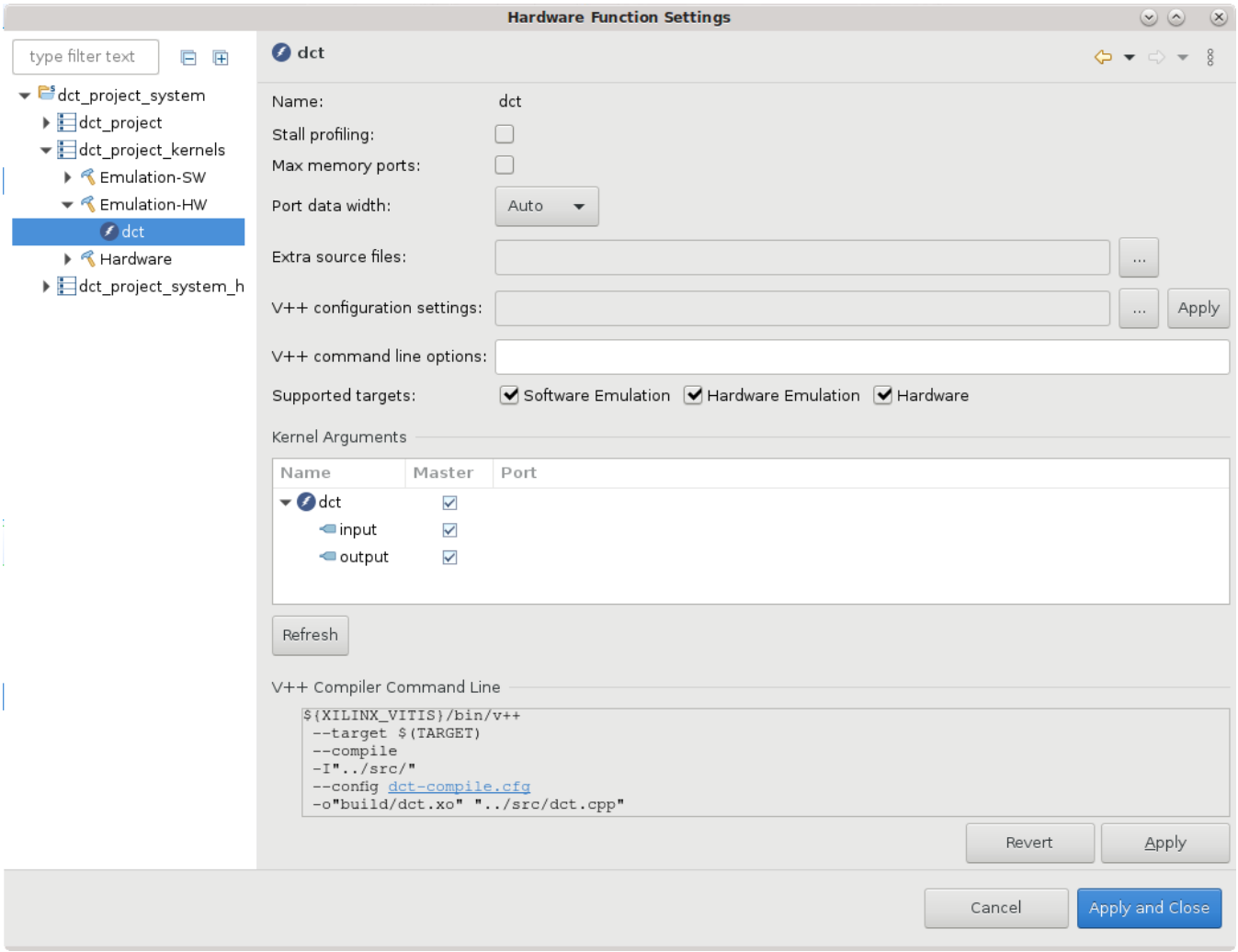
“Build Configuration”对话框还包含指向“Edit Toolchain Compiler Settings”和“Edit Toolchain Linker Settings”的链接。这些链接可提供对标准 Eclipse 环境下所有设置的访问权，并且可用于配置 G++ 编译器、V++ 编译器和 `emconfigutil` 命令，如 Vitis 工具链设置 中所述。

Vitis 硬件函数设置

要编辑硬件函数设置，请在“Assistant”视图中展开内核对象的构建目标、选择硬件函数，然后单击“Settings”按钮 ( )。这样即可显示“Hardware Function Settings”对话框，如下图所示。

提示：请注意在下图中，已选中 `dct` 内核（或硬件函数）作为硬件内核工程 (`dct_project_kernels`) 的 `Emulation-HW` 构建目标。

图 107：**Hardware Function Settings**



此对话框允许您设置与硬件函数相关的选项以及所选构建目标的 `v++` 编译进程。具体选项包括：

- “Stall Profiling”：为内核启用 `--profile.stall` 选项，如 --profile 选项 中所述。
- “Max memory ports”：对于 OpenCL 内核，启用更改选项时会为内核函数特征符中声明的每个全局存储缓冲器生成一个独立的物理存储器接口 (`m_axi`)。如果不启用该选项，则会为存储器映射的内核端口创建单个物理存储器接口。

- “Port data width”：对于 OpenCL 内核，请指定数据端口的宽度。
- “Extra source files”：定义此硬件函数所需的任何其它源文件，例如，输入数据文件。
- “V++ configuration settings”：指定要添加到编译器配置文件中的 Vitis 编译器选项。选择“Edit”命令 (“...”) 即可编辑要添加到配置文件中的选项。指定的选项将添加到 `compile.cfg` 文件中，在对话框底部显示的“V++ Compiler Command Line”中链接此文件。
- “V++ compiler options”：指定要添加到对话框底部显示的“V++ Compiler Command Line”中的 Vitis 编译器选项。
- “Supported targets”：在“Hardware Function Settings”对话框中指定您当前正在定义的 3 个构建目标。您可以选中其中任一或者全选所有构建目标。
- “Kernel Arguments”：显示硬件内核的实参的名称和属性。
- “V++ Compiler Command Line”：显示当前 `v++` 命令行以及您已指定的任何编译选项。

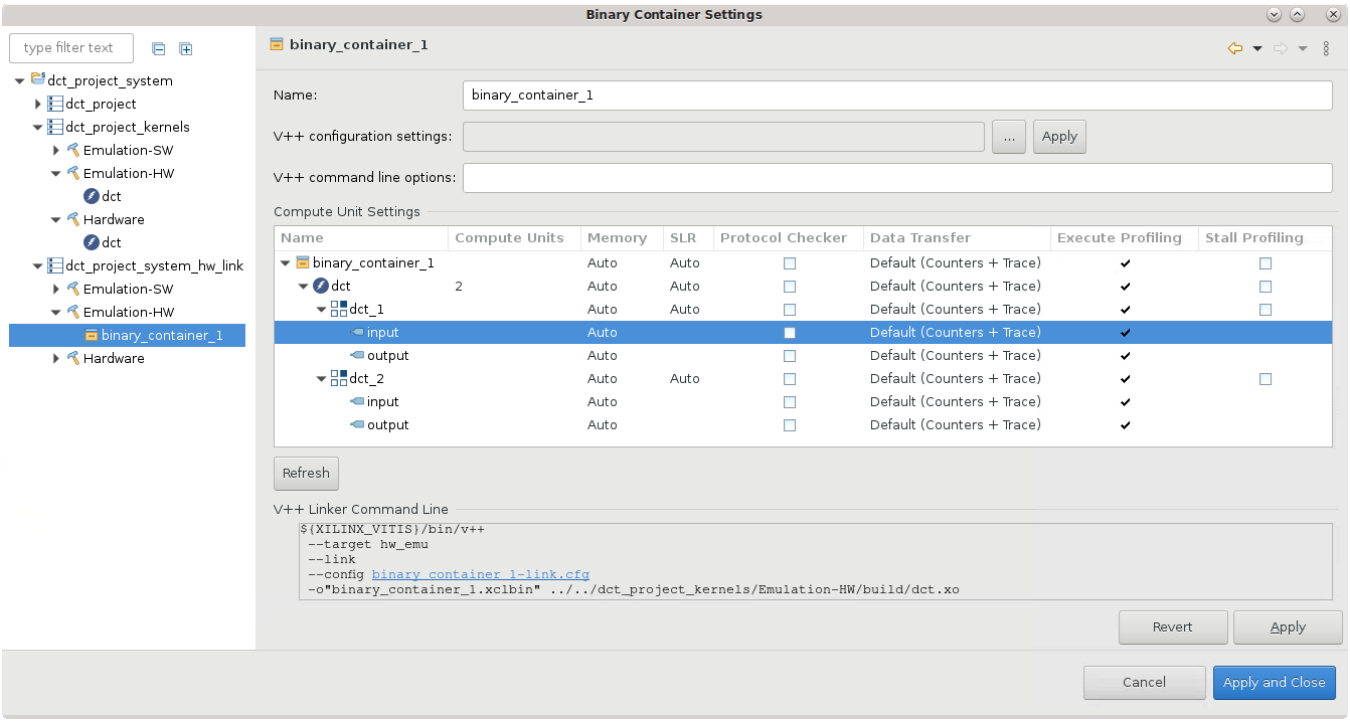
提示： “Hardware Function Settings”对话框指定的设置将写入配置文件，Vitis 编译器将通过 `--config` 选项使用此配置文件，如 Vitis 编译器配置文件 中所述。此配置文件为链接；当您将鼠标置于此链接上时，它会显示配置文件的内容。

Vitis 二进制容器设置

要编辑二进制容器的设置，请在“Assistant”视图中，展开 `hw_link` 工程的构建目标、选中 `binary_container`，然后单击“Settings”按钮 ( )。这样即可显示“Binary Container Settings”对话框，如下图所示。

提示： “Binary Container Settings”对话框中显示的选项取决于所选的具体构建目标，并且将因软件仿真、硬件仿真和硬件而异。

图 108：**二进制容器设置**



此对话框允许您为二进制容器指定新的名称，并为 `v++` 命令指定链接选项。具体选项包括：

- “Name”：指定二进制容器的名称。
- “V++ configuration settings”：指定要添加到配置文件中的 Vitis 连接器选项。选择“Edit”命令 (“...”) 即可编辑要添加到配置文件中的选项。指定的选项将添加到 `binary_container-link.cfg` 文件中，在对话框底部显示的“V++ Linker Command Line”中链接此文件。
- “V++ command line”：输入选定二进制容器的链接选项，这些选项将添加到对话框底部所显示的“V++ Linker Command Line”中。如需了解可用选项的更多信息，请参阅 Vitis 编译器命令。
- “Compute Units”：指定要添加到器件二进制文件的内核数量，如 --connectivity 选项 中所述。该字段可编辑，当您已输入值后，将在此对话框中显示其它内核。
- “Memory”：为计算单元的每个实参指定全局存储器分配，如 将内核端口映射到存储器 中所述。
- “SLR”：为内核的每个计算单元定义 SLR 布局，如 将计算单元分配给 SLR 中所述。
- “ChipScope Debug”：添加监控器以捕获硬件追踪调试信息。这样即可指定 `--debug.chipscope` 选项。
- “Protocol Checker”：将 AXI Protocol Checker 添加到您的设计中。与 `--debug.protocol` 选项相关。
- “Data Transfer”：添加性能监控器，以捕获计算单元与全局存储器之间传输的数据相关的信息。捕获的数据包括计数器和/或追踪。与 `--profile.data` 选项和 `xrt.ini` 文件设置相关，如 在应用中启用剖析 中所述。
- “Execute Profiling”：添加加速器监控器以捕获计算单元执行的开始和结束位置。与 `--profile.exec` 选项相关。
- “Stall Profiling”：添加加速器监控器，其中包含用于捕获内核内部、两个内核之间或者内核与外部存储器之间的数据流中的停滞的功能。与 `--profile.stall` 选项相关。

- “V++ Linker Command Line”：显示当前 `v++` 命令行，其中包含您已指定的任何链接选项。

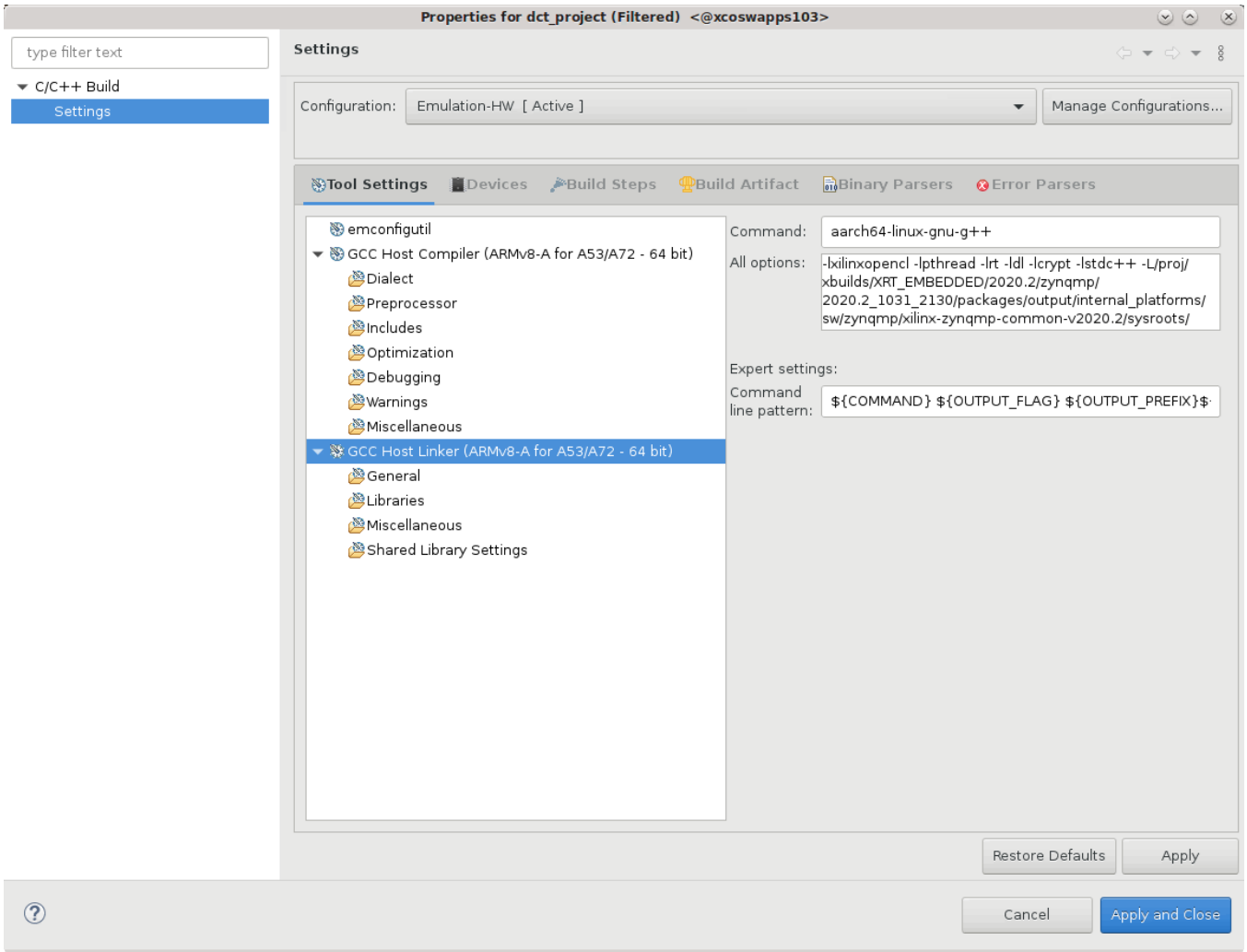
> **提示：** “Binary Container Settings”对话框指定的设置将写入配置文件，Vitis 编译器将通过 `--config` 选项使用此配置文件，如 Vitis 编译器配置文件 中所述。此配置文件为链接；当您将鼠标置于此链接上时，它会显示配置文件的内容。

Vitis 工具链设置

工具链设置可提供基于 Eclipse 的标准工程视图，为 Vitis IDE 中的 C/C++ 构建提供所有选项。

在“Build Configuration Settings”对话框中，单击“Edit Toolchain Compiler Settings”或“Edit Toolchain Linker Settings”以打开编译器和连接器的“Settings”对话框，其中包含所有 C/C++ 构建设置。此对话框允许您设置标准 C++ 路径，包括路径、库、全工程定义以及主机定义。

图 109：**工具链设置**

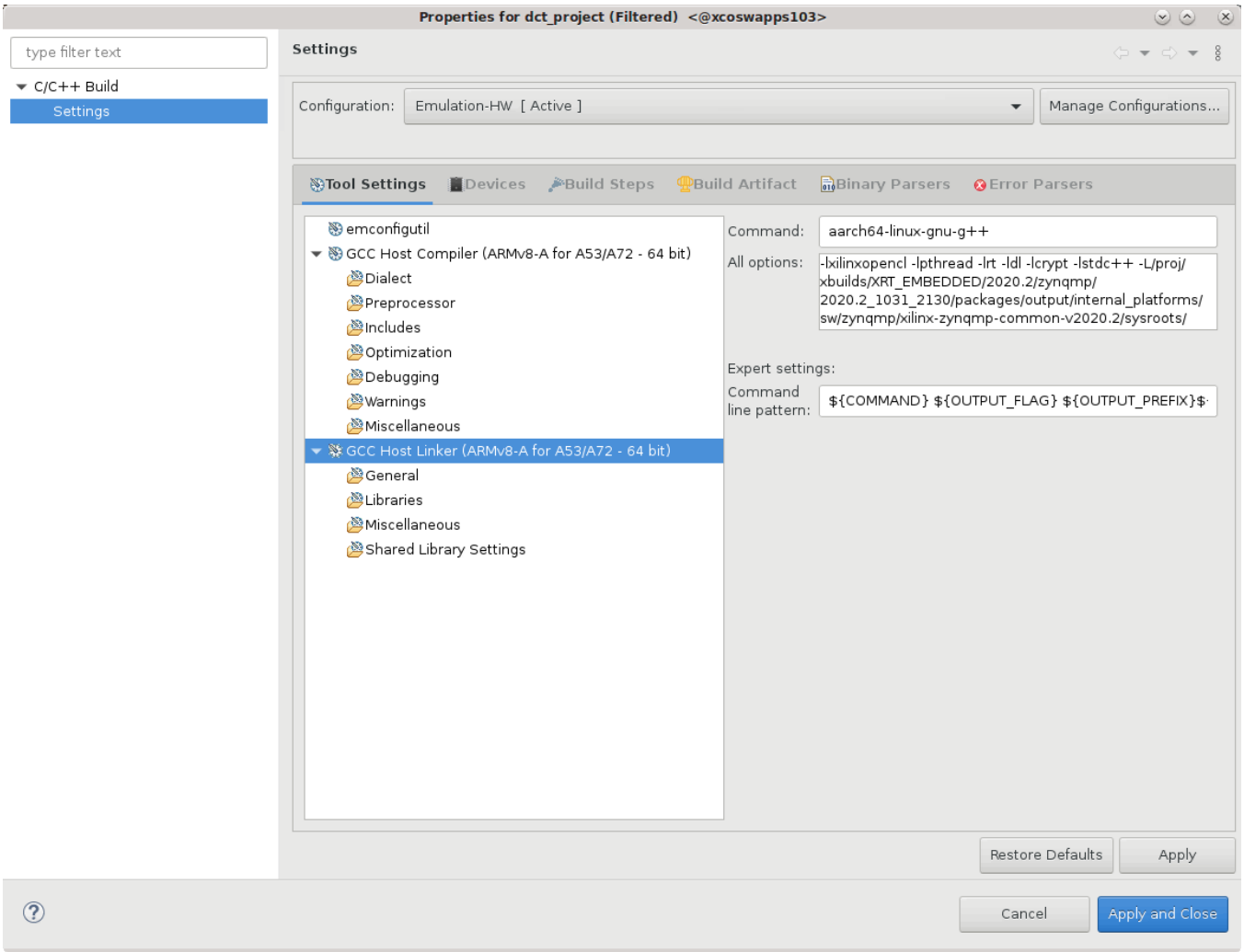


“Toolchain Settings”对话框的内容取决于您启动工具链设置的具体“Build Configuration”对话框。您可从主机应用工程、硬件内核工程或 `hw_link` 工程启动工具链设置。可用的具体设置包括：

- “emconfigutil”：指定 emconfigutil 实用工具 的命令行选项。请参阅 emconfigutil 设置。
- “GCC Host Compiler”：指定 `g++` 连接器实参，这些实参必须在主机编译进程中进行传递。请参阅 G++ 主机编译器和连接器设置。
- “GCC Host Linker”：指定 `g++` 连接器实参，这些实参必须在主机链接进程中进行传递。请参阅 G++ 主机编译器和连接器设置。
- “V++ Kernel Compiler”：指定 `v++` 命令以及在为内核编译进程调用 `v++` 命令时必须传递的所有其它选项。请参阅 V++ 编译器和连接器设置。
- “V++ Kernel Linker”：指定 `v++` 命令以及在为内核链接进程调用 `v++` 命令时必须传递的所有其它选项。请参阅 V++ 编译器和连接器设置。

G++ 主机编译器和连接器设置

图 110：**GCC 编译器和连接器设置**



您可从主机应用工程的“Build Configuration Settings”对话框打开“GCC Compiler and Linker Settings”对话框。Vitis 核开发套件所使用的 g++ 编译器的实参可在“Toolchain Settings”的“GCC Host Compiler”部分下访问。

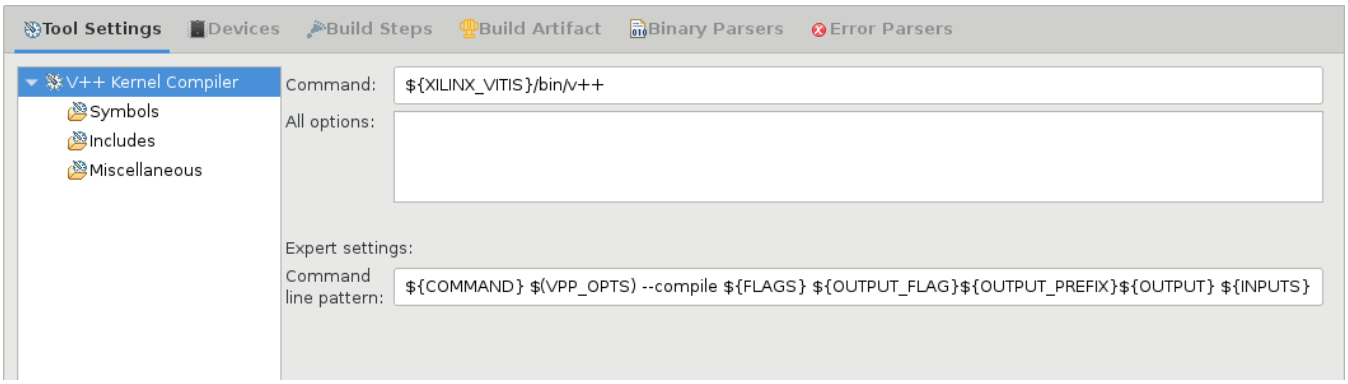
- “Dialect”：指定用于选择要使用的 C++ 语言标准的命令选项。标准语言选项包括：C++ 98、C++ 2011 和 C++ 2014 (1Y)。
- “Preprocessor”：指定主机编译器的处理器实参，例如，符号定义。已定义的默认符号包括平台，因此主机代码可以检查特定平台。
- “Includes”：指定包含路径和包含文件。
- “Optimization”：指定编译器最优化标记和其它最优化设置。
- “Debugging”：指定调试级别和其它调试标记。
- “Warnings”：指定与编译器警告相关的选项。
- “Miscellaneous”：指定传递到 g++ 编译器的任何其它标记。

GCC 连接器选项

Vitis 技术 G++ 主机连接器 (G++ Host Linker) 是通过此处可用的选项来提供的。具体各部分包括常规选项、库和库路径、其它连接器选项以及共享库。

V++ 编译器和连接器设置

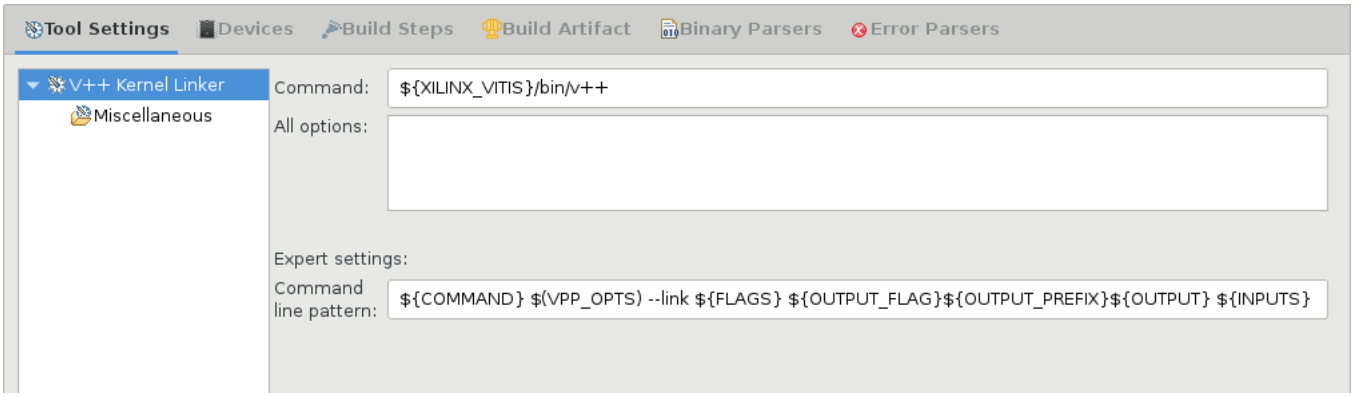
图 111：**V++ 编译器设置**



您可从硬件内核工程的“Build Configuration Settings”对话框中打开“V++ Compiler Settings”对话框。“V++ Kernel Compiler”设置可显示编译期间使用的 v++ 命令，以及调用 v++ 命令用于内核编译进程时必须传递的所有其它选项。v++ 命令选项可采用符号（包括路径）或其它有效选项。

- “Symbols”：单击 Vitis 编译器下的“Symbols”以定义调用 v++ 命令时通过 -D 选项传递的任何符号。
- “Includes”：要在 Vitis 编译器中添加 include 路径，请选中“Includes”，然后单击“Add”按钮 ( )。
- “Miscellaneous”：在“Miscellaneous”部分中可以添加特定于 Vitis 的设置（例如，不属于标准 C/C++ 工具链的 Vitis 编译器和连接器标记）作为标记。如需了解有关可用编译器选项的更多信息，请参阅 Vitis 编译器命令。

图 112：**V++ 连接器设置**



您可以从 `hw_link` 工程的“Build Configuration Settings”对话框打开“V++ Linker Settings”对话框。“V++ Linker Settings”可显示链接期间所用的 `v++` 命令以及调用 `v++` 命令用于内核编译进程时要传递的任何其它选项。

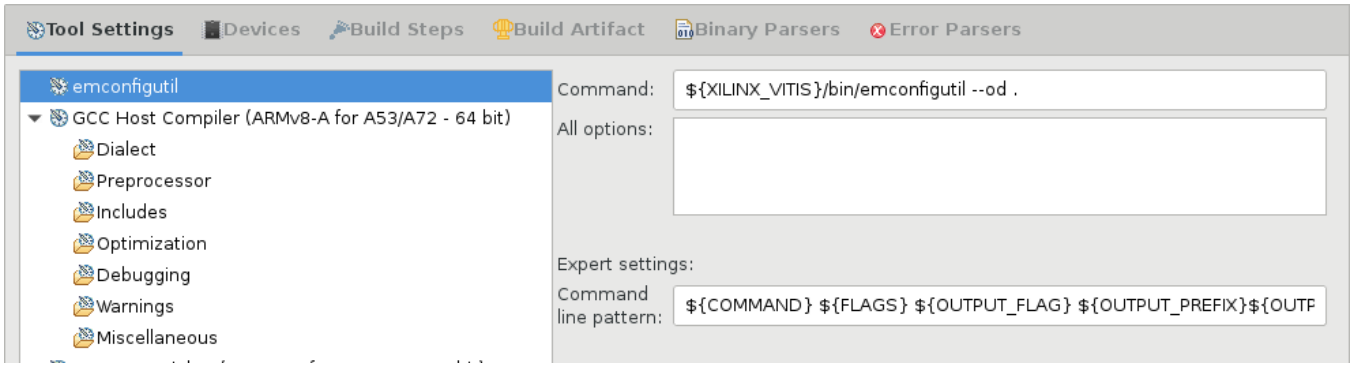
在“Miscellaneous”部分中可添加需要传递到 Vitis 编译器的任何其它选项作为标记。如需了解有关链接进程中可用选项的更多信息，请参阅 Vitis 编译器命令。

emconfigutil 设置

您可从主机应用工程的“Build Configuration Settings”对话框打开“GCC Compiler and Linker Settings”对话框，然后在此对话框中选中 `emconfigutil` 命令选项。“Command”字段可指定 `emconfigutil` 命令，此命令将启动 Vitis IDE，如 emconfigutil 实用工具 中所述。您也可以为命令行指定自己工程所需的任意选项。

在启动仿真运行配置前，Vitis IDE 会通过运行指定的 `emconfigutil` 命令来创建 `emconfig.json` 文件。

图 113：**emconfigutil 设置**



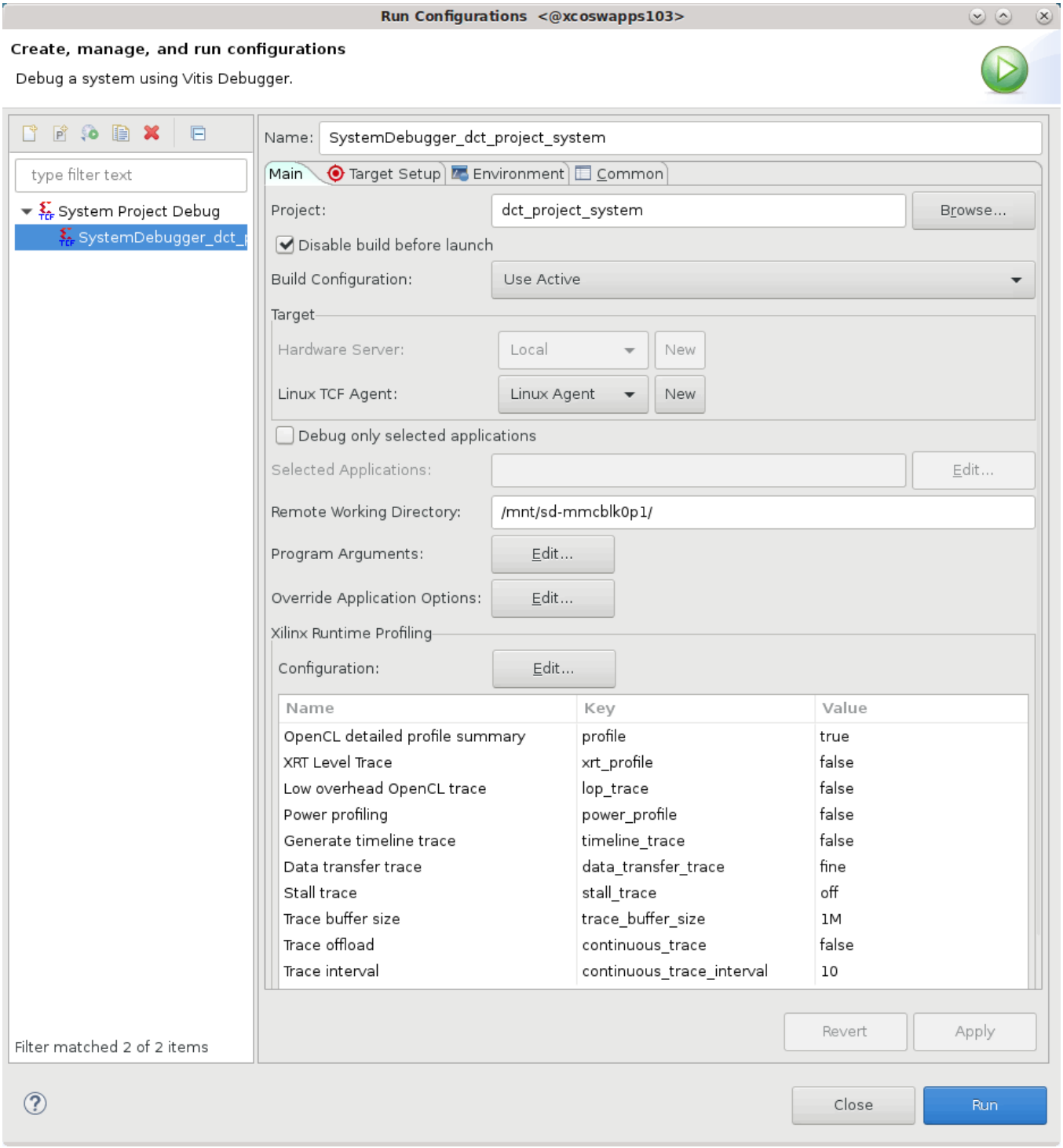
Vitis 运行和调试配置设置

启动已编译、已链接和已封装的应用后，要在 Vitis IDE 中运行或调试，需使用“Launch Configuration”对话框，如下图所示。当构建流程完成后，该工具会启用“Assistant”视图中的“Run”按钮和“Debug”按钮以便您指定要使用的“启动配置 (Launch Configuration)”。

提示：启动和调试应用所使用的启动配置是相同的。运行应用或启动调试器时该工具所遵循的步骤有些许差异，但在这两种情况下可使用相同的配置。

要编辑启动配置的设置，请选中构建目标，然后单击“Run”按钮以打开“Run Configurations”对话框。“Run Configuration”对话框（如下所示）支持您指定调试选项、对运行中的应用启用剖析并指定要收集的剖析数据的类型。

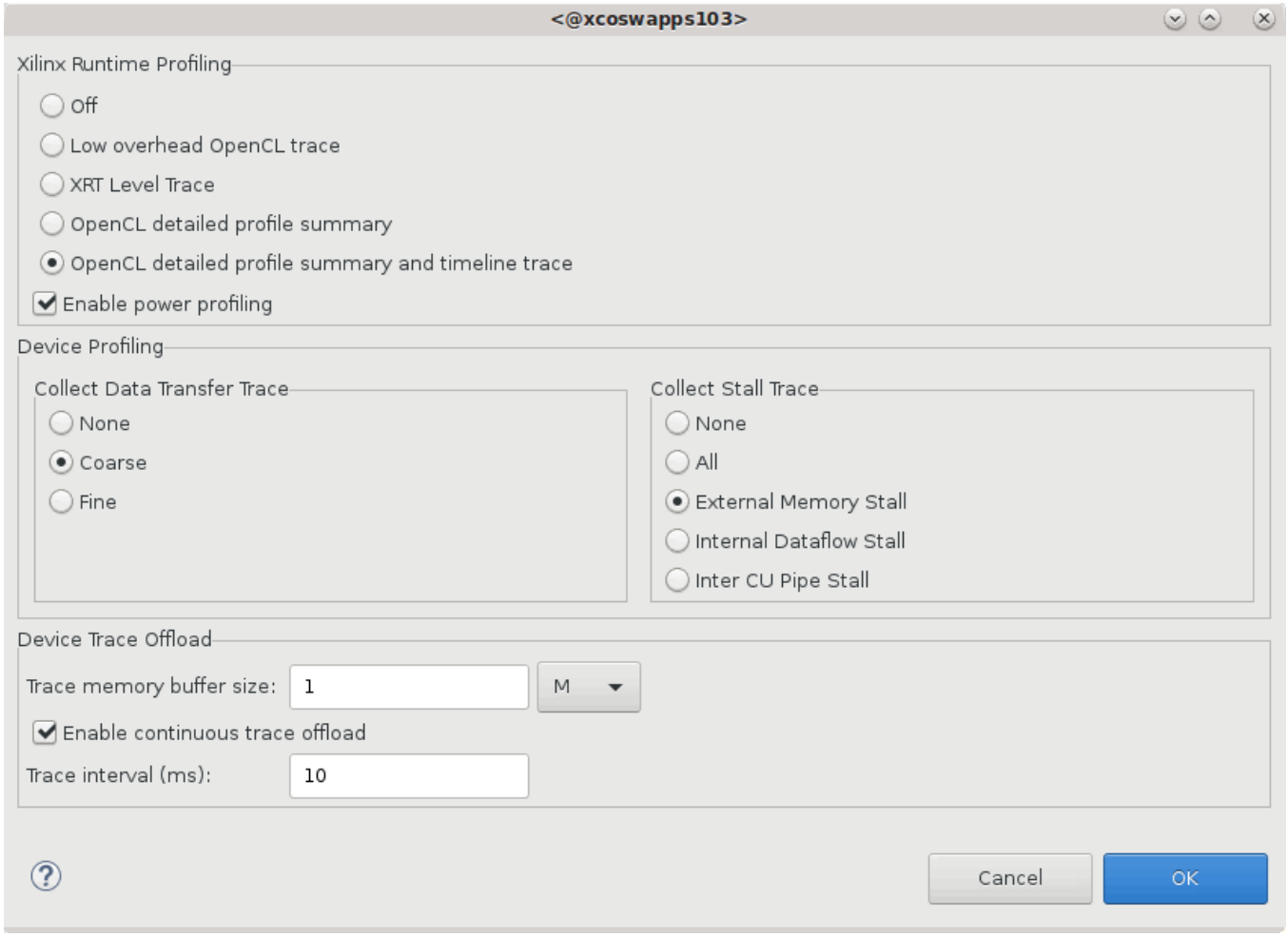
图 114：**运行配置设置**



提示：对于数据中心平台和嵌入式处理器平台、软件和硬件仿真 (emulation) 构建以及硬件构建，“Launch Configuration”对话框中显示的选项不尽相同。

- “Name”：指定运行配置的名称。Vitis IDE 会创建一个文件夹，此文件夹根据当前运行的特定构建配置中的运行配置来命名。例如，`./project/Emulation-HW/run_config`。应用运行的输出文件和日志将写入此文件夹。传递到主机程序的所有实参都应按与该文件夹的相对关系来写入。
- “Project”：显示当前工程，但可更改为其它打开的工程。
- “Build Configuration”：选择启动配置适用于的构建目标，或者它适用于处于活动状态的构建配置。
- “Disable Build Before Launch”：启用此复选框后，可避免该工具在运行或调试工程前对其进行重构。
- “Target”：指定配置的运行或调试目标。请注意，仿真 (emulation) 构建目标为 Linux TCF 代理，而硬件构建则需要 `hw_server`，如 调试应用与内核 中所述。
- 远程工作目录：对于嵌入式处理器系统，请指定 QEMU 环境的安装盘，或者指定物理器件的安装盘。
- “Program Arguments”：显示“程序实参 (Programs Argument)”对话框。此对话框允许您按需为应用指定命令行实参。启用“Automatically update arguments”复选框即可允许该工具自动指定 `xclbin` 文件作为该应用的实参。`xclbin` 文件将被追加到命令行末尾，位于任何其它已指定的实参之后。

  注释：程序实参应根据与运行期间创建的运行配置文件夹的相对关系来指定。您可参考 IDE 中的 Vitis 日志窗口，以复查用于启动此应用的命令行副本。
- “Override Application Options”：为 `launch_emulator.py` 命令创建选项文件，您可手动编辑此文件以根据自己的需求进行自定义。单击“Generate”按钮以创建新的 `launch_options.cfg` 文件，或者使用“Browse” 按钮来查找现有文件。
- “Xilinx Runtime Profiling”：指定应用运行期间启用的剖析和事件追踪功能。指定的选项存储在配置文件中，并写入 `xrt.ini` 文件以供在运行应用期间使用。单击“Edit”按钮以打开“Xilinx Runtime Profiling”对话框，如下所示。



- “Runtime Profiling”：指定为应用运行启用的剖析类型。您可指定剖析形式或者剖析和时间线捕获，如 在应用中启用剖析 中所述。

  提示：对于硬件构建，您还可为加速器卡启用“功耗剖析 (Power Profiling)”。

- “Device Profiling”：指定在硬件内核上执行的剖析级别。该选项与 `data_transfer_trace` 相关，如 xrt.ini 文件 中所述。粗糙的器件剖析可显示 CU 的数据传输活动。精细的器件剖析显示的是端口上的所有 AXI 级别的传输事务。
- “Collect Stall Trace”：表示各种条件下捕获的停滞数据，如 --profile 选项 和 xrt.ini 文件 中所述。
- “Device Trace Offload”：指定分配用于捕获追踪数据的全局存储器的量，并支持按指定时间间隔进行连续追踪卸载，以防止在中断进程中损失应用时间线数据，如 在应用中启用剖析 中所述。

“Launch Configuration”对话框还包含其它有助于为应用配置运行时环境的选项卡。包括以下 3 个选项卡：

- “Target Setup”：主要用于含裸机应用的嵌入式平台。它提供了用于对开发板、器件和程序进行初始化、管理和复位的选项。
- “Environment”：支持您设置和管理 Vitis IDE 所需的环境变量。
- “Common”：该选项卡派生自 Eclipse。这些是与 Eclipse 环境通用的设置和功能特性。

第 45 章

工程导出和导入

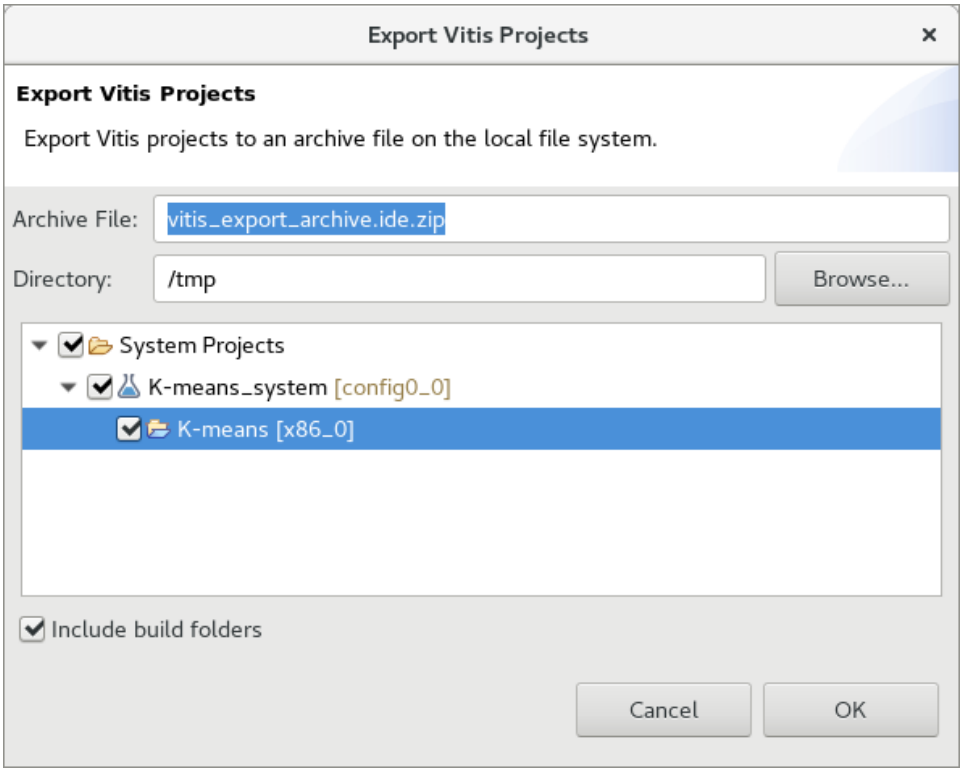
Vitis™ IDE 提供了简化的方法，用于在您的工作空间内导出或导入一个或多个 Vitis IDE 工程或者从 GIT 仓库导入工程。您可以选择包含关联的工程构建文件夹。

导出 Vitis 工程

导出工程时，该工程将存档为 zip 文件，其中包含导入另一个工作空间所需的所有相关文件。

1. 要导出工程，请从主菜单中依次选择“File” → “Export”。

   这样会打开“Export Vitis Projects”页面，您可在其中选择当前工作空间内要导出的一个或多个工程，如下图所示。



2. 要更改此存档文件的名称，请编辑“Archive File”字段。
3. 要包含当前构建配置，请启用窗口底部的“Include build folders”。

   提示：这样可能会显著增大存档文件大小，但在某些情况下需要执行此操作。

4. 要创建包含所选文件的存档，请单击“OK”以创建存档。

所选的 Vitis IDE 工程将存档到指定文件和位置中，并且可供其他用户导入位于其它计算机上的其它工作空间下的 Vitis IDE 中。

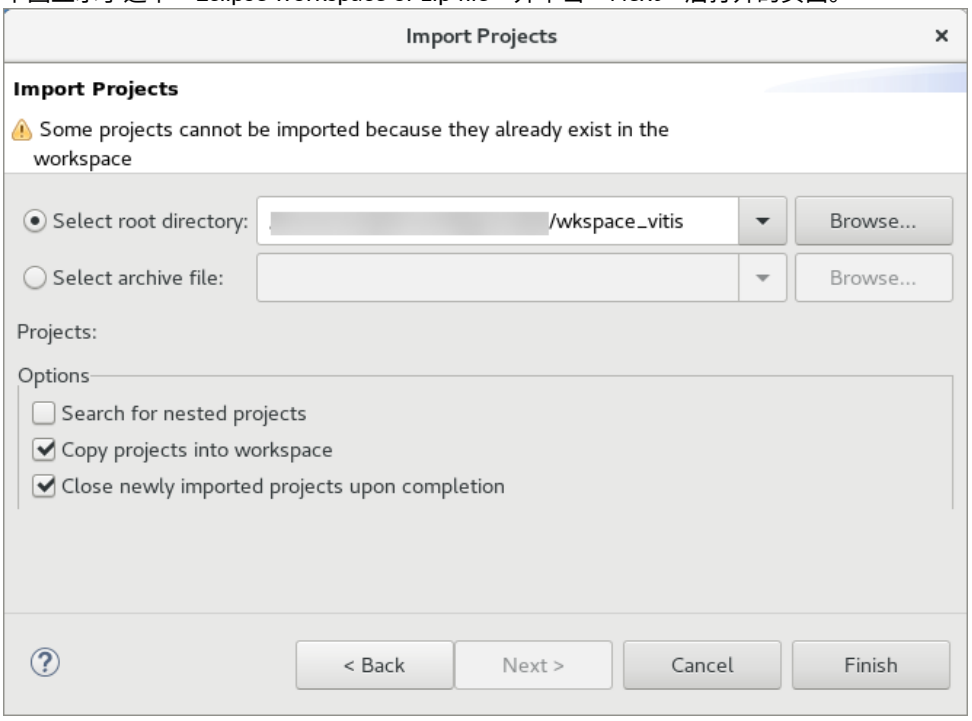
导入 Vitis 工程

1. 要导入工程，请从顶部菜单中选择“File” → “Import”。

   这样会打开“Import Projects”页面以供选择导入文件类型。有两种类型的文件可供您选择导入：

   - Vitis 工程导出的 zip 文件：允许您导入先前从 Vitis IDE 导出的工程，如 导出 Vitis 工程 中所述。
   - Eclipse 工作空间或 zip 文件：允许您从另一个 Vitis IDE 工作空间导入工程。
   - 从 Git 导入工程：允许您从先前本地克隆的 Git 仓库导入工程，或者从指定 Git URL 导入，如下一节主题所述。

2. 下图显示了选中“Eclipse workspace or zip file”并单击“Next”后打开的页面。



3. 对于“Select root directory”，请将其设置为指向 Vitis IDE 的工作空间，并按需指定以下选项：
   - “Search for nested projects”：在工作空间中的其它工程内查找工程。
   - “Copy projects into workspace”：在当前打开的工作空间内创建工程的物理副本。
   - “Close newly created imported projects upon completion”：在已打开的工作空间中创建工程后将其关闭。
4. 单击“Finish”以将工程导入 Vitis IDE 中已打开的工作空间。

从 Git 导入工程

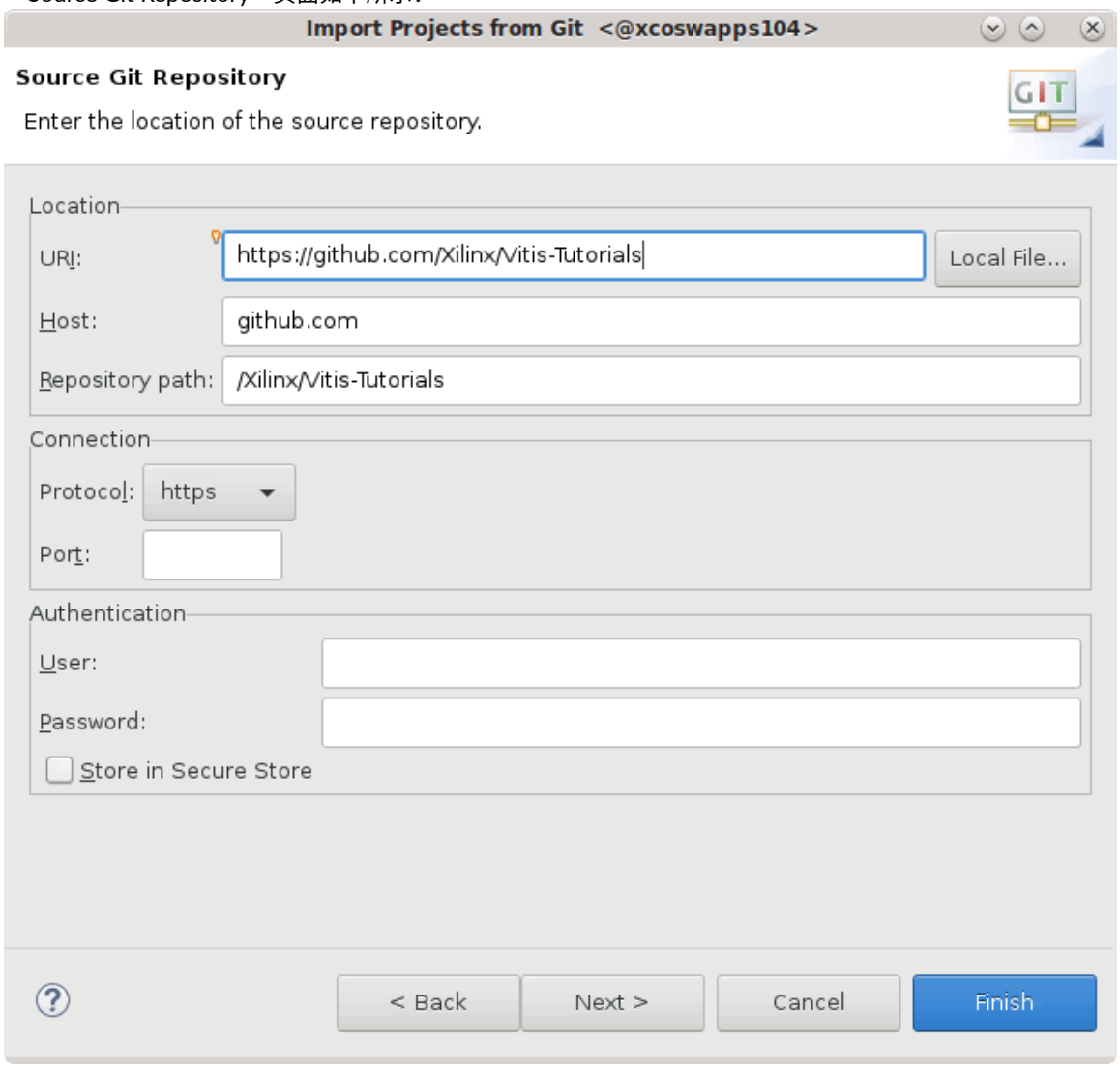
1. 在“Import Projects”Wizard 中，选中“Import projects from Git”选项，然后选择“Next”以继续。

   这样会打开“Select Repository Source”页面。其中有两种类型的仓库可供您选择：

   - Existing local repository：选择已克隆到本地的现有 Git 仓库。选择该选项时，“Select a Git Repository”页面会显示 Vitis IDE 找到的当前已克隆的本地仓库列表。单击“Next”以继续。

     提示：您可以按如下所述方式通过克隆 URL 来创建本地 Git 仓库，或者在 Linux shell 中使用 `git clone <url>` 命令来创建仓库。

   - Clone URL：允许您指定要克隆到指定位置的 Git URL。选择该选项时，“Import Projects”Wizard 的“Source Git Repository”页面如下所示：

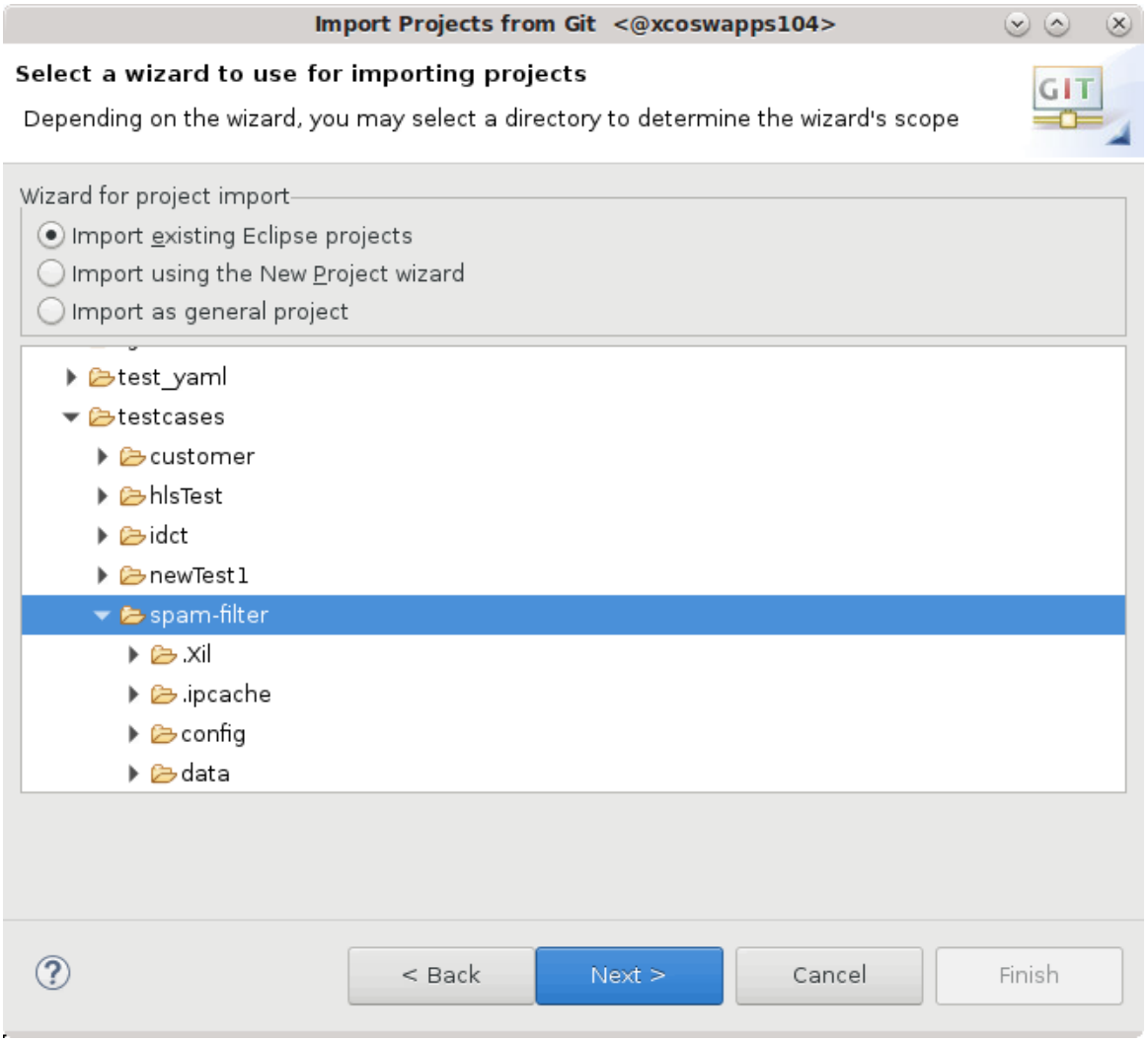


   1. 在“Source Git Repository”页面上，指定以下选项，然后单击“Next”以继续：

- Location：指定仓库的 URL。这样会从提供的 URL 中提取主机和仓库路径。用户可能感兴趣的 URL 包括：
  - Vitis 加速示例：https://github.com/Xilinx/Vitis_Accel_Examples
  - Vitis 库：https://github.com/Xilinx/Vitis_Libraries
  - Vitis 教程：https://github.com/Xilinx/Vitis-Tutorials
- Connection：指定用于连接的连接协议。这些字段可用于根据需要自定义连接。
- Authentication：指定用于访问仓库的用户 ID (User ID) 和密码 (Password)（如果需要）。

2. 在“Branch Selection”页面中，可选择克隆一个或多个分支。单击“Next”以继续。
3. 在“Local Destination”页面中，指定仓库将克隆到的“Destination Directory”。单击“Next”以继续。

2. 打开本地仓库或者克隆 URL 以创建新的本地仓库后，会打开“Import Project”Wizard 的“Select a wizard to import projects”页面。如下所示，此页面允许您导入 Eclipse 工程、请使用“New Project”Wizard 导入工程，或者导入通用工程。



3. 选择工程导入方法，然后单击“Next”以继续。根据所选方法，系统将指示您完成“New Project”Wizard（如 创建 Vitis IDE 工程 中所述），或者将指导您完成导入 Eclipse 工程或通用工程的流程。

第 46 章

入门示例

Vitis™ 核开发套件是随设计示例提供的。这些示例可以：

- 成为实用学习工具，供用户学习 Vitis IDE 和编译流程（如 makefile 流程）。
- 帮助您快速开始创建新的应用工程。
- 演示有用的编码样式。
- 突出重要的最优化技术。

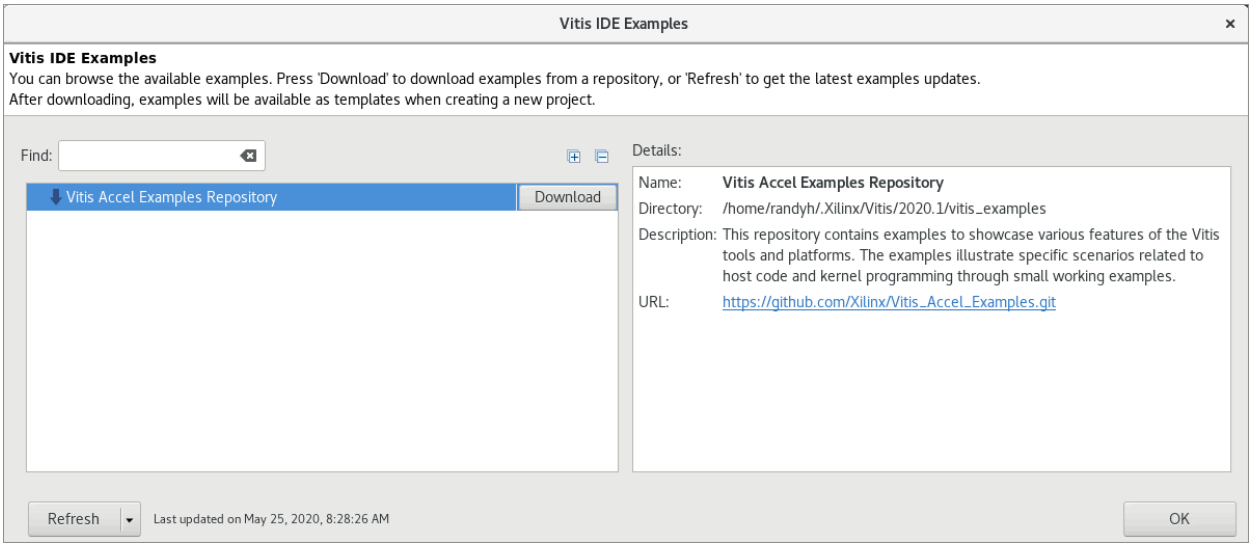
Vitis IDE 提供的每个目标平台都包含帮助您入门的样本设计，并且可以通过如 创建应用工程 中所述的工程创建流程访问。

在 `<vitis_root>/samples` 文件夹中提供了数量有限的设计样本，并且在赛灵思 GitHub 仓库中也有许多示例可供下载。其中每一项设计都随附有 Makefile，您可根据自己的偏好，完全在命令行上进行代码的构建、仿真和运行。

安装示例和库

您可通过以下方式下载并安装样本应用：在“New Application Project”Wizard 中可通过“Templates”页面，或者在现有工程内依次选择“Xilinx” → “Examples”。这样即可显示“Vitis IDE Examples”对话框，如下图所示。

图 115： **“Vitis IDE Examples”对话框**



此对话框的左侧显示“Vitis IDE Examples Repository”，并对应每个类别提供一条下载命令。对话框的右侧显示下载示例的目录以及下载示例的 URL。单击“Vitis IDE Examples”旁的“Download”以下载示例并填充对话框。

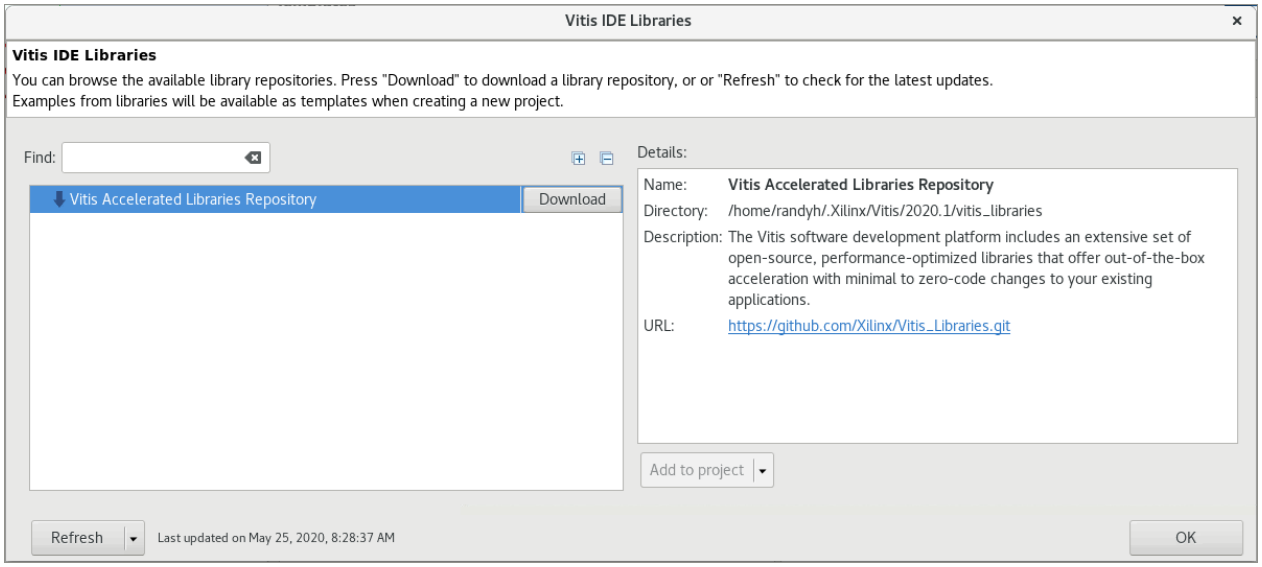
“Vitis IDE Examples”对话框左下角的命令菜单提供了两条用于管理示例存储库的命令：

- “Refresh”：刷新下载示例列表，以便从 Vitis 示例 GitHub 仓库下载任何更新。
- “Reset”：从 `.Xilinx` 文件夹删除下载的示例。

提示：公司防火墙可能限制出站连接。可能需要特定的代理设置。

您也可以从“New Application Project”Wizard 下载 Vitis 加速库，或者选择“Xilinx” → “Libraries”菜单命令进行下载。如需了解有关可用的库及其用法的更多信息，请参阅 Vitis 库。

图 116： **“Vitis IDE Libraries”对话框**



使用本地副本

虽然您必须在创建新工程时下载示例以添加模板，但 Vitis IDE 始终会将示例下载到您的本地 `.Xilinx/vitis/<version>` 文件夹中：

- Windows 上：`C:\Users\<user_name>\.Xilinx\vitis\<version>`
- Linux 上：`~/.Xilinx/vitis/<version>`

在 Vitis IDE 的“示例 (Examples)”对话框中无法更改下载目录。您可能希望将示例文件下载到除 `.Xilinx` 文件夹以外的其它位置。要执行此操作，请在命令 shell 中使用 `git` 命令为下载的示例指定新的目标文件夹：

```
git clone https://github.com/Xilinx/Vitis_Examples
<workspace>/examples
```

如上所示，使用 `git` 命令克隆示例时，您可以将示例文件作为应用与内核代码资源，以供在您自己的工程中使用。但是，许多文件使用 include 语句来包含在各种示例的 makefile 中管理的其它示例文件。当通过“New Vitis Project” Wizard 添加模板时，这些包含文件会自动填充到工程的 `src` 文件夹中。要使这些文件成为本地文件，请找到这些文件并将其手动设置为您的工程的本地文件。

您可通过在克隆的存储库所在位置中搜索所需的文件来找到这些文件。例如，您可在 `examples` 文件夹中运行以下命令来查找 `vadd` 示例所需的 `xcl2.hpp` 文件。

```
find -name xcl2.hpp
```

XILINX

第 47 章

RTL Kernel Wizard

为了确保将 RTL IP 封装到内核对象 (XO) 文件中以供 Vitis™ 编译器使用，需要执行相应的步骤，而 RTL Kernel Wizard 则可以自动执行其中部分步骤。RTL Kernel Wizard 可以：

- 指导您逐步完成指定 RTL 内核的接口要求的过程，并基于提供的信息生成顶层 RTL 封装文件。
- 自动生成 AXI4-Lite 接口模块，包括控制逻辑和寄存器文件，并将其包含在顶层封装文件内。
- 在顶层封装文件内包含内核 IP 模块示例，您可将其替换为自己的 RTL IP 设计，前提是确保您的 RTL IP 与该封装文件之间以正确方式连接。
- 自动生成 `kernel.xml` 文件，与来自该向导的内核规范匹配。
- 为生成的 RTL 内核封装文件生成简单的仿真测试激励文件。
- 生成主机程序示例，用于运行并调试 RTL 内核。

RTL Kernel Wizard 可从 Vitis IDE 或从 Vivado® IP 目录来访问。无论采用何种访问方式，它都会创建一个 Vivado 工程，其中包含设计示例以充当模板，用于定义您自己的 RTL 内核。

此设计示例包含一个简单的 RTL IP 加法器，称为 VADD，可用于全程指导您完成将您自己的 RTL IP 映射到生成的顶层封装文件的整个流程。连接对象包括各时钟、各复位、`s_axilite` 控制接口、`m_axi` 接口以及（可选）`axis` 数据流传输接口。

此向导还会为生成的 RTL 内核封装文件生成一个简单的测试激励文件，并生成主机代码样本用于实践此 RTL 内核示例。此测试激励文件示例和主机代码必须经过相应的更改才能用于测试您的 RTL IP 设计。

启动 RTL Kernel Wizard

“RTL Kernel”Wizard 可从 Vitis IDE 启动，也可从 Vivado IDE 启动。

提示：从 Vitis IDE 运行此向导会在此过程完成时，将生成的 RTL 内核与主机代码示例一起自动导入当前应用工程。

要从 Vitis IDE 内启动“RTL Kernel”Wizard，请从打开的应用工程中依次选中“Xilinx”→“RTL Kernel Wizard”菜单项。如需了解有关使用 GUI 的详细信息，请参阅 第七部分：使用 Vitis IDE。

要从 Vivado IDE 启动“RTL Kernel”Wizard，请执行以下操作：

1. 创建新的 Vivado 工程，并在选择工程的开发板时选择目标平台。
2. 在 Flow Navigator 中，单击“IP catalog”命令。
3. 在 IP 目录搜索框内键入 `RTL Kernel`。
4. 双击“RTL Kernel Wizard”以启动向导。

使用 RTL Kernel Wizard

“RTL 内核 (RTL Kernel)”Wizard 划分为多个页面，以细分 RTL 内核的定义流程。此向导包含下列页面：

1. 常规设置
2. 标量
3. 全局存储器
4. 数据流传输接口
5. 总结

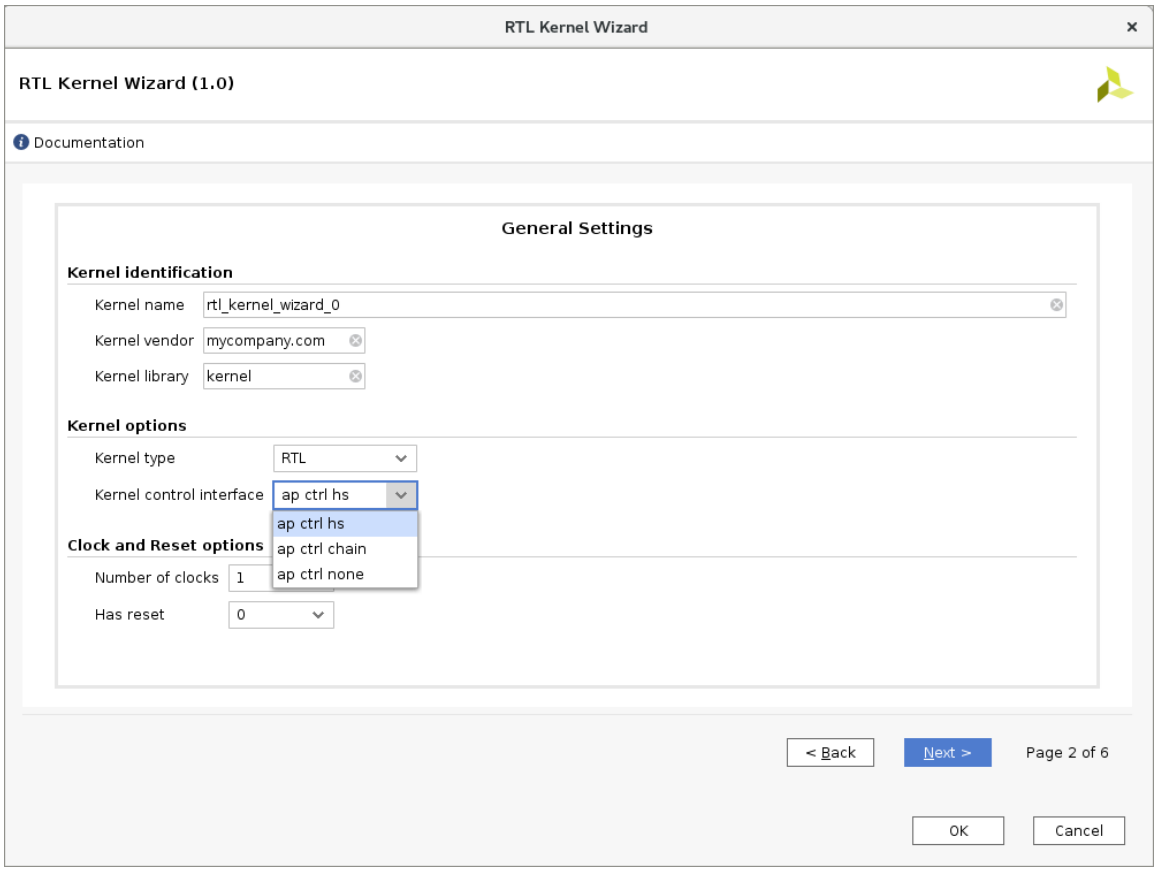
要浏览各页面，请按需单击“Next”和“Back”。

要最终完成内核并根据内核规格来构建工程，请单击“汇总 (Summary)”页面上的“OK”。

常规设置

下图显示了“General Settings”页面中的 3 项设置。

图 117：**RTL Kernel Wizard 常规设置页面**



以下显示的是“General Settings”页面中的 3 项设置。

“Kernel Identification”

- “Kernel name”：内核名称。这是 IP、顶层模块、内核和 C/C++ 函数模型的名称。该标识符应符合 C 语言和 Verilog 标识符命名规则。它还必须符合 Vivado IP integrator 命名规则，该规则禁止使用下划线，除非置于字母数字字符之间。
- “Kernel vendor”：供应商名称。用于 《Vivado Design Suite 用户指南：采用 IP 进行设计》(UG896) 中描述的供应商/库/名称/版本 (VLNV) 格式。
- “Kernel library”：库名称。用于 VLNV。必须符合相同的标识符规则。

“Kernel options”

- “Kernel type”：“RTL Kernel”Wizard 当前支持 2 种类型的内核：RTL 和块设计。
  - “RTL”：RTL 类型的内核由 Verilog RTL 顶层模块与该顶层模块内所包含的 Verilog 控制寄存器模块和 Verilog 内核示例组成。
  - “Block Design”：块设计类型的内核同样可提供 Verilog 顶层模块，但它在顶层模块内例化的是 IP integrator 模块框图。块设计包括 MicroBlaze™ 子系统，该子系统使用块 RAM 交换存储器来仿真控制寄存器。示例软件 MicroBlaze 随工程一起提供，以演示如何使用 MicroBlaze 控制内核。
- “Kernel control interface”：有 3 种类型的控制接口可用于 RTL 内核。分别是：`ap_ctrl_hs`、`ap_ctrl_chain` 和 `ap_ctrl_none`。此类接口可为 `<kernel>` 标签定义 `hwControlProtocol`，如 RTL 内核 XML 文件 中所述。

“Clock and Reset options”

- “Number of clocks”：设置内核使用的时钟数。每个 RTL 内核都有一个基准时钟（称为 `ap_clk`）和一个可选复位（称为 `ap_rst_n`）。内核上的所有 AXI 接口均由该时钟驱动。

  将“Number of clocks”设为 2 时，将提供辅助时钟和可选复位以供内核在内部使用。此辅助时钟和复位称为 `ap_clk_2` 和 `ap_rst_n_2`。该辅助时钟支持独立的频率缩放，并且独立于基准时钟。如果内核时钟需要以比 AXI4 接口更快或更慢的速率运行，辅助时钟非常有用，并且必须在基准时钟上进行时钟设置。

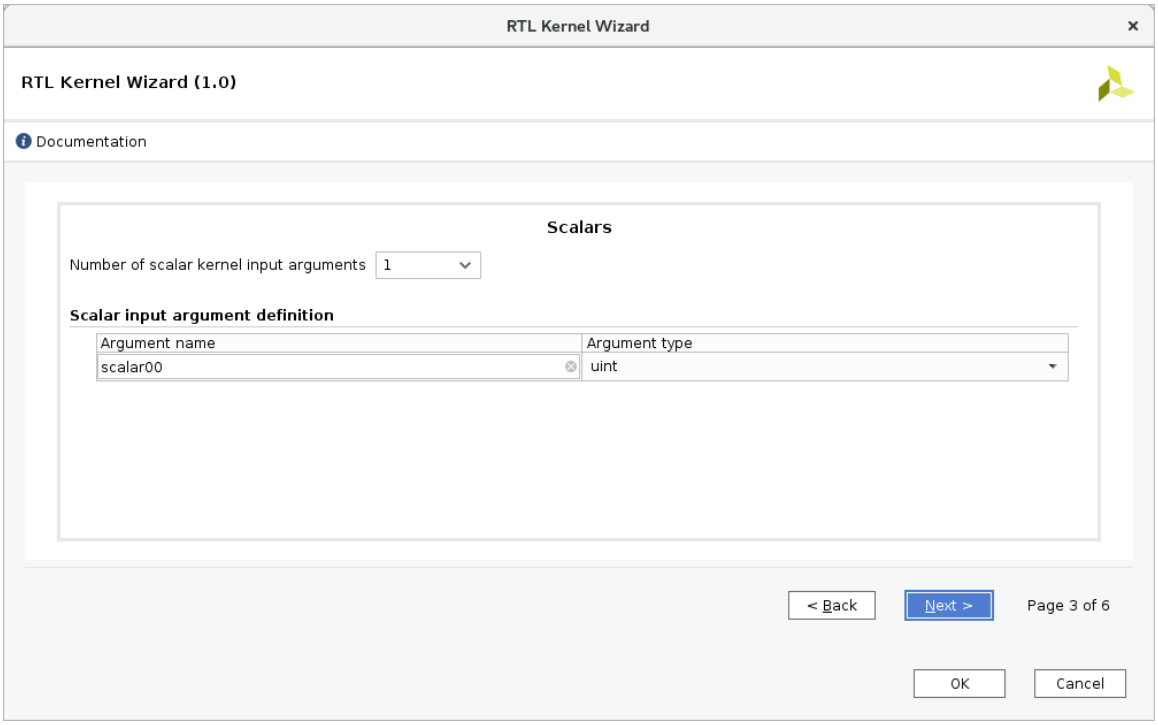
  重要提示！采用多时钟设计时，必须使用适当的时钟域交汇技术来确保所有时钟频率场景下的数据完整性。如需了解更多信息，请参阅 《UltraFast 设计方法指南（适用于赛灵思 FPGA 和 SoC）》(UG949)。

- “Has reset”：指定是否包含内核的顶层复位输入端口。省略复位对于改善大型设计的布线拥塞非常有用。通常在设计中具有复位的任何寄存器都应具有适当的初始值以确保正确性。如果启用，则每个时钟都包含一个复位端口。块设计类型的内核必须具有复位输入。

标量

标量实参用于将控制类型的信息传递给内核。标量实参无法从主机读取。对于指定的每个实参，创建相应的寄存器以便于将实参从软件传递到硬件。请参阅下图。

图 118： **“RTL Kernel”Wizard 的“Scalars”页面**



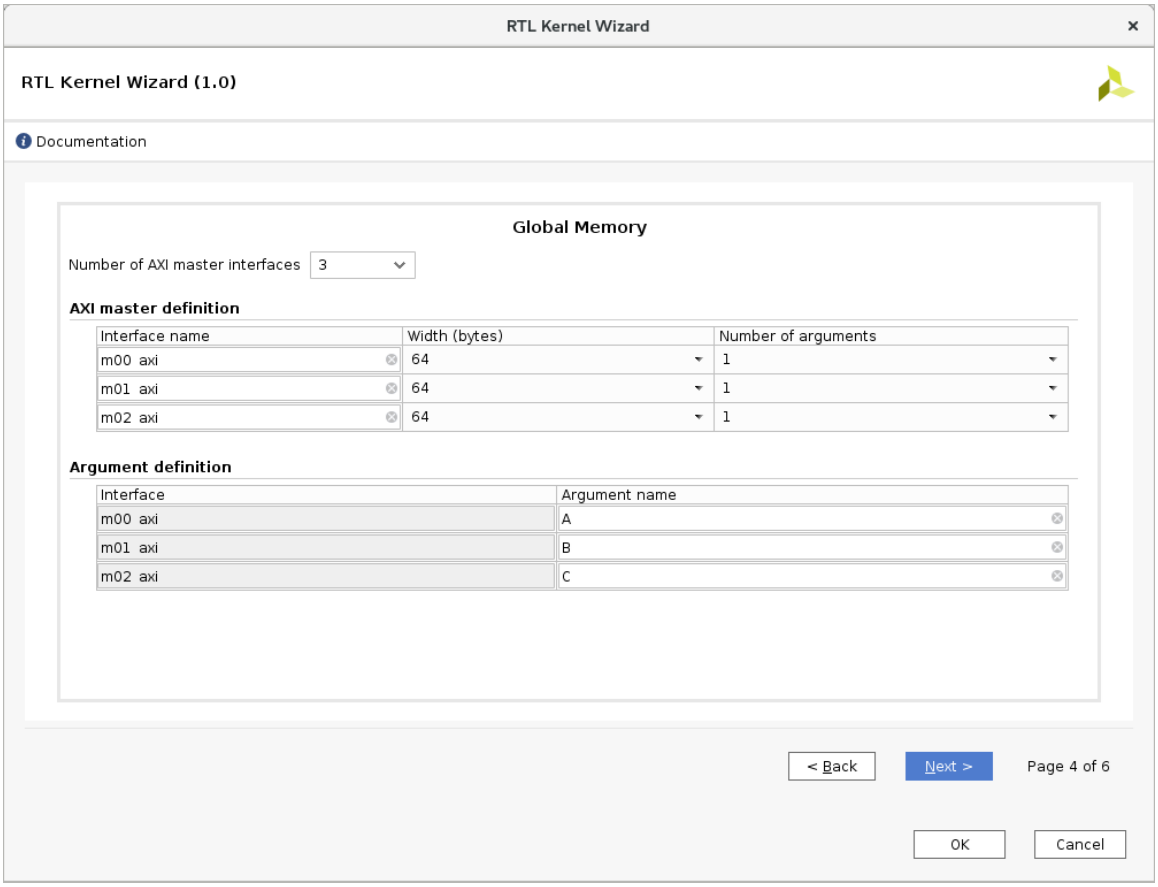
- “Number of scalar kernel input arguments”：指定要传递给内核的标量输入实参的数量。对于指定的每个数字，将生成一个表行，允许定制实参名称和实参类型。标量数目没有下限，此向导允许的最大数目为 64 个。

以下是标量输入实参的定义：

- “Argument name”：实参名称在生成的 Verilog 控制寄存器模块中用作输出信号。为每个实参赋予一个 ID 值。此 ID 值用于从主机软件访问该实参。在此向导的汇总页面上可以找到 ID 赋值方式。为确保最大兼容性，实参名称遵循与内核名称相同的标识符规则。
- “Argument type”：指定实参的数据类型，从而指定位宽。这会影响生成的 RTL 内核模块中的寄存器宽度。可用的数据类型仅限于 OpenCL C 规范版本 2.0 的“6.1.1 内置标量数据类型”部分中指定的数据类型。此规范为每种数据类型提供了关联的位宽。无论实参类型如何，RTL Wizard 都会在寄存器映射中为所有标量保留 64 位。如果实参类型为 32 位或更少，则 RTL Wizard 将把已分配的 64 位中的上半 32 位设置为保留地址位置。位宽大于 32 位的数据类型需要对控制寄存器进行两次写入操作。

全局存储器

图 119：**RTL Kernel Wizard 全局存储器页面**



内核可通过 AXI4 主接口来访问全局存储器。每个 AXI4 接口彼此独立运行，并且每个 AXI4 接口可以连接到一个或多个存储器控制器以连接到诸如 DDR4 等片外存储器。全局存储器主要用于在内核与主机之间进行大数据集的双向往来传递。它还可以用于在内核之间传递数据。如需了解有关如何设计这些接口以实现最优性能的建议，请参阅 AXI4 接口的存储器性能最优化。

提示：对于每个接口，“RTL Kernel”Wizard 会在顶层封装器中生成 AXI 主逻辑示例以作为起点，如果需要，可将其丢弃。

- “Number of AXI master interfaces”：指定内核上存在的接口数。最多 16 个接口。对于每个接口，您可以定制接口名称、数据宽度和关联实参的数量。每个接口都包含所有读写通道。“RTL Kernel”Wizard 建议的默认名称为 `m00_axi` 和 `m01_axi`。如果不更改这些名称，那么向全局存储器分配接口（如 将内核端口映射到存储器 中所述）时，这些名称可供使用。

“AXI master definition”**（表格中的列）**

- “Interface name”：指定接口的名称。为确保最大兼容性，实参名称遵循与内核名称相同的标识符规则。
- “Width (in bytes)”：指定 AXI 数据通道的数据宽度。赛灵思建议与存储器控制器 AXI4 从接口的本机数据宽度保持匹配。存储器控制器从接口宽度通常为 64 个字节（512 位）。

- “Number of arguments”：指定与此接口关联的实参数目。每个实参均表示一个指向全局存储器的数据指针，该全局存储器可供内核访问。

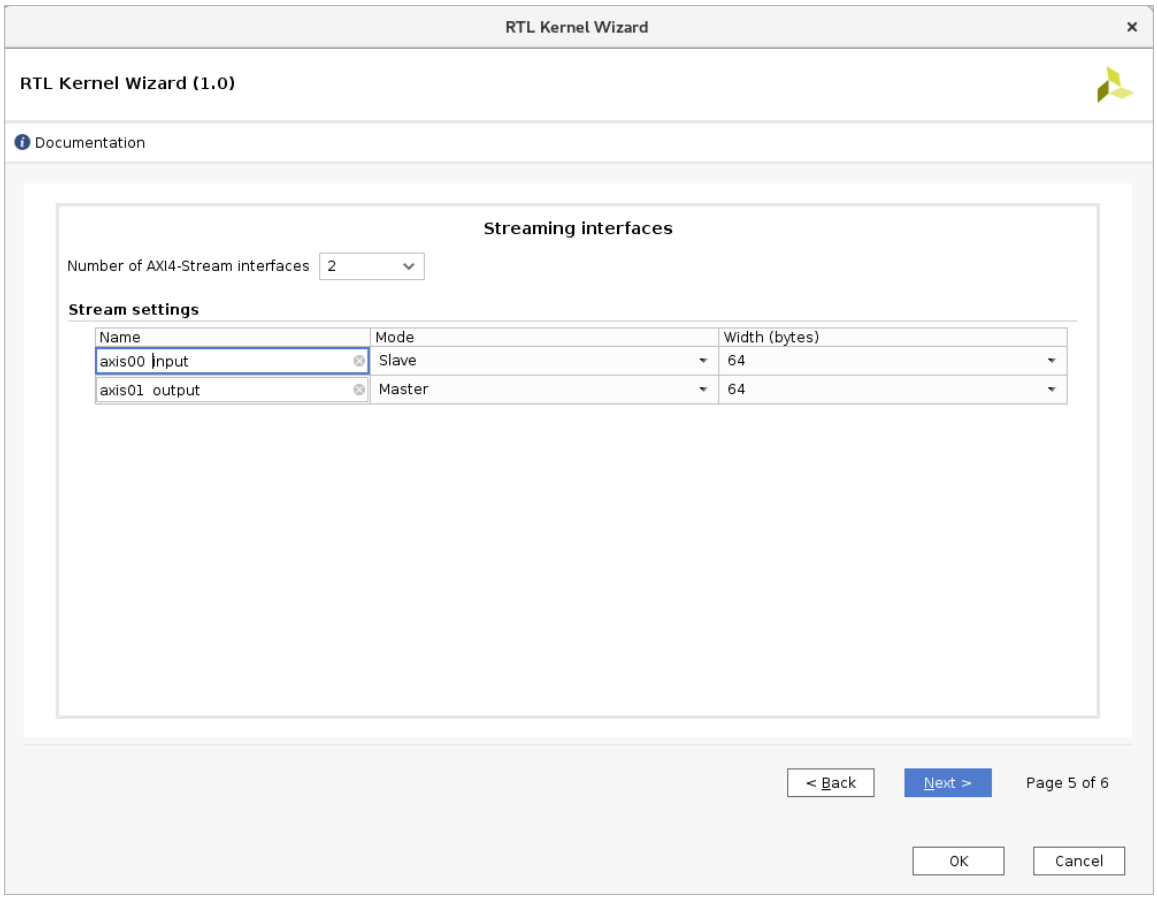
“Argument definition”

- “Interface”：指定 AXI 接口的名称。该值是从表格中定义的接口名称复制所得，且在此处无法修改。
- “Argument name”：指定指针实参的名称，此名称与函数原型特征符上显示的名称相同。为每个实参赋予一个 ID 值。此 ID 值用于从主机软件访问该实参，如 主机编程 中所述。在此向导的汇总页面上可以找到 ID 赋值方式。为确保最大兼容性，实参名称遵循与内核名称相同的标识符规则。在生成的 RTL 内核控制寄存器模块中，实参名称用作为输出信号。

数据流传输接口

“数据流传输 (Streaming Interfaces)”页面允许配置内核上的 AXI4-Stream 接口。数据流传输接口仅在选定的平台上可用，如果所选平台不支持数据流传输，则不显示此页面。数据流传输接口用于主机到内核与内核到主机的直接通信，以及连续操作内核，如 数据流传输 中所述。

图 120：**“RTL Kernel”Wizard 数据流传输接口页面**



- “Number of AXI4-Stream interfaces”：指定内核上存在的 AXI4-Stream 接口数。每个内核最多可启用 32 个接口。赛灵思建议尽量减少接口数量，以减少消耗的面积。
- “Name”：指定接口的名称。为确保最大兼容性，实参名称遵循与内核名称相同的标识符规则。

- “Mode”：指定接口为主接口还是从接口。AXI4-Stream 从接口为只读接口，可通过 `clWriteStream` API 从主机程序向 RTL 内核发送数据。AXI4-Stream 主接口为只写接口，主机程序可以通过 `clReadStream` API 经由该接口接收数据。
- “Width (bytes)”：指定 AXI4-Stream 接口的 `TDATA` 宽度（以字节为单位）。此接口宽度限制为 1 - 64 个字节（2 次幂）。

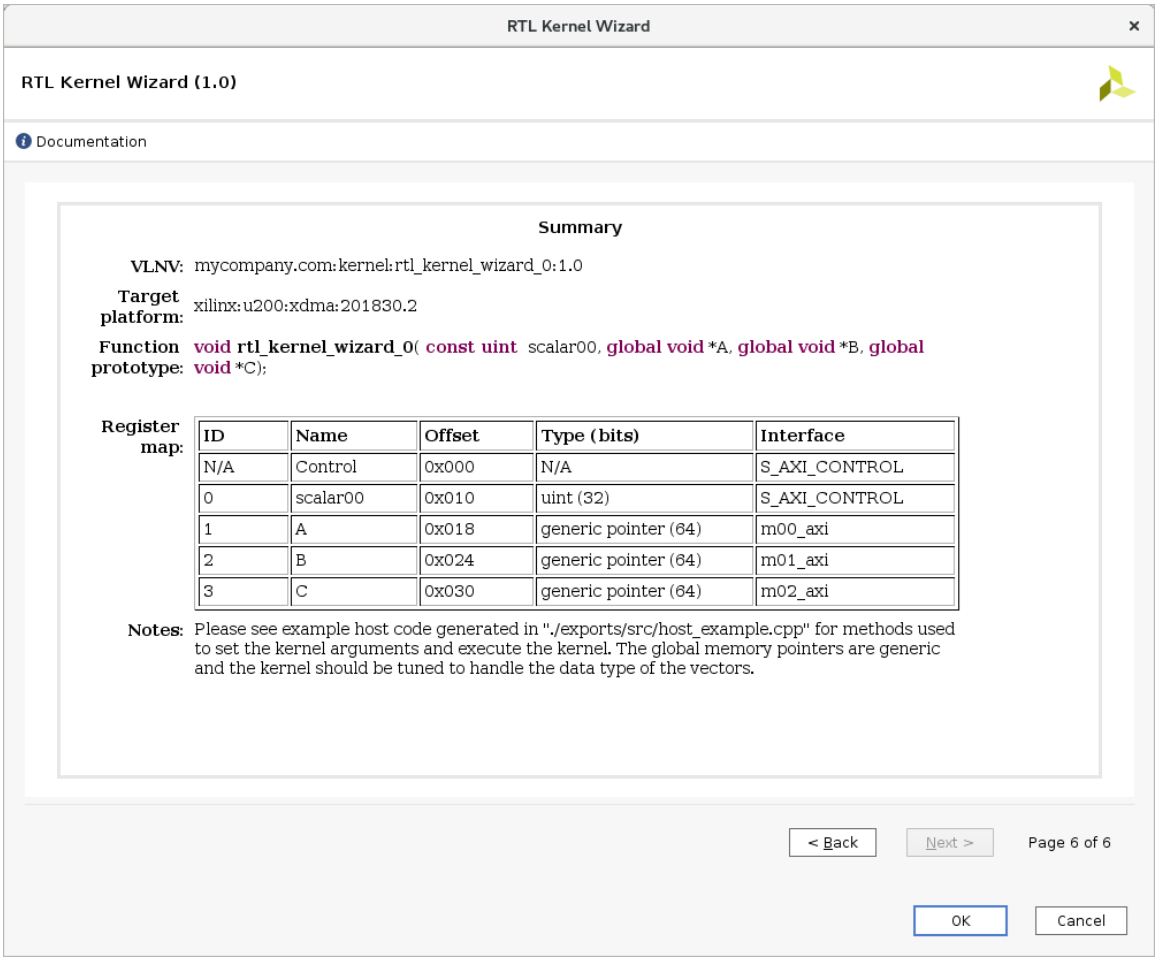
数据流传输接口使用 AXI4-Stream 协议的 `TDATA/TKEEP/TLAST` 信号。数据流传输事务由一系列传输组成，其中最终传输以 `TLAST` 信号断言有效来终止。数据流传输必须遵循如下规则：

- `TVALID/TREADY` 同时断言有效时，发生 AXI4-Stream 传输。
- `TDATA` 位宽必须为 8、16、32、64、128、256 或 512 位。
- `TLAST` 为 0 时，`TKEEP`（每个字节）必须全部为 1。
- `TLAST` 为 1 时，`TKEEP` 可用于发出尾部参差不齐信号。例如，在 4 字节接口上，`TKEEP` 只能是 `0b0001`、`0b0011`、`0b0111` 或 `0b1111`，以指定最后一次传输的大小分别为 1 字节、2 字节、3 字节或 4 字节。
- `TKEEP` 不能全为零（即使 `TLAST` 是 1 也是如此）。
- `TLAST` 必须在数据包的末尾断言有效。
- 如果未启动内核，则 `TREADY` 输入/`TVALID` 输出应为低电平，以避免丢失传输。

总结

本节总结了 RTL 内核 IP 的 VLNV、软件函数原型设计，以及基于前几页中选定选项创建的硬件控制寄存器。函数原型表述的是对 C 语言函数执行内核调用的方式。请参阅主机代码生成的示例，了解如何为内核调用设置内核实参。寄存器映射显示的是主机软件 ID、实参名称、硬件寄存器偏移、类型和关联接口之间的关系。在继续生成内核之前，请查看本节以了解正确的处理方式。

图 121： **“Kernel”Wizard 的“Summary”页面**



单击“OK”即可生成 RTL 内核的顶层封装文件、VADD 临时 RTL 内核 IP、`kernel.xml` 文件、仿真测试激励文件以及 `host.cpp` 代码示例。创建这些文件后，“RTL Kernel”Wizard 会在 Vivado Design Suite 中打开工程以便您完成内核开发。

在 Vivado IDE 中使用 RTL 内核工程

如果您从 Vitis IDE 启动“RTL Kernel”Wizard，单击“Summary”页面上的“OK”后，将打开 Vivado Design Suite，其中包含 IP 工程示例，您可根据此示例完成自己的 RTL 内核代码。

如果您从 Vivado IP 目录内启动“RTL Kernel”Wizard，那么单击“Summary”页面上的“OK”后，“RTL Kernel”Wizard 将例化到您的当前工程中。您必须在其中执行以下步骤：

1. 当“Generate Output Products”对话框出现时，单击“Skip”以将其关闭。
2. 右键单击已添加到“Sources”视图中的 `<kernel_name>.xci` 文件，然后选择“Open IP Example Design”。
3. 在“Open Example Design”对话框中，指定“Example project directory”，或者接受默认值，然后单击“OK”。

提示：这样会为 RTL 内核 IP 创建一个工程示例。此 IP 工程示例与从 Vitis IDE 启动“RTL Kernel”Wizard 时创建的工程示例相同，您将在其中完成内核开发工作。

4. 现在，您可以关闭从中启动“RTL Kernel”Wizard 的原 Vivado 工程了。

根据您为内核选项所选的“Kernel Type”，在 IP 工程示例中将填充顶层 RTL 内核文件，此文件中包含 Verilog 示例和控制寄存器（如 RTL 类型的内核工程 中所述）或者包含已例化的 IP integrator 块设计（如 块设计类型内核工程 中所述）。顶层 Verilog 文件包含预期的输入/输出信号和参数。这些顶层端口将与内核规范文件 (`kernel.xml`) 相匹配，并且可与您的 RTL 代码或块设计相结合以使 RTL 内核更完整。

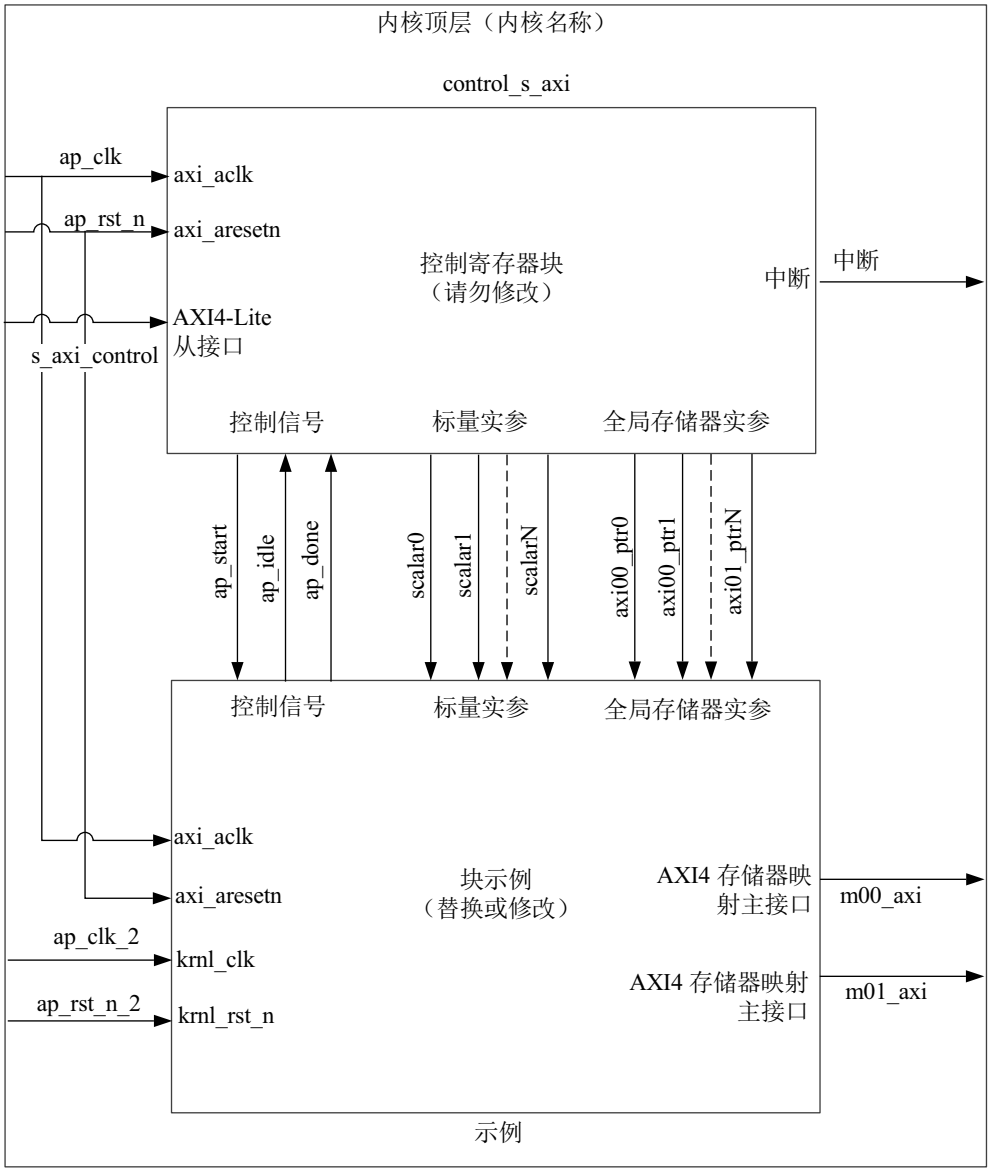
顶层文件中定义的 AXI4 接口包含生成高效且高吞吐量接口所需的一小部分 AXI4 信号。不显示的信号在连接到 AXI 系统其余部分时将继承经优化的默认值。这些最优化的默认值允许系统省略不需要的 AXI 功能，从而节省面积并降低复杂性。如果您的 RTL 代码或块设计包含已省略的 AXI 信号，那么您可将这些信号添加到顶层 RTL 内核文件中的端口，IP 封装器将对其进行相应调整。

该过程的下一步是自定义内核的内容，然后将这些内容封装到赛灵思对象 (`xo`) 文件中。

RTL 类型的内核工程

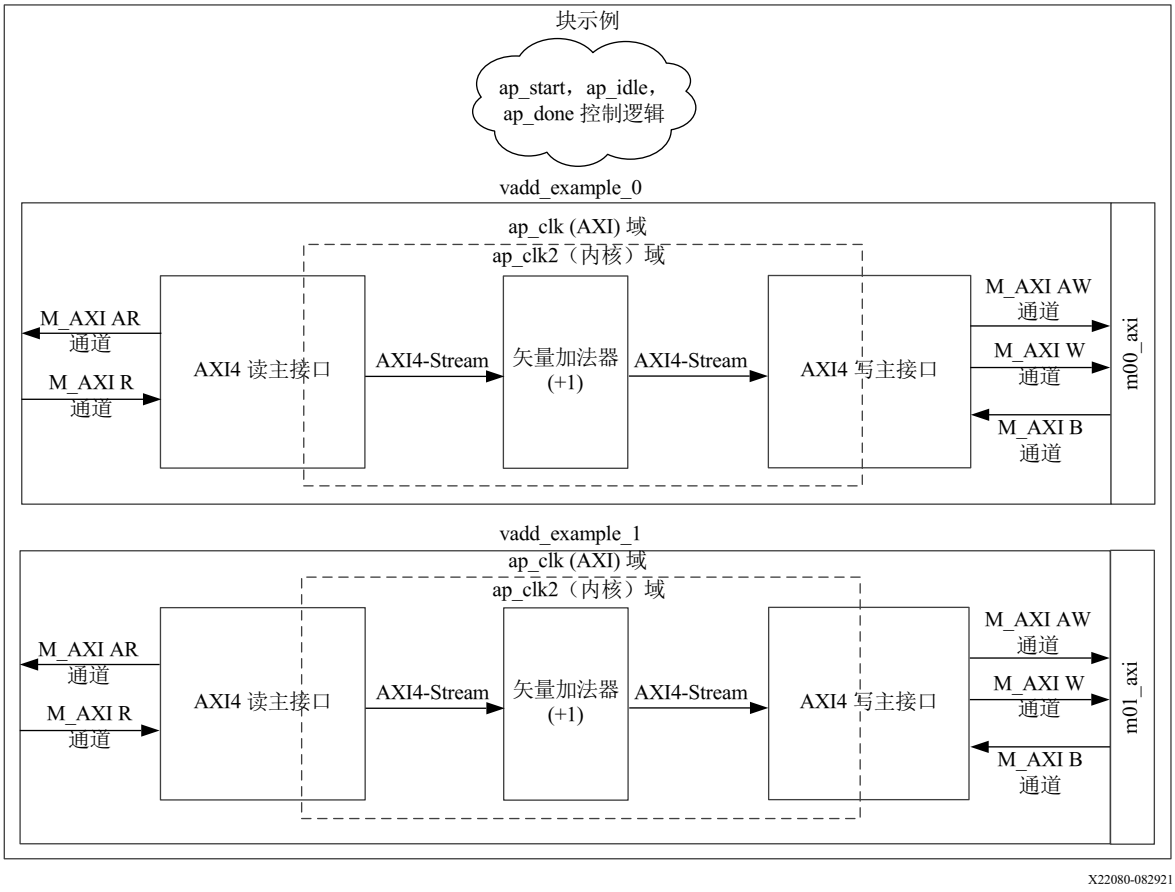
RTL 类型的内核可以提供顶层 Verilog 设计，此设计由控制寄存器和 `Vadd` 子模块设计示例组成。下图显示了配置有 2 个 AXI4 主接口的顶层设计。如果修改了控制寄存器模块以确保它仍与位于 Vivado 内核工程的导入目录中的 `kernel.xml` 文件保持一致，则应该谨慎处理。此块示例可替换为您的定制逻辑，或者用作为设计的起点。

图 122：**内核类型 RTL 顶层**



下图所示的 `Vadd` 块示例由简单的加法器函数、1 个 AXI4 读取主接口和 1 个 AXI4 写入主接口组成。每个已定义的 AXI4 接口都有一个独立的加法器代码示例。每个接口的第一个关联实参用作示例的数据指针。每个示例读取 16 KB 数据，执行 32 位加 1 操作，然后将 16 KB 数据写回原位（读写地址相同）。

图 123：**内核类型 RTL 示例**



下表描述了 IP 工程示例内的部分重要文件，这些文件与内核的 Vivado 工程根目录有关，其中 `<kernel_name>` 是您在“RTL Kernel”Wizard 中指定的内核名称。

表 57：**“RTL Kernel”Wizard 的源文件和测试激励文件**

文件名	描述	随内核类型提供
<kernel_name>_ex.xpr	Vivado 工程文件	全部
imports 目录		
<kernel_name>.v	内核顶层模块	全部
<kernel_name>_control_s_axi.v	RTL 控制寄存器模块	RTL
<kernel_name>_example.sv	RTL 块示例	RTL
<kernel_name>_example_vadd.sv	RTL AXI4 矢量加法块示例	RTL
<kernel_name>_example_axi_read_master.sv	RTL AXI4 读取主接口示例	RTL
<kernel_name>_example_axi_write_master.sv	RTL AXI4 写入主接口示例	RTL
<kernel_name>_example_adder.sv	RTL AXI4-Stream 加法器块示例	RTL
<kernel_name>_example_counter.sv	RTL 计数器示例	RTL
<kernel_name>_exdes_tb_basic.sv	仿真测试激励文件	全部
<kernel_name>_cmodel.cpp	用于软件仿真的软件 C 语言模型示例。	全部
<kernel_name>_ooc.xdc	非关联赛灵思约束文件	全部

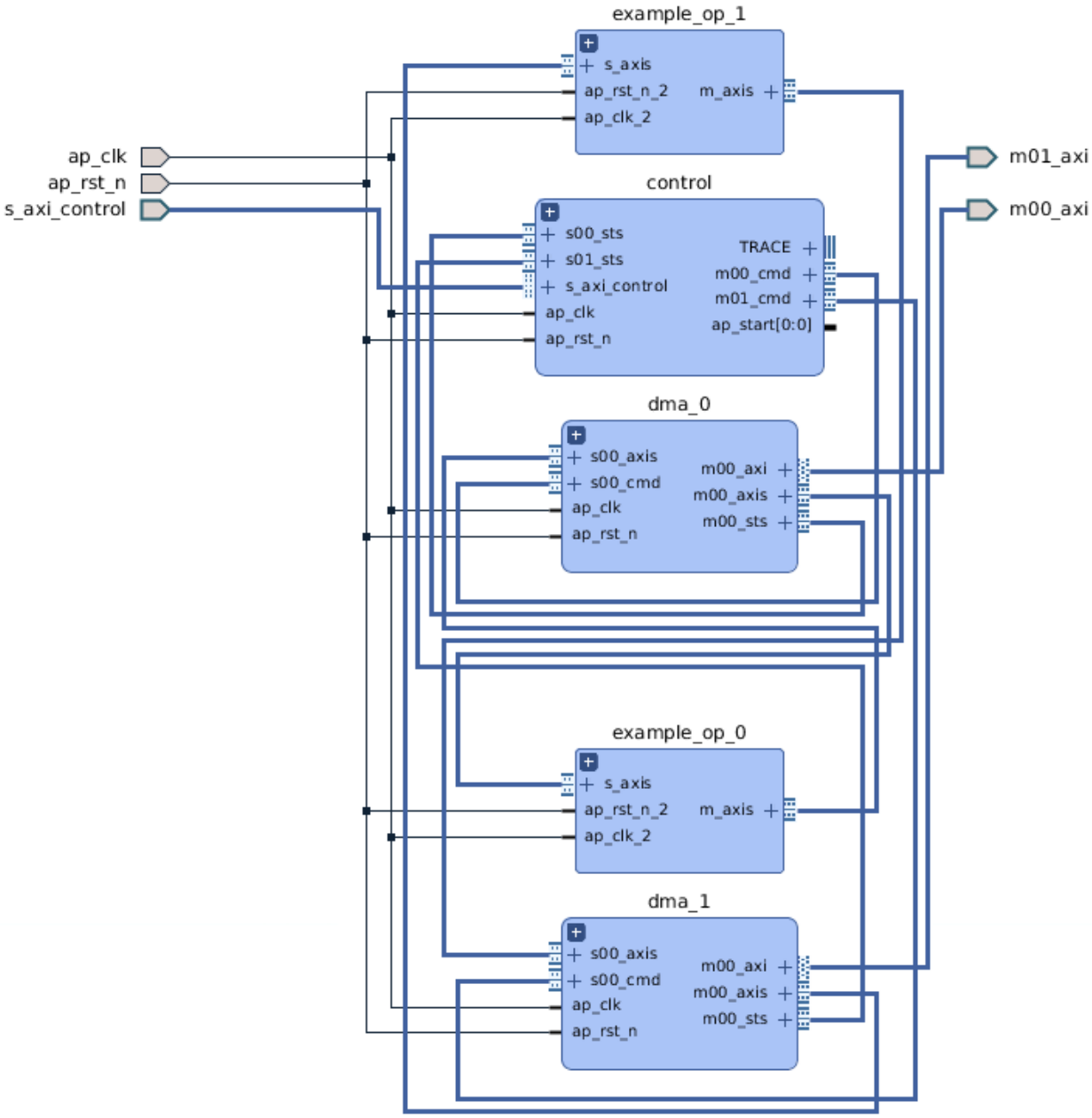
表 57： **“RTL Kernel” Wizard 的源文件和测试激励文件** (续)

文件名	描述	随内核类型提供
<kernel_name>_user.xdc	用于内核用户约束的赛灵思约束文件。	全部
kernel.xml	内核描述文件	全部
package_kernel.tcl	内核封装脚本 proc 定义	全部
post_synth_impl.tcl	Tcl 实现后文件	全部
exports 目录		
src/host_example.cpp	主机代码实例	全部
makefile	makefile 示例	全部

块设计类型内核工程

块设计类型内核可在工程示例顶层提供 IP integrator 块设计 (.bd)。MicroBlaze 处理器子系统用于对控制寄存器进行采样并控制内核流程。MicroBlaze 处理器系统则使用块 RAM 代替寄存器文件，用作为主机与内核之间的交换存储器。

图 124：**块设计类型内核**



对于每个 AXI 接口，都会创建 DMA 和数学运算子块以提供有关如何控制内核执行的示例。此示例使用 MicroBlaze AXI4-Stream 接口来控制 AXI Data Mover IP，以创建与 RTL 内核类型中的示例相同的示例。并且，其中还包含 Vitis IDE 工程，用于为 MicroBlaze 核编译和链接 ELF 文件。该 ELF 文件加载到 Vivado 内核工程中并直接初始化到 MicroBlaze 指令存储器中。

以下步骤可用于修改 MicroBlaze 处理器程序：

1. 如果设计已更新，您可能需要运行“Export Hardware”选项。该选项可在“File” → “Export” → “Export Hardware”菜单位置中找到。打开“Export Hardware”对话框后，请单击“OK”。
2. 现在，核开发套件应用已可供调用。请从主菜单中依次选中“Tools” → “Launch Vitis”。

3. 打开 Vitis IDE 后，单击“Welcome”选项卡上的文本右侧的“X”，以关闭欢迎对话框。这样即可显示该对话框下已加载的 Vitis IDE 工程。
4. 在“Project Explorer”中，源文件位于 `<Kernel Name>_control/src` 部分下方。请根据需要修改这些内容。
5. 更新完成后，请选中菜单选项“Project” → “Build All” → “Check for errors/warnings and resolve if necessary”以编译源文件。ELF 文件会在 IDE 中自动更新。
6. 运行仿真即可测试更新后的程序，并根据需要进行调试。

仿真测试激励文件

生成的 SystemVerilog 测试激励文件用于对 IP 工程示例进行仿真。此测试激励文件通过实践 RTL 内核来确保其正常执行运算。在此文件中会填充检查器函数来验证 `add one` 运算。

这个生成的测试激励文件可以用作验证内核功能的起点。它从控制寄存器写入/读取并多次执行内核，同时还包括简单的复位测试。它还可用于调试 AXI 问题、复位问题、多次迭代期间的错误以及内核功能。与硬件仿真相比，它对硬件极端情况执行更严格的测试，但不测试主机代码和内核之间的交互。

要运行仿真，请单击位于 GUI 左侧的“Vivado Flow Navigator” → “Run Simulation”，然后选中“Run Behavioral Simulation”。如果行为仿真按期望方式运行，那则可运行综合后功能仿真来确保综合结果与行为模型匹配。

非关联综合

Vivado 内核工程配置为在非关联 (OOC) 模式下运行综合和实现。设计中填充了赛灵思设计约束 (XDC) 文件，以便为此目的提供默认时钟频率。

应始终先综合 RTL 内核，然后再使用 `package_xo` 命令对其进行封装。运行综合有助于确定内核综合不含错误。它还可提供资源使用情况和运行频率的估算结果。如果不对 RTL 内核进行预综合，那么在 `v++` 链接进程中可能会遇到错误，并且错误原因的调试难度可能会更高。

要运行 OOC 综合，请单击“Vivado Flow Navigator” → “Synthesis”菜单中的“Run Synthesis”。

已综合的输出也可用于将 RTL 内核与网表源文件（而不是 RTL 源文件）封装在一起。

重要提示！块设计类型的内核则必须使用 `package_xo` 命令作为网表来进行封装。

软件模型和主机代码示例

在 `./imports` 目录中提供了 `add one` 运算示例的 C++ 软件模型 `<kernel_name>_cmodel.cpp`。您也可通过修改此软件模型来对自己的内核函数进行建模。运行 `package_xo` 时，此模型可包含在内核源文件中，以供为内核启用软件仿真。硬件仿真和系统构建始终使用内核的 RTL 描述。

在 `./exports/src` 目录中，提供了一个主机程序示例，称为 `host_example.cpp`。主机程序采用二进制容器作为程序的实参。主机代码会加载二进制文件，作为 `init` 函数的一部分。主机代码会例化内核、分配缓冲器、设置内核实参、执行内核，然后收集并检查 `add one` 函数示例的结果。

如需了解有关在应用中使用主机程序和内核代码的信息，请参阅 创建 Vitis IDE 工程。

生成 RTL 内核

在 Vivado IDE 的 IP 工程示例中设计并测试内核后，最后一个步骤是生成 RTL 内核对象 (XO) 文件以供 Vitis 编译器使用。

在“Vivado Flow Navigator” → “Project Manager”菜单中单击“Generate RTL Kernel”命令。这样会打开“生成 RTL 内核 (Generate RTL Kernel)”对话框，其中包含 3 个主要的封装选项：

- “Sources only kernel”：直接使用 RTL 设计源文件来封装内核。
- “Pre-synthesized kernel”：将内核与 RTL 设计源文件封装在一起，并在其中包含已综合且已缓存的输出，此输出可供稍后在流程中使用，以避免重新综合。如果目标平台发生更改，则封装的内核可能会回退到 RTL 设计源文件，而不是使用缓存的输出。
- “Netlist (DCP) based kernel”：使用内核的已综合输出所生成的网表，将内核封装为黑盒。如有必要，可以选择加密此输出。如果目标平台发生更改，则内核可能无法重新定位新器件，并且必须从源文件重新生成。如果设计包含块设计，则基于网表 (DCP) 的内核是唯一可用的封装选项。

（可选）“Software Emulation Sources”字段允许您指定内核的软件模型，此模型可在软件仿真期间使用。如果软件模型包含多个文件，请在“源文件”列表中的每个文件之间提供空格，或者使用 GUI，在选择文件时按住 “CTRL”键来选择多个文件。

单击“OK”后，将生成内核输出文件。如果选择了预综合的内核或网表内核选项，则可以运行综合。如果先前已运行综合，那么它将使用这些输出，即使这些输出已过时也是如此。内核的赛灵思对象 (XO) 文件是在 Vivado 内核工程的 `exports` 目录中生成的。

此时，您可以关闭 Vivado 内核工程。如果 Vivado 内核工程是从 Vitis IDE 调用的，则会将名为 `host_example.cpp` 的主机代码示例与内核赛灵思对象 (XO) 文件自动导入 Vitis IDE 中的应用工程的 `./src` 文件夹。

修改从向导生成的现有 RTL 内核

在 Vitis IDE 中，您可从使用现有 RTL 内核的应用工程的 `./src` 文件夹中选中该内核。右键单击“Project Explorer”视图中的 XO 文件，然后选中“RTL Kernel Wizard”。这样 Vitis IDE 会尝试为选定的 RTL 内核打开 Vivado 工程。

提示：如果 Vitis IDE 无法找到 Vivado 工程，它会返回错误，且不允许您编译 RTL 内核。

此时会显示一个对话框，其中包含 2 个用于编辑现有 RTL 内核的选项。选择“Edit Existing Kernel Contents”会重新打开 Vivado 工程，以便您修改和重新生成内核内容。选择“Re-customize Existing Kernel Interfaces”则会打开“RTL Kernel”Wizard。除“Kernel Name”外，其它选项均可修改，并且可替换原先的 Vivado 工程。

重要提示！创建经过更新的 RTL 内核工程后，原先 Vivado 工程中的所有文件和更改都将丢失。

第八部分

使用 Vitis 嵌入式平台

本部分包含以下章节：

第 48 章

Vitis 嵌入式平台

引言

Vitis™ 统一软件平台可提供“平台 + 内核”结构，帮助开发者专注于处理应用程序。平台与内核相分离有助于使平台可供复用于多种内核，且反之亦然。

赛灵思为 Alveo™ 和嵌入式评估板提供了预构建的平台。您既可以自由创建自己的嵌入式平台，也可以自定义赛灵思嵌入式平台。

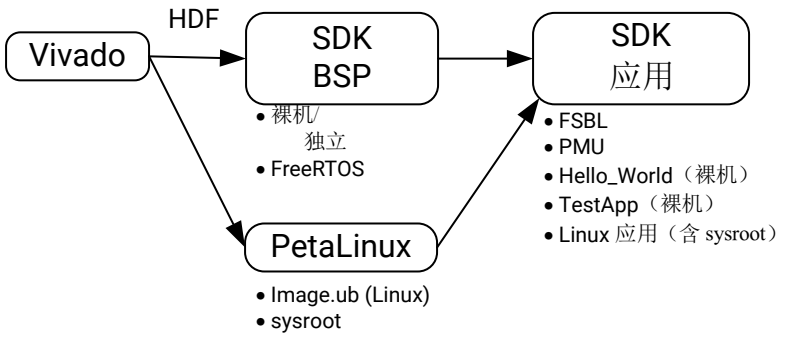
Vitis 软件平台提供了相应的环境，适合用于在基于 FPGA、Zynq®-7000 SoC 和 Zynq® UltraScale+™ MPSoC 的异构平台上创建嵌入式软件和加速应用。本文档侧重于介绍如何将嵌入式平台用于 Zynq UltraScale+ MPSoC。

平台类型

Vitis 目标平台可通过独特的硬件和软件组件来进行自定义。平台一般有两种类型：固定平台和可扩展平台。固定平台支持嵌入式软件开发，它直接模拟先前用于通过赛灵思 SDK 工具进行软件开发的硬件定义文件。可扩展平台支持应用加速开发流程，包含用于支持加速内核和为 Versal™ ACAP 控制 AI 引擎的硬件，并包含适用于运行 Linux 和赛灵思的 Xilinx Runtime (XRT) 库的软件。如需了解有关 XRT 库的更多信息，请访问 https://github.com/Xilinx/XRT。

下图显示的是用于嵌入式软件应用开发的传统 SDK 流程。赛灵思硬件设计文件 (HDF) 是从 Vivado® Design Suite 导出的。它可供 SDK 用于生成板级支持包 (BSP) 以及创建应用该 BSP 的软件应用。

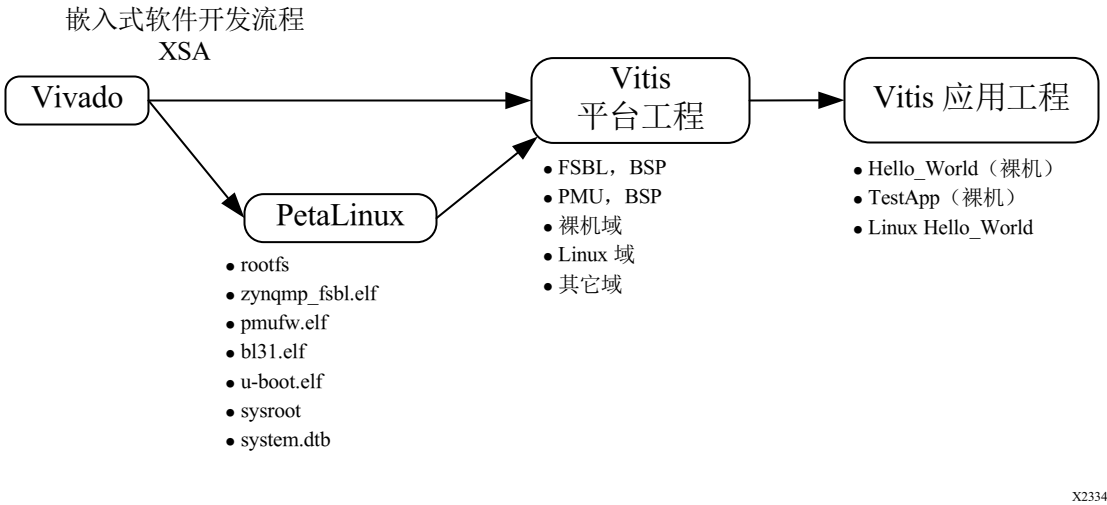
图 125：**2019.2 版本之前的 SDK 流程**



X233443-102119

下图显示的是 Vitis 嵌入式软件开发流程，此流程从 2019.2 起替换 SDK 流程。硬件规格现在包含在赛灵思 Shell 存档 (XSA) 中，从 Vivado 设计导出，但格式不同于 HDF，并使用 `.xsa` 作为文件扩展名。

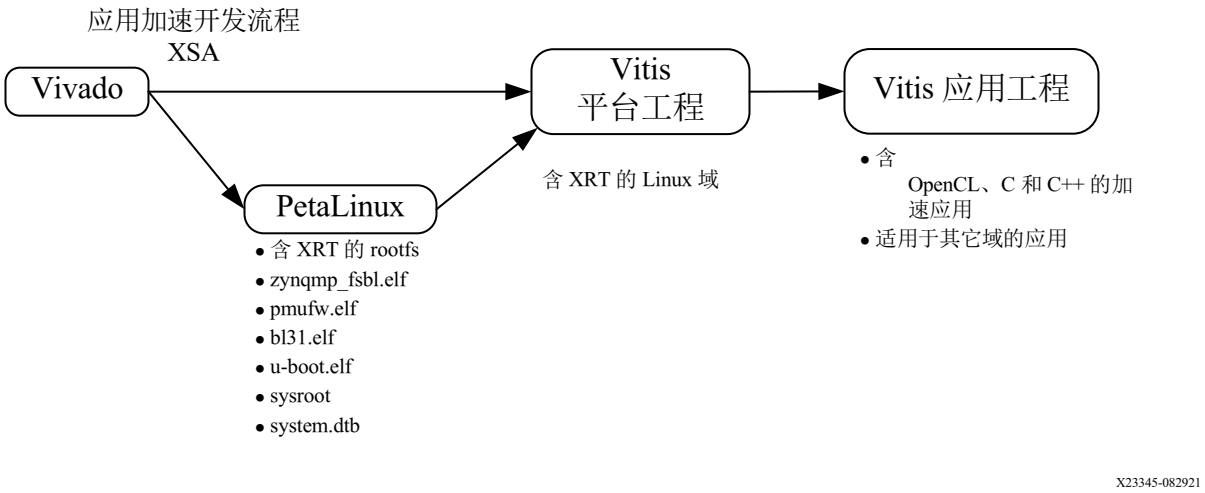
图 126：**Vitis 嵌入式软件开发流程**



Vitis 核工具会为固定 XSA 创建平台、BSP 和软件启动组件（例如，FSBL 和 PMU 固件），并且这些工具与 Vitis 平台关联。以固定平台为目标的软件应用可以通过 Vitis 嵌入式软件开发流程来进行开发。固定平台无需 Linux 和 XRT 库，但同样能够以运行裸机和 RTOS 操作系统的处理器域为目标。如需了解更多信息，请参阅《Vitis 统一软件平台文档》(UG1416) 中的 Vitis 嵌入式软件开发流程文档。

在 Vitis 应用加速开发流程中，Vivado Design Suite 还用于生成和写入可扩展 XSA，其中包含额外的 IP 块和元数据，用于支持内核连接。下图显示了加速软件开发流程。

图 127：**Vitis 加速内核流程**



Vitis 核工具支持以多种语言（OpenCL™、C 和 C++）来进行应用开发，但这些应用适用目标必须为 Vitis 平台。目标平台由硬件和软件组件组成，如下图所示。该页面左侧的目标平台视图适用于 Vitis 嵌入式软件开发流程，而页面右侧则显示支持加速内核的平台。两者之间的差异包括含 Linux + XRT 的目标的加速内核要求、元数据及内核接口声明等方面。

注释：在 https://github.com/Xilinx/Vitis_Embedded_Platform_Source 中提供了定制平台生成源代码。

图 128：**Vitis 目标平台**

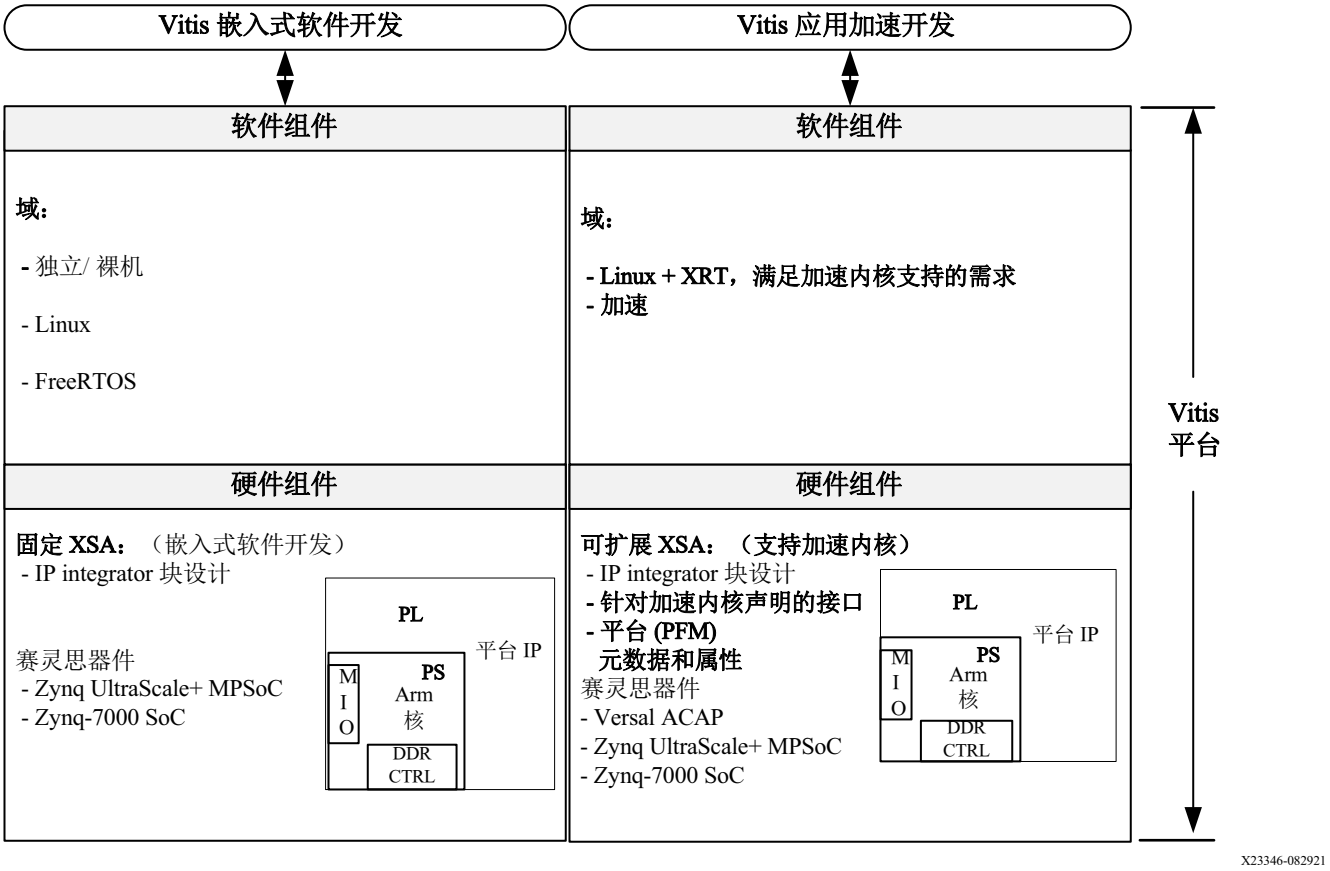
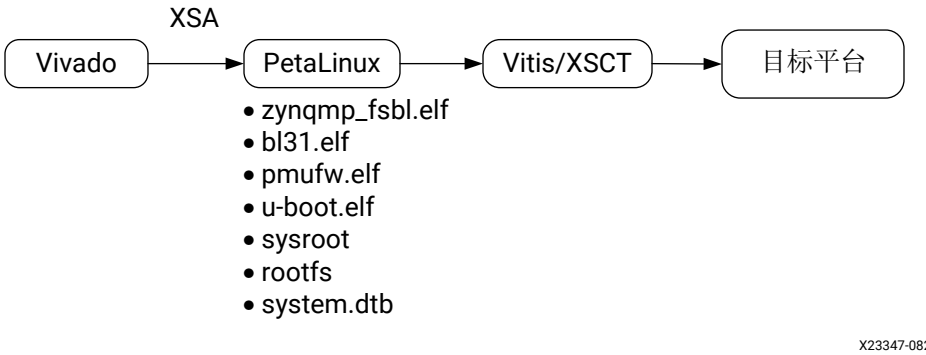


图 129：**Vitis 平台工程流程**



平台命名约定

在 Vitis 或目标平台中使用 Vitis 编译器 (v++) 创建加速应用时，会使用平台名称。预构建的平台镜像也会使用相同的文件名和目录名称。

预构建的 Vitis 嵌入式平台使用以下命名约定。

```
<Vendor>_<Board>_<Feature>_<Supported Vitis Tool Version>_<Release
                          Version>
```

其中：

- <Vendor>：开发板供应商。对于所有赛灵思创建的预构建平台，此处均使用 `xilinx`。
- <Feature>：此平台的特殊功能。例如：
  - `base` 表示它用于连接所有可用资源，以供您在加速应用中使用。
  - `DFX` 表示它支持赛灵思 Dynamic Function eXchange (DFX)。
- <Supported Vitis Tool Version>：Vitis 开发平台的特定版本，平台是专为此版本设计的。此项也表示创建预构建平台的 Vivado® Design Suite 工具的版本。
- <Release Version>：平台的发行版本。第一个版本为 `1`。

例如，以下平台名称遵循此命令约定：

- `xilinx_zcu102_base_202020_1`
- `xilinx_zcu104_base_202020_1`
- `xilinx_zc706_base_202020_1`
- `xilinx_zcu102_base_dfx_202020_1`

注释：平台源代码使用 git 分支来进行版本控制。目录名称为 `<Board>_<Feature>`（例如，`zcu102_base`）。根据 https://github.com/Xilinx/Vitis_Embedded_Platform_Source 中的源代码生成的平台名称为 `xilinx_zcu102_base_202020_1`。

嵌入式平台组件和架构

平台是 Vitis 设计的起点。Vitis 应用是基于平台构建的。

嵌入式平台包含硬件平台和软件平台。

硬件平台

硬件平台即硬件设计中保持静态不变的部分。它包含赛灵思支持存档 (XSA) 文件，此文件是从 Vivado Design Suite 导出的。

硬件平台描述了可供加速应用使用的平台硬件设置和加速资源，例如，输入和输出接口、时钟、AXI 总线和中断。Vitis 会根据需要向硬件设计添加内核和基础架构模块，以便于数据移植。加速内核能够与平台 IP 共享数据，但不能对其进行更改或修改。如需了解有关硬件平台设置的信息，请参阅 安装 Xilinx Runtime 和平台。

软件平台

软件平台作为环境，可供在其中运行软件来为加速应用控制加速内核。它包含域设置和启动组件设置。

默认情况下，所有赛灵思预构建的平台都包含 Linux 域，其中已启用赛灵思的 Xilinx Runtime (XRT) 以便加速应用可在此环境内运行。Linux 内核镜像和 rootfs 的预构建二进制文件位于 PetaLinux 下载页面上的独立下载文件中。请参阅赛灵思下载中心的“嵌入式 Vitis 平台通用镜像”部分。由于设备树对于每个平台都是唯一的，因此它作为组件随 Linux XRT 域一起在平台内部提供。

如果嵌入式平台中包含 Linux 域，则必须提供 Linux 域组件。这些组件可由 PetaLinux、Yocto 或第三方框架生成。由于这些组件可在给定 FPGA 系列的所有赛灵思演示板之间共享，因此为 Zynq-7000 SoC 器件和 Zynq UltraScale+ MPSoC 器件提供了公用的 Linux 组件镜像（由 PetaLinux 生成）。

以下 Linux 镜像可从 PetaLinux 下载页面下载：

- 根文件系统 (RFS)：包括 Linux 文件系统的二进制文件、库和设置。在赛灵思提供的通用 rootfs 中，已安装 XRT 以使加速应用可在此 Linux 环境上运行。
- 内核镜像：已编译的 Linux 内核。由赛灵思提供的通用内核镜像包含大部分赛灵思外设驱动程序。
- Sysroot：用于交叉编译。它可提供多个库，以供在为目标系统编译应用时链接。

注释： （可选）您可将 Linux 域组件打包到嵌入式平台中。在 Vitis IDE 中创建 Linux 应用时，如果在平台中已设置默认和初始设置，那么平台设置中的 Linux 域组件将采用这些设置。您可使用别处安装的组件来覆盖这些设置。

赛灵思预构建的嵌入式平台和预构建的通用 Linux 组件通过独立下载文件来提供。您可从 Vitis 嵌入式平台 GitHub 仓库上托管的平台源文件来重新生成通用 Linux 组件，具体方法是先设置环境变量 `COMMON_RFS_KRNL_SYSROOT=FALSE`，然后再运行 `make`。

安装嵌入式平台

预构建的 Vitis 嵌入式平台必须从赛灵思网站下载：

- 从 Vitis 嵌入式平台下载页面下载所需的 Vitis 平台。
- 从 PetaLinux 下载页面的“Vitis 嵌入式平台的常用镜像”部分下载常用 Linux 组件。

下载嵌入式平台后，有 3 种途径可将其包含到您的工程中：

- 将预构建平台解压到 `/opt/xilinx/platforms`。
- 将预构建平台解压到任意文件夹。将 `PLATFORM_REPO_PATHS` 环境变量设置为文件夹路径。

```
export PLATFORM_REPO_PATHS=/path/to/platforms
```

- 将预构建平台解压到任意文件夹。在 Vitis IDE 中，依次选中“Xilinx” → “Add Custom Platform”，然后选择文件夹路径。

常用 Linux 组件可解压到任意文件夹。创建 Linux 应用时，您将在“Vitis New Application Project”Wizard 中为组件指定路径。

XILINX

第 49 章

使用 Vitis 嵌入式平台

封装镜像

在 2020.1 版的 Vitis™ 编译器 (v++) 中添加了新的封装阶段。

在 2019.2 中，v++ 具有 2 个阶段：

- `-c` 或 `--compile`，用于编译加速内核
- `-l` 或 `--link`，用于将加速内核与平台逻辑相链接。

在 2020.1 版本中，新增阶段为 `-p` 或 `--package`。`V++ --package` 命令可生成 `boot.bin` 和 sd_card 镜像文件。

`--package` 命令支持 initramfs 镜像和 Ext4 rootfs 镜像。

在 Vitis IDE 中，在构建进程中将自动调用封装阶段。您可通过双击 `.sprj` 文件在系统工程详细信息页面中添加其它封装选项。封装日志文件、命令配置文件以及输出文件存储在 `package` 目录的 `Emulation-SW`、`Emulation-HW` 或 `Hardware` 目录下。

在命令行模式下，您可将封装选项作为 v++ 选项或以配置文件的形式来传入命令行。如需了解有关 `v++ --package` 选项的详细信息，请参阅 Vitis 编译器命令 或 `v++ -help` 命令以及赛灵思 Vitis 加速示例 GitHub 仓库。

在 Vitis IDE 中利用 Ext4 rootfs 封装镜像

向 Vitis IDE 提供 `ext4 rootfs` 时，生成的 `sd_card.img` 文件包含：

- 用于 PL 内核的 `xclbin` 文件
- 主机应用
- Linux 内核镜像
- 设备树
- U-Boot 配置文件：`boot.scr`
- 平台初始化脚本 `init.sh` 和平台名称（包含在 `platform_desc.txt` 内）。

  注释： `init.sh` 会设置环境变量 `XILINX_XRT` 并将 `platform_desc.txt` 文件复制到 `/etc/xocl.txt`。此操作必须手动执行。
- `ext4 rootfs`（包含在 Ext4 分区内）

要在 Vitis IDE 中使用 Ext4 rootfs 来封装镜像，请执行以下操作：

1. 依次选中“File” → “New” → “Application Project”，在 Vitis IDE 中创建新应用工程。
2. 选择平台（例如，`xilinx_zcu102_base_202010_1`），然后单击“Next”。

3. 为应用工程提供名称（例如，vadd）
4. 对于“System Project”，请选中“Create New”。
5. 对于“Target processor”，请选择可运行 Linux 域的处理器（例如，psu_cortexa53 SMP），然后单击“Next”。
6. 在“Domain”页面中，选中“xrt”并提供以下应用设置：
   - “Sysroot path”（例如，xilinx-zynqmp-common-v2020.1/sysroots/aarch64-xilinx-linux）
   - “Root FS”（例如，xilinx-zynqmp-common-v2020.1/rootfs.ext4）
   - “Kernel Image”（例如，xilinx-zynqmp-common-v2020.1/Image）
7. 单击“Next”。
8. 选择应用模板（例如，“Vector Addition”），然后单击“Finish”。
9. 选择系统工程，然后单击“Build”按钮 () 以构建工程。
10. 验证在 Emulation-SW、Emulation-HW 或 Hardware 目录下的 package 目录中是否已创建 sd_card.img 文件。

提示：创建应用工程后，要更改 sysroot、rootfs 或 kernel 的路径，请双击 .sprj 文件，然后在“Options”对话框中更改路径。

在 Vitis IDE 中使用 initramfs rootfs 封装镜像

向 Vitis IDE 提供 initramfs rootfs (rootfs.cpio) 时，生成的 sd_card.img 包含：

- 用于 PL 内核的 xclbin 文件
- 主机应用
- Linux 内核镜像
- 设备树
- boot.scr
- init.sh、platform_desc.txt 和 initramfs rootfs（包含在 FAT32 分区内）

注释：sd_card.img 文件不包含 ext4 分区。

1. 在 Vitis IDE 中，依次选中“File” → “New” → “Application Project”以创建新应用工程。
2. 选择平台（例如，xilinx_zcu102_base_202010_1），然后单击“Next”。
3. 提供应用工程名称（例如，vadd）。
4. 选择“Create New”。
5. 对于目标处理器，请选择可运行 Linux 域的处理器（例如，psu_cortexa53 SMP），然后单击“Next”。
6. 在“域 (Domain)”页面中，选中 xrt 域，并提供如下应用设置：
   a. Sysroot 路径（例如，your_linux_component_dir/sysroots/aarch64-xilinx-linux）
   b. 根文件系统（例如，your_linux_component_dir/rootfs.cpio.gz.u-boot）
   c. 内核镜像（例如，your_linux_component_dir/Image）
7. 单击“Next”。
8. 选择应用模板（例如，“Vector Addition”）。

9. 选择系统工程，然后单击“Build”按钮 () 以构建工程。
10. 验证在 `Emulation-SW`、`Emulation-HW` 或 `Hardware` 目录下的 `package` 目录中是否已创建 `sd_card.img` 文件。

注释：常用 Linux 组件包不提供 initramfs rootfs。如需了解有关生成 initramfs rootfs 的更多信息，请参阅《PetaLinux 工具文档：参考指南》(UG1144)。

提示：创建应用工程后，要更改 `sysroot`、`rootfs` 或 `kernel` 的路径，请双击 `.sprj` 文件，然后在“Options”对话框中更改路径。

将镜像写入 SD 卡

您可以使用 Vitis 统一软件平台加速流程将嵌入式平台定为目标。这样有助于以 RootFS 作为 EXT4 分区来封装并创建 SD 镜像，因为 `initramfs` 使用双倍数据速率 SDRAM (DDR SDRAM) 来作为文件系统存储器。当文件系统大小增大时，它可为 Linux 内核和应用限制实际可用的 DDR 存储器。重新启动后，它无法保留 RootFS 更改。

要将 EXT4 RootFS 写入 SD 卡：

1. 请准备 SD 卡二进制镜像文件，其中包含 FAT32 分区用于启动，并包含 EXT4 分区用于 RootFS。
2. 将 SD 卡镜像写入 SD 卡。您可使用各种工具来执行此操作，例如，Windows 上的 Etcher 或 Linux 上的 `dd` 命令。

   注释：请参阅赛灵思答复记录 73711，以获取有关这些工具的详细信息。

有各种方式可用于准备 SD 卡镜像。您可以使用 v++ 封装工具来生成镜像，或者也可以使用开源工具。在基本平台包中还提供了预构建的 SD 卡镜像，您可从 Vitis 嵌入式平台下载页面下载此镜像。

预构建的 SD 卡镜像 (`pre-built/sd_card.img`) 具有 2 个分区：

- FAT32 分区：大小为 1 GB，以 Linux 公共组件提供的内核镜像来进行初始化。
- EXT4 分区：大小为 2 GB，以 Linux 公共组件提供的 RootFS 来进行初始化。

要启动预构建的 SD 卡镜像，必须将以下启动组件复制到 FAT32 分区：

- `pre-built/BOOT.BIN`
- `xrt/image` 目录中的 `boot.scr`、`system.dtb`、`init.sh` 和 `platform_desc.txt`

预构建的 SD 卡镜像可用于评估，也可供 Windows 用户使用。它无需安装 Vitis 或 PetaLinux。

注释：在 Windows 上不支持含 Ext4 分区的 `v++ --package`。

注释：`init.sh` 会设置环境变量 `XILINX_XRT` 并将 `platform_desc.txt` 文件复制到 `/etc/xocl.txt`。您必须在启动 Linux 后先手动运行此程序，然后才能运行任何加速应用。

在 DFX 平台和非 DFX 平台中配置 PL 内核

赛灵思 Dynamic Function eXchange (DFX) 功能特性支持在更改 PL 功能的某些块的同时，保持 PL 的其它区域正常运行，这样您即可实时动态配置 PL 内核。要使用 DFX 功能特性，请在生成 `xclbin` 文件时，通过您的主机应用来配置此文件。xclbin 中的新内核将立即生效，无需重新启动。

对于无 DFX 功能特性的平台，PL 内核必须封装到 `boot.bin` 中。请将其复制到 SD 卡上的 FAT32 分区，然后重新启动系统。随后，使用您的主机应用来配置 `xclbin` 文件。

这样 `xclbin` 文件就会包含 PL 内核的位元文件以及用于描述这些内核功能特性和连接的元数据。在 DFX 平台上对 `xclbin` 文件进行编程会加载此位元文件和元数据，在非 DFX 平台上执行编程则仅加载元数据。

在开发板上运行加速应用

如果您使用由赛灵思提供的通用 Linux 组件，请执行以下操作以在平台上运行加速应用：

1. 通过 Vitis 编译器命令 `v++ --package` 生成 `sd_card.img` 并将其写入 SD 卡。
2. 启动开发板。
3. 运行 `cd /mnt/sd-mmcblk0p1/` 命令。
4. 运行 `source init.sh` 命令。
5. 运行加速应用。例如，要添加矢量，请运行 `./vadd ./binary_container_1.xclbin`。

加速应用使用赛灵思的 Xilinx Runtime (XRT) 与加速内核进行通信。要为 XRT 设置环境，请运行 `init.sh`。此命令用于：

- 将 `XILINX_XRT` 环境变量设置为 `/usr` 以允许应用查找 XRT 环境。
- 将 `platform_desc.txt` 复制到 `/etc/xocl.txt` 以将其运行平台告知 XRT。

注释：在 2019.2 版的 Vitis 中，对于嵌入式平台，此操作原先是自动完成的。由于自动运行 `init.sh` 可能引发安全性违规，因此默认情况下，通用 Linux rootfs 不运行 `init.sh`。

注释：如果 `sd_card.img` 文件已写入 SD 卡，并且您只需更新应用，那么您可以将所有文件从 `<Vitis System Project>/Hardware/package/sd_card` 复制到 SD 卡上的 FAT32 分区以替换现有文件，这样可以节省调试阶段的时间。Ext4 分区在 `sd_card.img` 中不发生更改。

PetaLinux rootfs 中的软件包管理

软件包管理功能是 Vitis 2020.1 版中的新增功能。所有 PetaLinux rootfs 软件包都在 http://petalinux.xilinx.com/sswreleases/rel-v2020/feeds 上进行托管。您可在目标板上运行 Linux 时将这些软件包安装到 rootfs 中，前提是开发板有权访问互联网。

要使用此功能，您必须启用 rootfs 中的软件包管理器 DNF。赛灵思提供的预构建 Linux 组件中的 rootfs 默认提供 DNF 软件包管理功能。

要设置软件包订阅源 URL：

1. 访问软件包订阅源存储库目录 http://petalinux.xilinx.com/sswreleases/rel-v2020/generic/rpm/repos/。
2. 下载存储库文件，此文件用于将您的 SoC 器件匹配到目标板。

```
# Example: ZCU102 uses ZU9EG devices
wget http://petalinux.xilinx.com/sswreleases/rel-v2020/generic/rpm/repos/
zynqmp-generic_eg.repo
# Example: ZCU104 uses ZU7EV devices
wget http://petalinux.xilinx.com/sswreleases/rel-v2020/generic/rpm/repos/
zynqmp-generic_ev.repo
```

3. 将下载的存储库文件复制到 `/etc/yum.repos.d/`。
4. 清除高速缓存。

```
dnf clean all
```

要管理软件包，请使用 DNF：

- 列出可用软件包：使用 `dnf` repoquery 命令。
- 从赛灵思存储库安装软件包：使用 `dnf install <pkg name>` 命令。

以下列出了您可通过 DNF 软件包管理器运行的一些基本功能。

- 列出可用软件包：使用 `dnf repoquery` 命令。
- 从赛灵思存储库安装软件包：使用 `dnf install <pkg name>` 命令。
- 从本地软件包文件安装软件包：使用 `dnf install <pkg file name>` 命令。
- 将软件包安装到 sysroot：

  在运行中的开发板的 rootfs 上安装软件包后，目标板上即包含最新二进制文件和库。如需在主机上进行交叉编译，则必须将这些库添加到主机侧 sysroot。

  在 XRT 中提供了 `sysroot_overlay` 脚本，用于提取 RPM 并更新 sysroot。此脚本将提取 RPM 库并在 sysroot 中包含文件更新。

  除 XRT 外，此脚本还支持将所有 RPM 用于各种软件包。

- 获取 sysroot_overlay.sh：使用 `wget https://github.com/Xilinx/XRT/blob/master/src/runtime_src/tools/scripts/sysroots_overlay.sh` 命令。

`sysroot` 命令描述为：

```
./sysroots_overlay.sh --sysroot --rpms-file
```

其中：

- `--sysroot` 是要覆盖的 sysroot。
- `--rpms-file` 是 RPM 文件，其中包含要覆盖的 RPM 文件路径。

示例

以下示例演示了用于将更新后的 XRT 安装到公用 sysroot 的命令：

```
./sysroots_overlay.sh -s sysroots/aarch64-xilinx-linux/ -r $PWD/rpm.txt
```

此示例显示了 `rpm.txt` 文件的内容：

```
./xrt-dev-202010.2.6.0-r0.aarch64.rpm
./xrt-202010.2.6.0-r0.aarch64.rpm
```

注释：此脚本仅适用于本地 RPM。您必须将 RPM 下载到自己的主机上才能将其安装到公用 sysroot。

第 50 章

在 Vitis 中创建嵌入式平台

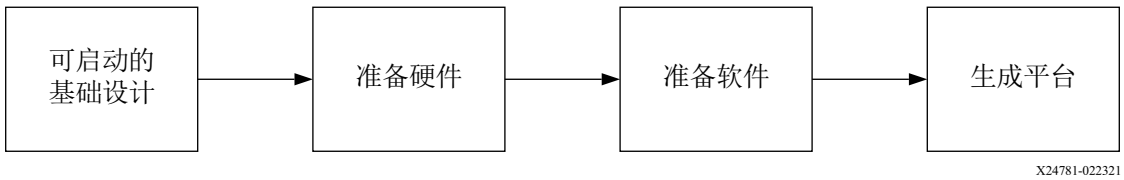
平台创建基础

在 Vitis™ 环境的加速应用开发流程中，工程分为 2 个不同要素：平台和处理子系统。平台包含基本 IP 块（例如，用于 SoC 的 PS、NoC 和用于 Versal™ ACAP 的 AI 引擎）以及开发板接口 IP 块（例如，高速 I/O 和存储器控制器）。处理子系统包含特定于应用的系统部分，这部分由可编程逻辑与 AI 引擎块组成。此方法有助于分离关注点、促进并发开发并鼓励复用。应用开发者能够免于应付平台的低层次细节，转而专注于解决处理子系统的细节问题。平台开发者则能够集中应对系统初始化和 I/O 性能调整，而无需担心处理子系统的问题。这意味着应用开发者能够在不同平台上集成子系统，并且可在不同处理子系统间复用平台。

赛灵思为 Alveo™ 卡和嵌入式评估板提供了预构建平台。您可从赛灵思下载中心下载这些平台。此方法论的核心是有效利用平台与子系统的解绑，这同时也是 Vitis 环境所提供的生产力增益的核心。对于嵌入式设计，赛灵思建议采用并行开发进程，即应用团队使用赛灵思预构建平台开始处理子系统，而平台团队则独立处理定制平台的初始化。以此方式开展工作即可快速取得进展。使用预构建平台意味着可使用经过预验证且已知有效的基本平台来对子系统进行独立开发、集成和测试。当子系统达到足够先进且稳定的状态后，该子系统即可与自定义平台的相应版本进行集成。总而言之，此方法能够显著精简系统集成进程。

下图显示了如何创建自定义的嵌入式平台。

图 130：**平台创建**



要创建平台，您必须具备可启动的基础设计作为起点。此设计可以是赛灵思基本平台设计、现有正常运作的设计或者也可以是从头开始创建的设计。在可启动的基础设计中，必须包含下列基础组件：

- 从 Vivado® Design Suite 导出的基础硬件设计
- 基础软件设计，其中包含 Linux 内核、根文件系统和设备树

通过 Vivado® Design Suite 设计获得正常运行的硬件和开发板后，将其转换为 Vitis 环境平台需要为基础组件添加属性以满足 Vitis 环境要求。总体上，平台创建由以下步骤组成：

1. 在 Vivado® Design Suite 工程中添加硬件接口参数和中断支持，并导出 XSA。
2. 更新软件平台组件以启用应用加速软件堆栈（启用 XRT、更新设备树等）。
3. 使用 XSCT 命令或 Vitis IDE 封装并生成平台。

Vitis 环境使用硬件工程中的属性来识别平台中的资源，并将内核链接至平台。Vitis 环境使用软件堆栈来控制内核。

如需了解有关 Vitis 环境内的嵌入式平台创建的详细信息，请参阅《Vitis 统一软件平台文档》(UG1416)。要获取分步指示信息，请参阅 Vitis 平台创建教程。

平台创建要求

您在 Vitis 平台中创建的基本设计在平台创建进程完成后保持静态。

Vitis 会基于某些 IP（例如，SmartConnect 或 NoC）通过添加额外的主接口/从接口来修改参数。在某些情况下，在平台中还可以修改 PS/CIPS 接口并例化 Versal™ ACAP 和 AI 引擎 IP。

下表显示了用于确认开发板上的基本系统的工作流程。

表 58：**平台工作流程**

工作流程	开发	确认
基础开发板启动	处理器基础参数设置。	独立 Hello world 和存储器测试应用正常运行。
高级硬件设置	在处理器系统中启用高级 I/O（例如，USB、以太网、闪存、PCIe® 或 RC）。 在 PL 中添加 I/O 相关 IP（例如，MIPI、EMAC 或 HDMI_TX）。 添加非 Vitis IP（例如，AXI BRAM Controller 或 Video Processing Subsystem (VPSS) IP）。	如果这些 IP 具有独立驱动程序，则对其进行测试。
基本软件设置	基于硬件平台创建 PetaLinux 工程。 启用内核驱动程序。 配置启动模式。 配置 rootfs。	Linux 成功启动。 在 Linux 中，外设正常工作。

基本组件要求

每个硬件平台设计都必须包含来自 IP 目录的 Processing System IP 块。

- 支持 Versal ACAP、Zynq® UltraScale+™ MPSoC 和 Zynq-7000 SoC 器件。
- 不支持使用 MicroBlaze™ 处理器来控制加速内核，但它可作为基本硬件的一部分。

创建嵌入式平台

添加硬件接口

下表显示了加速嵌入式平台可能的 Vitis 输入以及最低要求。

表 59：**Vitis 的可用接口**

输入	Vitis 可使用的类型	AXI MM 内核的最低要求
控制接口	AXI 主接口，源自 PS 或者源自 AXI Interconnect IP 或 SmartConnect IP	1 个 AXI4-Lite 主接口，用于内核控制

表 59：**Vitis 的可用接口** (续)

输入	Vitis 可使用的类型	AXI MM 内核的最低要求
存储器接口	AXI 从接口	1 个存储器接口，用于数据交换
数据流传输接口	AXI4-Stream 接口	非必需。
时钟	多个时钟信号	1 个时钟
中断	多个中断信号	1 个中断

一般要求

重要提示！ Vivado 工程的所有元素的源文件必须位于工程本地，随后才能将其作为 XSA 进行导出，否则在 Vitis 工具中使用平台时可能会返回错误。

- 平台设计中所用不属于标准 Vivado IP 目录的每个 IP 都必须位于 Vivado Design Suite 工程本地。创建可扩展 XSA 时，不支持引用工程外部的 IP 存储库路径。
- 供 Vitis 编译器用于链接至内核的任意平台接口都必须为以下任一接口类型：AXI4、AXI4-Lite、AXI4-Stream、中断、时钟或复位类型。
- 具有 AXI 接口（供 Vitis 编译器用于链接至内核）的任何平台 IP 都必须同时包含关联的时钟管脚，才能使 `v++` 根据需要正确推断和插入时钟域交汇逻辑。
- 不支持通过 `v++` 连接器 `--connectivity.sp` 和 `--connectivity.sc` 指令来处理平台或内核上的定制总线类型和硬件接口。如果 Vitis 编译器需要将含定制总线类型的数据总线连接到内核，则必须将其转换为 AXI4、AXI4-Lite 或 AXI4-Stream 接口。

工程类型

Vivado 工程类型需设置为“extensible Vitis platform”类型。

创建新工程时，请选择“Project is an extensible Vitis platform”。

要将现有 Vivado 工程更改为可扩展的 Vitis 平台工程，请依次选中“Project Manager” → “Settings in the Flow Navigator”，然后启用“Project is an extensible Vitis platform”。

```
set_property platform.extensible true [current_project]
```

添加平台接口

如果块设计中的组件具有 PFM 属性，那么该组件可通过 `v++` 连接器来识别，并且可供加速内核使用。

在 Vivado IDE 中，PFM 接口属性可在“Platform Setup”窗口中进行设置，前提是该工程创建为可扩展的平台工程。单击“Window menu” → “Platform Setup”即可打开设置。这些设置可在 Tcl 控制台中手动定义，或者也可通过 Tcl 脚本来定义。

4 大平台接口 Tcl API：

- AXI 存储器映射接口：

```
set_property PFM.AXI_PORT { <port_name> {parameters} <port2>
{parameters} ...} [get_bd_cells <cell_name>]
```

- AXI4-Stream 接口：

```
set_property PFM.AXIS_PORT { <port_name> {parameters} <port2>
{parameters} ...} [get_bd_cells <cell_name>]
```

- 时钟与复位：

```
set_property PFM.CLOCK { <port_name> {parameters} <port2>
{parameters} ...} [get_bd_cells <cell_name>]
```

- 中断：

```
set_property PFM.IRQ {pin_name {id id_number range irq_count}}
[get_bd_cells <cell_name>]
```

PFM 属性的要求包括：

- PFM 接口属性的值必须指定为 Tcl 词典，即：名称/"值" 对组成的列表。

> **重要提示！** "值" 必须加引号，并且名称和值均为区分大小写。

- `bd_cell` 可具有多个 PFM 接口定义。但对于每种类型的 PFM 接口，必须通过一条 `set_property` Tcl 命令来设置所有端口。
- 对于每个 PFM 接口属性，针对端口对象指定的名称必须与 `bd_cell` 上的外部端口或接口名称相匹配。每个外部端口或接口对象都只能有一个 PFM 接口定义。
- 每种不同类型的 PFM 接口都可具有不同的参数。
- 以 NULL ("") 字符串来设置 PFM 属性将导致删除先前定义的 PFM 接口。

添加 AXI 接口

为支持 AXI 存储器映射内核，平台需随 AXI 存储器映射主端口 (M_AXI_GP) 声明至少 1 个 AXI 控制接口，并随 AXI 从端口声明至少 1 个存储器接口。这些接口可从 PS 块直接导出或者连接 Interconnect IP。如果此平台不使用 AXI 存储器映射内核，则不需要这些接口。

以下是 Tcl 命令语法：

```
set_property PFM.AXI_PORT { <port_name> {parameters} <port2>
{parameters} ...} [get_bd_cells <cell_name>]
```

AXI 控制接口与 AXI 存储器接口共享同一个 `PFM.AXI` 属性。但其 `memport` 类型不同。

- AXI 控制接口可定义为 M_AXI_GP。存储器接口则使用其它类型：S_AXI_HP、S_AXI_ACP、S_AXI_HPC 或 MIG。
- 针对 M_AXI_GP 端口不支持 sptag 属性。
- sptag ID：（可选）用户定义的 ID，应以字母字符开头。ID 区分大小写。系统端口标签 (sptag) 属于符号标识，表示平台端口连接的类，例如，S_AXI_HP、S_AXI_ACP 或 M_AXI_GP。多个块设计平台端口可共享同一个 sptag。
- memory：（可选）指定关联的 MIG IP 实例和 `address_segment`。memory 标签属于唯一标识，用于将 IP integrator 的“地址编辑器 (Address Editor)”中的“单元名称 (Cell Name)”列和“基本名称 (Base Name)”列组合在一起。此标签将与 Memory Subsystem HIP 连接相关联，在其中，多个块设计平台端口可共享相同的 memory 标签。

导出 AXI Interconnect 主端口和从端口包含下列要求：

- 平台内使用的互连上的所有端口按索引顺序都必须位于任意已声明的平台接口之前。
- 端口索引之间可不含间隙。
- S_AXI_ACP 端口的主 ID 最大数量为 8，因此在已连接的 AXI Interconnect 上，要声明的可用端口必须为 {S00_AXI, S01_AXI, ..., S07_AXI} 之一。请勿声明平台本身内部使用的任何端口。尽可能声明更多端口将能够使 `sds++` 避免级联 `axi_interconnects`。

- S_AXI_HP 或 MIG 端口的主 ID 最大数量为 16，因此在已连接的 AXI Interconnect 上，要声明的可用端口必须为 {S00_AXI, S01_AXI, ..., S15_AXI} 之一。请勿声明平台本身内部使用的任何端口。尽可能声明更多端口将能够使 `v++` 避免在生成的用户系统中出现级联 `axi_interconnects`。
- 在已连接到 M_AXI_GP 端口的互连上声明的主端口的最大数量为 64，因此在已连接的 AXI Interconnect 上，要声明的可用端口必须为 {M00_AXI, M01_AXI, ..., M63_AXI} 之一。请勿声明平台本身内部使用的任何端口。尽可能声明更多端口将能够使 `v++` 避免在生成的用户系统中出现级联 `axi_interconnects`。

以下显示了在 AXI Interconnect IP 上定义 AXI 主端口的示例：

```
set parVal []
for {set i 2} {$i < 64} {incr i} {
lappend parVal M[format %02d $i]_AXI \
{memport "M_AXI_GP"}
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /axi_interconnect_0]
```

以下显示了在 SmartConnect IP 上定义含 MIG 的 AXI 存储器端口的示例：

```
set parVal []
for {set i 1} {$i < 16} {incr i} {
lappend parVal S[format %02d $i]_AXI
{memport "MIG" sptag "Bank0"}
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /smartconnect_0]
```

以下提供了控制接口和存储器接口的 `PFM.AXI_PORT` 设置示例。

```
set_property PFM.AXI_PORT {
M_AXI_HPM1_FPD {memport "M_AXI_GP"}
S_AXI_HPC0_FPD {memport "S_AXI_HPC" sptag "HPC0" memory "zynq_ultra_ps_e_0
HPC0_DDR_LOW"}
S_AXI_HPC1_FPD {memport "S_AXI_HPC" sptag "HPC1" memory "zynq_ultra_ps_e_0
HPC1_DDR_LOW"}
S_AXI_HP0_FPD {memport "S_AXI_HP" sptag "HP0" memory "zynq_ultra_ps_e_0
HP0_DDR_LOW"}
S_AXI_HP1_FPD {memport "S_AXI_HP" sptag "HP1" memory "zynq_ultra_ps_e_0
HP1_DDR_LOW"}
S_AXI_HP2_FPD {memport "S_AXI_HP" sptag "HP2" memory "zynq_ultra_ps_e_0
HP2_DDR_LOW"}
} [get_bd_cells /ps_e]
```

注释：

- `zynq_ultra_ps_e_0` 是 Zynq UltraScale+ MPSoC 模块的实例名称。
- `HPC0_DDR_LOW` 是地址范围名称。

添加 AXI4-Stream 接口

为支持 AXI4-Stream 流传输内核，平台需声明对应的 AXI4-Stream 主接口或从接口。

AXI4-Stream 内核接口是使用 PFM.AXIS_PORT sptag 接口属性和与 `v++` 连接器匹配的 `connectivity.sc` 命令实参指定的。

以下是 Tcl 命令语法：

```
set_property PFM.AXIS_PORT { <port_name> {parameters} <port2>
{parameters} ...} [get_bd_cells <cell_name>]
```

实参描述

- Port_name：AXI4-Stream 端口名称。
- 参数：类型值：流传输接口端口类型。有效的类型值包括：
  - M_AXIS：通用 AXI 主端口
  - S_AXIS：高性能 AXI 从端口

示例

```
set_property PFM.AXIS_PORT {AXIS_P0 {type "S_AXIS"}} [get_bd_cells /
zynq_ultra_ps_e_0]
```

注释：请参阅 `v++` 连接器配置文件语法，以了解如何在内核和平台之间链接 AXI4-Stream 接口 (--connectivity.sc).

添加时钟和复位

您可随平台导出任意时钟源，但对于每个时钟，您还必须使用平台中的 Processor System Reset IP 块来导出已同步的复位信号。PFM.CLOCK 属性可在 BD 单元、外部端口或外部接口上进行设置。

以下是用于设置 PFM.CLOCK 属性的 Tcl 命令：

```
set_property PFM.CLOCK { <port_name> {parameters} \
<port2> {parameters} ...} [get_bd_cells <cell_name>]
```

添加中断

Vitis 提供了一种方法，可在 `v++` 链接阶段将内核输出 IRQ 信号自动连接到平台中的 IRQ。以下显示了此 Tcl 命令的语法：

```
set_property PFM.IRQ {pin_name {id id_number}} bd_cell
set_property PFM.IRQ {port_name {id id_number range irq_count}}
[get_bd_cell <cell_name>]
```

实参描述

- Port_name：`bd_cell` 的 IRQ 端口名称。
- id_number：范围介于 0 到 127 之间的整数，用于指定 IRQ 编号和起始编号（如果已指定范围）。
- irq_count：用于标记接口，否则这些接口可能用于参数传输以指定总线大小（例如，中断控制器 intr 接口）。

此示例显示了如何对 `axi_intc_0` intr 端口启用 32 个 IRQ 输入。

```
set_property PFM.IRQ {intr {id 0 range 32}} [get_bd_cells /axi_intc_0]
```

此示例显示了如何在 VCK190 基本平台中启用 63 个含级联中断控制器的 IRQ。

```
set_property PFM.IRQ {intr {id 0 range 32}}  [get_bd_cells /
axi_intc_cascaded_1]
set_property PFM.IRQ {In0 {id 32} In1 {id 33} In2 {id 34} In3 {id 35} In4
{id 36} In5 {id 37} In6 {id 38} In7 {id 39} In8 {id 40} \
                                In9 {id 41} In10 {id 42} In11 {id 43} In12
{id 44} In13 {id 45} In14 {id 46} In15 {id 47} In16 {id 48} In17 {id 49}
In18 {id 50} \
```

```
                                In19 {id 51} In20 {id 52} In21 {id 53} In22
{id 54} In23 {id 55} In24 {id 56} In25 {id 57} In26 {id 58} In27 {id 59}
In28 {id 60} \
                                In29 {id 61} In30 {id 62}} [get_bd_cells /
xlconcat_0]
```

导出可扩展平台

硬件平台以 XSA 文件格式进行封装。有 2 种类型的 XSA 格式：用于软件开发的固定 XSA 和用于加速工程的可扩展 XSA。要为加速流程创建 Vitis 嵌入式平台，必须使用可扩展 XSA。

当 Vivado 工程类型设置为“extensible Vitis platform”时，即可在“File”→“Export or Window”→“Platform Setup”→“Export Platform”按钮下使用导出平台菜单。

在“Export Hardware Platform”窗口中，选择平台类型。平台分为 4 种类型：如果导出的平台将仅用于生成二进制文件以在硬件开发板上运行，请选择“硬件 (hardware)”。如果要通过此平台运行硬件仿真，请选择“硬件仿真 (hardware emulation)”或者选择“硬件和硬件仿真 (hardware and hardware emulation)”。这 2 个选项之间的差异在于：如果仿真不支持当前设计中的某些模块，那么您应创建仿真专用设计、导出为“hardware emulation”平台，然后使用“组合 XSA (Combine XSAs)”选项来将硬件 XSA 与硬件仿真 XSA 组合为能够执行这两项作业的单一 XSA。

对于简单设计：

1. 选择“Hardware and Hardware Emulation”，然后单击“Next”。
2. 选择“Pre-synthesis for Platform State”。仅当创建 DFX 平台时才需要执行实现后操作。单击“Next”。
3. 输入“Platform Properties”。单击“Next”。
4. 输入 XSA 文件名，然后导出目标目录。单击“Next”。
5. 检查汇总信息，然后单击“Finish”。

您还可在命令行中使用以下命令来执行此操作：

```
set_property pfm_name {vendor:board:name:version} [get_files <bd_file>]
write_hw_platform -hw -force <XSA file>
```

要创建硬件 XSA 与硬件仿真 XSA 并将两者组合，请使用以下命令：

```
write_hw_platform -hw <hw_platform>
write_hw_platform -hw_emu <hw_emu_platform>
combine_hw_platform -hw <hw_platform> -hw_emu <hw_emu_platform> -o
<combined_platform>
```

更新软件组件

向根文件系统添加 XRT

Vitis 加速应用使用 XRT 来控制硬件。XRT 为从 Alveo™ 数据中心加速器卡到嵌入式等各种用例提供了统一的编程接口。

您必须将 XRT 内核驱动程序 (zocl) 和用户空间库 (xrt-dev) 添加到 `rootfs` 和 `sysroot`。封装 xrt-dev 支持您对使用 XRT API 的 Vitis 应用进行编译。

为 ZOCL 更新设备树

zocl 驱动程序接口需要设备树节点才能启用中断连接。

以下提供了 zocl 器件节点的示例。

```
&amba {
    zyxclmm_drm {
        compatible = "xlnx,zocl";
        status = "okay";
        interrupt-parent = <&axi_intc_0>;
        interrupts = <0  4>, <1  4>, <2  4>, <3  4>,
                     <4  4>, <5  4>, <6  4>, <7  4>,
                     <8  4>, <9  4>, <10 4>, <11 4>,
                     <12 4>, <13 4>, <14 4>, <15 4>,
                     <16 4>, <17 4>, <18 4>, <19 4>,
                     <20 4>, <21 4>, <22 4>, <23 4>,
                     <24 4>, <25 4>, <26 4>, <27 4>,
                     <28 4>, <29 4>, <30 4>, <31 4>;
    };
};
```

欲知详情，请参阅 XRT 文档：https://xilinx.github.io/XRT/master/html/yocto.html。

更新中断控制器输入值

在模块框图中，中断控制器未连接到加速内核。自动生成的设备树反映了模块框图的硬件设计，且未考量 `v++` 连接器将把内核中断信号连接到中断控制器的情况。为了启用这些中断，请覆盖中断控制器的中断输入值。

以下示例演示了如何在 `system-user.dtsi` 中覆盖 AXI Interconnect 节点参数。

```
&axi_intc_0 {
    xlnx,kind-of-intr = <0x0>;
    xlnx,num-intr-inputs = <0x20>;
    interrupt-parent = <&gic>;
    interrupts = <0 89 4>;
};
```

通过 /etc/xocl.txt 来声明平台

平台名称可写入嵌入式平台 rootfs 的 `/etc/xocl.txt` 中，以便 XRT 确定其平台。主机应用可使用 XRT API 来获取平台名称并检查与 `xclbin` 的比较以及主机应用与平台的比较。

调整 CMA 大小

XRT 使用 CMA 来进行缓冲器对象分配。您必须为 `bootargs` 中的 CMA 或设备树保留足够的存储器，以免加速应用运行时期间出现存储器不足。

封装 Vitis 加速平台

满足 Vitis 加速平台的所有要求后，您即可将其封装在一起，并生成最终 Vitis 加速平台。您可使用 Vitis IDE 或赛灵思软件命令行工具 (XSCT) 来执行此操作。

- 在 Vitis IDE 中，依次选中“File” → “New” → “Platform Project”以创建 Vitis 平台。
- 您可通过 XSCT 来使用 `platform` 命令创建平台，并使用 `domain` 命令将域添加到平台中。如需了解有关 XSCT 的更多信息，请参阅 《Vitis 统一软件平台文档》(UG1416) 的“嵌入式软件开发流程”中的赛灵思软件命令行工具

平台是通过对多个硬件和软件组件加以封装而构成的。此封装可便于面向硬件的工程师将其交付给应用开发者。

在平台中封装有以下文件和信息。

- 硬件规格：这是指可扩展的 XSA 文件。
- 软件组件：这些组件添加到平台后，作为支持 OpenCL 运行时的 Linux 域来使用。软件组件包括：
  - 启动组件
    - BIF 文件，用于描述启动组件及其属性，以供 Bootgen 生成 `boot.bin` 文件。
    - 启动组件目录，其中包含 BIF 文件中描述的所有文件。
  - 镜像文件（可选）：此目录中的内容将被复制到最终 SD 卡镜像的 FAT32 分区内。
  - Linux 域：平台需要 1 个 Linux 域。内核、RootFS 和 sysroot 信息可在创建平台时或创建应用时添加。
  - 仿真支持文件（可选）

根文件系统

Vitis 支持 FAT32 和 Ext4 分区类型。根文件系统在平台创建步骤中为可选，因为它可在 Vitis 应用创建步骤中指定。

在平台创建期间需设置镜像目录。此目录中的所有内容都将被封装到最终 SD 卡镜像内。如果目标文件系统为 FAT32，那么这些文件将置于 SD 卡根目录下；如果目标文件系统为 Ext4，那么这些文件将置于第一个 FAT32 分区的根目录下。

启动组件

必须提供 BIF 文件，以便应用构建进程能够封装启动镜像。

以下是 BIF 文件示例：

```
/* linux */
the_ROM_image:
{
  [fsbl_config] a53_x64
  [bootloader] <fsbl.elf>
  [pmufw_image] <pmufw.elf>
  [destination_device=pl] <bitstream>
  [destination_cpu=a53-0, exception_level=el-3, trustzone] <bl31.elf>
  [destination_cpu=a53-0, exception_level=el-2] <u-boot.elf>
}
```

应提供启动组件目录，其中包含 BIF 中描述的所有文件。在此示例中，组件目录提供了 `fsbl.elf`、`pmufw.elf`、`bl31.elf` 和 `u-boot.elf`。这些启动组件可由 PetaLinux 生成。

在 Vitis 应用构建和封装状态下，`v++` 会在启动组件目录下查找这些文件，并将占位符替换为真实的文件名和路径。随后，它会调用 Bootgen 以生成 `BOOT.BIN`。

平台测试

向应用开发者交付平台之前，应运行一些基本平台测试来确保它能正常运行加速应用。

通常，需确保平台能够成功完成下列测试：

- 启动测试：Vivado 工程生成的实现结果 BIT 文件（源自 添加硬件接口）和 PetaLinux 生成的镜像（源自 更新软件组件）都应能够成功启动至 Linux 控制台。

- Platforminfo 测试：封装 Vitis 加速平台 中生成的平台应包含正确的 platforminfo 报告，用于提供时钟和存储器信息。
- XRT 基本测试：XRT `xbutil query` 实用工具应能在目标开发板上运行并正确报告平台信息。
- Vadd 测试：Vitis 可用于生成矢量加法样本应用以搭配平台使用。生成的应用和 `xclbin` 应在开发板上打印 `test pass`。

为可扩展 XSA 启用硬件仿真

以下步骤可供定制平台开发者使用。

1. 使用必要的 BD、RTL、测试激励文件和其它源文件创建 Vivado 工程。
   a. 请注意，在 2020.1 中，硬件仿真 (HW EMU) 中只能使用 BD，但从 2020.2 起，将允许使用其它源文件。
   b. （仅限于 Versal ACAP）测试激励文件需包含 BD 封装器而不是直接包含 BD，因为 Vitis 会在此级别执行作业以将 NoC 插入仿真。
   c. （仅限于 Versal ACAP）要在 Vitis 平台中启用 AI 引擎，AI 引擎块需配置为仅启用 1 个 AXI4 存储器映射从端口，并连接至 NoC。基于 AI 引擎 Graph 软件的 Vitis 将在 `v++` 链接阶段中执行其它自动连接。
   d. 对于 DFX 平台，请在动态区域 BD 内指定正确的 PFM 属性，以便 Vitis 工具能够正确连接加速器。
2. 将设计硬件仿真封装更新到 XSA 中。
   a. 在将设计封装到 XSA 中之前，重要的是设计正确完成仿真器的每个步骤。
   b. （仅限于 Versal ACAP）准备平台设计以支持 SystemC 模型。更新 CIPS 和 NoC IP 设置，将 SELECTED_SIM_MODEL 属性更改为 TLM。这样即可确保对于 CIPS IP，软件可在设计所使用的 QEMU 模型上运行。同样，对于 Zynq®-7000 和 Zynq® UltraScale+™ MPSoC 器件，请在处理器系统 IP 实例上设置 SELECTED_SIM_MODEL。以下 Tcl 命令可在设计中使用。并设置参数，以在 Vivado 中支持 SystemC 仿真：

```
foreach tlmCell [get_bd_cells * -hierarchical -filter {VLNV =~
"*:*:axi_noc:*" || VLNV =~ "*:*:versal_cips:*"}] {set_property
SELECTED_SIM_MODEL tlm $tlmCell }
set_param bd.generateHybridSystemC true
```

   c. 在 `sim_1 fileset` 文件集中创建测试激励文件，并例化设计的 `<top>` 模块。对于 Versal ACAP，Vivado 要求用户测试激励文件不应直接例化 `<top>` 模块。而应改为例化 `<top>_sim_wrapper` 模块。调用 `launch_simulation -scripts_only` 命令时，会生成名为 `<top>_sim_wrapper.v` 的文件。此模块的接口与 `<top>` 模块相同，但它会例化其它仿真模型，这些例化的仿真模型与从设计中例化的各逻辑 NoC 模块创建的聚合 NoC 模块相关。以下 Vivado Tcl 命令可用于生成必要的 NoC 仿真文件，并在仿真源中使用。

```
# Ensure that your top of synthesis module is also set as top for
simulation

set_property top <rtl_top> [get_filesets sim_1]

# Generate simulation top for your entire design which would include
# aggregated NOC in the form of xlnoc.bd

launch_simulation -scripts_only
update_compile_order -fileset sim_1

# Set the auto-generated <rtl_top>_sim_wrapper as the sim top

set_property top <rtl_top>_sim_wrapper [get_filesets sim_1]
update_compile_order -fileset sim_1
```

```
#Generate the final simulation script which will compile
# the <syn_top>_sim_wrapper and xlnoc.bd modules also
launch_simulation -scripts_only
launch_simulation -step compile
launch_simulation -step elaborate
```

d. 编译设计，逐步执行上述步骤，然后开始仿真。由于设计配置为使用 QEMU，CIPS IP 将不会生成任何传输事务，因为在 Vivado 仿真器中执行仿真时不存在任何软件。在 Vivado 仿真中，您将看到如下 ERROR 消息，但它表明基本设计在仿真器中已正确加载。

```
##################################################################
#
 #  Simulation does not work as Versal CIPS Emulation
(SELECTED_SIM_MODLE=tlm) only works with Vitis
tool(launch_emulator.py tool in Vitis)
#
##################################################################

ERROR: [Simtcl 6-50] Simulation engine failed to start: The
Simulation shut down unexpectedly during initialization.
```

注释：为确认设计是否将包含正确的传输事务，您可以选择首先使用 CIPS VIP 来执行设计仿真，然后再将其改为使用 TLM (QEMU)。首先，必须针对 NoC 和 CIPS IP 将 SELECTED_SIM_MODEL 属性保留设为 RTL。同时，创建另一个测试激励文件用于驱动 CIPS VIP 并同时满足 NoC Verilog 模型的要求。请参阅 CIPS VIP 和 NoC IP 文档，以获取有关如何设置测试激励文件用于基于 Verilog 的仿真的更多详细信息。

3. 对仅含硬件仿真的 XSA 进行封装。

重要提示！Vivado 工程的所有元素的源文件必须位于工程本地，随后才能将其作为 XSA 进行导出，否则在 Vitis 工具中使用平台时可能会返回错误。

a. 使用“Vivado File” → “Export” → “Export Platform”以导出硬件仿真平台，或者使用以下 Tcl 命令：

```
set_property platform.platform_state "pre_synth" [current_project]
write_hw_platform -hw_emu -file  platform_hw_emu.xsa
```

b. 此 XSA 可搭配预构建的 Linux 镜像一起使用或者搭配 PetaLinux 一起使用以创建定制 Linux 镜像，用于创建完整平台。随后，可使用其它 Vitis 工具来通过 XRT 将内核添加到设计中。

有关嵌入式平台创建的特殊注意事项

按平台和内核拆分逻辑函数

虽然 FPGA 和 SoC 上的设计日益复杂，但大部分开发者或团队都通过协作来进行设计。Vitis 软件平台为应用开发者和平台开发者提供了清晰的边界。平台开发者可包括开发板开发者、BSP 开发者、系统软件开发者等。

从系统架构师角度而言，部分逻辑函数可能处于灰色区域：这些逻辑函数可能与平台组合在一起，或者可能充当加速内核来使用。为了帮助拆分系统块，以下提供了一些通用准则。

- 将函数分类为内核或平台的基本考量标准是：它是否属于应用相关的逻辑。
- 平台应比应用更稳定。应用函数更改应仅发生在软件和内核中。
- 平台把硬件抽象化。更改硬件开发板时，应用应无需更改，或者只需极少量更改，以匹配新硬件的需求。
- 遵循 Vitis 工具的约束和限制进行操作。例如：
  - Vitis 加速内核仅支持 3 种类型的接口：AXI MM、AXI4-Lite 和 AXI4-Stream。
  - AXI 内核不支持外部 I/O 管脚。

下表显示了针对逻辑类型建议的平台和内核。

表 60：**建议的平台与内核**

逻辑	平台	内核
硬核处理器（Zynq 和 Zynq UltraScale+ MPSoC 的 PS）	仅限在平台内使用	
软核处理器	首选在平台内使用	可作为 RTL 内核来使用
I/O 块（外部管脚、MIPI、PHY 等）	仅限在平台内使用	
I/O 块的相关 IP（DMA for PCIe®、MAC for Ethernet 等）	通常在平台内使用，因为 I/O 与 IP 间的接口并非 AXI。	可作为内核使用，前提是 I/O 块与 IP 之间的接口为 AXI。
含非 AXI 接口的 IP	仅限在平台内使用	如果接口可更改为 AXI MM 或 AXI4-Stream，则可用
具有 Linux 驱动程序（VPSS 等）的传统存储器映射 IP	仅限在平台内使用	
HLS AXI Memory Mapped IP	可在平台内使用。您必须编写控制软件。	首选作为内核使用。受 XRT 控制。
加速存储器映射 IP 遵循 Vitis 内核寄存器标准，并可接受 XRT		首选作为内核使用
Vitis 库		仅限作为内核使用
含 AXI4-Stream 接口的自由运行 IP	OK	OK

参考资料

如需了解有关嵌入式平台的更多信息，请访问以下链接：

- Vitis 嵌入式加速平台创建教程
- Vitis 嵌入式平台源文件

验证嵌入式平台

您应遵循 平台创建要求 中所述步骤来验证创建的平台。此外，您也可以通过下列章节所描述的附加操作来检验并实践嵌入式平台上的软件、图形与内核开发工作，以确保平台满足您的需求和期望。

检查平台元数据

在平台创建进程中，您将定义平台中可用的资源。您可使用 `platforminfo` 命令对平台定义及可用资源执行快速检验。该实用工具可以打印平台元数据，以便您确认封装的平台中定义的资源能否满足您的期望。

使用平台来处理简单的测试案例

通过使用平台来处理简单的测试案例，即可确保系统按期望方式运行，而无需调试主机应用、图形或内核交互。简单测试无需通过复杂的设计即可测试平台的特定功能组合。赛灵思提供了多个示例和教程，可作为测试案例来使用，部分示例和教程位于：

- https://github.com/Xilinx/Vitis_Accel_Examples
- https://github.com/Xilinx/Vitis-Tutorials

简单的测试有助于检查时钟和复位、AXI4 接口设置以及中断是否正确并按期望方式工作。

在固定平台上运行裸机软件

要进行平台验证，您可以在可扩展平台的固定版本上运行裸机主机应用。这样您即可快速测试嵌入式平台，并将结果与通过裸机应用访问各硬件元件所需的驱动程序进行比对。这有助于平台开发者通过简单的应用来确认平台外设功能特性和平台中的定制 IP。为执行此确认，您必须创建可扩展平台的固定版本，用于裸机软件开发。此确认无需 Linux 软件组件。

要创建可扩展平台的固定形式，请使用以下步骤：

- 从可扩展硬件平台导出固定 XSA：如 平台类型 中所述，平台一般有两种类型：可扩展平台和固定平台。固定平台支持嵌入式软件开发，不允许您修改器件二进制文件定义的 PL 内核或目标平台。但 AI 引擎流程可允许您在固定平台上修改和重新编译 Graph 应用。

  在 Vitis 编译器 (`v++`) 链接进程中构建“应用工程 (Application Project)”时，即可生成硬件平台的固定版本 (`.xsa`)，如 构建器件二进制文件 中所述。

  在 Vitis IDE 中，通过启用“Hardware Link Project Settings”视图中的“Export hardware (XSA)”复选框即可生成固定 XSA。对于命令行流程，则必须在运行 `v++ --link` 命令期间，给使用的配置文件添加以下选项来启用该功能：

  ```
  [advanced]
  param=compiler.addOutputTypes="hw_export"
  ```

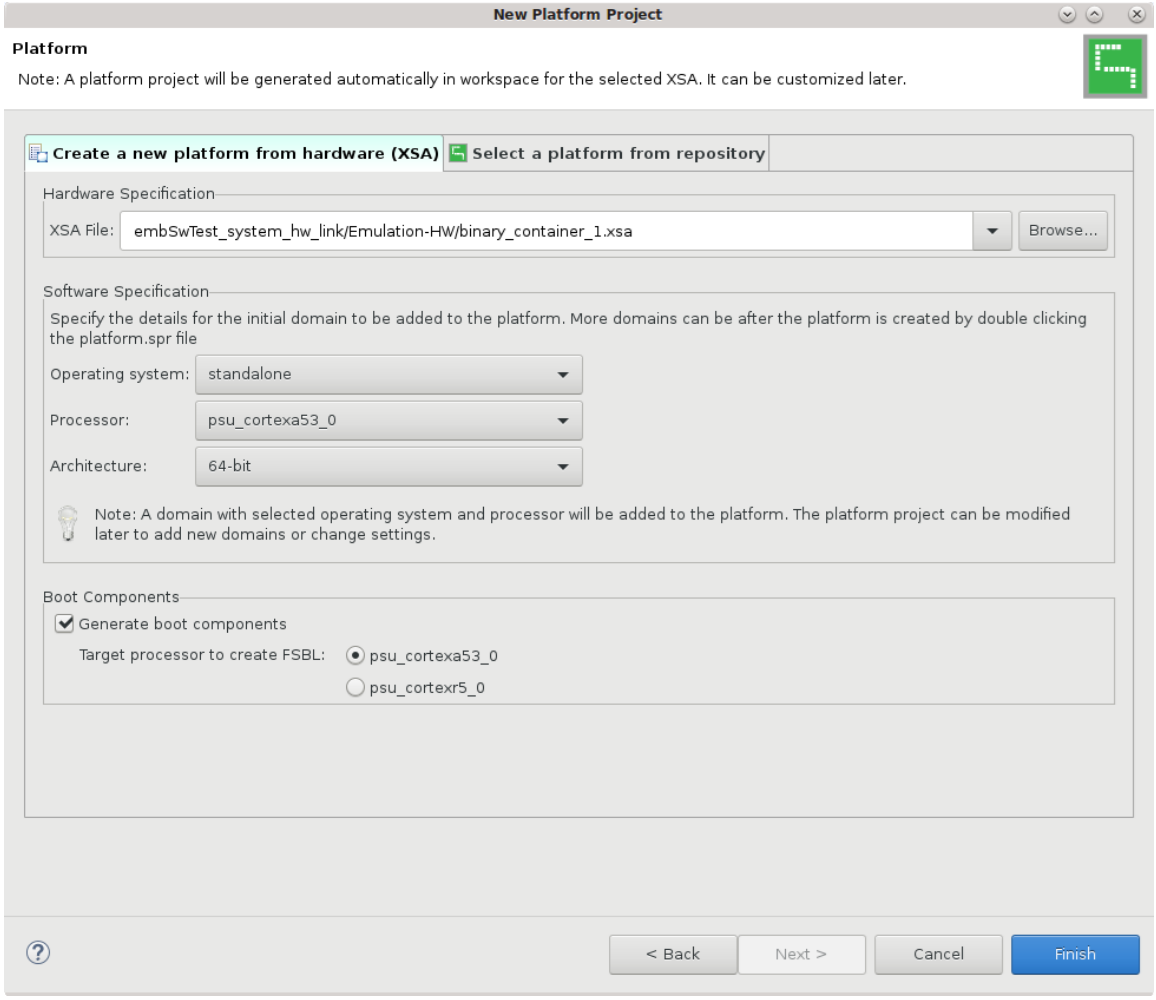
  Vitis 编译器会导出固定硬件规格，其中包含目标平台的各元素，并且所有加速内核与 AI 引擎 Graph 均已固定到硬件内。固定 XSA 可以为系统设计提供 BSP，供您用于开发嵌入式软件应用。

  重要提示！此固定 XSA 与 Vitis 编译器的指定构建目标相匹配。因此，从 `hw_emu` 构建目标会生成支持硬件仿真的固定 XSA，从 `hw` 构建目标则会生成支持硬件的固定 XSA。

- 生成裸机平台：

  1. 固定 XSA 可用于创建或生成嵌入式处理器平台的裸机固定版本，用于确认平台。打开 Vitis IDE 并依次选中“File” → “New” → “Platform Project”。这样会打开“New Platform Project”Wizard。
  2. 指定“Platform project name”，然后单击“Next”。这样会打开“Platform”页面，如下图所示。

3. 在此对话框中，您可通过如下方式来定义平台：为平台选择“XSA File”的名称、选择“Operating system”、选择“Processor”域，然后选择“Generate boot components”。



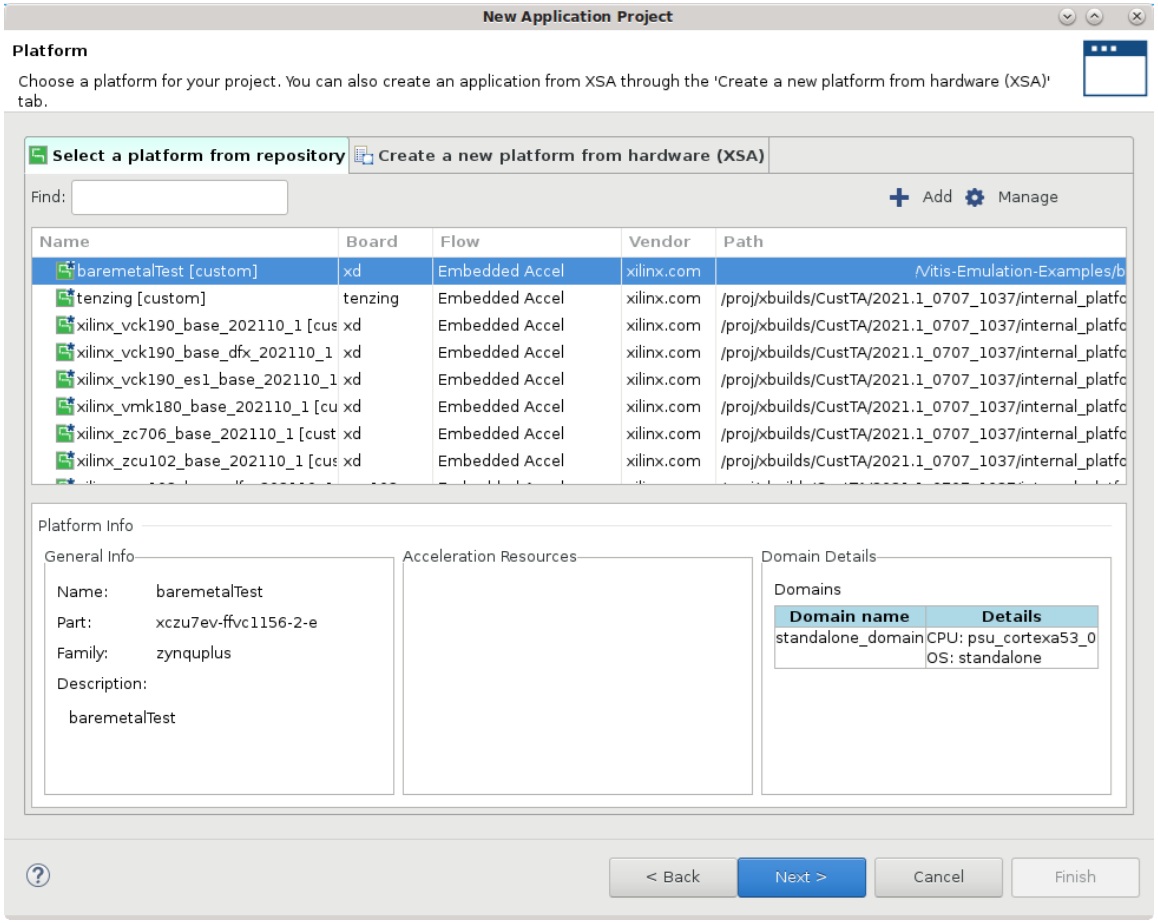
针对“Operating system”选中“standalone”，指定“Processor”和“Architecture”即可完成域定义。如上所示启用“Generate boot components”，然后单击“Finish”即可生成裸机嵌入式平台。

平台设置完成后，您应可看到此平台被添加到可用平台存储库中。您将使用它来创建裸机应用工程，如下文所述。

- 利用应用工程来确认裸机平台：构建完裸机平台后，您可使用 Arm 处理器上运行的简单应用来确认平台。此方法允许您利用已编译的 ELF 文件来快速实践平台的各项功能特性。

1. 在 Vitis IDE 中，依次选中“File”→“New”→“Application Project”。这样会打开“New Application Project”Wizard。

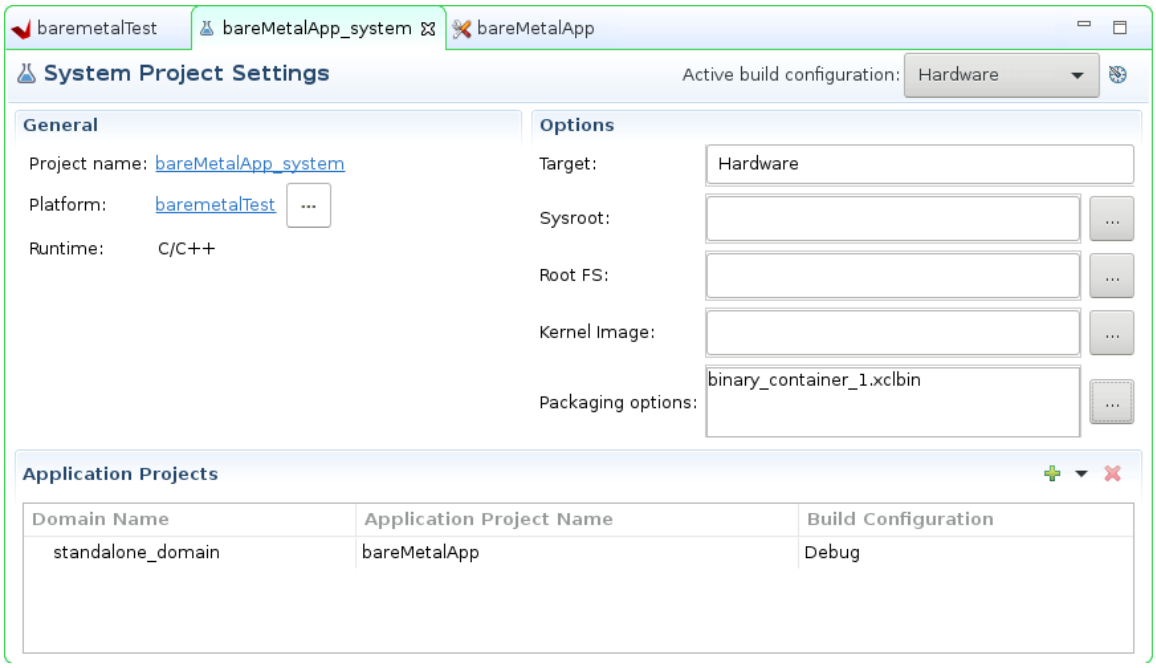
   在“Platform”页面上，选中您在上一步中创建的裸机平台，然后单击“Next”以继续。



2. 随后，“New Application Project”Wizard 就会显示“Application Project Details”页面。指定“Project name”和“System project name”，然后单击“Next”以继续。
3. 这样“Domain”页面会显示选定平台的域信息。单击“Next”以继续。
4. 这样“Templates”页面会显示选定平台的“Embedded software development templates”列表。选择“Hello World”模板或另一个简单应用，或者选择“Empty Application”以便您指定自己的源文件，然后单击“Finish”以创建应用工程。
5. 当裸机平台获得裸机应用工程的支持后，您即可开始执行确认。您可以使用此裸机平台，并且仍采用原先用于创建固定 XSA 的构建目标：
   - 如果固定 XSA 是从硬件仿真构建 (`v++ --link -t hw_emu`) 生成的，那么它可用于在裸机应用工程内执行“硬件仿真 (Emulation-HW)”构建。
   - 如果固定 XSA 是从硬件构建 (`v++ --link -t hw`) 生成的，那么它可用于在裸机应用工程内执行“硬件 (Hardware)”构建。

   在“Application System Project Settings”窗口中，指定与裸机平台的固定 XSA 兼容的“Active build configuration”，如下所示。

提示：您可能还需要在“Packaging options”字段中为裸机应用工程指定 `.xclbin` 文件，如下图所示。硬件构建需要指定此文件，您可将 `.xclbin` 文件从固定 XSA 的源工程复制到裸机应用工程内。



6. 对于 Emulation-HW 构建配置，您可以在 QEMU 环境内运行应用，并查看此应用与固定平台的交互，以确认系统。要运行仿真构建，请从“Assistant”视图或工具栏主菜单中选择“Launch HW Emulator”命令。通过 TCF 代理即可启动 QEMU，您可在“Emulation Console”视图中观察 QEMU 启动和 PS 应用运行的文字记录。

提示：您也可以选择“Debug As”命令以在 Vitis IDE 中启动交互式调试环境。

第九部分

附加信息

本部分包含以下章节：

XILINX

第 51 章

OpenCL 编程

OpenCL 主机应用

在 Vitis™ 核开发套件中，主机代码是使用赛灵思的 Xilinx Runtime (XRT) API 或业界标准 OpenCL™ API 以 C 或 C++ 语言编写的。如需了解有关 XRT 本机 API 的说明，请访问 XRT 网站，网址为：https://xilinx.github.io/XRT/2020.2/html/xrt_native_apis.html。Vitis 核开发套件支持 OpenCL 1.2 API，如 https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf 中所述。如需了解有关 OpenCL 的 XRT 扩展的说明，请访问 https://xilinx.github.io/XRT/2020.2/html/opencl_extension.html。

提示：此处文本中所示代码示例使用的是 OpenCL C 语言 API。

总之，主机代码结构可分为 3 个部分：

1. 设置环境。
2. 核命令执行包括执行一个或多个内核。
3. 资源的后处理和发布。

提示：Vitis 核开发套件支持 OpenCL 可安装客户端驱动 (ICD) 扩展 (`cl_khr_icd`)。此扩展支持在同一系统上共存多个 OpenCL 实现。如需了解详细信息以及安装指示信息，请参阅 OpenCL 可安装客户端驱动程序加载器。

注释：要对主机程序启用多线程，从 Vitis 核开发套件应用调用 `fork()` 系统调用时请谨慎处理。`fork()` 不会复制所有运行时线程。因此，在 Vitis 核开发套件中，子进程无法作为完整应用来运行。建议使用 `posix_spawn()` 系统调用从 Vitis 软件平台应用启动另一个进程。

设置 OpenCL 环境

Vitis 核开发套件中的主机代码遵循 OpenCL 编程范例。为正确设置运行时环境，主机应用需初始化标准 OpenCL 结构，包括：目标平台、器件、上下文环境、命令队列和程序。

提示：本文档中使用的主机代码示例和 API 命令遵循 OpenCL C API。但 XRT 也支持 OpenCL C++ 封装文件 API，许多 Vitis 示例都是使用 C++ API 编写的。如需了解有关此 C++ 封装文件 API 的更多信息，请参阅 https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf。

平台

初始化时，主机应用需要识别由一个或多个赛灵思器件组成的平台。以下代码片段显示了识别赛灵思平台的常用方法。

```
cl_platform_id platform_id;         // platform id

err = clGetPlatformIDs(16, platforms, &platform_count);

// Find Xilinx Platform
for (unsigned int iplat=0; iplat<platform_count; iplat++) {
  err = clGetPlatformInfo(platforms[iplat],
    CL_PLATFORM_VENDOR,
    1000,
    (void *)cl_platform_vendor,
    NULL);

  if (strcmp(cl_platform_vendor, "Xilinx") == 0) {
  // Xilinx Platform found
  platform_id = platforms[iplat];
  }
}
```

OpenCL API 调用 clGetPlatformIDs 用于发现给定系统的可用 OpenCL 平台组合。随后，clGetPlatformInfo 用于识别基于赛灵思器件的平台，方法是将 cl_platform_vendor 与字符串 "Xilinx" 相匹配。

建议：虽然在前述代码或者本章全文所使用的其它主机代码示例中并未明确显示，但最好在每次 OpenCL API 调用后使用纠错功能，这是规范的编码实践。这样有助于调试，并且在仿真流程中或者硬件执行期间，当您执行主机与内核代码调试时，这也有助于提升效率。以下代码片段演示的正是 clGetPlatformIDs 命令的纠错代码示例。

```
err = clGetPlatformIDs(16, platforms, &platform_count);
if (err != CL_SUCCESS) {
  printf("Error: Failed to find an OpenCL platform!\n");
  printf("Test failed\n");
  exit(1);
}
```

器件

找到赛灵思平台后，应用需识别对应的赛灵思器件。

以下代码演示了使用 API clGetDeviceIDs 查找所有赛灵思器件（上限为 16 个器件）的过程。

```
cl_device_id devices[16];  // compute device id
char cl_device_name[1001];

err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR,
  16, devices, &num_devices);

printf("INFO: Found %d devices\n", num_devices);

//iterate all devices to select the target device.
for (uint i=0; i<num_devices; i++) {
  err = clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 1024, cl_device_name,
0);
  printf("CL_DEVICE_NAME %s\n", cl_device_name);
}
```

重要提示！ clGetDeviceIDs API 通过 platform_id 和 CL_DEVICE_TYPE_ACCELERATOR 来调用，以接收所有可用的赛灵思器件。

子器件

在 Vitis 核开发套件中，有时器件包含单一内核或不同内核的多个内核实例。虽然 OpenCL API clCreateSubDevices 允许主机代码将器件分割为多个子器件，但 Vitis 核开发套件仅支持均等分割的子器件（使用 CL_DEVICE_PARTITION_EQUALLY），且每个子器件包含一个内核实例。

以下示例显示：

1. 按均等大小分区创建的子器件，每个子器件执行一个内核实例。
2. 基于子器件列表迭代，使用独立的上下文和命令队列在每个子器件上执行内核。
3. 简单起见，其中并未显示与内核执行（和对应的相关缓冲器）代码相关的 API，但在 run_cu 函数中将对此类 API 进行描述。

```
cl_uint num_devices = 0;
  cl_device_partition_property props[3] = {CL_DEVICE_PARTITION_EQUALLY,1,0};

  // Get the number of sub-devices
  clCreateSubDevices(device,props,0,nullptr,&num_devices);

  // Container to hold the sub-devices
  std::vector<cl_device_id> devices(num_devices);

  // Second call of clCreateSubDevices
  // We get sub-device handles in devices.data()
  clCreateSubDevices(device,props,num_devices,devices.data(),nullptr);

  // Iterating over sub-devices
  std::for_each(devices.begin(),devices.end(),[kernel](cl_device_id sdev) {

      // Context for sub-device
      auto context = clCreateContext(0,1,&sdev,nullptr,nullptr,&err);

      // Command-queue for sub-device
      auto queue = clCreateCommandQueue(context,sdev,
      CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,&err);

      // Execute the kernel on the sub-device using local context and
    queue run_cu(context,queue,kernel); // Function not shown
  });
```

重要提示！ 如以上示例所示，您必须为每个子器件创建单独的上下文。虽然 OpenCL 支持可包含多个器件和子器件的上下文，但 XRT 要求每个器件和子器件都具有独立的上下文。

内容

clCreateContext API 用于创建上下文环境，其中包含将与主机进行通信的赛灵思器件。

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

在代码示例中，clCreateContext API 用于创建包含 1 个赛灵思器件的上下文环境。赛灵思建议仅为每个器件或每个子器件创建一个上下文环境。但如果使用多个子器件并且每个子器件对应一个上下文环境，那么主机程序应使用多个上下文环境。

命令队列

clCreateCommandQueue API 会为每个器件创建一个或多个命令队列。FPGA 可包含多个内核，这些内核可能相同，也可能不同。开发主机应用时，有 2 种主要的编程方法可用于在器件上执行内核。

1. 单个无序命令队列：可通过同一命令队列请求多次内核执行。XRT 会尽快以任意顺序分派内核，以允许在 FPGA 上并发执行内核。
2. 多个有序命令队列：从不同的有序命令队列请求每次内核执行。在此类情况下，XRT 会分派来自不同命令队列的内核，通过在器件上并发运行这些内核来提升性能。

建议：为了提升性能，赛灵思建议使用单个无序命令队列，并显式管理事件依赖关系和同步，而不是使用多个命令队列。

以下是创建有序和无序命令队列的标准 API 调用示例。

```
// Out-of-order Command queue
commands = clCreateCommandQueue(context, device_id,
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);

// In-order Command Queue
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

程序

主机与内核代码分别进行编译，以创建独立的可执行文件：即主机程序可执行文件和 FPGA 二进制文件 (.xclbin)。当运行主机应用时，它必须使用 clCreateProgramWithBinary API 加载 .xclbin 文件。

以下代码示例显示了如何使用标准 OpenCL API 基于 .xclbin 文件构建程序。

```
unsigned char *kernelbinary;
char *xclbin = argv[1];

printf("INFO: loading xclbin %s\n", xclbin);

int size=load_file_to_memory(xclbin, (char **) &kernelbinary);
size_t size_var = size;

cl_program program = clCreateProgramWithBinary(context, 1, &device_id,
                     &size_var,(const unsigned char **) &kernelbinary,
                     &status, &err);

// Function
int load_file_to_memory(const char *filename, char **result)
{
  uint size = 0;
  FILE *f = fopen(filename, "rb");
  if (f == NULL) {
    *result = NULL;
    return -1; // -1 means file opening fail
  }
  fseek(f, 0, SEEK_END);
  size = ftell(f);
  fseek(f, 0, SEEK_SET);
  *result = (char *)malloc(size+1);
  if (size != fread(*result, sizeof(char), size, f)) {
    free(*result);
    return -2; // -2 means file reading fail
```

```
    }
    fclose(f);
    (*result)[size] = 0;
    return size;
}
```

此示例执行的步骤如下：

1. 从命令行实参 `argv[1]` 传入内核二进制文件 `.xclbin`。

   提示： 通过命令行实参来传递 `.xclbin` 只是其中一种方法。您也可以在主机程序中对内核二进制文件进行硬编码，使用环境变量来定义此文件，从定制初始化文件来读取它，或者也可以采用其它合适的机制。

2. `load_file_to_memory` 函数用于在主机存储器空间中加载文件内容。
3. `clCreateProgramWithBinary` API 用于在指定上下文和器件中完成程序创建进程。

在 FPGA 中执行命令

完成 OpenCL 环境初始化后，主机应用即可随时向器件发出命令并与内核进行交互。这些命令包括：

1. 设置内核。
2. 往来 FPGA 的缓冲器传输。
3. FPGA 上的内核执行。
4. 事件同步。

设置内核

设置运行时环境（如识别器件、创建上下文环境、命令队列和程序）后，主机应用应识别将在器件上执行的内核并设置内核实参。

应使用 OpenCL API `clCreateKernel` 访问 `.xclbin` 文件中所包含的内核（“程序”）。`cl_kernel` 对象用于识别加载到 FPGA 内的程序中的内核，此内核可供主机应用运行。以下代码示例可识别加载的程序中定义的 2 个内核。

```
kernel1 = clCreateKernel(program, "<kernel_name_1>", &err);
kernel2 = clCreateKernel(program, "<kernel_name_2>", &err);  // etc
```

设置内核参数

在 Vitis 软件平台中，可为内核对象设置两种类型的实参：

1. 标量实参用于小型数据传输，例如，常量或配置类型数据。从主机应用角度来看，这些实参为只读实参，即内核的输入。
2. 存储缓冲器实参则用于大型数据传输。值是所创建的存储器对象的指针，其上下文与程序和内核对象相关联。这些实参是内核的输入或输出。

内核实参可使用 `clSetKernelArg` 命令来设置，以下示例演示了如何为 2 个标量实参和 2 个缓冲器实参设置内核实参。

```
// Create memory buffers
cl_mem dev_buf1 = clCreateBuffer(context, CL_MEM_WRITE_ONLY |
CL_MEM_USE_HOST_PTR, size, &host_mem_ptr1, NULL);
cl_mem dev_buf2 = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_USE_HOST_PTR, size, &host_mem_ptr2, NULL);
```

```
int err = 0;
// Setup scalar arguments
cl_uint scalar_arg_image_width = 3840;
err |= clSetKernelArg(kernel, 0, sizeof(cl_uint), &scalar_arg_image_width);
cl_uint scalar_arg_image_height = 2160;
err |= clSetKernelArg(kernel, 1, sizeof(cl_uint),
&scalar_arg_image_height);

// Setup buffer arguments
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_buf1);
err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_buf2);
```

重要提示！虽然 OpenCL 允许在内核入队前随时设置内核实参，但您应尽早设置内核实参。如果在 XRT 确定器件上缓冲器的放置位置之前，您尝试移植该缓冲器，那么 XRT 将出错。因此，请在任意缓冲器上执行任意入队操作（例如，`clEnqueueMigrateMemObjects`）之前设置内核实参。

对于所有内核缓冲器实参，您必须在器件全局存储器上分配缓冲器。但有时，开始内核执行之前无需缓冲器内容。例如，仅在内核执行期间填充输出缓冲器内容，因此内核执行前，这些内容无关紧要。在此情况下，您应指定 `clEnqueueMigrateMemObject`（含 `CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED` 标记），这样缓冲器移植就不涉及主机与器件之间的 DMA 操作，从而即可改善性能。

器件上的缓冲器分配

默认情况下，当内核链接到平台时，来自所有内核的存储器接口都连接到单个默认的全局存储体。因此，每次在该全局存储体上只能有 1 个计算单元 (CU) 执行数据传输，这就限制了应用的总体性能。

如果器件仅包含 1 个全局存储体，那么这是唯一选项。但是，如果器件包含多个全局存储体，那么您可以通过在链接期间修改内核的存储器接口连接来自定义全局存储体连接。如需了解有关执行此操作的方法的详细信息，请参阅 将内核端口映射到存储器。针对不同内核或计算单元使用不同的独立存储体可以支持多个内核存储器接口进行数据并发读写，从而提升总体性能。

重要提示！XRT 必须检测内核的存储器连接，以将数据从主机程序发送到内核中正确的存储器位置。如果在缓冲器上执行任何入队操作（例如，`clEnqueueMigrateMemObject`）之前使用了 `clSetKernelArgs`，那么 XRT 将从内核二进制文件中自动查找缓冲器位置。

缓冲器创建和数据传输

主机程序与硬件内核之间的交互依赖于创建缓冲器并在器件中的存储器上传入和传出数据。这一过程会利用诸如 `clCreateBuffer` 和 `clEnqueueMigrateMemObjects` 等函数。

重要提示！单一缓冲器大小不能超过 4 GB，但为了最大限度提升从主机到全局存储器的吞吐量，赛灵思还建议尽可能保留大小至少为 2 MB 的缓冲器。

有两种方法可用于分配存储缓冲器和传输数据：

1. 由 XRT 分配缓冲器
2. 使用主机指针缓冲器

如由 XRT 分配缓冲器，请使用 `enqueueMapBuffer` 来捕获缓冲器句柄。在第二种情况下，使用 `CL_MEM_USE_HOST_PTR` 来直接分配缓冲器，因此您无需捕获缓冲器句柄。

有多种编码实践可供您用于最大程度提升性能和高精度控制。OpenCL API 还支持通过其它命令来读写缓冲器。例如，您可使用 clEnqueueWriteBuffer 和 clEnqueueReadBuffer 命令代替 clEnqueueMigrateMemObjects。但其中部分命令具有不同的影响，使用时必须明确其作用。例如，clEnqueueReadBufferRect 可以读取缓冲器对象的矩形区域并传输到主机应用，但它不会将数据从器件全局存储器传输到主机。您必须首先使用 clEnqueueReadBuffer 将数据从器件全局存储器传输到主机应用，然后使用 clEnqueueReadBufferRect 来读取目标矩形部分，并将其传输到主机应用中。

由 XRT 分配缓冲器

在数据中心平台上，按对齐到 4k 页面边界来分配存储器更高效。在嵌入式平台上，执行连续存储器分配更高效。无论在何种情况下，您都可在创建缓冲器时交由 XRT 来分配主机存储器。创建缓冲器时，分配是通过使用 CL_MEM_ALLOC_HOST_PTR 标记来完成的，随后使用 clEnqueueMapBuffer 将已分配的存储器映射到用户空间指针。利用此方法则无需创建对齐到 4K 边界的主机空间指针。

clEnqueueMapBuffer API 可映射指定的缓冲器，并将 XRT 创建的指针返回到该映射区域。随后，使用您的数据来填充主机侧指针，再使用 clEnqueueMigrateMemObject 进行往来器件的数据传输。以下代码示例使用此编码样式：

```
// Two cl_mem buffer, for read and write by kernel
cl_mem dev_mem_read_ptr = clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR |
CL_MEM_READ_ONLY,
                      sizeof(int) * number_of_words, NULL, NULL);

cl_mem dev_mem_write_ptr = clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR |
CL_MEM_WRITE_ONLY,
                      sizeof(int) * number_of_words, NULL, NULL);


cl::Buffer in1_buf(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY,
sizeof(int) * DATA_SIZE, NULL, &err);

// Setting arguments
clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_mem_read_ptr);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_mem_write_ptr);

// Get Host side pointer of the cl_mem buffer object
auto host_write_ptr =
clEnqueueMapBuffer(queue,dev_mem_read_ptr,true,CL_MAP_WRITE,0,bytes,0,nullpt
r,nullptr,&err);
auto host_read_ptr =
clEnqueueMapBuffer(queue,dev_mem_write_ptr,true,CL_MAP_READ,0,bytes,0,nullpt
r,nullptr,&err);

// Fill up the host_write_ptr to send the data to the FPGA

for(int i=0; i< MAX; i++) {
    host_write_ptr[i] = <.... >
}

// Migrate
cl_mem mems[2] = {host_write_ptr,host_read_ptr};
clEnqueueMigrateMemObjects(queue,2,mems,0,0,nullptr,&migrate_event));

// Schedule the kernel
clEnqueueTask(queue,kernel,1,&migrate_event,&enqueue_event);

// Migrate data back to host
clEnqueueMigrateMemObjects(queue, 1, &dev_mem_write_ptr,
                           CL_MIGRATE_MEM_OBJECT_HOST,1,&enqueue_event,
```

```
&data_read_event);

clWaitForEvents(1,&data_read_event);

// Now use the data from the host_read_ptr
```

要使用 clEnqueueMapBuffer 来处理示例，请参阅 Vitis 示例 GitHub 仓库中的数据传输 (C)。

使用主机指针缓冲器

重要提示！对于嵌入式平台，不建议使用 CL_MEM_USE_HOST_PTR。嵌入式平台需持续性存储器分配，应使用 CL_MEM_ALLOC_HOST_PTR 方法，如 由 XRT 分配缓冲器 中所述。

cl_mem 对象包含 2 个主要部分：主机侧指针和器件侧指针。在内核开始操作之前，器件侧指针是在器件侧存储器上（例如，在器件的全局存储器内部的特定位置）隐式分配的，而缓冲器则驻留在器件上。使用 clEnqueueMigrateMemObjects 意味着先执行此分配并进行数据传输，并且这两项操作远早于内核执行。如果主机多次对相同内核执行操作，那么这对于软件流水打拍非常有帮助，因为当内核仍在对前一个数据集执行操作的同时，即可为后一项传输事务执行数据传输，从而将连续内核执行的数据传输时延隐藏起来。

OpenCL 框架可提供大量 API，用于在主机和器件之间传输数据。通常，数据传输 API（如 clEnqueueWriteBuffer 和 clEnqueueReadBuffer）会在存储器对象完成排队后，将这些对象隐式移植到器件中。由于这些 API 无法保证何时进行数据传输，导致主机应用难以将存储器对象的移动与对数据执行的计算进行同步。

赛灵思建议使用 clEnqueueMigrateMemObjects 代替 clEnqueueWriteBuffer 或 clEnqueueReadBuffer 以提升性能。通过使用此 API 即可在执行相关命令之前显式执行存储器移植。这样主机应用即可通过常规命令队列调度来抢先更改存储器对象的关联，从而为其它后续命令做好准备。这样还可支持应用在实际需要使用存储器对象之前，将这些存储器对象的布局与其它不相关操作进行重叠，从而隐藏或者减小数据传输时延。当与 clEnqueueMigrateMemObjects 关联的事件被标记为完成后，主机程序即可知晓这些存储器对象已成功完成移植。

提示：clEnqueueMigrateMemObjects 的另一大优势在于它可在单次 API 调用内移植多个存储器对象。这样即可减小通过调度和调用函数来为多个存储器对象传输数据所需的开销。

以下代码显示了 clEnqueueMigrateMemObjects 的用法：

```
int host_mem_ptr[MAX_LENGTH]; // host memory for input vector

// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
  host_mem_ptr[i] = <... >
}

cl_mem dev_mem_ptr = clCreateBuffer(context,
                     CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
                     sizeof(int) * number_of_words, host_mem_ptr, NULL);

clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_mem_ptr);

err = clEnqueueMigrateMemObjects(commands, 1, dev_mem_ptr, 0, 0,
      NULL, NULL);
```

分配页面对齐的主机存储器

XRT 在 4K 边界内分配存储器空间，用于内部存储器管理。如果主机存储器指针未对齐到页面边界，那么 XRT 会执行额外的 memcpy 以使其对齐。因此，您应将主机存储器指针与 4K 边界对齐，以免执行额外的存储器复制操作。

以下示例演示了如何为主机存储器空间指针使用 posix_memalign 以代替 malloc。

```
int *host_mem_ptr; // = (int*) malloc(MAX_LENGTH*sizeof(int));
// Aligning memory in 4K boundary
posix_memalign(&host_mem_ptr,4096,MAX_LENGTH*sizeof(int));

// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
  host_mem_ptr[i] = <... >
}

cl_mem dev_mem_ptr = clCreateBuffer(context,
                     CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
                     sizeof(int) * number_of_words, host_mem_ptr, NULL);

err = clEnqueueMigrateMemObjects(commands, 1, dev_mem_ptr, 0, 0,
      NULL, NULL);
```

子缓冲器

虽然子缓冲器并不常用，但在某些特定情况下很有用。以下章节探讨了子缓冲器能够发挥实际作用的情境。

从器件缓冲器中读取某一特定部分

假设某一内核根据其输入会生成不同数据量。例如，某个压缩引擎的输出大小因输入数据模式和相似性而异。主机仍可使用 clEnqueueMigrateMemObjects 来读取整体输出缓冲器，但是这并非最优方法，因为产生的存储器传输可能超出必要的量。理想情况下，主机程序读取的数据量应与内核写入的数据量完全相同。

有一种方法是由内核在开始编写输出数据时即写入输出数据的量。主机应用可以使用 2 次 clEnqueueReadBuffer，第一次读取返回的数据量，第二次则基于第一次读取的信息来读取内核返回的精确数据量。

```
clEnqueueReadBuffer(command_queue,device_write_ptr, CL_FALSE, 0,
sizeof(int) * 1,
                    &kernel_write_size, 0, nullptr, &size_read_event);
clEnqueueReadBuffer(command_queue,device_write_ptr, CL_FALSE,
DATA_READ_OFFSET,
                    kernel_write_size, host_ptr, 1, &size_read_event,
&data_read_event);
```

相比于 clEnqueueReadBuffer 或 clEnqueueWriteBuffer，更推荐使用 clEnqueueMigrateMemObject，您可以通过使用子缓冲器来采用类似的方法。如以下代码样本所示。

提示：此代码样本仅显示部分命令，用作概念演示。

```
//Create a small sub-buffer to read the quantity of data
cl_buffer_region buffer_info_1={0,1*sizeof(int)};
cl_mem size_info = clCreateSubBuffer (device_write_ptr, CL_MEM_WRITE_ONLY,
      CL_BUFFER_CREATE_TYPE_REGION, &buffer_info_1, &err);

// Map the sub-buffer into the host space
auto size_info_host_ptr = clEnqueueMapBuffer(queue, size_info,,,, );

// Read only the sub-buffer portion
clEnqueueMigrateMemObjects(queue, 1, &size_info,
CL_MIGRATE_MEM_OBJECT_HOST,,,);

// Retrive size information from the already mapped size_info_host_ptr
kernel_write_size = ..........
```

```
// Create sub-buffer to read the required amount of data
cl_buffer_region buffer_info_2={DATA_READ_OFFSET, kernel_write_size};
cl_mem  buffer_seg = clCreateSubBuffer (device_write_ptr,
CL_MEM_WRITE_ONLY,
      CL_BUFFER_CREATE_TYPE_REGION, &buffer_info_2,&err);

// Map the subbuffer into the host space
auto read_mem_host_ptr = clEnqueueMapBuffer(queue, buffer_seg,,,);

// Migrate the subbuffer
clEnqueueMigrateMemObjects(queue, 1, &buffer_seg,
CL_MIGRATE_MEM_OBJECT_HOST,,,);

// Now use the read data from already mapped read_mem_host_ptr
```

在多个存储器端口或多个内核之间共享器件缓冲器

有时，内核的存储器端口只需少量数据。但管理小型缓冲器以及传输少量数据都可能对应用引发潜在性能问题。或者，您的主机程序可以创建更大的缓冲器，并将其分割为较小的子缓冲器。对于需要少量数据的每个存储器端口，将把每个子缓冲器分配为一个内核实参，如 设置内核参数 中所述。

创建子缓冲器后，就会在主机代码中以类似常规缓冲器的方式来使用这些子缓冲器。由于 XRT 只需通过单一传输事务来处理大型缓冲器，而无需处理多个小型缓冲器和多项传输事务，因此性能可以得到提升。

内核执行

通常主机应用所需的计算密集型任务可在单个内核内进行定义，并且此内核仅执行一次以处理整个范围内的所有数据。由于内核多次执行存在相关联的开销，因此调用单一整体式内核可以提升性能。虽然内核仅执行一次，并处理整个范围内的所有数据，但并行化是在内核硬件内部的 FPGA 上实现的。只要经过正确编码，内核即可通过多种方法来实现并行化，例如，指令级并行化（循环流水线）和功能级并行化（数据流）。如需了解有关这些不同内核编码方法的信息，请参阅 C/C++ 内核。

在将内核编译到 FPGA 上的单一硬件实例（或 CU）中时，执行内核的最简单的方法是使用 `clEnqueueTask`，如下所示。

```
err = clEnqueueTask(commands, kernel, 0, NULL, NULL);
```

XRT 会调度工作负载或者从内核实参流经 OpenCL 缓冲器的数据，并调度内核任务，以在赛灵思 FPGA 上运行加速器。

重要提示！ 虽然支持使用 `clEnqueueNDRangeKernel`（仅限 OpenCL 内核），但赛灵思建议使用 `clEnqueueTask`。

但有时由于各种原因，无法使用单一 clEnqueueTask 来运行内核。例如，内核代码可能过大且过于复杂，导致在单次执行过程中执行所有计算密集型任务时，难以对代码进行最优化。有时，可将多个内核设计为在 FPGA 上并行执行不同任务，这需要执行多条排队命令。或者主机应用可能在很长一段时间内持续接受数据，且无法一次性同时处理所有数据。因此，根据情况以及应用，您可能需要将内核的数据和任务拆分为多条 `clEnqueueTask` 命令。在此情况下，可通过无序命令队列或有序命令队列来判定内核任务的处理方式，如 命令队列 中所述。此外，可将多个内核任务作为阻塞事件或者非阻塞事件来实现，如 事件同步 中所述。这些都可能影响设计性能。

以下主题探讨的各种方法可供您用于在加速器上运行内核、运行多个内核或者运行相同内核的多个实例。

使用不同内核实现任务并行

有时，主机应用所需的计算密集型任务可分为多个不同内核，这些不同内核设计为在 FPGA 上并行执行不同任务。例如，通过按无序命令队列来使用多条 `clEnqueueTask` 命令，即可让多个执行不同任务的内核并行运行。这样即可在 FPGA 上实现任务并行。

空间数据并行化：增加计算单元数量

有时候，主机应用所需的计算密集型任务可以跨同一内核的多个硬件实例或者跨计算单元来处理数据，以在 FPGA 上实现数据并行化。如果单个内核已编译为多个 CU，那么在单一无序命令队列中可以多次调用 `clEnqueueTask` 命令来支持数据并行化。每次调用 `clEnqueueTask` 都会调度不同 CU 中的数据工作负载，并且对其进行并行处理。

暂时性数据并行化：主机到内核数据流

有时，计算单元所处理的数据经由内核中的某一个处理阶段，进入下一个处理阶段。在此情况下，内核的第一个阶段可能处于空闲状态，可开始处理一组新数据。从本质上来看，内核就像是工厂组装线，它可以在接受新数据的同时，使原始数据沿组装线下行。

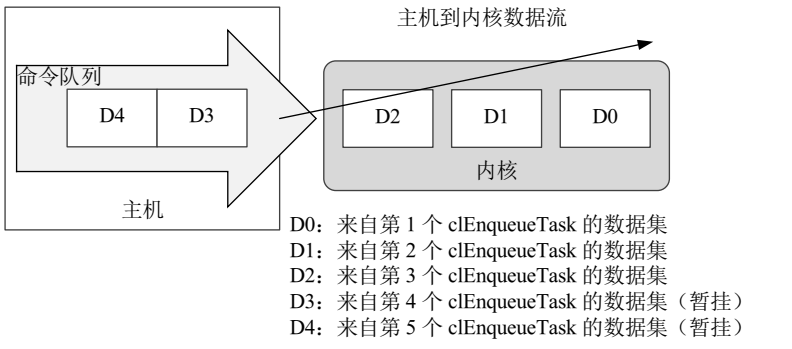
为了理解这种方法，假定内核在 FPGA 上只有 1 个 CU，主机应用会以多组不同数据对内核进行多次排队。如 使用主机指针缓冲器 中所示，主机应用可以在内核执行前将数据移植到器件全局存储器上，从而隐藏内核执行产生的数据传输时延，实现软件流水打拍。

但默认情况下，每个内核只有在完成处理当前数据组之后才能开始处理一组新数据。虽然 `clEnqueueMigrateMemObject` 可以隐藏数据传输时间，但多次内核执行仍保留按顺序执行的方式。

通过启用主机到内核数据流，可以进一步提升加速器性能，方法是在内核仍在处理前一组数据的同时就以一组新数据来重新启动该内核。如 启用主机到内核的数据流 中所述，内核必须实现 `ap_ctrl_chain` 接口，并且必须编写为允许分阶段处理数据。在此情况下，一旦内核能够接受新数据，XRT 就会重新启动内核，从而重叠多次内核执行。但主机应用必须使用请求来保持命令队列处于填满状态，以便内核一旦准备好接受新数据后即可立即重新启动。

以下是主机到内核数据流的概念图。

图 131：**主机到内核数据流**



内核从始至终处理一组数据耗时越长，使用主机到内核数据流来提升性能的可能性就越大。因为这样无需等待至内核完成处理一组数据，只需等待至内核准备好开始处理下一组数据即可。这样即可实现时间并行化，即同一内核的不同阶段以流水打拍方式处理来自多个 `clEnqueueTask` 命令的不同数据组。

对于高级设计，您可将空间并行化（通过多个 CU 处理数据）与时间并行化（使用主机到内核数据流）有机结合，从而在每个计算单元上重叠内核执行。

重要提示！嵌入式处理器平台不支持主机到内核数据流功能。

启用主机到内核的数据流

如果内核在操作来自先前传输事务的数据的同时仍能接受更多数据，那么 XRT 即可发送下一批数据。随后，内核会在不同算法阶段并行处理多个数据集，从而改善性能。为了支持主机到内核数据流，内核必须使用 pragma HLS interface 为 return 函数实现 `ap_ctrl_chain` 协议：

```
void kernel_name( int *inputs,
                  ...          )// Other input or Output ports
{
#pragma HLS INTERFACE  .....    // Other interface pragmas
#pragma HLS INTERFACE ap_ctrl_chain port=return bundle=control
```

重要提示！为了充分利用主机到内核数据流，内核还必须分阶段写入处理数据，例如，在循环级别进行流水打拍（如 循环流水打拍 中所述）或者在任务级别进行流水打拍（如 数据流最优化 中所述）。

对称计算单元和非对称计算单元

如 创建内核的多个实例 中所述，在内核链接进程中，可在 FPGA 上例化单一内核的多个计算单元 (CU)。根据同一内核中不同 CU 之间的相互关系，可将 CU 称为对称或非对称 CU。

- 对称：如果 CU 间具有完全相同的 `connectivity.sp` 选项，因而与全局存储器的连接都相同，则可视为对称 CU。这些 CU 即可供赛灵思的 Xilinx Runtime 互换使用。调用 `clEnqueueTask` 可能导致调用对称 CU 组中的任意实例。
- 非对称：如果 CU 间没有完全相同的 `connectivity.sp` 选项，因而与全局存储器的连接不同，则可视为非对称 CU。使用相同的输入和输出缓冲器设置的情况下，XRT 无法以互换方式来执行非对称 CU。

内核句柄和计算单元

针对给定内核对象首次调用 `clSetKernelArg` 时，XRT 会识别一组对称 CU 以供在后续执行此内核时使用。为此内核调用 `clEnqueueTask` 时，该组中的任意对称 CU 均可用于处理此任务。

如果给定内核的所有 CU 均为对称 CU，那么单个内核对象即足以访问任意 CU。但如果存在非对称 CU，则主机应用需为每一组非对称 CU 创建唯一的内核对象。在此情况下，调用 `clEnqueueTask` 时必须指定要用于该任务的内核对象，并且该内核的任意匹配 CU 均可供 XRT 使用。

为特定计算单元创建内核对象

为创建与特定计算单元关联的内核，`clCreateKernel` 命令支持在主机程序创建内核对象时指定 CU。该命令的语法如下所示：

```
// Create kernel object only for a specific compute unit
cl_kernel kernelA = clCreateKernel(program,"<kernel_name>:
{compute_unit_name}",&err);
// Create a kernel object for two specific compute units
cl_kernel kernelB = clCreateKernel(program, "<kernel_name>:{CU1,CU2}",
&err);
```

重要提示！如 创建内核的多个实例 中所述，CU 数量是由配置文件中的 `connectivity.nk` 选项指定的，此配置文件供 `v++` 命令在链接期间使用。因此，无论主机程序中指定任何对象，要创建内核对象或对其进行排队，所使用的选项必须与链接期间使用的配置文件所指定的选项相匹配。

在此情况下，赛灵思的 Xilinx Runtime 会在创建内核时识别特定 CU 或 CU 组的内核句柄（kernelA 和 kernelB）。这样您即可在主机程序中使用 clEnqueueTask 时，控制所使用的内核配置或特定 CU 实例。这对于非对称 CU 或者执行 CU 的负载和优先级管理都很有用。

使用计算单元名称来获取所有非对称计算单元的句柄

如果内核对多个非对称 CU 进行例化，可指定含 CU 名称的 clCreateKernel 命令来创建不同的 CU 组。在此情况下，主机程序可使用 clCreateKernel 返回的 cl_kernel 句柄来引用特定 CU 组。

在以下示例中，内核 mykernel 具有 5 个 CU：K1、K2、K3、K4 和 K5。K1、K2 和 K3 CU 均为对称组，在器件上具有对称连接。同样，CU K4 和 K5 构成第二个对称 CU 组。以下代码片段显示了如何使用 cl_kernel 句柄对特定 CU 组进行寻址。

```
// Kernel handle for Symmetrical compute unit group 1: K1,K2,K3
cl_kernel kernelA = clCreateKernel(program,"mykernel:{K1,K2,K3}",&err);

for(i=0; i<3; i++) {
  // Creating buffers for the kernel_handle1
  .....
  // Setting kernel arguments for kernel_handle1
  .....
  // Enqueue buffers for the kernel_handle1
  .....
  // Possible candidates of the executions K1,K2 or K3
  clEnqueueTask(commands, kernelA, 0, NULL, NULL);
  //
}

// Kernel handle for Symmetrical compute unit group 1: K4, K5
cl_kernel kernelB = clCreateKernel(program,"mykernel:{K4,K5}",&err);

for(int i=0; i<2; i++) {
  // Creating buffers for the kernel_handle2
  .....
  // Setting kernel arguments for kernel_handle2
  .....
  // Enqueue buffers for the kernel_handle2
  .....
  // Possible candidates of the executions K4 or K5
  clEnqueueTask(commands, kernelB, 0, NULL, NULL);
}
```

事件同步

基于 OpenCL 队列的所有 API 调用均为异步调用。在命令队列中，当命令入队后，将立即返回这些命令。要暂停主机程序以等待结果，或者要解决命令之间的任何依赖关系，都可使用 API 调用（如 clFinish 或 clWaitForEvents）来阻止执行主机程序。

以下代码显示了 clFinish 和 clWaitForEvents 的示例。

```
err = clEnqueueTask(command_queue, kernel, 0, NULL, NULL);
// Execution will wait here until all commands in the command queue are
finished
clFinish(command_queue);

// Create event, read memory from device, wait for read to complete, verify
results
cl_event readevent;
// host memory for output vector
```

```
int host_mem_output_ptr[MAX_LENGTH];
//Enqueue ReadBuffer, with associated event object
clEnqueueReadBuffer(command_queue, dev_mem_ptr, CL_TRUE, 0, sizeof(int) *
number_of_words,
  host_mem_output_ptr, 0, NULL, &readevent );
// Wait for clEnqueueReadBuffer event to finish
clWaitForEvents(1, &readevent);
// After read is complete, verify results
...
```

请注意以上示例中这些命令的使用方式：

1. `clFinish` API 已显式用于阻止主机执行，直至内核执行完成为止。这是很有必要的，否则主机可能过早尝试从 FPGA 缓冲器回读，并且可能读取垃圾数据。
2. 从 FPGA 存储器到本地主机的数据传输是通过 `clEnqueueReadBuffer` 完成的。此处 `clEnqueueReadBuffer` 的最后一个实参会返回事件对象，用于识别这条特定读取命令，并且可用于查询事件或者等待这条特定命令完成。`clWaitForEvents` 命令可指定单一事件（读取事件），并等待以确保数据完成后再验证数据。

后处理和 FPGA 清理

在主机代码的最后，应使用适当的释放功能来释放所有已分配的资源。如果未正确释放资源，Vitis 核开发套件可能无法生成正确的性能相关剖析和分析报告。

```
clReleaseCommandQueue(Command_Queue);
clReleaseContext(Context);
clReleaseDevice(Target_Device_ID);
clReleaseKernel(Kernel);
clReleaseProgram(Program);
free(Platform_IDs);
free(Device_IDs);
```

OpenCL 内核开发

以下 OpenCL™ 内核讨论内容是基于 C/C++ 内核 主题中所提供的信息来给出的。对于 C/C++ 和 OpenCL 内核，可使用相同的编程技巧来加速内核性能。但 OpenCL 内核使用 `__attribute` 语法代替编译指示。如需了解有关可用属性的详细信息，请参阅 OpenCL 属性。

以下代码示例演示的是 OpenCL 内核的部分元素，这些元素适用于 Vitis™ 应用加速开发流程。本篇并非旨在作为有关 OpenCL 或内核开发的入门读物，而仅仅只是为了突显 OpenCL 内核与 C/C++ 内核之间的一些主要差异。

内核特征符

在 C/C++ 内核中，在 Vitis 编译器命令行上使用 `v++ --kernel` 选项来识别内核。但在 OpenCL 代码中，`__kernel` 关键字可用于识别代码中的内核。您可在单一 `.cl` 文件中定义多个内核，但除非您指定 `--kernel` 选项以识别要编译的内核，否则 Vitis 编译器将编译所有内核。

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void apply_watermark(__global const TYPE * __restrict input,
    __global TYPE * __restrict output, int width, int height) {
{
    ...
}
```

提示：以上内核函数 `apply_watermark` 的完整代码可在 Vitis 加速示例 GitHub 仓库的全局存储器双存储体示例 (CL) 示例中找到。

在以上示例中，您可以看到 watermark 内核具有 2 个指针类型的实参：`input` 和 `output`，并具有 2 个标量类型的整数实参：`width` 和 `height`。

在 C/C++ 内核中，需使用 `HLS INTERFACE` 编译指示来识别这些实参。但在 OpenCL 内核中，Vitis 编译器和 Vitis HLS 可识别内核实参，并按需对其进行编译：将指针实参编译到 `m_axi` 接口中，将标量实参编译到 `s_axilite` 接口中。

内核最优化

由于内核在目标平台上的可编程逻辑中运行，因此根据环境进行任务最优化是应用设计中的重要一环。C/C++ 内核 中提到的大部分最优化技巧均可应用于 OpenCL 内核。但对于 OpenCL 内核，使用的并非先前用于 C/C++ 内核的 HLS 编译指示，而是改为使用 `__attribute__` 关键字，如 OpenCL 属性 中所述。下面给出 1 个示例：

```
// Process the whole image
__attribute__((xcl_pipeline_loop))
image_traverse: for (uint idx = 0, x = 0 , y = 0  ; idx < size ; ++idx, x+=
DATA_SIZE)
{
    ...
}
```

以上示例指定 `for` 循环 `image_traverse` 应通过流水打拍来提升内核性能。在此例中，目标 II 为 1。如需了解更多信息，请参阅 xcl_pipeline_loop。

在以下代码示例中，watermark 函数使用 `opencl_unroll_hint` 属性，让 Vitis 编译器展开循环从而降低时延并提升性能。但在此例中，`__attribute__` 仅作为建议，编译器可按需忽略。如需了解详细信息，请参阅 opencl_unroll_hint。

```
//Unrolling below loop to process all 16 pixels concurrently
__attribute__((opencl_unroll_hint))
watermark: for ( int i = 0 ; i < DATA_SIZE ; i++)
{
    ...
}
```

如需了解更多信息，请复查 OpenCL 属性 主题，了解 OpenCL 内核支持的特定最优化，并复查 C/C++ 内核 内容，以查看如何在您的内核设计中应用这些最优化。

在 OpenCL 内核中设置数据宽度

对于 OpenCL 内核，API 可提供属性以支持和使用增量 AXI 数据宽度。为了消除手动修改代码的需求，通过解读以下 OpenCL 属性可执行数据路径拓宽和算法矢量化：

- `vec_type_hint`
- `reqd_work_group_size`
- `xcl_zero_global_work_offset`

查看以下案例中的组合功能：

```
__attribute__((reqd_work_group_size(64, 1, 1)))
__attribute__((vec_type_hint(int)))
__attribute__((xcl_zero_global_work_offset))
__kernel void vector_add(__global int* c, __global const int* a, __global
const int* b) {
    size_t idx = get_global_id(0);
    c[idx] = a[idx] + b[idx];
}
```

在此案例中，硬编码接口是一个位宽为 32 位的数据路径 `(int *c, int* a, int *b)`，如果直接实现，会极大限制存储器的吞吐量。但是，根据三个属性的值，可应用自动拓宽和变换。

- `__attribute__((vec_type_hint(int)))`：声明 `int` 是用于计算和存储器传输（32 位）的主要类型。此知识用于根据 AXI 接口（512 位）的目标带宽计算矢量化/拓宽因数。在本示例中，该因数为 16 = 512 位/32 位。这表示，在理论上，如果可以应用矢量化，则可以处理 16 个值。
- `__attribute__((reqd_work_group_size(X, Y, Z)))`：定义工作项的总数（其中 `X`、`Y` 和 `Z` 为正常数）。`X*Y*Z` 是工作项的最大数量，因此请定义尽可能大的矢量化因数，该因数将填满存储器带宽。在本示例中，工作项的总数是 `64*1*1=64`。

  实际使用的矢量化因数为实际编码类型或 `vec_type_hint` 所定义的矢量化因数与通过 `reqd_work_group_size` 定义的最大矢量化因数的最大公约数。

  最大矢量化因数的商除以实际矢量化因数将得到 OpenCL 描述的剩余循环计数。由于此循环经过流水打拍，因此最好对剩余循环进行多次迭代，以便充分利用经流水打拍的实现。如果矢量化 OpenCL 代码具有很长的时延，尤其如此。
- `__attribute__((xcl_zero_global_work_offset))`：`__attribute__((xcl_zero_global_work_offset))` 会告知编译器，在运行时不使用任何全局偏移参数，并且所有访问都对齐。这样即可向编译器提供有关工作组对齐的宝贵信息，通常此信息会被传播至存储器访问的对齐（硬件更少）。

应注意，应用这些变换会改变要综合的实际设计。部分展开的循环要求重塑存储数据的本地阵列。此操作通常会顺利进行，但在极少情况下交互可能会很差。

例如：

- 对于分区的阵列，当分区因数不能被展开/矢量化因数整除时。
  - 生成的访问需要大量多路复用器，从而给调度器制造难题（可能显著增加存储器使用量和编译时间）。赛灵思建议使用 2 次幂作为分区因数，因为矢量化因数始终为 2 次幂值）。
- 如果进行矢量化的循环具有未关联的资源约束，则调度器会警告无法满足 II。
  - 这不一定与性能损失相关（通常其性能表现仍然更好），因为 II 是在未展开的循环上进行计算的（因而每次迭代的吞吐量都成倍增加）。
  - 调度器会通知您可能产生的资源约束，并告知您解决这些约束将进一步提升性能。
  - 请注意，一种常见的现象是局部阵列不会自动重塑（通常是因为在后续代码段中会以非矢量化的方法对它进行访问）。

在 OpenCL 内核中减小内核间通信时延

OpenCL API 2.0 规格引入了一种称为管道的新存储器对象。管道以先进先出 (FIFO) 方式来存储数据。管道对象只能使用从管道读取和写入的内置函数来访问。管道对象无法从主机进行访问。管道可用于在 FPGA 内部的不同内核之间进行数据流传输，而无需使用外部存储器，这样即可显著改善总体系统时延。如需了解更多信息，请参阅来自 Khronos Group 的 OpenCL C 规范 2.0 版的管道函数。

在 Vitis IDE 中，管道必须在所有内核函数外部进行静态定义。不支持使用 OpenCL 2.x `clCreatePipe` API 进行动态管道分配。管道深度必须在管道声明中使用 OpenCL 属性 `xcl_reqd_pipe_depth` 来指定。如需了解更多信息，请参阅 xcl_reqd_pipe_depth。

根据 `xcl_reqd_pipe_depth` 中的指定，有效深度值如下：16、32、64、128、256、512、1024、2048、4096、8192、16384 和 32768。

在不同内核中，一个给定的管道可以有且只能有一个生产者和消费者。

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
```

在阻塞模式下，管道可使用赛灵思的 `read_pipe_block()` 和 `write_pipe_block()` 扩展函数来访问。

提示：不支持使用非阻塞 `read_pipe()` 或 `write_pipe()` 函数来读取或写入管道。

可以使用 OpenCL `get_pipe_num_packets()` 和 `get_pipe_max_packets()` 内置函数来查询管道的状态。

以下函数特征符是当前受支持的管道函数，其中 `gentype` 表示内置的 OpenCL C 语言标量整数或浮点数据类型。

```
int read_pipe_block (pipe gentype p, gentype *ptr)
int write_pipe_block (pipe gentype p, const gentype *ptr)
```

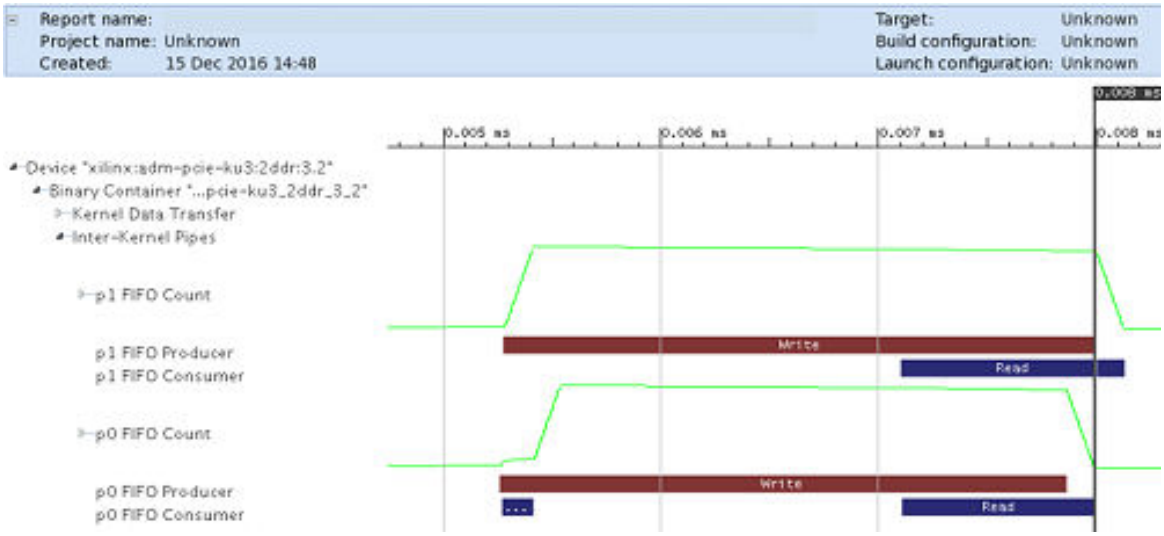
以下来自 GitHub 上的赛灵思入门示例的“dataflow/dataflow_pipes_ocl”使用管道通过阻塞函数 `read_pipe_block()` 和 `write_pipe_block()` 将数据从某一个处理阶段传递到另一个处理阶段。

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel : Read Data from Global Memory and write into Pipe P0
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_rd: for (int i = 0 ; i < size ; i++)
    {
        //blocking Write command to pipe P0
        write_pipe_block(p0, &input[i]);
    }
}
// Adder Stage Kernel: Read Input data from Pipe P0 and write the result
// into Pipe P1
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void adder_stage(int inc, int size)
{
    __attribute__((xcl_pipeline_loop))
    execute: for(int i = 0 ; i < size ;  i++)
    {
        int input_data, output_data;
        //blocking read command to Pipe P0
        read_pipe_block(p0, &input_data);
        output_data = input_data + inc;
```

```
        //blocking write command to Pipe P1
        write_pipe_block(p1, &output_data);
    }
}
// Output Stage Kernel: Read result from Pipe P1 and write the result to
Global
// Memory
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_wr: for (int i = 0 ; i < size ; i++)
    {
        //blocking read command to Pipe P1
        read_pipe_block(p1, &output[i]);
    }
}
```

“Device Traceline”视图显示在硬件仿真运行后 OpenCL 管道上的详细活动和停滞情况。此信息可用于选择正确的 FIFO 规模，以便获得最佳的应用区域和性能。

图 132： **“Device Traceline”视图**



OpenCL 属性

本节旨在描述 OpenCL™ 属性，这些属性可通过 Vitis 核开发套件和 Vitis HLS 工具综合添加到源代码中，以便辅助系统最优化。

Vitis 核开发套件可以提供 OpenCL 属性用于代码最优化，以便简化数据移动和提升内核性能。数据移动最优化的目的是通过最大程度提升接口带宽利用率和 DDR 带宽利用率来最大程度提升系统级别数据吞吐量。内核计算最优化的目标是创建处理逻辑，此逻辑可在数据到达内核接口时立即使用所有数据。为此，最常用的方法是展开处理代码，并将数据路径与诸如函数内联和流水打拍、循环展开、阵列分区、数据流等方法搭配使用。

下表包含按类型指定的 OpenCL 属性。

表 61：**OpenCL 属性（按类型）**

类型	属性
内核最优化	· reqd_work_group_size · vec_type_hint · work_group_size_hint · xcl_latency · xcl_max_work_group_size · xcl_zero_global_work_offset
函数内联	· always_inline
任务级流水线	· xcl_dataflow · xcl_reqd_pipe_depth
流水线	· xcl_pipeline_loop · xcl_pipeline_workitems
循环最优化	· opencl_unroll_hint · xcl_loop_tripcount · xcl_pipeline_loop
阵列最优化	· xcl_array_partition · xcl_array_reshape **注释：**阵列变量仅接受单一阵列最优化属性。

提示：Vitis 编译器还支持许多受 `gcc` 支持的标准属性，例如：
- ALWAYS_INLINE
- NOINLINE
- UNROLL
- NOUNROLL

always_inline

描述

ALWAYS_INLINE 属性用于指示函数必须为内联函数。该属性是 GCC 的标准功能，也是 Vitis 编译器的标准功能。

提示：NOINLINE 属性也是 GCC 的标准功能，同时也受 Vitis 编译器支持。

该属性支持编译器最优化在调用函数中包含内联函数。此内联函数将消隐，不再显示为 RTL 中的层级的独立层次。

在某些情况下，内联函数允许通过调用函数中的周围运算来共享和最优化该函数内的运算。但内联函数无法再与其它函数共享，因此在内联函数与可广泛共享的函数的独立实例之间可能存在重复的逻辑。虽然这样可以改善性能，但也将增加实现 RTL 所需的面积。

对于 OpenCL 内核，Vitis 编译器使用其自己的规则来判断是否对函数进行内联。要直接控制内联函数，请使用 ALWAYS_INLINE 或 NOINLINE 属性。

默认情况下，内联只能在函数层级的下一级上执行，而不能对子函数执行。

重要提示！搭配 XCL_DATAFLOW 属性来使用时，编译器将忽略 ALWAYS_INLINE 属性，而不对函数执行内联。

语法

将该属性置于 OpenCL API 源代码中函数定义之前的位置，这样只要调用该函数就会将其内联。

```
__attribute__((always_inline))
```

示例

此示例给 `foo` 函数添加了 ALWAYS_INLINE 属性：

```
__attribute__((always_inline))
  void foo ( a, b, c, d ) {
  ...
}
```

此示例则阻止 `foo` 函数的内联：

```
__attribute__((noinline))
  void foo ( a, b, c, d ) {
  ...
}
```

另请参阅

- https://gcc.gnu.org

opencl_unroll_hint

描述

重要提示！这是编译器提示，编译器可将其忽略。

循环展开是 Vitis 编译器中可用的最优化技巧之一。循环展开最优化的目的是向编译器公开并发。这一新公开的并发可以降低时延并改善性能，但占用更多 FPGA 互连结构资源。

OPENCL_UNROLL_HINT 属性是 OpenCL 规范的一部分，用于指定循环（`for`、`while`、`do`）可由 Vitis 编译器展开。如需了解更多信息，请参阅 循环展开。

OPENCL_UNROLL_HINT 属性限定符必须显示在受影响的循环之前并与之紧邻。您可使用该属性来完全展开循环、按指定量部分展开或者禁用循环展开。

语法

将该属性置于 OpenCL 源代码中循环定义前的位置：

```
__attribute__((opencl_unroll_hint(<n>)))
```

其中：

- <n> 是可选循环展开因子，必须为正整数，或为编译时间常量表达式。展开因子为 1 表示禁用展开。

提示：如果不指定 <n>，那么编译器会自动确定循环的展开因子。

示例

以下示例按因子 2 展开 `for` 循环。这样将生成 2 次并行循环迭代，而不是 4 次顺序迭代，以供计算单元完成操作。

```
__attribute__((opencl_unroll_hint(2)))
for(int i = 0; i < LENGTH; i++) {
bufc[i] = bufa[i] * bufb[i];
}
```

从概念上讲，编译器会将以上循环变换为以下代码。

```
for(int i = 0; i < LENGTH; i+=2) {
bufc[i] = bufa[i] * bufb[i];
bufc[i+1] = bufa[i+1] * bufb[i+1];
}
```

另请参阅

- https://www.khronos.org/
- 《OpenCL C 语言规范》

reqd_work_group_size

描述

在 OpenCL 器件上提交 OpenCL API 内核以供执行时，这些内核将在索引空间内执行，此索引空间称为 ND 范围，可包含 1、2 或 3 个维度。在 OpenCL API 中，这被称为全局大小。工作组大小用于定义每个内核计算单元 (CU) 的单一调用可处理的 ND 范围量。工作组大小在 OpenCL API 中也被称为局部大小。OpenCL 编译器可基于内核和选定器件的属性来判定工作组大小。确定工作组大小（局部大小）后，ND 范围（全局大小）将被自动拆分为工作组，并对这些工作组进行调度以便在器件上执行。

虽然 OpenCL 编译器可以定义工作组大小，但强烈建议使用内核上用于定义工作组大小的 REQD_WORK_GROUP_SIZE 属性规格来执行内核的 FPGA 实现。建议在为内核生成定制逻辑期间使用该属性进行性能最优化。

提示：在执行 FPGA 实现时，强烈建议使用 REQD_WORK_GROUP_SIZE 属性的规格，原因在于它可用于在为内核生成定制逻辑期间进行性能最优化。

针对 ND 范围索引空间内的每个点，OpenCL 内核函数仅执行一次。ND 范围内每个点的工作单元称为工作项 (work-item)。工作项会被分组到工作组中，这些工作组将作为工作单元被调度到计算单元上。可选 REQD_WORK_GROUP_SIZE 属性可定义计算单元的工作组大小，这些工作组大小必须被用作为 `clEnqueueNDRangeKernel` 的 `local_work_size` 实参。这样即可允许编译器以适当的方式为此内核优化生成的代码。

语法

将该属性置于内核定义之前或针对内核指定的主函数之前的位置。

```
__attribute__((reqd_work_group_size(<X>, <Y>, <Z>)))
```

其中：

- <X>, <Y>, <Z>：指定内核的 ND 范围。这表示三维矩阵的每个维度，用于指定内核的工作组大小。

示例

一下 OpenCL C 内核代码显示的矢量加法设计中，两个数据阵列经求和得到第三个阵列。工作组所需大小为 16x1x1。此内核将执行 16 次以生成有效结果。

```
#include <clc.h>
// For VHLS OpenCL C kernels, the full work group is synthesized
__attribute__ ((reqd_work_group_size(16, 1, 1)))
__kernel void
vadd(__global int* a,
__global int* b,
__global int* c)
{
int idx = get_global_id(0);
c[idx] = a[idx] + b[idx];
}
```

另请参阅

- https://www.khronos.org/
- 《OpenCL C 语言规范》

vec_type_hint

描述

重要提示！这是编译器提示，编译器可将其忽略。

可选 `__attribute__((vec_type_hint(<type>)))` 是 OpenCL 语言规范的一部分，也是 OpenCL 编译器提示，表示内核的计算宽度，在编译器对代码进行自动矢量化时，它可作为基础用于计算处理器带宽使用情况。

默认情况下，内核假定包含 `__attribute__((vec_type_hint(int)))` 限定符。它支持您指定不同的矢量化类型。

自动矢量化隐含如下假定：从内核调用的任何库都必须可在运行时重新编译，以便处理编译器决定合并或拆分工作项的情况。这意味着，这些库不允许包含硬编码二进制文件，或者硬编码的二进制文件必须随附源代码或者某些可重定向的中间表示法。在某些情况下，这可能导致代码安全性问题。

语法

将该属性置于内核定义之前或针对内核指定的主函数之前的位置。

```
__attribute__((vec_type_hint(<type>)))
```

其中：

- <type>：是下表所列的内置矢量类型之一，或者是标量元素类型之一。

  注释：如不指定，内核假定包含 INT 类型。

表 62：**矢量类型**

类型	描述
char<n>	矢量 <n>，8 位有符号 2 的补码整数值。
uchar<n>	矢量 <n>，8 位无符号整数值。
short<n>	矢量 <n>，16 位有符号 2 的补码整数值。
ushort<n>	矢量 <n>，16 位无符号整数值。
int<n>	矢量 <n>，32 位有符号 2 的补码整数值。
uint<n>	矢量 <n>，32 位无符号整数值。
long<n>	矢量 <n>，64 位有符号 2 的补码整数值。
ulong<n>	矢量 <n>，64 位无符号整数值。
float<n>	矢量 <n>，32 位浮点值。
double<n>	矢量 <n>，64 位浮点值。

注释： <n> 如未指定，则假定为 1。以上定义的矢量数据类型的名称（其中 <n> 为除 2、3、4、8 和 16 以外的任意值）同样为保留值。因此，< n> 只能指定为 2、3、4、8 和 16。

示例

以下自动矢量化示例假定采用双宽整数作为基本计算宽度。

```
#include <clc.h>
// For VHLS OpenCL C kernels, the full work group is synthesized
__attribute__((vec_type_hint(double)))
__attribute__ ((reqd_work_group_size(16, 1, 1)))
__kernel void
...
```

另请参阅

- https://www.khronos.org/
- 《OpenCL C 语言规范》

work_group_size_hint

描述

重要提示！这是编译器提示，编译器可将其忽略。

OpenCL API 标准中的工作组大小可定义 ND 范围空间的大小，即内核计算单元的单次调用可处理的大小。在 OpenCL 器件上提交 OpenCL 内核以供执行时，这些内核将在索引空间内执行，此索引空间称为 ND 范围，可包含 1、2 或 3 个维度。

针对 ND 范围索引空间内的每个点，OpenCL 内核函数仅执行一次。ND 范围内每个点的工作单元称为工作项 (work-item)。不同于 C 语言中的按顺序有序执行迭代的 `for` 循环，OpenCL 运行时和器件能够按任意顺序并行执行工作项。

工作项会被分组到工作组中，这些工作组将作为工作单元被调度到计算单元上。在 OpenCL 语言规范中包含可选 WORK_GROUP_SIZE_HINT 属性，该属性作为编译器提示，表示 `local_work_size` 实参最有可能指定给 `clEnqueueNDRangeKernel` 的工作组大小值。这样即可允许编译器根据期望的值对生成的代码进行最优化。

提示：在执行 FPGA 实现时，强烈建议使用 REQD_WORK_GROUP_SIZE 属性的规格代替 WORK_GROUP_SIZE_HINT，原因在于前者可用于在为内核生成定制逻辑期间进行性能最优化。

语法

将该属性置于内核定义之前或针对内核指定的主函数之前的位置：

```
__attribute__((work_group_size_hint(<X>, <Y>, <Z>)))
```

其中：

- <X>, <Y>, <Z>：指定内核的 ND 范围。这表示三维矩阵的每个维度，用于指定内核的工作组大小。

示例

以下示例可作为编译器提示，表示内核将很有可能按工作组大小 1 来执行。

```
__attribute__((work_group_size_hint(1, 1, 1)))
__kernel void
...
```

另请参阅

- https://www.khronos.org/
- 《OpenCL C 语言规范》

xcl_array_partition

描述

重要提示！阵列变量仅接受 1 个属性。虽然 XCL_ARRAY_PARTITION 可支持多维阵列，但每个属性只能重构阵列的 1 个维度。

对于 OpenCL 程序而言，使用 FPGA 相比于其它计算器件的优势在于，应用程序员能够在整个系统中和计算单元内自定义存储器架构。默认情况下，Vitis 编译器可在计算单元内生成存储器架构，从而根据内核代码的静态代码分析最大程度提升本地和专用存储器带宽。根据内核源代码中的属性，可进一步对这些存储器进行最优化，这些属性可用于指定本地和专用存储器的物理布局和实现。Vitis 编译器中用于控制计算单元中的存储器的物理布局的属性是 `array_partition`。

对于一维阵列，XCL_ARRAY_PARTITION 属性会将内核代码中声明的任一阵列实现为多个物理存储器，而不是单个物理存储器。根据具体应用及其性能目标来选择要使用的分区方案。Vitis 编译器中可用的阵列分区方案为 `cyclic`、`block` 和 `complete`。

语法

将属性与阵列变量定义置于一处。

```
__attribute__((xcl_array_partition(<type>, <factor>,
<dimension>)))
```

其中：

- <type>：指定以下分区类型之一：
  - cyclic：周期分区是将阵列实现作为一组较小的物理存储器，可供计算单元内的逻辑同时访问。该阵列按周期进行分区，具体方式是在每个存储器中放入一个元素，然后回到第一个存储器以重复该周期直至阵列完全完成分区为止。
  - block：块分区是将阵列的物理实现作为一组较小的存储器，可供计算单元内的逻辑同时访问。在此情况下，每个存储器块内都填满来自该阵列的元素后再移至下一个存储器。
  - complete：完全分区可将阵列分解为多个独立元素。对于一维阵列，这对应于将存储器解析为独立寄存器。<type> 默认值为 complete。
- <factor>：对于周期类型的分区，<factor> 可指定在内核代码中，原始阵列分区到的物理存储器的数量。对于块类型分区，<factor> 可指定要在每个物理存储器中存储的来自原始阵列的元素数量。

  重要提示！对于 complete 类型的分区，不指定 <factor>。
- <dimension>：指定要分区的阵列维度。指定为范围介于 1 到 <N> 之间的整数。Vitis 核开发套件支持 N 维阵列，可在任何单一维度上对阵列进行分区。

示例 1

例如，请考虑以下阵列声明。

```
int buffer[16];
```

名为 buffer 的整数阵列存储了 16 个值（位宽为 32 位）。周期分区可通过以下声明应用于此阵列。

```
int buffer[16] __attribute__((xcl_array_partition(cyclic,4,1)));
```

在此示例中，周期 <partition_type> 属性告知 Vitis 编译器在 4 个物理存储器之间分发阵列内容。该属性可按因子 4 来为访问阵列缓冲器的操作增大即时存储器带宽。

在 Vitis 核开发套件上下文中的任一计算单元内的所有阵列均可维持最多 2 个并发访问。通过将代码中的原始阵列分为 4 个物理存储器，生成的计算单元可以维持对该阵列缓冲器最多 8 个并发访问。

示例 2

同样使用示例 1 中的整数阵列，块分区可通过以下声明应用于该阵列。

```
int buffer[16] __attribute__((xcl_array_partition(block,4,1)));
```

由于块的大小为 4，Vitis 编译器将生成 4 个物理存储器，并以来自该阵列的数据按顺序填充每个存储器。

示例 3

同样使用示例 1 中的整数阵列，完整分区可通过以下声明应用于该阵列。

```
int buffer[16] __attribute__((xcl_array_partition(complete, 1)));
```

在此示例中，阵列完全分区为分布式 RAM，或内核的可编程逻辑内的 16 个独立寄存器。由于完全分区为默认方式，因此通过以下声明也可以实现相同效果。

```
int buffer[16] __attribute__((xcl_array_partition));
```

虽然由此创建的实现含有最高的存储器带宽，但它并不适用于所有应用。Vitis 编译器必须围绕每个寄存器构建相应数量的支持逻辑，以确保功能与原始代码中的用法相同，而内核代码通过常数索引或数据相关索引来访问数据的方式则会影响此支持逻辑的数量。Vitis 核开发套件的一般最佳实践准则是，如果通过使用常数索引访问阵列中的至少 1 个维度，则此类阵列最适合完整分区属性。

另请参阅

- xcl_array_reshape
- pragma HLS array_partition
- Vitis HLS 流程

xcl_array_reshape

描述

重要提示！阵列变量仅接受 1 个属性。虽然 XCL_ARRAY_RESHAPE 属性可支持多维阵列，但每个属性只能重构阵列的 1 个维度。

该属性用于将阵列分区与垂直阵列映射相结合。

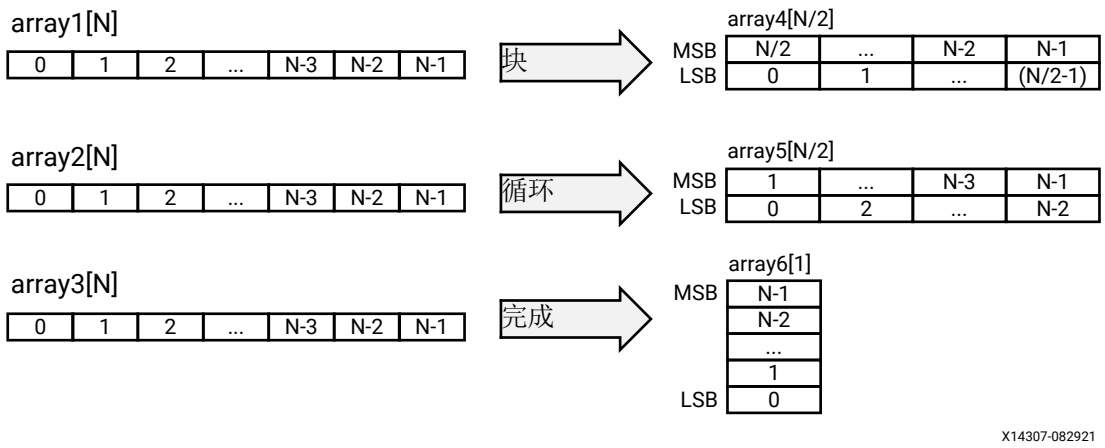
XCL_ARRAY_RESHAPE 属性可与 XCL_ARRAY_PARTITION 的效果相结合、将任一阵列拆分为多个较小的阵列，并通过增加位宽来将阵列元素连接起来。这样即可减少耗用的块 RAM 数量，同时提供对数据的并行访问。该属性能够创建新阵列，其中包含元素数量更少但位宽更大，这样即可在单一时钟周期内访问更多数据。

给定如下代码：

```
void foo (...) {
int array1[N] __attribute__((xcl_array_reshape(block, 2, 1)));
int array2[N] __attribute__((xcl_array_reshape(cycle, 2, 1)));
int array3[N] __attribute__((xcl_array_reshape(complete, 1)));
...
}
```

ARRAY_RESHAPE 属性可将阵列转换为下图所示形式。

图 133：**ARRAY_RESHAPE**



语法

将属性与阵列变量定义置于一处。

```
__attribute__((xcl_array_reshape(<type>,<factor>,
<dimension>)))
```

其中：

- `<type>`：指定以下分区类型之一：
  - `cyclic`：周期分区是将阵列实现作为一组较小的物理存储器，可供计算单元内的逻辑同时访问。该阵列按周期进行分区，具体方式是在每个存储器中放入一个元素，然后回到第一个存储器以重复该周期直至阵列完全完成分区为止。
  - `block`：块分区是将阵列的物理实现作为一组较小的存储器，可供计算单元内的逻辑同时访问。在此情况下，每个存储器块内都填满来自该阵列的元素后再移至下一个存储器。
  - `complete`：完全分区可将阵列分解为多个独立元素。对于一维阵列，这对应于将存储器解析为独立寄存器。`<type>` 默认值为 `complete`。
- `<factor>`：对于周期类型的分区，`<factor>` 可指定在内核代码中，原始阵列分区到的物理存储器的数量。对于块类型分区，`<factor>` 可指定要在每个物理存储器中存储的来自原始阵列的元素数量。

  重要提示！对于 `complete` 类型的分区，则不应指定 `<factor>`。

- `<dimension>`：指定要分区的阵列维度。指定为范围介于 1 到 <N> 之间的整数。Vitis 核开发套件支持 <N> 维阵列，可在任何单一维度上对阵列进行分区。

示例 1

使用块映射将含 17 个元素的 8 位阵列 AB[17] 重塑（分区并映射）为含 5 个元素的全新 32 位阵列。

```
int AB[17] __attribute__((xcl_array_reshape(block,4,1)));
```

提示：`<factor>` 为 4 表示该阵列应分割为 4 个阵列。因此，17 个元素重塑为 1 个含 5 个元素的阵列，位宽增加至原来的 4 倍。在此案例中，最后一个元素 AB[17] 映射到第五个元素的下 8 位，第五个元素的其余部分则为空。

示例 2

将二维阵列 AB[6][4] 重塑为新阵列（维度为 [6][2]），其中维度 2 位宽增至原来的 2 倍：

```
int AB[6][4] __attribute__((xcl_array_reshape(block,2,2)));
```

示例 3

将函数 `foo` 中的三维 8 位阵列 AB[4][2][2] 重塑为全新的单元素阵列（寄存器），位宽为 128 位 (4×2×2×8)：

```
int AB[4][2][2] __attribute__((xcl_array_reshape(complete,0)));
```

提示：<dimension> 为 0 表示重塑该阵列的所有维度。

另请参阅

- xcl_array_partition
- pragma HLS array_reshape
- Vitis HLS 流程

xcl_dataflow

描述

启用任务级流水打拍，允许函数和循环在其操作过程中重叠，增加 RTL 实现的并发度，并增加设计的整体吞吐量。

在 C 语言描述中，所有操作均按顺序执行。如无限制资源的任何指令（如 `pragma HLS allocation`），则 Vitis HLS 工具会尝试最大限度减小时延并提高并发。但是，数据依赖关系可能对此施加限制。例如，访问阵列的函数或循环必须先完成对阵列的所有读写访问后才能完成操作。这样会阻止下一个使用该数据的函数或循环开始操作。数据流最优化支持函数或循环中的操作在前一个函数或循环尚未完成其所有操作时就开始操作。

当指定数据流最优化后，HLS 工具会分析顺序函数或循环之间的数据流，并基于乒乓 RAM 或 FIFO 创建通道，以允许使用者 (consumer) 函数或循环在生产者 (producer) 函数或循环完成前即开始操作。这样即可允许函数或循环并行操作，从而降低时延，并提升 RTL 吞吐量。

如果未指定启动时间间隔（任一函数或循环与下一个函数或循环的开始时间间隔的周期数），那么 HLS 工具会工具尝试最大限度减小启动时间间隔，且数据一旦可用即开始操作。

提示：HLS 工具可提供数据流配置设置。`config_dataflow` 命令可用于指定数据流最优化中使用的默认存储器通道和 FIFO 深度。

要使 DATAFLOW 最优化正常工作，数据必须逐一流经设计中的每个任务。以下编码样式会阻止 HLS 工具执行 DATAFLOW 最优化：

- 单一生产者使用者违例
- 绕过任务
- 任务间的反馈
- 任务的有条件执行
- 含多个退出条件的循环

重要提示！如果存在上述任一编码样式，则 HLS 工具会发出一条消息，且不会执行 DATAFLOW 最优化。

最后，DATAFLOW 最优化不含任何分层实现。如果子函数或循环包含可能受益于 DATAFLOW 最优化的其它任务，那么必须对该循环、子函数或子函数的内联应用 DATAFLOW 最优化。

语法

在函数定义或循环定义之前，对 XCL_DATAFLOW 属性进行赋值：

```
__attribute__((xcl_dataflow))
```

示例

指定数据流最优化在函数 `foo` 内执行。

```
__attribute__((xcl_dataflow))
void foo ( a, b, c, d ) {
...
}
```

另请参阅

- pragma HLS dataflow
- Vitis HLS 流程

xcl_latency

描述

XCL_LATENCY 属性可指定最小时延值和/或最大时延值，用于完成函数、循环和区域。时延定义为生成输出所需的时钟周期数。函数或区域时延是代码计算所有输出值并返回所需的时钟周期数。循环时延是执行所有循环迭代的周期数。请参阅《Vitis 高层次综合用户指南》(UG1399) 的性能指标示例。

Vitis HLS 工具始终尝试最大程度降低设计中的时延。指定 XCL_LATENCY 属性后，工具行为如下所示：

- 当时延大于最小值时，或者小于最大值时：满足约束。不再执行进一步最优化。
- 当时延小于最小值时：如果 HLS 工具可以实现小于最小指定值的时延，那么它可将时延扩展至指定值，这样可能增加共享。
- 当时延大于最大值时：如果 HLS 工具无法调度到最大限值范围内，那么它会尽力实现指定约束。如果仍无法满足最大时延，则会发出警告，并以超出最大限值前提下可实现的最小时延来生成设计。

提示：您还可以使用 XCL_LATENCY 属性来限制该工具寻找最优解决方案的范围。在代码中指定时延的约束范围（循环、函数或区域），减少此范围内的可能解决方案数量，并改善工具运行时间。如需了解更多信息，请参阅《Vitis 高层次综合用户指南》(UG1399) 的改善综合运行时间和容量。

语法

将 XCL_LATENCY 属性分配到函数、循环或区域主体之前：

```
__attribute__((xcl_latency(min, max)))
```

其中：

- <min>：指定代码的函数、循环或区域的最小时延。
- <max>：指定代码的函数、循环或区域的最大时延。

示例

将 `test` 函数中的 `for` 循环指定为最小时延为 4，最大时延为 8。

```
__kernel void test(__global float *A, __global float *B, __global float *C,
int id)
{
  for (unsigned int i = 0; i < id; i++)
__attribute__((xcl_latency(4, 12))) {
   C[id] = A[id] * B[id];
 }
}
```

另请参阅

- pragma HLS latency
- Vitis HLS 流程

xcl_loop_tripcount

描述

XCL_LOOP_TRIPCOUNT 属性可应用于循环，以指定循环所执行的迭代总数。

重要提示！XCL_LOOP_TRIPCOUNT 属性仅用于分析，不影响综合结果。

Vivado 高层次综合 (HLS) 可报告每个循环的总时延，即执行循环的所有迭代的时钟周期数。因此，循环时延即为循环迭代次数（或循环次数）的函数。

循环次数可为常量值。它取决于循环表达式（例如，`x<y`）中使用的变量值或循环内使用的控制语句。在某些情况下，HLS 工具无法判定循环次数，因此时延未知。在此类情况下，用于判定循环次数的变量可能是：

- 输入实参，或者
- 采用动态运算计算所得的变量。

如果循环时延未知或者无法计算，那么 XCL_LOOP_TRIPCOUNT 属性会要求您指定循环迭代次数的最小、最大和平均值。这样该工具即可分析循环时延给报告中的设计总时延造成的影响，并帮助您为设计决定合适的最优化措施。

语法

将该属性置于 OpenCL 源代码中循环声明前的位置。

```
__attribute__((xcl_loop_tripcount(<min>, <max>, <average>)))
```

其中：

- <min>：指定循环迭代次数的最小值。
- <max>：指定循环迭代次数的最大值。
- <avg>：指定循环迭代次数的平均值。

示例

在此示例中，`f` 函数中的 WHILE 循环指定的循环次数最小值为 2，最大值为 64，平均值为 33。

```
__kernel void f(__global int *a) {
unsigned i = 0;
__attribute__((xcl_loop_tripcount(2, 64, 33)))
  while(i < 64) {
    a[i] = i;
    i++;
  }
}
```

另请参阅

- pragma HLS occurence
- Vitis HLS 流程

xcl_max_work_group_size

描述

如果要指定大小超过 4K 的大型内核，请使用该选项代替 REQD_WORK_GROUP_SIZE。

增大 Vitis 核开发套件中通过 `reqd_work_group_size` 属性所支持的工作组默认最大大小。Vitis 核开发套件支持大小超过 4096 的工作组通过 XCL_MAX_WORK_GROUP_SIZE 属性。

注释：实际工作组大小限制取决于针对平台所选的赛灵思器件。

语法

将该属性置于内核定义之前或针对内核指定的主函数之前的位置：

```
__attribute__((xcl_max_work_group_size(<X>, <Y>, <Z>)))
```

其中：

- <X>, <Y>, <Z>：指定内核的 ND 范围。这表示三维矩阵的每个维度，用于指定内核的工作组大小。

示例

以下是未经最优化的加法器的内核源代码。针对此设计未指定任何属性，仅指定工作组大小等于矩阵大小（例如，64x64）。即，迭代整个工作组将完整添加输入矩阵 a 和 b 并输出结果。这三者全都是全局整数指针，这意味着矩阵中每个值均为 4 个字节，并存储在片外 DDR 全局存储器中。

```
#define RANK 64
__kernel __attribute__ ((reqd_work_group_size(RANK, RANK, 1)))
void madd(__global int* a, __global int* b, __global int* output) {
int index = get_local_id(1)*get_local_size(0) + get_local_id(0);
output[index] = a[index] + b[index];
}
```

由此生成的本地工作组大小为 (64, 64, 1)，与全局工作组大小相同。此设置创建的工作组总计大小为 4096。

注释：这是 Vitis 核开发套件使用标准 OpenCL 属性 REQD_WORK_GROUP_SIZE 可支持的最大工作组大小。Vitis 核开发套件可通过赛灵思的 `xcl_max_work_group_size` 属性支持大小超过 4096 的工作组。

大于 64x64 的任何矩阵都只需使用一个维度即可定义整个工作组大小。即，一个 128x128 矩阵可通过含大小为 (128, 1, 1) 的工作组的内核来执行运算，每次调用都会对数据的整个行或列进行运算。

另请参阅

- https://www.khronos.org/
- 《OpenCL C 语言规范》

xcl_pipeline_loop

描述

您可以通过对循环进行流水打拍来缩小时延，并最大程度提升内核吞吐量和性能。

虽然展开循环可以增加并发，但它无法解决始终将所有元素都保留在内核数据路径内的问题。即使在展开循环的情况下，循环控制依赖关系仍可能导致顺序行为。顺序操作行为会导致硬件空闲并损失性能。

赛灵思解决此问题的方法是在 OpenCL 2.0 API 规范的基础上引入矢量扩展，通过使用 XCL_PIPELINE_LOOP 属性开展循环流水打拍。

默认情况下，`v++` 编译器会自动对循环进行流水打拍（循环次数大于 64），或者展开循环（循环次数小于 64）。这样应可提供良好的结果。但是，您可以选择通过在循环之前显式指定 NOUNROLL 属性和 XCL_PIPELINE_LOOP 属性来选择循环流水打拍（而不是自动展开）方法。

语法

将该属性置于 OpenCL 源代码中循环定义前的位置：

```
__attribute__((xcl_pipeline_loop(<II_number>)))
```

其中：

- <II_number>：指定期望的流水线启动时间间隔 (II)。Vitis HLS 工具会尝试满足此请求；但基于数据依赖关系，循环可能采用更大的启动时间间隔。如果不指定 II，则默认值为 1。

示例

以下示例为指定函数中的 `for` 循环指定的 II 目标为 3：

```
__kernel void f(__global int *a) {
  __attribute__((xcl_pipeline_loop(3)))
  for (unsigned i = 0; i < 64; ++i)
    a[i] = i;
}
```

另请参阅

- pragma HLS pipeline
- Vitis HLS 流程

xcl_pipeline_workitems

描述

对工作项进行流水打拍，以改善时延和吞吐量。工作项流水打拍是对内核工作组进行循环流水打拍的一项扩展操作。其必要性在于它可以最大程度提升内核吞吐量和性能。

语法

将该属性置于 OpenCL API 源代码中位于要流水打拍的元素之前的位置：

```
__attribute__((xcl_pipeline_workitems))
```

示例 1

为了处理以下属性中的 `reqd_work_group_size` 属性，Vitis 技术自动插入循环嵌套以处理 ND 范围 (3,1,1) 的三维特征。由于添加的循环嵌套，导致此内核的执行方式类似于未流水打拍的循环。添加 XCL_PIPELINE_WORKITEMS 属性能够添加并发性，并改善代码的吞吐量。

```
kernel
__attribute__ ((reqd_work_group_size(3,1,1)))
void foo(...)
{
...
__attribute__((xcl_pipeline_workitems)) {
int tid = get_global_id(0);
op_Read(tid);
op_Compute(tid);
op_Write(tid);
}
...
}
```

示例 2

以下示例为内核的相应元素添加了工作项流水打拍：

```
__kernel __attribute__ ((reqd_work_group_size(8, 8, 1)))
void madd(__global int* a, __global int* b, __global int* output)
{
int rank = get_local_size(0);
__local unsigned int bufa[64];
__local unsigned int bufb[64];
__attribute__((xcl_pipeline_workitems)) {
int x = get_local_id(0);
int y = get_local_id(1);
bufa[x*rank + y] = a[x*rank + y];
bufb[x*rank + y] = b[x*rank + y];
}
barrier(CLK_LOCAL_MEM_FENCE);
__attribute__((xcl_pipeline_workitems)) {
int index = get_local_id(1)*rank + get_local_id(0);
output[index] = bufa[index] + bufb[index];
}
}
```

另请参阅

- pragma HLS pipeline
- Vitis HLS 流程

xcl_reqd_pipe_depth

描述

重要提示！管道必须以小写字母数字来声明。在管道中，也不支持将 `printf()` 与变量搭配使用。

OpenCL 框架 2.0 规范引入了一种新的存储器对象，称为管道。管道以先进先出 (FIFO) 方式来存储数据。管道可用于在 FPGA 中的不同内核之间执行数据流传输，不使用外部存储器，这样可以显著缩短总体系统时延。

在 Vitis 核开发套件中，管道必须在所有内核函数外部进行静态定义。在管道声明中，管道深度必须使用 XCL_REQD_PIPE_DEPTH 属性来指定。

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)));
```

在非阻塞模式下，管道只能使用标准 OpenCL `read_pipe()` 和 `write_pipe()` 内置函数来访问，在阻塞模式下，管道必须使用赛灵思的 `read_pipe_block()` 和 `write_pipe_block()` 扩展函数来访问。

重要提示！Vitis HLS 仅支持通过阻塞模式来读取和写入管道。不支持非阻塞 read_pipe/write_block 函数。

重要提示！在不同内核中，一个给定的管道可以有且只能有一个生产者和消费者。

管道对象无法从主机 CPU 进行访问。可以使用 OpenCL `get_pipe_num_packets()` 和 `get_pipe_max_packets()` 内置函数来查询管道的状态。如需了解有关这些内置函数的详细信息，请参阅来自 Khronos OpenCL 工作组的《OpenCL C 语言规范》。

语法

该属性必须在声明管道对象时赋值：

```
pipe int <id> __attribute__((xcl_reqd_pipe_depth(<n>)));
```

其中：

- <id>：指定管道的标识符，必须由小写字母数字组成。例如，<infifo1>，不得使用 <inFifo1>。
- <n>：指定管道深度。有效值为 16、32、64、128、256、512、1024、2048、4096、8192、16384 和 32768。

示例

以下是来自赛灵思 GitHub 的 `dataflow_pipes_ocl` 示例，此示例使用管道以及 `read_pipe_block()` 和 `write_pipe_block()` 阻塞函数将数据从一个处理阶段传递到下一个处理阶段：

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel : Read Data from Global Memory and write into Pipe P0
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
```

```
{
__attribute__((xcl_pipeline_loop))
mem_rd: for (int i = 0 ; i < size ; i++)
{
//blocking Write command to pipe P0
write_pipe_block(p0, &input[i]);
}
}
// Adder Stage Kernel: Read Input data from Pipe P0 and write the result
// into Pipe P1
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void adder_stage(int inc, int size)
{
__attribute__((xcl_pipeline_loop))
execute: for(int i = 0 ; i < size ; i++)
{
int input_data, output_data;
//blocking read command to Pipe P0
read_pipe_block(p0, &input_data);
output_data = input_data + inc;
//blocking write command to Pipe P1
write_pipe_block(p1, &output_data);
}
}
// Output Stage Kernel: Read result from Pipe P1 and write the result to
// Global Memory
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
__attribute__((xcl_pipeline_loop))
mem_wr: for (int i = 0 ; i < size ; i++)
{
//blocking read command to Pipe P1
read_pipe_block(p1, &output[i]);
}
}
```

另请参阅

- https://www.khronos.org/
- 《OpenCL C 语言规范》

xcl_zero_global_work_offset

描述

如果您使用 `clEnqueueNDRangeKernel` 并将 `global_work_offset` 设置为 NULL 或全部为 0，请使用该属性告知编译器，`global_work_offset` 始终为 0。

如果您具有如下存储器访问权，那么该属性可提升存储器性能：

```
A[get_global_id(x)] = ...;
```

注释：您可同时指定 REQD_WORK_GROUP_SIZE、VEC_TYPE_HINT 和 XCL_ZERO_GLOBAL_WORK_OFFSET 来最大程度提升性能。

语法

将该属性置于内核定义或针对内核指定的主函数之前的位置。

```
__kernel __attribute__((xcl_zero_global_work_offset))
void test (__global short *input, __global short *output, __constant short
*constants) { }
```

另请参阅

- reqd_work_group_size
- vec_type_hint
- clEnqueueNDRangeKernel

XILINX

第 52 章

移植到新的目标平台

此移植内容主要面向需要将其 Vitis™ 加速技术应用从一个目标平台移植到另一个目标平台的用户。例如，将应用从 Alveo™ U200 数据中心加速器卡移植到 Alveo U280 卡。

此内容包含下列章节：

- 设计移植：设计移植进程概述，包括 FPGA 器件物理方面的信息。
- 移植版本：使用新版本时对主机代码和设计约束进行的任何更改。
- 修改内核布局：控制内核布局和 DDR 接口连接。
- 满足时序要求：新目标平台中的时序问题，此类问题可能需要通过其它选项才能满足性能要求。

设计移植

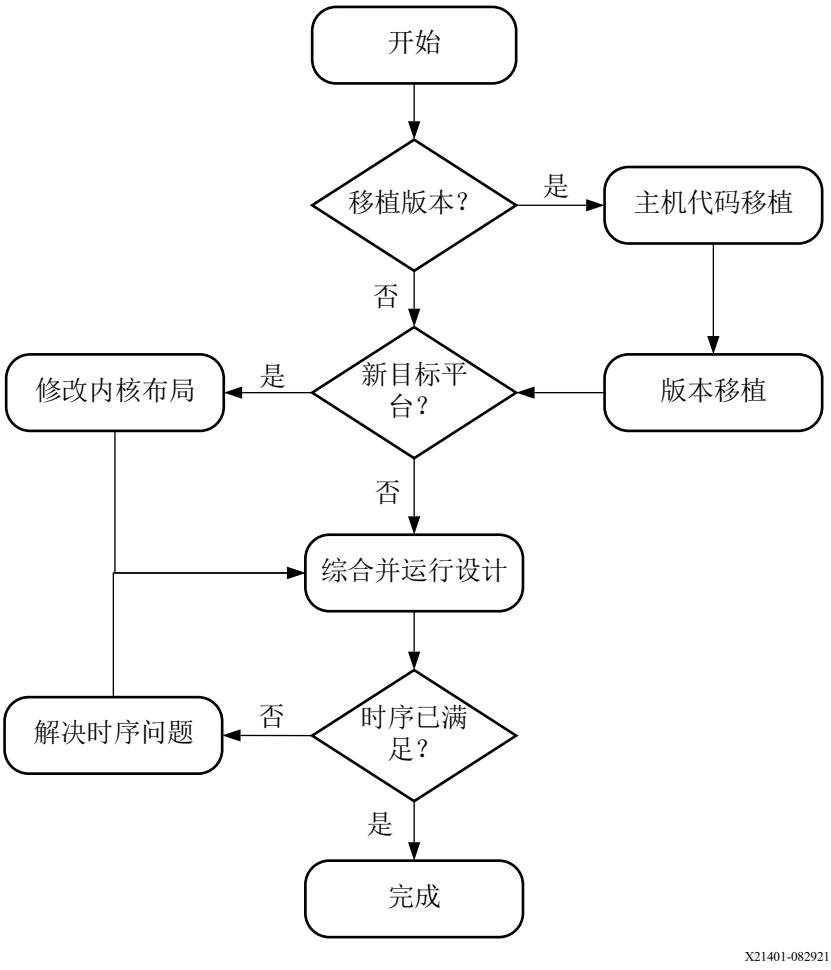
将某一目标平台中实现的应用移植到另一个目标平台上时，重要的是了解两个目标平台之间的差异以及这些差异对于设计所产生的影响。

关键考虑因素：

- 发布的版本较前版本是否有变化？
- 新目标平台是否包含不同的目标平台？
- 内核是否需要在超级逻辑区域 (SLR) 中重新分配？
- 设计是否满足新平台所需的频率（时序）性能？

下图总结了本指南中描述的移植流程以及移植过程中要考虑的主题。

图 134：**目标平台移植流程图**



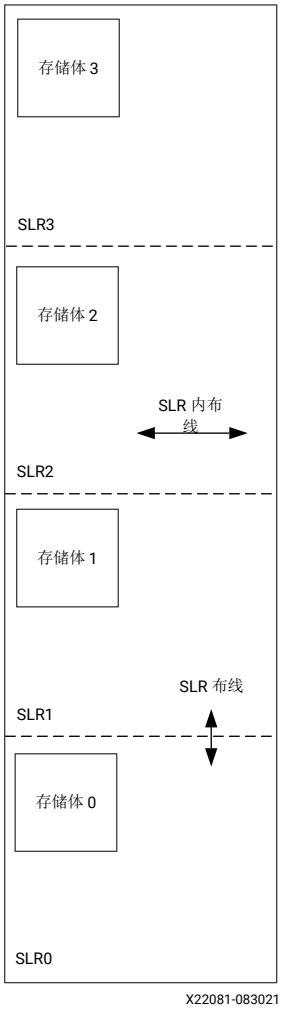
重要提示！在开始设计移植之前，了解 FPGA 和目标平台的架构非常重要。

理解 FPGA 架构

在将任何设计移植到新的目标平台之前，您应该对 FPGA 架构有基本的了解。下图显示了赛灵思 FPGA 器件的布局规划。要理解的概念是：

- SSI 器件
- SLR
- SLR 的布线资源
- 存储器接口

图 135：**具有四个 SLR 区域的赛灵思 FPGA 的物理视图**



提示：上面显示的 FPGA 布局规划适用于具有四个 SLR 的 SSI 器件，其中每个 SLR 包含一个 DDR 存储器接口。

堆叠硅片互联 (SSI) 器件

在 SSI 器件中，有多个硅裸片经由硅片互联连接到一起并且封装到单个器件中。SSI 器件通过提供更多数量的连接来实现多个裸片之间的高带宽连接。与多 FPGA 或多芯片模块方法相比，它还具有更低的时延和功耗，同时可在单个封装内集成大量互连逻辑、收发器和片上资源。如需了解有关 SSI 器件优势的详细信息，请参阅《赛灵思堆叠硅片互连 (SSIT) 提供突破性 FPGA 容量、带宽和电源效率》(WP380)。

超级逻辑区域 (SLR)

SLR 是 SSI 器件内包含的单一 FPGA 裸片切片 (slice)。多个 SLR 组件经组装构成单一 SSI 器件。每个 SLR 都包含大多数赛灵思 FPGA 器件通用的有源电路。该电路包括大量的：

- LUT
- 寄存器
- I/O 组件

- 千兆位收发器
- 块存储器
- DSP 块

在任一 SLR 内可实现一个或多个内核。如果需要，单一内核可跨多个 SLR 进行布局。

SLR 的布线资源

FPGA 上实现的定制硬件通过片上布线资源来连接。SSI 器件中有两种类型的布线资源：

- SLR 内部资源：SLR 内部布线资源是用于连接硬件逻辑的快速资源。Vitis 技术在实现内核时，会自动使用最优化的资源来连接硬件元件。
- 超长线路 (SLL) 资源：SLL 是在 SLR 之间运行的布线资源，用于在不同区域之间连接逻辑。这些布线资源比 SLR 内部布线更慢。但是，如果内核与其连接的 DDR 布局在不同的 SLR 内，那么 Vitis 技术会自动实现专用硬件，以使用 SLL 布线资源，而不会对性能产生任何影响。如需了解有关管理布局的更多信息，请参阅 修改内核布局。

存储器接口

每个 SLR 都包含一个或多个存储器接口。这些存储器接口用于连接 DDR 存储器，在内核执行之前，主机缓冲器中的数据会被复制到此 DDR 存储器中。每个内核会从该 DDR 存储器读取数据，并将结果重新写入该 DDR 存储器。存储器接口连接到 FPGA 上的管脚，此接口还包含存储器控制器逻辑。

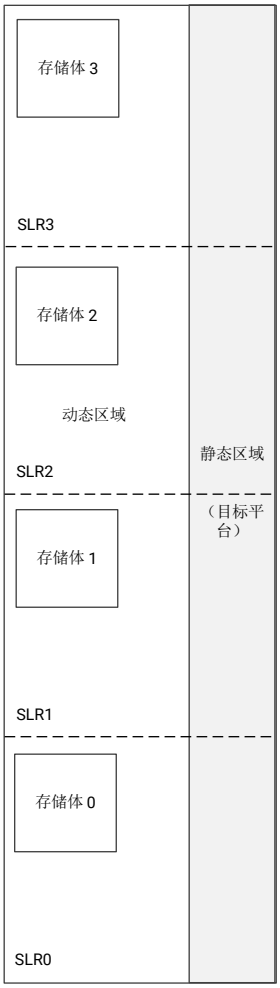
了解目标平台

在 Vitis 技术中，目标平台是在添加任何定制逻辑或加速器之前实现到 FPGA 上的硬件设计。目标平台可定义 FPGA 的属性，它由两个区域组成：

- 静态区域，包含内核和器件管理逻辑。
- 动态区域，加速内核的定制逻辑即布局在此区域中。

下图显示了已应用目标平台的 FPGA。

图 136：**含 4 个 SLR 区域的 FPGA 上的目标平台**



此处目标平台为无法修改的静态区域，其中包含操作 FPGA 所需的逻辑，以及在平台与动态区域之间进行数据传输所需的逻辑。静态区域（上图中显示为灰色）可能存在于单一 SLR 内，或者也可能跨多个 SLR（如上例所示）。静态区域包含：

- DDR 存储器接口控制器
- PCIe® 接口逻辑
- XDMA 逻辑
- 防火墙逻辑等

动态区域为上图中的白色区域。此区域包含目标平台的所有可重配置组件，并且所有加速器内核即布局在此区域内。

由于静态区域会耗用器件上的部分可用硬件资源，因此要在动态区域中实现的定制逻辑只能使用剩余资源。如上例所示，目标平台定义为 FPGA 上的全部 4 个 DDR 存储器接口均可供使用。这将需要在 DDR 接口中使用存储器控制器的资源。

如需了解在每个目标平台的动态区域内有多少逻辑可供实现，请参阅 Vitis 软件平台版本说明。在 修改内核布局 中也对此话题进行了阐述。

移植版本

在移植到新目标平台之前，您还应确定是否需要将新平台设为使用不同版本的 Vitis 技术。如果您打算使用新版本，赛灵思强烈建议首先将现有平台设为使用新软件版本，以确认无需更改，然后再移植到新目标平台。

将现有平台设定为使用新版本时，需遵循两个步骤：

- 主机代码移植
- 版本移植

重要提示！在移植到新版本之前，赛灵思建议您查看 Vitis 软件平台版本说明。

主机代码移植

`XILINX_XRT` 环境变量用于指定 XRT 库环境的位置，必须在编译主机代码之前完成设置完成 XRT 库环境安装后，可通过查找相应的 `/opt/xilinx/xrt/setup.csh` 或 `/opt/xilinx/xrt/setup.sh` 文件来设置 `XILINX_XRT` 环境变量。其次，请确保 `LD_LIBRARY_PATH` 变量同样指向 XRT 库的安装区域。

要编译并运行主机代码，请从 Vitis 安装找到 `<INSTALL_DIR>/settings64.csh` 或 `<INSTALL_DIR>/settings64.sh` 文件。

如果您使用的是 GUI，那么它将自动整个新 XRT 库的位置，并在您构建工程时生成 `makefile`。

但如果您使用自己的定制 `makefile`，您必须使用 `XILINX_XRT` 环境变量来设置 XRT 库。

- 包含目录现在指定为 `-I${XILINX_XRT}/include` 和 `-I${XILINX_XRT}/include/CL`
- 库路径现在是：`-L${XILINX_XRT}/lib`
- OpenCL 库则是：`libxilinxopencl.so`，请使用 `makefile` 中的 `-lxilinxopencl`

版本移植

移植主机代码后，在现有目标平台上使用新版本的 Vitis 技术构建代码。验证您是否可在 Vitis 统一软件平台中使用新版本技术来运行工程，确保工程成功完成，并满足时序要求。

使用新版本时可能出现的问题包括：

- C 库或库文件发生更改。
- 内核路径名发生更改。
- 内核代码中嵌入的 HLS 编译指示或编译指示选项发生更改。
- C/C++/OpenCL 编译器支持发生更改。
- 内核性能发生更改：可能需要对现有内核代码中的编译指示进行调整。

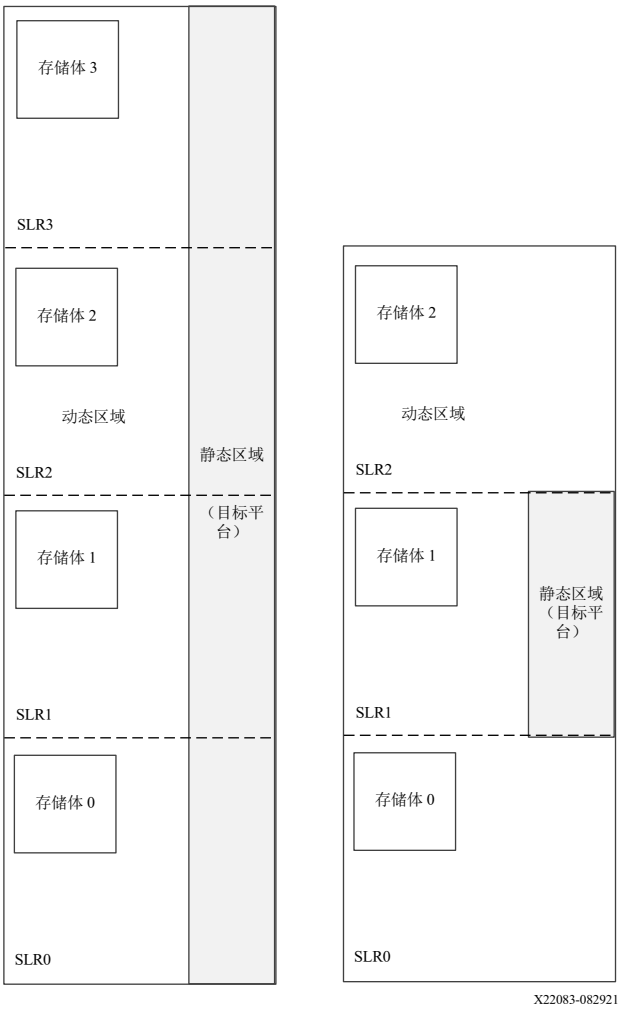
请使用在开发任何内核期间使用的相同技术来解决这些问题。在此阶段，确保使用新版本的目标平台的吞吐量性能满足您的要求。如果最终时序（最大时钟频率）发生更改，您可以在移植到新目标平台后解决这些问题。欲知详情，请参阅满足时序要求。

修改内核布局

以新平台为目标的主要问题在于如何确保现有内核布局能在新目标平台上正常工作。每个目标平台都有按静态区域定义的 FPGA。如下图所示，目标平台可能不尽相同。

- 左侧目标平台包含 4 个 SLR，静态区域分布在全部 4 个 SLR 中。
- 右侧目标平台则仅有 3 个 SLR，且静态区域完全包含在 SLR1。

图 137：**硬件平台的目标平台比较**



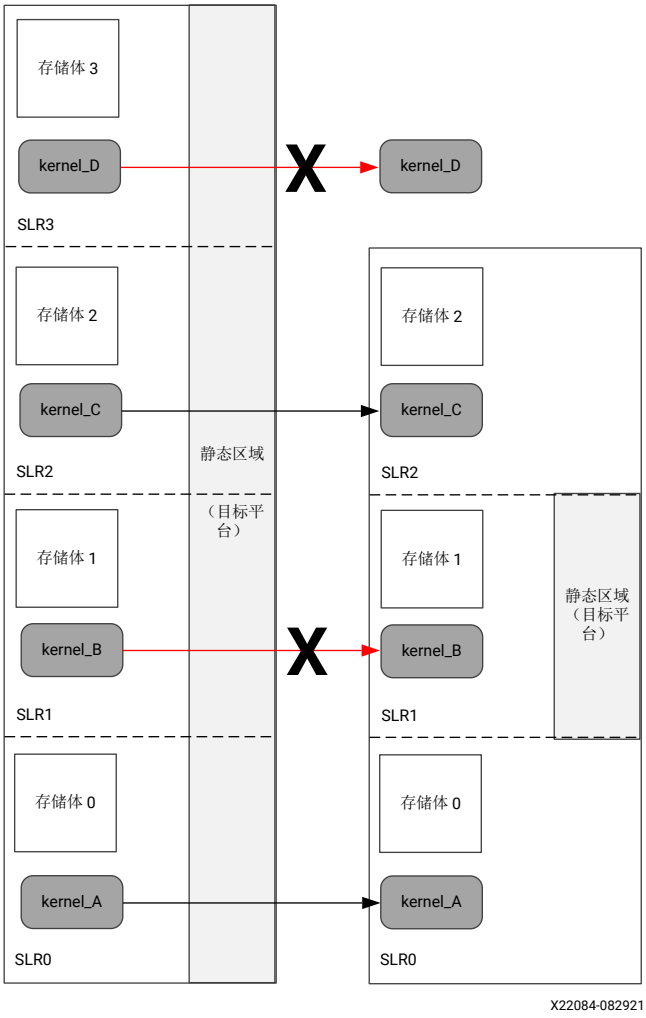
本节旨在介绍如何修改内核布局。

新硬件平台的含义

下图着重演示了移植到新目标平台时的内核布局问题。在下面的示例中：

- 现有内核 kernel_B 太大，无法适应新目标平台的 SLR2，因为大部分 SLR 都被静态区域占用。
- 必须将现有内核 kernel_D 重新定位到新 SLR，因为新目标平台不像现有平台一样拥有 4 个 SLR。

图 138：**移植平台 - 内核布局**



移植到新平台时，您需要执行以下操作：

- 了解新目标平台的每个 SLR 中可用的资源，如 Vitis 软件平台版本说明 所述。
- 了解设计中每个内核所需的资源。
- 使用 `v++ --config` 选项来指定每个内核所在的 SLR 以及每个内核连接的 DDR 存储体。欲知详情，请参阅 将计算单元分配给 SLR 和 将内核端口映射到存储器。

本章节的其余部分将介绍这些项目。

确定内核的布局位置

要确定内核的布局位置，需要以下两项信息：

- 硬件平台的每个 SLR 中的可用资源 (`.xsa`)。
- 每个内核所需的资源。

通过这两项信息，您将确定目标平台的每个 SLR 中可布局的一个或多个内核。

执行这些计算时请谨记，系统基础架构可以使用 10% 的可用资源：

- 如果内核必须跨 SLR 边界，则可以使用基础架构逻辑将内核连接到 DDR 接口。
- 在 FPGA 中，资源也用于信号布线。在 FPGA 中永远不可能使用 100% 的可用资源，因为信号布线也需要资源。

可用的 SLR 资源

在 Vitis 软件平台版本说明 中可找到各版本支持的各平台的每个 SLR 的可用资源。下表显示了目标平台示例。在此示例中：

- SLR 描述指明了哪个 SLR 包含静态区域和/或动态区域。
- 其中列出了每个 SLR 可用的资源（LUT、寄存器、RAM 等）。

这使您能够确定每个 SLR 中有哪些资源可用。

表 63：**硬件平台的 SLR 资源**

区域	SLR 0	SLR 1	SLR 2
SLR 描述	器件底部；专属于动态区域。	器件中间；由动态和静态区域资源共享。	器件顶部；专属于动态区域。
动态区域 Pblock 名称	pfa_top_i_dynamic_region_pblock_dynamic_SLR0	pfa_top_i_dynamic_region_pblock_dynamic_SLR1	pfa_top_i_dynamic_region_pblock_dynamic_SLR2
计算单元布局语法	set_property CONFIG.SLR_ASSIGNMENTS SLR0[get_bd_cells<cu_name>]	set_property CONFIG.SLR_ASSIGNMENTS SLR1[get_bd_cells<cu_name>]	set_property CONFIG.SLR_ASSIGNMENTS SLR2[get_bd_cells<cu_name>]
动态区域中可用的全局存储器资源			
存储器通道；系统端口名称	bank0 (16 GB DDR4)	bank1（16 GB DDR4，位于静态区域） bank2（16 GB DDR4，位于动态区域）	bank3 (16 GB DDR4)
动态区域中的大致可用互连结构资源			
CLB LUT	388K	199K	388K
CLB 寄存器	776K	399K	776K
块 RAM 拼块	720	420	720
UltraRAM	320	160	320
DSP	2280	1320	2280

内核资源

在“System Estimate”报告中可获取每个内核的资源。

完成硬件仿真或硬件运行后，在“Assistant”视图中会提供“System Estimate”报告。此报告的示例如下所示。

图 139：**系统估算报告**

```
Area Information
Compute Unit     Kernel Name    Module Name    FF    LUT   DSP  BRAM
---------------  -------------  -------------  ----  ----  ---  ----
smithwaterman_1  smithwaterman  smithwaterman  2925  4304  1    10
-------------------------------------------------------------------------
```

- FF 是指每个 SLR 的平台资源中记录的 CLB 寄存器。
- LUT 是指每个 SLR 的平台资源中记录的 CLB LUT。
- DSP 是指每个 SLR 的平台资源中记录的 DSP。
- BRAM 是指每个 SLR 的平台资源中记录的块 RAM tile。

此信息可帮助您确定每个内核的正确 SLR 分配。

将内核分配给 SLR

设计中的每个内核均可使用 `connectivity.slr` 选项分配给任一 SLR 区域，该选项可在为 `v++ --config` 命令行选项指定的配置文件中找到。如需了解更多信息，请参阅 将计算单元分配给 SLR。

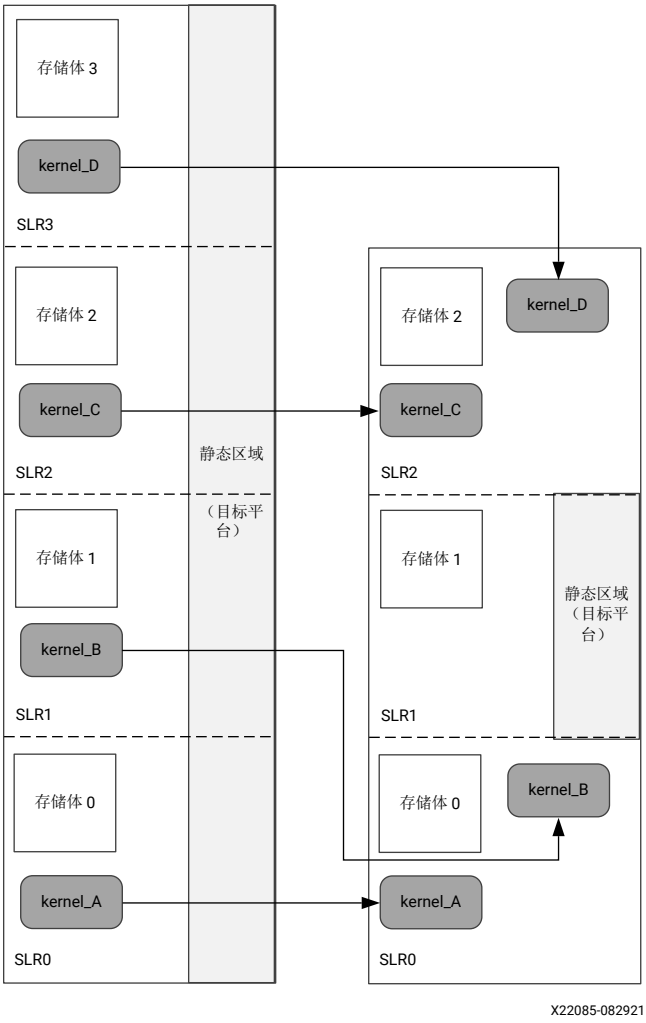
执行内核布局时，赛灵思建议使用 `connectivity.sp` 配置选项来分配内核将连接到的特定 DDR 存储体，如 将内核端口映射到存储器 中所述。

例如，下图显示了包含 4 个 SLR 的现有目标平台和包含 3 个 SLR 的新目标平台。这两个平台之间的静态区域构造方式也不同。在此移植示例中：

- Kernel_A 映射到 SLR0。
- Kernel_B 不再适合置于 SLR1 内，它被重新映射到具有可用资源的 SLR0。
- Kernel_C 映射到 SLR2。
- Kernel_D 被重新映射到具有可用资源 SLR2。

内核映射如下图所示。

图 140：**跨 SLR 的内核映射**



指定内核布局

对于以上示例，用于分配内核的配置文件如下所示：

```
[connectivity]
nk=kernel:4:kernel_A.lernel_B.kernel_C.kernel_D

slr=kernel_A:SLR0
slr=kernel_B:SLR0
slr=kernel_C:SLR2
slr=kernel_D:SLR2
```

用于对上图所示每个内核进行布局的 `v++` 命令行如下：

```
v++ -l --config config.cfg ...
```

指定内核 DDR 接口

在指定内核布局时，还应指定内核 DDR 存储器接口。指定 DDR 接口可确保对内核到其它 SLR 中的 DDR 接口间的连接进行自动流水打拍。这确保了时序不会降级，时序降级会降低最大时钟频率。

在此示例中，使用上图中的内核布局：

- Kernel_A 连接到存储体 0。
- Kernel_B 连接到存储体 1。
- Kernel_C 连接到存储体 2。
- Kernel_D 连接到存储体 1。

用于执行这些连接的配置文件如下所示，且该配置文件通过 `v++ --config` 命令来传递：

```
[connectivity]
nk=kernel:4:kernel_A.kernel_B.kernel_C.kernel_D

slr=kernel_A:SLR0
slr=kernel_B:SLR0
slr=kernel_C:SLR2
slr=kernel_D:SLR2

sp=kernel_A.arg1:DDR[0]
sp=kernel_B.arg1:DDR[1]
sp=kernel_C.arg1:DDR[2]
sp=kernel_D.arg1:DDR[1]
```

重要提示！ 使用 `connectivity.sp` 选项将内核端口分配给存储体时，必须映射内核的所有接口/端口。如需了解更多信息，请参阅 将内核端口映射到存储器。

满足时序要求

执行系统运行，如果完成且没有违例，则移植成功。

如果未满足时序要求，则可能需要指定一些定制约束以帮助满足时序要求。如需了解有关满足时序的信息，请参阅《UltraFast 设计方法时序收敛快捷参考指南》(UG1292)。

定制约束

用于内核布局规划、布局和时序的定制 Tcl 约束需要在新目标平台上下文 (`.xsa`) 中进行复查。例如，如果内核需要移植到新目标平台的不同 SLR 中，那么也需要修改该内核的布局约束。

通常，预计在基于 9P Virtex UltraScale 器件的不同目标平台之间的时序是可比较的。时序收敛的任何定制 Tcl 约束都将需要经过评估，并且可能需要针对新平台进行修改。

定制约束可以使用 `v++` 配置文件的 `[advanced]` 指令（通过 `--config` 选项来指定）传递到 Vivado® 工具。如需了解更多信息，请参阅 管理 Vivado 综合与实现结果。

时序收敛的考虑因素

跨 Vitis 版本或目标平台进行移植时，设计性能和时序收敛可能不尽相同，满足以下任一条件时差异尤为明显：

- 需通过布局规划约束来收敛时序。
- 器件或 SLR 资源利用率高于典型指标：
  - LUT 利用率高于 70%
  - DSP、RAMB 和 UltraRAM 利用率高于 80%
  - FD 利用率高于 50%
- 需要大量编译策略才能收敛时序。

利用率指标提供了一个阈值，如果高于该阈值，则设计编译耗时更长，或者性能可能低于初始估算值。对于通常需要使用多个 SLR 的大型设计，在验证任何布局规划约束的同时指定内核或 DDR 与 `v++ --config` 选项的关联（如 将内核端口映射到存储器 中所述）即可确保：

- 每个 SLR 的利用率都低于建议的指标。
- 如果某一种类型的硬件资源需高于指标，则在各 SLR 之间平衡利用率。

对于整体利用率较高的设计，以更高的时延为代价增加内核中的流水打拍量可以极大帮助实现时序收敛并实现更高的性能。

为便于快速复查以上列出的所有方面，请使用快速失败 (fail-fast) 报告，此报告是在整个 Vitis 应用加速开发流程中使用 `-R` 选项生成的，如下所述（欲知详情，请参阅 控制报告生成）：

- `v++ -R 1`
  - `report_failfast` 在每个内核综合步骤结束时运行
  - `report_failfast` 在对整个设计执行 `opt_design` 之后运行
  - `opt_design` DCP 已保存
- `v++ -R 2`
  - 所生成的报告与使用 `-R 1` 时相同，并且：
  - `report_failfast` 是每个 SLR 的后布局操作
  - 生成其它报告和中间 DCP

所有报告和 DCP 均可在实现目录中找到，包括内核综合报告：

```
<runDir>/_x/link/vivado/prj/prj.runs/impl_1
```

如需了解有关时序收敛和快速失败报告的更多信息，请参阅《UltraFast 设计方法指南（适用于赛灵思 FPGA 和 SoC）》(UG949)。

XILINX

第 53 章

旧版本参考信息

xbutil 实用工具 - 旧版本

提示：本节描述了旧版本的 `xbutil` 命令，此命令可通过调用 `xbutil --legacy` 命令来访问。

本赛灵思开发板实用工具 (`xbutil`) 是一种独立的命令行实用工具，包含在赛灵思的 Xilinx Runtime (XRT) 安装包内。`xbutil` 命令仅支持 Alveo 数据中心加速器卡上的平台和基于嵌入式处理器的平台。

加速器卡划分为用户函数和管理函数，以提供不同的卡访问级别。用户函数允许用户加载并运行其应用，而管理函数则供系统管理员用于对卡进行管理。`xbutil` 实用工具能够与用户函数进行交互。`xbmgmt` 实用工具则需要根用户权限，用于与管理函数进行交互。将函数访问拆分为两种实用工具的原因是为了给该工具的管理功能特性提供一定程度的安全性。

在卡上需安装并识别 XRT。对于 Vivado 流程中自定义的 Alveo 卡设置，则不使用 `xbutil`。它包含多项命令，用于确认和识别已安装的加速器卡，以及有关这些卡的其它详细信息，包括卡存储器、主机接口、目标平台名称和系统信息等。这些信息可用于卡管理和应用调试。

重要提示！虽然 `xbutil` 支持嵌入式处理器平台，对于这些平台，只能使用以下命令：`dump`、`help`、`program`（仅适用于 DFX 平台）、`query`、`scan` 和 `status`。

`xbutil` 命令行格式如下：

```
xbutil <command> [options]
```

以下提供了可用命令和选项。

- clock
- dmatest
- dump
- m2mtest
- mem --read
- mem --write
- p2p
- program
- query
- reset
- scan (xbutil)

- status
- top
- validate
- version

提示：您可使用 `help` 命令来列出可用的 `xbutil` 命令和选项：

```
xbutil help
```

clock

重要提示！该选项不适用于嵌入式处理器平台。

`clock` 命令允许您在 `xclbin` 内更改计算单元 (CU) 的一个或多个时钟的时钟频率。它采用如下命令行格式：

```
xbutil clock [-d card] [-r region] [-f clock1_freq_MHz]
[-g clock2_freq_MHz] [-h clock3_freq_MHz]
```

使用 `-f` 开关指定的时钟频率将应用于所有 CU。个别 CU 的时钟频率无法单独更改。此外，必须对 `xclbin` 进行编程，使其支持指定的时钟频率或者以指定的时钟频率运行。使用 Vitis 工具生成的 CU 仅含 clock1。基于 RTL 的内核则可连接 clock2 和 clock3。

下表中列出了可用选项。

表 64：**xbutil clock 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否
`-r <region>`	已弃用，无影响。在后续版本中将移除该选项。	否
`-f <clock1_freq_MHz>`	为第一个时钟指定时钟频率 (MHz)。所有平台均采用该时钟。	是
`-g <clock2_freq_MHz>`	为第二个时钟指定时钟频率 (MHz)。部分平台可能不含该时钟。	否
`-h <clock3_freq_MHz>`	为第三个时钟指定时钟频率 (MHz)。部分平台可能不含该时钟。	否

xclbinutil 实用工具 工具可用于列出可用的 `xclbin` 时钟。

在更改时钟频率前，需要先对 `xclbin` 进行编程。请参阅 program 以了解有关 `xclbin` 编程的信息。完成 `xclbin` 编程后，即可更改时钟频率。

例如，要将 card_ID = 0 中的 clock1 更改为 100 MHz，请运行以下命令：

```
xbutil clock -d 0 -f 100
```

同样，要更改 card_ID = 0 中的 2 个时钟（例如，将 clock1 设置为 200 MHz，将 clock2 设置为 250 MHz），请运行以下命令：

```
xbutil clock -d 0 -f 200 -g 250
```

以下示例是成功运行此命令后的输出：

```
INFO: Found total 1 card(s), 1 are usable
INFO: xbutil clock succeeded.
```

如果不对 `xclbin` 进行编程，则将显示如下消息。请在运行 `clock` 命令前对 `xclbin` 进行编程。

```
INFO: Found total 1 card(s), 1 are usable
WARNING: 'uuid' invalid, unable to find uuid.
Has the bitstream been loaded? See 'xbutil program'.
ERROR: xbutil clock failed.
```

dmatest

重要提示！该选项不适用于嵌入式处理器平台。

`dmatest` 命令用于确认卡存储器吞吐量，方法是在主机与指定卡上的全局存储器之间执行数据传输测试。`dmatest` 作为 validate 命令的一部分来运行。

它采用如下命令行格式：

```
xbutil dmatest [-d card] [-b [0x]block_size_KB]
```

下表中列出了可用选项。

表 65：**xbutil dmatest 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否
`-b blocksize`	指定测试块大小 (KB)。如不指定 `-b` 选项，块大小默认为 65536 (KB)。块大小必须为 2 次幂值。 块大小可以用十进制或十六进制格式指定。例如，`-b 1024` 和 `-b 0x400` 都可以设置块的大小为 1024 KB。	否

在运行 `dmatest` 前，需要对 `xclbin` 进行编程。请参阅 program 以了解有关 `xclbin` 编程的信息。

`dmatest` 命令仅在编程到卡的 `xclbin` 所访问的 DDR 存储体或 HBM 伪通道 (PC) 上执行吞吐量测试。

以下示例显示了在 U200 上通过 `xclbin` 使用 DDR 存储体 0、1、2 和 3 所生成的命令输出：

```
INFO: Found total 1 card(s), 1 are usable
Total DDR size: 65536 MB
Reporting from mem_topology:
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 11341.5 MB/s
```

```
Host <- PCIe <- FPGA read bandwidth = 11097.3 MB/s
Data Validity & DMA Test on bank1
Host -> PCIe -> FPGA write bandwidth = 11414.6 MB/s
Host <- PCIe <- FPGA read bandwidth = 10981.7 MB/s
Data Validity & DMA Test on bank2
Host -> PCIe -> FPGA write bandwidth = 11345.1 MB/s
Host <- PCIe <- FPGA read bandwidth = 11189.2 MB/s
Data Validity & DMA Test on bank3
Host -> PCIe -> FPGA write bandwidth = 11121.7 MB/s
Host <- PCIe <- FPGA read bandwidth = 11375.7 MB/s
INFO: xbutil dmatest succeeded.
```

同样，以下示例显示了在 U50 上通过 `xclbin` 使用 HBM 端口 0、1、2 和 3 所生成的命令输出：

```
INFO: Found total 1 card(s), 1 are usable
Total DDR size: 0 MB
Reporting from mem_topology:
Data Validity & DMA Test on HBM[0]
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 11950.9 MB/s
Host <- PCIe <- FPGA read bandwidth = 11940.3 MB/s
Data Validity & DMA Test on HBM[1]
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 11947 MB/s
Host <- PCIe <- FPGA read bandwidth = 11958.1 MB/s
Data Validity & DMA Test on HBM[2]
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 12077.2 MB/s
Host <- PCIe <- FPGA read bandwidth = 12064.1 MB/s
Data Validity & DMA Test on HBM[3]
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 11989.5 MB/s
Host <- PCIe <- FPGA read bandwidth = 11976 MB/s
INFO: xbutil dmatest succeeded.
```

如果不对 `xclbin` 进行编程，则将显示如下消息。

```
INFO: Found total 1 card(s), 1 are usable
'uuid' invalid, please re-program xclbin.
```

dump

`dump` 命令可返回有关卡和系统的广泛信息，包括操作系统、XRT、开发板、电子设备、散热以及 `xclbin`（JSON 格式）以支持脚本编制流程。对此命令执行更改后，将落实输出格式和内容，并且输出格式和内容均向后兼容。

它采用如下命令行格式：

```
xbutil dump [-d card]
```

下表中列出了可用选项。

表 66：**xbutil dump 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否

命令输出示例如下所示：

```
{
    "version": "1.1.0",
    "system": {
        "sysname": "Linux",
        "release": "4.15.0-74-generic",
        "version": "#84-Ubuntu SMP Thu Dec 19 08:06:28 UTC 2019",
        "machine": "x86_64",
        "glibc": "2.27",
        "linux": "Ubuntu 18.04.3 LTS",
        "cores": "48",
        "memory": "31812",
        "model": "Precision 7920 Tower",
        "now": "Mon Jan 13 15:57:59 2020"
    },
    "runtime": {
        "build": {
            "version": "2.4.19",
            "hash": "be6279809c82b3b6abfd6a6baed6343bd4bda232",
            "date": "2020-01-09 10:57:59",
            "branch": "2019.2_PU1",
            "xocl": "2.4.19,be6279809c82b3b6abfd6a6baed6343bd4bda232",
            "xclmgmt": "2.4.19,be6279809c82b3b6abfd6a6baed6343bd4bda232"
        }
    },
    "board": {
        "info": {
            "dsa_name": "xilinx_u250_xdma_201830_2",
            "vendor": "0x10ee",
            "device": "0x5005",
            "subdevice": "0x000e",
            "subvendor": "0x10ee",
            "xmcversion": "2019107",
            "xmc_oem_id": "0x0",
            "serial_number": "21320493802N",
            "max_power": "225W",
            "sc_version": "4.2.0",
            "ddr_size": "68719476736",
            "ddr_count": "4",
            "clock0": "250",
            "clock1": "500",
            "clock2": "0",
            "pcie_speed": "3",
            "pcie_width": "16",
            "dma_threads": "2",
            "mig_calibrated": "true",
            "idcode": "0x4b57093",
            "fpga_name": "xcu250-figd2104-2L-e",
            "dna": "",
            "p2p_enabled": "0"
        },
```

```
        "physical": {
            "thermal": {
                "pcb": {
                    "top_front": "51",
                    "top_rear": "41",
                    "btm_front": "50"
                },
                "fpga_temp": "53",
                "tcrit_temp": "51",
                "fan_presence": "A",
                "fan_speed": "1262",
                "cage": {
                    "temp0": "0",
                    "temp1": "0",
                    "temp2": "0",
                    "temp3": "0"
                }
            },
            "electrical": {
                "12v_pex": {
                    "voltage": "12100",
                    "current": "1480"
                },
                "12v_aux": {
                    "voltage": "12121",
                    "current": "1369"
                },
                "3v3_pex": {
                    "voltage": "3349",
                    "current": "0"
                },
                "3v3_aux": {
                    "voltage": "3316"
                },
                "ddr_vpp_bottom": {
                    "voltage": "2500"
                },
                "ddr_vpp_top": {
                    "voltage": "2500"
                },
                "sys_5v5": {
                    "voltage": "5492"
                },
                "1v2_top": {
                    "voltage": "1207"
                },
                "1v2_btm": {
                    "voltage": "1199"
                },
                "1v8": {
                    "voltage": "1824"
                },
                "0v85": {
                    "voltage": "856",
                    "current": "0"
                },
                "mgt_0v9": {
                    "voltage": "908"
                },
                "12v_sw": {
                    "voltage": "12038"
                },
                "mgt_vtt": {
```

```
                    "voltage": "1203"
                },
                "vccint": {
                    "voltage": "850",
                    "current": "16668"
                },
                "vcc3v3": {
                    "voltage": "0"
                },
                "hbm_1v2": {
                    "voltage": "0"
                },
                "vpp2v5": {
                    "voltage": "0"
                },
                "vccint_bram": {
                    "voltage": "0"
                }
            },
            "power": "34"
        },
        "error": {
            "firewall": {
                "firewall_level": "0",
                "firewall_status": "0",
                "firewall_time": "0",
                "status": "(GOOD)"
            }
        },
        "pcie_dma": {
            "transfer_metrics": {
                "chan": {
                    "0": {
                        "h2c": "6240 MB",
                        "c2h": "12160 MB"
                    },
                    "1": {
                        "h2c": "6240 MB",
                        "c2h": "6144 MB"
                    }
                }
            }
        },
        "memory": {
            "mem": {
                "0": {
                    "ecc_status": "(None)",
                    "ecc_ce_cnt": "0",
                    "ecc_ue_cnt": "0",
                    "ecc_ce_ffa": "0",
                    "ecc_ue_ffa": "0",
                    "type": "MEM_DDR4",
                    "temp": "41",
                    "tag": "bank0",
                    "enabled": "true",
                    "size": "16 GB",
                    "mem_usage": "0 Byte",
                    "bo_count": "0"
                },
                "1": {
                    "ecc_status": "(None)",
                    "ecc_ce_cnt": "0",
                    "ecc_ue_cnt": "0",
```

```
                    "ecc_ce_ffa": "0",
                    "ecc_ue_ffa": "0",
                    "type": "MEM_DDR4",
                    "temp": "41",
                    "tag": "bank1",
                    "enabled": "true",
                    "size": "16 GB",
                    "mem_usage": "0 Byte",
                    "bo_count": "0"
                },
                "2": {
                    "ecc_status": "(None)",
                    "ecc_ce_cnt": "0",
                    "ecc_ue_cnt": "0",
                    "ecc_ce_ffa": "0",
                    "ecc_ue_ffa": "0",
                    "type": "MEM_DDR4",
                    "temp": "54",
                    "tag": "bank2",
                    "enabled": "true",
                    "size": "16 GB",
                    "mem_usage": "0 Byte",
                    "bo_count": "0"
                },
                "3": {
                    "ecc_status": "(None)",
                    "ecc_ce_cnt": "0",
                    "ecc_ue_cnt": "0",
                    "ecc_ce_ffa": "0",
                    "ecc_ue_ffa": "0",
                    "type": "MEM_DDR4",
                    "temp": "48",
                    "tag": "bank3",
                    "enabled": "true",
                    "size": "16 GB",
                    "mem_usage": "0 Byte",
                    "bo_count": "0"
                },
                "4": {
                    "type": "**UNUSED**",
                    "temp": "0",
                    "tag": "PLRAM[0]",
                    "enabled": "false",
                    "size": "128 KB",
                    "mem_usage": "0 Byte",
                    "bo_count": "0"
                },
                "5": {
                    "type": "**UNUSED**",
                    "temp": "0",
                    "tag": "PLRAM[1]",
                    "enabled": "false",
                    "size": "128 KB",
                    "mem_usage": "0 Byte",
                    "bo_count": "0"
                },
                "6": {
                    "type": "**UNUSED**",
                    "temp": "0",
                    "tag": "PLRAM[2]",
                    "enabled": "false",
                    "size": "128 KB",
                    "mem_usage": "0 Byte",
```

```
                    "bo_count": "0"
                },
                "7": {
                    "type": "**UNUSED**",
                    "temp": "0",
                    "tag": "PLRAM[3]",
                    "enabled": "false",
                    "size": "128 KB",
                    "mem_usage": "0 Byte",
                    "bo_count": "0"
                }
            }
        },
        "xclbin": {
            "uuid": "c5b9a584-9b70-4902-ae32-addf5c1c6e0c"
        },
        "compute_unit": {
            "0": {
                "name": "bandwidth1:kernel_1",
                "base_address": "25165824",
                "status": "(IDLE)"
            },
            "1": {
                "name": "bandwidth2:kernel_2",
                "base_address": "25231360",
                "status": "(IDLE)"
            }
        }
    },
    "debug_profile": {
        "device_info": {
            "error": "0",
            "device_index": "0",
            "user_instance": "129",
            "nifd_instance": "0",
            "device_name": "\/dev\/dri\/renderD129",
            "nifd_name": "\/dev\/nifd0"
        }
    }
}
```


如果提供的卡索引无效，将显示以下消息：

```
ERROR: Card index 1 is out of range
```

m2mtest

重要提示！该选项不适用于嵌入式处理器平台。

`m2mtest` 命令在指定卡上的两个器件存储体之间执行吞吐量数据传输测试。请注意，仅限支持 M2M 功能的平台才能运行此命令，请参阅《Alveo 数据中心加速器卡平台用户指南》(UG1120) 了解详情。此外，在使用 2 个存储体的卡上需要先下载 `xclbin`，然后才能运行 `m2mtest`，否则，运行此命令会返回错误。`m2mtest` 命令只能对已下载到卡上的 `xclbin` 所访问的存储体执行吞吐量测试。

它采用如下命令行格式：

```
xbutil m2mtest [-d card]
```

下表中列出了可用选项。

表 67：**xbutil m2mtest 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否

以下示例显示了使用 DDR 组 0、1、2 和 3 的 `xclbin` 命令的输出结果：

```
INFO: Found total 2 card(s), 2 are usable
bank0 -> bank1 M2M bandwidth: 12050.5 MB/s
bank0 -> bank2 M2M bandwidth: 12074.3 MB/s
bank0 -> bank3 M2M bandwidth: 12082.9 MB/s
bank1 -> bank2 M2M bandwidth: 12061.8 MB/s
bank1 -> bank3 M2M bandwidth: 12105.2 MB/s
bank2 -> bank3 M2M bandwidth: 12065.8 MB/s
INFO: xbutil m2mtest succeeded.
```

如果未加载 `xclbin`，则将显示以下错误消息：

```
'uuid' invalid, please re-program xclbin.
```

如果在不支持 M2M 功能的平台上运行此命令，则将显示以下错误：

```
M2M is not available. Skipping validation
ERROR: xbutil m2mtest failed.
```

mem --read

重要提示！该选项不适用于嵌入式处理器平台。

`mem --read` 命令可读取从指定存储器地址开始的指定数量的字节，并将内容写入输出文件。

它采用如下命令行格式：

```
xbutil mem --read [-d card] [-a [0x]start_addr]
[-i size_bytes] [-o output filename]
```

下表中列出了可用选项。

表 68：**xbutil mem --read 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否

表 68：**xbutil mem --read 命令选项** (续)

选项	描述	是否必需
`-a <start_addr>`	指定有效起始地址，可采用十六进制格式或十进制格式。十六进制格式需前置 `0x`，即，`0x100`。默认地址为 `0x0`。 有效地址可使用 Linux `dmesg` 命令来获取，如下所述。	否
`-i <size_bytes>`	指定十六进制格式或十进制格式的存储器传输大小（以字节为单位）。十六进制格式需前置 `0x`，即，`0x100`。默认大小为 `0x20000`。	否
`-o <output_file_name>`	输出文件名。如果不指定输出文件名，那么默认输出文件为 `memread.out`。	否

以下显示的是使用以下命令搭配 `xclbin` 并使用 DDR 存储体 0、1、2 和 3 所生成的输出示例。

```
xbutil mem --read -a 0x0 -d2 -i 0x10
```

```
INFO: Found total 3 card(s), 3 are usable
INFO: Reading from single bank, 256 bytes from DDR/HBM/PLRAM address
0x4000000000
INFO: Read size 0x100 B. Total Read so far 0x100
INFO: Read data saved in file: memread.out; Num of bytes: 256 bytes
INFO: xbutil mem succeeded.
```

以下提供了使用以上命令生成的文件的内容示例。其中使用了 Linux 十六进制转储命令 `xxd` 来显示此文件。

```
00000000: 3d3d 3d3d 5354 4152 5420 6f66 2044 4452  ====START of DDR
00000010: 2044 6174 613d 3d3d 3d3d 3d3d 0a00   Data==========..
00000020: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000030: 0a3d 3d3d 3d3d 454e 4420 6f66 2044 4452  .=====END of DDR
00000040: 2044 6174 613d 3d3d 3d3d 3d3d 0a00   Data==========..
```

如果使用的起始地址无效，则会返回以下错误。起始地址必须在器件的地址空间范围内。在此示例中，`0x400` 为无效的起始地址：

```
ERROR: Start address 0x400 is not valid
Available memory banks:
ERROR: xbutil mem failed.
```

提示：请使用 grep 来显示可用的地址空间。例如，以下命令可显示 DDR 存储器基址：

```
dmesg | grep -A 10 -i ddr
```

Linux `dmesg` 输出将提供各种 DDR 存储器的基址。DDR[1] 输出样本如下所示：

```
[23174.283512] xocl 0000:a6:00.1: xocl_init_mem: Memory Bank: DDR[1]
[23174.283514] xocl 0000:a6:00.1: xocl_init_mem: Base Address:0x8000000000
[23174.283515] xocl 0000:a6:00.1: xocl_init_mem: Size:0x400000000
```

将以上 `-i ddr` 搜索项替换为 `-i hbm` 即可查找 HBM 存储器的基址。

要编写已知字节模式，请参阅 mem --write。

mem --write

重要提示！该选项不适用于嵌入式处理器平台。

`mem --write` 命令用于将已定义的模式写入一组指定的存储器位置。

它采用如下命令行格式：

```
xbutil mem --write [-d card] [-a [0x]start_addr]
[-i size_bytes] [-e pattern_byte]
```

下表中列出了可用选项。

表 69：**xbutil mem --write 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否
`-a <start_addr>`	指定有效起始地址，可采用十六进制格式或十进制格式。十六进制格式需前置 `0x`，即，`0x100`。默认地址为 `0x0`。 有效地址可使用 Linux `dmesg` 命令来获取，如下所述。	否
`-i <size_bytes>`	指定十六进制格式或十进制格式的存储器传输大小（以字节为单位）。十六进制格式需前置 `0x`，即，`0x100`。	否
`-e <pattern>`	指定写入所有已定义的字节位置的字节模式，可采用十六进制 (hex) 格式或十进制 (decimal) 格式。十六进制格式需前置 `0x`，即，`0xEF`。	否

以下显示的是使用以下命令搭配 `xclbin` 并使用 DDR 存储体 0、1、2 和 3 所生成的输出示例。

```
xbutil mem --write -a 0x0 -d2 -i 0x10 -e 0xef
```

```
INFO: Found total 1 card(s), 1 are usable
INFO: Writing to single bank, 16 bytes from DDR/HBM/PLRAM address 0x0
INFO: Writing DDR/HBM/PLRAM with 16 bytes of pattern: 0xef from address 0x0
INFO: xbutil mem succeeded.
```

如果使用的起始地址无效，则会返回以下错误。起始地址必须在器件的地址空间范围内。在此示例中，`0x400` 为无效的起始地址。

```
ERROR: Start address 0x400 is not valid
Available memory banks:
ERROR: xbutil mem failed.
```

提示：请使用 grep 来显示可用的地址空间。例如，以下命令可显示 DDR 存储器基址：

```
dmesg | grep -A 10 -i ddr
```

Linux `dmesg` 输出将提供各种 DDR 存储器的基址。DDR[1] 输出样本如下所示：

```
[23174.283512] xocl 0000:a6:00.1: xocl_init_mem: Memory Bank: DDR[1]
[23174.283514] xocl 0000:a6:00.1: xocl_init_mem: Base Address:0x8000000000
[23174.283515] xocl 0000:a6:00.1: xocl_init_mem: Size:0x400000000
```

将以上 `-i ddr` 搜索项替换为 `-i hbm` 即可查找 HBM 存储器的基址。

要读取存储器地址，请参阅 mem --read。

p2p

重要提示！该选项不适用于嵌入式处理器平台。

`p2p` 命令用于启用/禁用 PCIe 点对点支持功能特性并检查当前配置。

重要提示！仅限支持 P2P 功能的平台才能运行此命令，请参阅《Alveo 数据中心加速器卡平台用户指南》(UG1120) 了解详情。

P2P 配置在热启动过程中持续不变。

注释：启用或禁用 P2P 需根用户权限。

它采用如下命令行格式：

```
xbutil p2p [-d card] --[enable | disable | validate]
```

下表中列出了可用选项。

表 70：**xbutil p2p 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否
`--enable`	启用 `p2p` 选项 `--enable`、`--disable` 和 `--validate` 为互斥。命令中只能包含其中任一选项。 如果针对 P2P 返回的 `xbutil query` 状态为 `no_iomem`，则表示需要热重启。	--enable， --disable， --validate 互斥
`--disable`	启用 `p2p` 选项 `--enable`、`--disable` 和 `--validate` 为互斥。命令中只能包含其中任一选项。 可能需要热重启才能完全禁用。	--enable， --disable， --validate 互斥
`--validate`	确认 `p2p` 功能。 选项 `--enable`、`--disable` 和 `--validate` 为互斥。命令中只能包含其中任一选项。 如果需要热重启，请在热重启后运行。	--enable， --disable， --validate 互斥

使用 `xbutil query` 来显示 P2P 的当前状态。以下是 `xbutil query` 命令的部分输出，其中在“P2P Enabled”标题下显示了当前状态。

```
PCIe             DMA chan(bidir) MIG Calibrated  P2P Enabled
GEN 3x16         2               true            false
```

表 71：**P2P 启用后返回的值定义**

值	注解
`true`	P2P 已启用。
`false`	P2P 已禁用。
`no_iomem`	P2P 在器件中已启用，但系统无法分配 I/O 存储器，需热重启。

如果尝试确认时未启用 P2P，那么将跳过确认并显示以下消息：

```
P2P BAR is not enabled. Skipping validation
```

如果卡平台不支持 P2P，则将显示以下消息：

```
ERROR: P2P is not supported on this platform
```

如果未对 `xclbin` 进行编程，则将显示以下消息：

```
'uuid' invalid, please re-program xclbin.
```

program

重要提示！该选项支持用于嵌入式处理器 DFX 平台。

`program` 命令用于将指定 `xclbin` 二进制文件下载到卡上的可编程区域。

它采用如下命令行格式：

```
xbutil program [-d card] [-r region] -p <xclbin_filename>
```

下表中列出了可用选项。

表 72：**xbutil program 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否
`-r <region>`	已弃用，无影响。在后续版本中将移除该选项。	否
`-p <xclbin_filename>`	指定要下载到卡的 `xclbin` 二进制文件的文件名。	是

当 xclbin 已成功下载到卡上后，就会显示以下消息：

```
INFO: Found total 1 card(s), 1 are usable
INFO: xbutil program succeeded.
```

如果后续使用的 xbutil 程序使用相同的 xbutil_filename，那么将不会下载 xbutil，因为它在卡上已存在，但仍将显示上述相同的消息。

如果指定的 xclbin 文件不存在，则将显示以下消息：

```
ERROR: Cannot open <my_xclbin>.xclbin. Check that it exists and is readable.
ERROR: xbutil program failed.
```

query

重要提示！该选项不适用于嵌入式处理器平台。

query 命令会以人类可读格式返回有关卡状态的详细信息。请参阅 dump 以查看 JSON 格式的输出。

它采用如下命令行格式：

```
xbutil query [-d card [-r region]]
```

下表中列出了可用选项。

表 73：**xbutil query 命令选项**

选项	描述	是否必需
-d <card>	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** xbutil scan 命令可用于显示已安装的卡的 card_id 和 BDF。	否
-r <region>	已弃用，无影响。在后续版本中将移除该选项。	否

此输出会返回大量信息。输出示例如下所示。为了更好地描述内容，将输出划分为几个单独的部分。

系统配置

表 74：**系统配置 (System Configuration) 字段定义**

字段	描述
OS Name	机器上运行的操作系统 (OS) 的名称
Release	OS 发行版号
Version	OS 版本
Machine	基于 CPU 的架构
Glibc	已安装的 GLIBC 版本
Distribution	分发版
Now	当前日期和时间

系统配置示例如下所示：

```
System Configuration
OS name:     Linux
Release:     4.15.0-74-generic
Version:     #83~16.04.1-Ubuntu SMP Wed Dec 18 04:56:23 UTC 2019
Machine:     x86_64
Glibc:       2.23
Distribution:   Ubuntu 16.04.6 LTS
Now:         Wed Jan 22 15:30:36 2020
```

XRT 信息

表 75：**XRT 字段定义**

字段	描述
Version	XRT 版本
Git Hash	关联的 GIT 散列
Git Branch	关联的 GIT 分支
Build Date	XRT 构建日期
XOCL	XOCL 版本
XCLMGMT	XCLMGMT 版本

```
XRT Information
Version:     2.3.1301
Git Hash:    192e706aea53163a04c574f9b3fe9ed76b6ca471
Git Branch: 2019.2
Build Date: 2019-10-24 20:04:29
XOCL:        2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471
XCLMGMT:     2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471
```

卡平台 (Shell) 信息

表 76：**卡平台 (Shell) 字段定义**

字段	描述
Shell	卡上安装的平台
FPGA	FPGA 名称
IDCode	平台的 ID 代码
Vendor	供应商 ID
Device	器件 ID
SubDevice	子器件 ID
SubVendor	子供应商 ID
SerNum	卡的唯一序列号
DDR Size	卡上可用的 DDR RAM 总量（以 GB 为单位）
DDR Count	卡上已安装的 DDR DIMM 总数
Clock0	Clock0 频率（以 MHz 为单位）
Clock1	Clock1 频率（以 MHz 为单位）

表 76：**卡平台 (Shell) 字段定义** (续)

字段	描述
Clock2	Clock2 频率（以 MHz 为单位）
PCIe	训练后的 PCIe 链路状态
DMA chan(bidir)	卡上的 DMA 通道数
MIG Calibrated	MIG 为 TRUE 表示已校准，FALSE 表示 MIG 尚未校准。
P2P Enabled	返回 P2P 的状态。状态为以下值之一： · true：已启用 P2P · false：已禁用 P2P · no_iomem：P2P 在器件中已启用，但系统无法分配 I/O 存储器，需热重启
OEM ID	OEM 所使用的 ID
Interface UUID	此唯一标识可用于判定包含平台的各分区的部分比特流彼此之间是否存在逻辑兼容且物理兼容关系。
Logic UUID	此唯一标识可用于判定包含平台的各分区的部分比特流彼此之间是否存在逻辑兼容且物理兼容关系。

卡信息输出示例如下所示：

```
Shell                          FPGA                                IDCode
xilinx_u50_gen3x16_xdma_201920_3                                   0x14b77093
Vendor         Device          SubDevice       SubVendor           SerNum
0x10ee         0x5021          0x000e          0x10ee              00501201A030
DDR size       DDR count       Clock0          Clock1              Clock2
0 Byte         0               250             500                 450
PCIe           DMA chan(bidir) MIG Calibrated  P2P Enabled         OEM ID
GEN 3x16       2               true            false               0x0
Interface UUID
862c7020a250293e32036f19956669e5
Logic UUID
f465b0a3ae8c64f619bc150384ace69b
DNA
```

温度

下文提供了卡的功耗和散热信息。其中报告的温度以摄氏度为单位。

表 77：**温度字段定义**

字段	描述
PCB Top Front	位于 PCB 顶层前部的温度（摄氏度）
PCB Top Rear	位于 PCB 顶层后部的温度（摄氏度）
PCB BTM Front	位于 PCB 底层前部的温度（摄氏度）
FPGA Temp	FPGA 核温度（摄氏度）
TCRIT Temp	风扇控制器的临界温度（以摄氏度为单位）。该字段针对有源卡和无源卡均存在。
Fan Presence	表示卡上是否存在风扇。 · A - 主动散热。卡上存在风扇。 · P - 被动散热。卡上不存在风扇，必须由主机服务器进行散热。
Fan Speed	风扇速度 (RPM)。针对无源卡返回 N/A。
QSFP 0,1,2,3	QSFP 模块温度（摄氏度）

温度输出示例如下所示：

```
Temperature(C)
PCB TOP FRONT    PCB TOP REAR     PCB BTM FRONT
42               37               42
FPGA TEMP        TCRIT Temp       FAN Presence     FAN Speed(RPM)
44               42               A                1108
QSFP 0           QSFP 1           QSFP 2           QSFP 3
0                0                0                0
```

电气

提供各种电压 (mV) 和电流 (mA) 读数。

表 78：**电气字段定义**

字段	描述
12V PEX	电压测量（12V 额定）值，来自 PCIe 连接器的 12V 电源。
12V AUX	电压测量（12V 额定）值，来自 12V 的 6 针或 8 针 PCIe 辅助电源线。
12V PEX Current	PCIe 连接器汲取的电流测量值。
12V AUX Current	6 针或 8 针 PCIe 辅助电源线汲取的电流测量值。
3V3 PEX	PCIe 连接器的 3.3V 电源的电压测量（3.3V 额定）值。
3V3 AUX	PCIe 连接器供电的 3.3V 辅助电源的电压测量（3.3V 额定）值。
DDR VPP BOTTOM	电压测量（2.5V 额定）值，用于为卡的下半部分 DDR4 器件的板载 VPP 供电。
DDR VPP TOP	电压测量（2.5V 额定）值，用于为卡的上半部分 DDR4 器件的板载 VPP 供电。
SYS 5V5	电压测量（5.5V 额定）值，用于为板载 VCC_INT 调节器供电。仅在 U2XX 卡上可用。
1V2 TOP	电压测量（1.2V 额定）值，用于为卡的上半部分 DDR4 器件的板载 VDD 供电。
1V8 TOP	电压测量（1.8V 额定）值，用于为 FPGA 所使用的板载 VCCAUX、VCCAUXIO 和 MGTAVVCAUX 调节器供电。
0V85	FPGA 所使用的板载 VCCINTIO 和 VCCBRAM 调节器的电压测量（0.85V 额定）值。
MGT 0V9	FPGA 所使用的 GTY 的板载 MGTAVCC 调节器的电压测量（0.9V 额定）值。
12V SW	12V 的 6 针或 8 针 PCIe 辅助电源线的电压测量（12V 额定）值。
MGT VTT	FPGA 所使用的 GTY 的板载 MGTAVTT 调节器的电压测量（1.2V 额定）值。
1V2 BTM	电压测量（1.2V 额定）值，用于为卡的下半部分 DDR4 器件的板载 VDD 调节器供电。
VCCINT VOL	FPGA 的板载 VCCINT 调节器的电压测量（0.72-0.85V 额定）值。
VCCINT CURR	卡汲取的 VCCINT 电源的电流测量值。
VCCINT BRAM VOL	FPGA 所使用的板载 VCCINT、VCCINTIO 和 VCCBRAM 调节器的电压测量（0.85V 额定）值。
VCC3V3 VOL	QSFP 和其它电路所使用的板载 3.3V 调节器的电压测量（3.3V 额定）值。
3V3 PEX CURR	卡汲取的 3.3V 主板 PCIe 电源轨的电流测量值。
VCC0V85 CURR	卡汲取的 VCCINTIO 和 VCCBRAM 电源的电流测量值。
HBM1V2 VOL	1.2V 电压值，用于为卡上的 DDR4 HBM 器件的板载 VDD 供电。
VPP2V5 VOL	2.5V 电压值，用于为卡上的 DDR4 器件的板载 VPP 供电。

输出示例如下所示：

```
Electrical(mV|mA)
12V PEX          12V AUX          12V PEX Current 12V AUX Current
12101            12202            1505            1268
3V3 PEX          3V3 AUX          DDR VPP BOTTOM  DDR VPP TOP
3357             3326             2500            2500
SYS 5V5          1V2 TOP          1V8 TOP         0V85
5515             1204             1836            855
MGT 0V9          12V SW           MGT VTT         1V2 BTM
910              12064            1207            1209
VCCINT VOL       VCCINT CURR      VCCINT BRAM VOL VCC3V3 VOL
851              15894            0               0
3V3 PEX CURR     VCC0V85 CURR     HBM1V2 VOL      VPP2V5 VOL
0                0                0               0
```

卡的电源

该字段可返回卡所耗用的总功耗 (W)。

输出示例如下所示：

```
Card Power(W)
33
```

防火墙最后错误状态

当在硬件中检测到错误时，防火墙会提供信息。其中包括时间戳和防火墙等级。防火墙有 3 个等级，如 AXI 防火墙脱扣 中所述。在以下输出中，未检测到任何防火墙错误。

表 79：**防火墙最后错误状态字段定义**

字段	描述
Tag	存储体 (memory bank) 名称
Errors	指示是否发生错误
CE Count	可纠正错误的数量。 此数值可长久保留，但可通过 `xbmgmt reset` 来复位。
UE Count	不可纠正错误的数量。此计数可长久保留，但可使用 `xbmgmt reset` 来复位。

输出示例如下所示：

```
Firewall Last Error Status
Level 0 : 0x0(GOOD)

ECC Error Status
Tag     Errors      CE Count  UE Count  CE FFA  UE FFA
bank0   (None)      0         0         0x0     0x0
bank1   (None)      0         0         0x0     0x0
bank2   (None)      0         0         0x0     0x0
bank3   (None)      0         0         0x0     0x0
```

在某些卡（例如，U50）上，卫星控制器 (SC) 会监控运行状况。如果卡超出电气或散热限值，SC 将对卡上的工作负载进行复位。在某些情况下，在 `xbutil query` 输出中，将此视为防火墙脱扣。它将显示发生脱扣的时间。脱扣后的状态示例如下所示：

```
Firewall Last Error Status
Level 3 : 0x80004(RECS_CONTINUOUS_RTRANSFERS_MAX_WAIT|
RECS_WRITE_TO_BVALID_MAX_WAIT)
Error occurred on: Tue 2020-04-28 15:16:47 MDT
```

在此情况下，应可正常使用该卡。

存储器状态

下文提供了存储器拓扑结构和 DMA 传输指标，随后提供了数据流传输信息。DMA 指标包括主机和卡之间的数据传输。主机到卡的传输由 `h2c` 表示，而卡到主机的传输由 `c2h` 定义。

输出示例如下所示。如果未加载 `xclbin`，则将不显示存储器状态。

```
Memory Status
      Tag          Type         Temp(C)  Size     Mem Usage        BO count
[ 0] bank0        MEM_DDR4     37       16 GB    16 MB            1
[ 1] bank1        MEM_DDR4     39       16 GB    16 MB            1
[ 2] bank2        MEM_DDR4     47       16 GB    16 MB            1
[ 3] bank3        MEM_DDR4     43       16 GB    16 MB            1
[ 4] PLRAM[0]     **UNUSED**   N/A      128 KB   0 Byte           0
[ 5] PLRAM[1]     **UNUSED**   N/A      128 KB   0 Byte           0
[ 6] PLRAM[2]     **UNUSED**   N/A      128 KB   0 Byte           0
```

DMA 传输指标

输出示例如下所示。如果未加载 `xclbin`，则将不显示任何指标。

```
DMA Transfer Metrics
Chan[0].h2c:  12384 MB
Chan[0].c2h:  15200 MB
Chan[1].h2c:  6240 MB
Chan[1].c2h:  6144 MB
```

数据流传输

仅可用于数据流传输平台。

输出示例如下所示：

```
Streams
Tag Flow ID Route ID Status Total (B/#) Pending (B/#)
```

Xclbin UUID

用于显示 `xclbin` UUID。输出示例如下所示。如果未加载 `xclbin`，返回的 UUID 全部为 0。

```
Xclbin UUID
dfd5a66a-36aa-41c6-88bb-c85a86d15512
```

计算单元状态

显示已加载到卡上的 `xclbin` 中存在的计算单元 (CU)。对于每个 CU 来说，显示名称、PCIe BAR 地址和状态，状态包括 IDLE、START 和 DONE。下面的输出显示的是 `xclbin` ID 和两个处于 IDLE 状态的 CU。

输出示例如下所示。如果未加载 `xclbin`，则不显示任何 CU 状态。

```
Compute Unit Status
CU[ 1]: bandwidth1:kernel_1          @0x1c00000          (IDLE)
CU[ 0]: bandwidth2:kernel_2          @0x1800000          (IDLE)
```

reset

重要提示！该选项不适用于嵌入式处理器平台。

`reset` 命令能够复位卡上的可编程区域。该区域内的所有正在运行的 CU 均停止并复位。

它采用如下命令行格式：

```
xbutil reset [-d card]
```

下表中列出了可用选项。

表 80：**xbutil reset 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否

以下示例显示了成功运行此命令后的输出：

```
All existing processes will be killed.
Are you sure you wish to proceed? [y/n]: y
```

scan (xbutil)

`xbutil scan` 命令可返回详细的系统信息，包括：

- 系统配置详细信息
- XRT 信息
- 系统中已安装的所有卡的列表（处于 GOLDEN 状态的卡除外）。

它采用如下命令行格式，且没有任何选项。

```
xbutil scan
```

下表列出了 `xbutil scan` 命令的各部分返回的字段。

表 81：**系统配置 (System Configuration) 字段定义**

字段	描述
OS Name	机器上运行的操作系统 (OS) 的名称
Release	OS 发行版号
Version	OS 版本
Machine	基于 CPU 的架构
Model	机器型号
CPU Cores	机器上的 CPU 核数
Memory	机器上安装的存储器总量（以 MB）
Glibc	已安装的 GLIBC 版本
分发版	分发版
Now	当前日期和时间

表 82：**XRT 字段定义**

字段	描述
Version	XRT 版本
Git Hash	关联的 GIT 散列
Git Branch	关联的 GIT 分支
Build Date	XRT 构建日期
XOCL	XOCL 版本
XCLMGMT	XCLMGMT 版本

此外还将返回系统上已安装的每张卡的列表。每张卡单独显示一行。单卡输出示例如下。它提供了多个字段用于详述已安装的卡。这些字段以空格来分隔。

```
[0] 0000:65:00.1 xilinx_u50_gen3x16_xdma_201920_3 user(inst=128)
```

下表中定义了这些字段。

表 83：**已安装的卡字段定义**

字段	描述
[card_id]	基于驱动程序数据结构中的卡枚举，提供已分配的 card_id。针对检测到的每张卡分配唯一的 card_id。冷重启或热重启后顺序可能发生更改。
Bus : Device : Function	为已安装的每张卡提供枚举的卡 Bus:Device:Function (BDF)。它采用如下格式： [Bus : Device : Function] 卡的 BDF 是根据它插入的 PCIe 插槽来确定。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。
Platform name	平台名称，格式如下： `<company>_<card>_<customization>_<major_release>_<minor_release>` 如需了解平台命名信息，请参阅《Alveo 数据中心加速器卡平台用户指南》(UG1120)。

表 83：**已安装的卡字段定义** (续)

字段	描述
user (inst = <value>)	返回与卡关联的用户函数实例值。实例值使您能够轻松找到每个函数的器件节点。 在 Linux 操作系统中，可在 `/dev/dri/renderD<inst>` 中找到器件节点 此外，将 `dmesg` 信息映射到指定卡时，实例也很有用。

以下是 `xbutil scan` 输出示例。系统配置和 XRT 信息部分首先显示，随后显示检测到的卡。在以下示例中，检测到一张卡，并为其分配 card_ID 0，BDF 为 `0000:65:00.1`。卡上烧写并运行的平台为 `xilinx_u50_gen3x16_xdma_201920_3`，并且已为用户实例赋值 128。

```
INFO: Found total 1 card(s), 1 are usable
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
System Configuration
OS name:     Linux
Release:     4.15.0-96-generic
Version:     #97~16.04.1-Ubuntu SMP Wed Apr 1 03:03:31 UTC 2020
Machine:     x86_64
Model:       Super Server
CPU cores:   16
Memory:      15703 MB
Glibc:       2.23
Distribution:   Ubuntu 16.04.6 LTS
Now:         Tue Apr 14 21:08:05 2020
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
XRT Information
Version:     2.5.309
Git Hash:    9a03790c11f066a5597b133db737cf4683ad84c8
Git Branch: 2019.2_PU2
Build Date: 2020-02-23 18:51:37
XOCL:        2.5.309,9a03790c11f066a5597b133db737cf4683ad84c8
XCLMGMT:     2.5.309,9a03790c11f066a5597b133db737cf4683ad84c8
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 [0] 0000:65:00.1 xilinx_u50_gen3x16_xdma_201920_3 user(inst=128)
```

如果安装有多张卡，则将展开检测到的卡的列表。在以下显示的示例中，检测到 3 张卡，并分别为其分配 card_ID 0、1 和 2。

```
[0] 0000:a6:00.1 xilinx_u280_xdma_201920_2 user(inst=130)
[1] 0000:73:00.1 xilinx_u250_xdma_201830_2 user(inst=129)
[2] 0000:17:00.1 xilinx_u200_xdma_201830_2 user(inst=128)
```

列出的卡前带有星号表示该卡尚未准备就绪。针对不可用的卡提供的消息示例如下：

```
*[0] 0000:a6:00.1 xilinx_u280_xdma_201920_2(ts=0x5e172e16) user(inst=130)
WARNING: card(s) marked by '*' are not ready, run xbmgmt flash --scan --
verbose to further check the details.
```

注释：处于黄金 (golden) 状态的卡（未烧写任何分区）将不予显示。

status

`status` 命令用于报告 Vitis Performance Monitor (SPM) 和 Lightweight AXI Protocol Monitor (LAPC) 调试 IP 的状态，这些 IP 包含在编程到卡上的 `xclbin` 内。

它采用如下命令行格式：

```
xbutil status [-d <card>] [--debug_ip_name]
```

下表中列出了可用选项。

表 84：**xbutil status 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否
`--<ip_name>`	返回指定调试 IP 的状态。	否

`status` 命令可显示加速器卡上的调试 IP 的类型和数量。

```
xbutil status
```

此命令的输出示例如下所示。它会列出已找到的所有调试 IP。

```
INFO: Found total 1 card(s), 1 are usable
Number of IPs found: 9
IPs found [<ipname>(<count>)]: aim(5) tracefunnel(1) monitorfifolite(1)
monitorfifofull(1) accelmonitor(1)
Run 'xbutil status' with option --<ipname> to get more information about
the IP
INFO: xbutil status succeeded.
```

可用 IP 列表由 `xclbin` 文件决定，此文件经编译后可在加速器卡上使用，此列表中包括：

- Accelerator Monitor (AM)：对计算单元的执行进行计数和追踪。性能监控器 (Performance Monitor) 可使用 `--profile.exec` 选项来添加，如 --profile 选项 中所述。
- AXI Interface Monitor (AIM)：对 AXI4 连接上的传输事务进行计数和追踪。
- AXI4-Stream Monitor (ASM)：对 AXI4-Stream 上的传输事务进行计数和追踪。
- Lightweight AXI Protocol Monitor (LAPC)：对 AXI4 进行协议检查。协议检查器 (Protocol Checker) 可使用 `--debug.protocol` 选项来添加，如 --debug 选项 中所述。
- Streaming Protocol Checker (SPC)：对 AXI4-Stream 进行协议检查。
- Trace Funnel：收集来自所有监控器的追踪事件。如果存在追踪事件，则在编译期间已使用 `--profile.data` 选项启用追踪（如 --profile 选项 中所述），在运行时则使用 `opencl_trace` 启用追踪（如 xrt.ini 文件 中所述）。
- FIFO Lite：控制用于存储追踪事件的 PL FIFO。如果存在追踪事件，则在编译和运行时期间已启用追踪，在编译期间使用 `--trace_memory` 选项将存储器卸载指定为 PL 中的 FIFO，如 Vitis 编译器常规选项 中所述。追踪行为受到 xrt.ini 文件 中的设置的影响。
- FIFO Full：用于存储追踪事件的 PL FIFO 的数据卸载。如果存在追踪事件，则在编译和运行时期间已启用追踪，在编译期间使用 `--trace_memory` 选项将存储器卸载指定为 PL 中的 FIFO。

- TS2MM：提取追踪事件，并将其卸载至存储器资源（DDR、HBM 或 PLRAM）。如果存在追踪事件，则在编译和运行时期间已启用追踪，在编译期间使用 `--trace_memory` 选项将存储器卸载指定为存储器资源，如 Vitis 编译器常规选项 中所述。

您可使用以下命令语法获取特定 IP 的状态：

```
$ xbutil status --<ipname>
```

使用 `--aim` 选项的输出示例如下所示：

```
$ xbutil status --aim
```

```
INFO: Found total 1 card(s), 1 are usable
AXI Interface Monitor Counters
Region or CU Type or Port            Wr Bytes Wr Trans. Rd Bytes Rd Tranx. Outstanding Cnt
runOnfpga_1  m_axi_maxiport0-DDR[1] 0        0         0        0         0
runOnfpga_1  m_axi_maxiport1-DDR[1] 0        0         0        0         0
shell        Memory to Memory       0        0         0        0         0
shell        Host to Device         0        0         0        0         0
shell        Peer to Peer           0        0         0        0         0
INFO: xbutil status succeeded.
```

以下是各列的代码延续：

```
INFO: Found total 1 card(s), 1 are usable
AXI Interface Monitor Counters
Region or CU Type or Port            Last Wr Addr Last Wr Data Last Rd Addr Last Rd Data
runOnfpga_1  m_axi_maxiport0-DDR[1] 0x0          0x0          0x0          0x0
runOnfpga_1  m_axi_maxiport1-DDR[1] 0x0          0x0          0x0          0x0
shell        Memory to Memory       0x0          0x0          0x0          0x0
shell        Host to Device         0x0          0x0          0x0          0x0
shell        Peer to Peer           0x0          0x0          0x0          0x0
INFO: xbutil status succeeded.
```

如果在 `xclbin` 中未找到任何调试 IP，则会显示以下消息：

```
INFO: Found total 1 card(s), 1 are usable
INFO: Failed to find any debug IPs on the platform. Ensure that a valid
bitstream with debug IPs (SPM, LAPC) is successfully downloaded.
INFO: xbutil status succeeded.
```

如需了解有关在您的设计中添加性能监控器计数器（AM、AIM 或 ASM）和 LAPC 的更多信息，请参阅 应用挂起调试技巧。

top

重要提示！该选项不适用于嵌入式处理器平台。

`top` 命令可输出卡统计数据，包括存储器拓扑结构、DMA 传输指标和计算单元使用数据。该命令与 Linux `top` 命令相似。运行该命令时，它会持续操作，直至在终端窗口中输入 `q` 后才会退出。

它采用如下命令行格式：

```
xbutil top [-d card] [-i seconds]
```

下表中列出了可用选项。

表 85：**xbutil top 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否
`-i <seconds>`	刷新率（以秒为单位）默认为 1 秒。	否

例如，以下命令将以 2 秒刷新率来执行 `top`。

```
xbutil top -i 2
```

表 86：**顶层字段定义**

字段	描述
Device Memory Usage	每个存储体所用的存储器百分比。 将按所用存储器比例显示图形化用量条。所显示的条形如下所示： ||||||||||
Power	卡的总功耗
Mem Topology	分配给存储体的标记
	存储器类型（即 DDR 或 HBM）
	存储体的温度
	每个存储体的可用存储器总量
	当前存储器使用情况
	已分配的缓冲器数量
Total DMA Transfer Metrics	由于重启而传输的累计字节数
CU Usage	由该 CU 执行的命令数。数量累计至 `xclbin` 发生更改为止，更改后数量复位为 0。

下列示例即为运行此命令的输出结果：

```
Device Memory Usage
 [0] bank0      [ ||||||||||                                          25.0% ]
 [1] bank1      [                                                     0.00% ]
 [2] bank2      [                                                     0.00% ]
 [3] bank3      [                                                     0.00% ]


Power
34.0W

Mem Topology                                        Device Memory Usage
Tag             Type         Temp        Size       Mem Usage        BO nums
 [0] bank0      MEM_DDR4     36 C        16 GB      4 GB             16
 [1] bank1      MEM_DDR4     38 C        16 GB      0 Byte           0
 [2] bank2      MEM_DDR4     46 C        16 GB      0 Byte           0
 [3] bank3      MEM_DDR4     41 C        16 GB      0 Byte           0

Total DMA Transfer Metrics:
  Chan[0].h2c:  75 GB
  Chan[0].c2h:  78 GB
```

```
  Chan[1].h2c:  61600 MB
  Chan[1].c2h:  61440 MB

################################################################################

Compute Unit Usage:
CU[@0x1800000] : 68
CU[@0x1c00000] : 68
CU[@0x250000] : 6

################################################################################
```

如果未加载 `xclbin`，则将显示：

```
Device Memory Usage
 [1] bank1      [                                                      0.00% ]

Power
23W

Mem Topology                                        Device Memory Usage
Tag              Type         Temp      Size       Mem Usage         BO nums
 [1] bank1       MEM_DDR4     36        64 GB      0 Byte            0

Total DMA Transfer Metrics:
  Chan[0].h2c:  0 Byte
  Chan[0].c2h:  0 Byte
  Chan[1].h2c:  0 Byte
  Chan[1].c2h:  0 Byte

################################################################################

Compute Unit Usage:

################################################################################
```

validate

重要提示！该选项不适用于嵌入式处理器平台。

`validate` 命令可为已安装的卡生成易于阅读的高层次摘要。该选项通过执行下列测试来确认安装是否正确：

1. 确认发现的卡。
2. 检查 PCI Express 链路状态。
3. 在卡上运行验证内核。
4. 执行下列数据带宽测试：
   a. DMA 测试：主机与卡存储器之间通过 PCI Express 执行的数据传输。
   b. DDR 或 HBM 测试：内核与卡存储器之间执行的数据传输。

它采用如下命令行格式：

```
xbutil validate [-d card]
```

下表中列出了可用选项。

表 87：**xbutil validate 命令选项**

选项	描述	是否必需
`-d <card>`	指定目标卡。<card> 可以 card_id 形式或 Bus:Device:Function (BDF) 形式来指定。如未指定，则默认为 card_id = 0。 **注释：** `xbutil scan` 命令可用于显示已安装的卡的 card_id 和 BDF。	否

下列示例演示的是运行此命令的输出结果：

```
INFO: Found 1 cards

INFO: Validating card[0]: xilinx_u200_xdma_201830_2
INFO: == Starting AUX power connector check:
INFO: == AUX power connector check PASSED
INFO: == Starting PCIE link check:
INFO: == PCIE link check PASSED
INFO: == Starting verify kernel test:
INFO: == verify kernel test PASSED
INFO: == Starting DMA test:
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 8775.99 MB/s
Host <- PCIe <- FPGA read bandwidth = 12136.8 MB/s
INFO: == DMA test PASSED
INFO: == Starting device memory bandwidth test:
...........
Maximum throughput: 52428 MB/s
INFO: == device memory bandwidth test PASSED
INFO: == Starting PCIE peer-to-peer test:
P2P BAR is not enabled. Skipping validation
INFO: == PCIE peer-to-peer test SKIPPED
INFO: == Starting memory-to-memory DMA test:
bank0 -> bank1 M2M bandwidth: 12010.3 MB/s
bank0 -> bank2 M2M bandwidth: 12051.6 MB/s
bank0 -> bank3 M2M bandwidth: 12063.5 MB/s
bank1 -> bank2 M2M bandwidth: 12052.7 MB/s
bank1 -> bank3 M2M bandwidth: 12048.2 MB/s
bank2 -> bank3 M2M bandwidth: 12052.2 MB/s
INFO: == memory-to-memory DMA test PASSED
INFO: Card[0] validated successfully.

INFO: All cards validated successfully.
```

version

重要提示！ 该选项不适用于嵌入式处理器平台。

`version` 命令用于返回 XRT 构建版本详细信息。其作用与 version 命令相同。它采用如下命令行格式。没有可用选项。

```
xbutil version
```

下表列出了 `xbutil version` 命令返回的字段。

表 88：**版本字段定义**

字段	描述
XRT Build Version	XRT 构建版本
Build Version Branch	构建版本分支
Build Version Hash	构建版本散列
Build Version Hash Date	构建版本分支日期
Build Version Date	构建版本日期
XOCL	XOCL 版本
XCLMGMT	XCLMGMT 版本

以下显示了针对安装有 3 张卡的系统执行 `xbutil version` 的输出示例。

```
      XRT Build Version: 2.3.1301
   Build Version Branch: 2019.2
     Build Version Hash: 192e706aea53163a04c574f9b3fe9ed76b6ca471
Build Version Hash Date: Thu, 24 Oct 2019 19:27:30 -0700
     Build Version Date: Thu, 24 Oct 2019 20:04:29 -0700
                   XOCL: 2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471
                XCLMGMT: 2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471
```

要返回更多卡详细信息，请使用 `xbmgmt flash --scan`。

xbmgmt 实用工具 - 旧版本

提示：本节描述了旧版本的 `xbmgmt` 命令，此命令可通过调用 `xbmgmt --legacy` 命令来访问。

赛灵思开发板管理 (`xbmgmt`) 实用工具是一个独立命令行工具，随附于赛灵思的 Xilinx Runtime (XRT) 安装包内。`xbmgmt` 命令支持 Alveo 数据中心加速器卡和基于嵌入式处理器的平台。

加速器卡划分为用户函数和管理函数，以提供不同的卡访问级别。用户函数允许用户加载并运行其应用，而管理函数则供系统管理员用于对卡进行管理。`xbutil` 实用工具能够与用户函数进行交互。`xbmgmt` 实用工具则需要根用户权限，用于与管理函数进行交互。将函数访问拆分为两种实用工具的原因是为了给该工具的管理功能特性提供一定程度的安全性。

重要提示！ `xbmgmt` 实用工具仅适用于包含赛灵思提供的 shell 或平台的 Alveo 卡。XRT 不适用于定制的 Vivado® 设计。

该实用工具用于卡的安装和管理，并且运行时需要 `sudo` 权限。`xbmgmt` 支持的任务包括烧写卡固件和扫描当前器件配置。

`xbmgmt` 命令行格式为：

```
xbmgmt <command> [options]
```

以下提供了受支持的子命令。

- config：解析或更新守护程序/器件配置

- flash：更新器件上的 SC 固件或 shell
- help：打印输出子命令的帮助消息
- partition：显示分区，并将其下载至器件上
- scan：列示所有已检测到的 `mgmt` PCIe® 函数
- version：打印输出 XRT 构建版本

提示：您可以使用 `help` 命令来列出可用的 `xbmgmt` 命令和选项，并使用以下命令来访问各项命令的帮助信息：

```
xbmgmt help <command>
```

如需获取各条子命令的详细帮助信息，请使用以下命令：

```
xbmgmt help <subcommand>
```

使用以下脚本设置 `xbmgmt` 命令：

- 对于 csh shell：

```
$ source /opt/xilinx/xrt/setup.csh
```

- 对于 bash shell：

```
$ source /opt/xilinx/xrt/setup.sh
```

config

`xbmgmt config` 命令支持您配置 Alveo 加速器卡的存储器保留功能。

表 89：**xbmgmt config 子命令**

子命令	描述
`--enable_retention`	此功能特性可在 DFX-2RP 目标平台中执行 `xclbin` 交换期间，保留 DDR 数据内容。
`--disable_retention`	此功能特性用于禁用在 DFX-2RP 目标平台中执行 `xclbin` 交换期间保留 DDR 数据内容的功能。

`config` 命令具有下列命令行选项。

```
sudo xbmgmt config --enable_retention --ddr [--card <bdf>]
```

表 90：**xbmgmt config 子命令选项**

选项	描述	是否必需
--ddr	启用或禁用 DDR 存储器内容保留功能。	是

表 90：**xbmgmt config 子命令选项** (续)

选项	描述	是否必需
`--card <bdf>`	指定按加速器卡的 Bus:Device:Function (BDF) 标签所标识的方式来对该卡进行配置。 **提示：** `xbmgmt flash --scan` 可用于获取卡的 BDF。 如果未找到 BDF，那么您将收到如下消息，其中 <bdf> 表示输入的 BDF 值： `ERROR: No mgmt PF found for <bdf>`	否

flash

重要提示！该选项不适用于嵌入式处理器平台。

`flash` 命令具有 3 条子命令，如下表中所述。

`xbmgmt flash --scan` 命令将报告是否加载平台库。您将需要使用 `xbmgmt partition --scan` 来查看是否加载 shell。请参阅 Alveo 平台加载概述，以获取有关平台库和 shell 的更多信息。

表 91：**xbmgmt flash 子命令**

子命令	描述
`--scan`	查询卡的可烧写分区，此分区在 FPGA 上运行并安装在主机系统上
`--update`	将目标平台（可烧写分区）烧写到卡上
`--factory_reset`	将卡重置为出厂状态

以下详述了每条子命令。

--scan

`--scan` 子命令用于返回每个卡上安装的可烧写分区以及主机系统上安装的可烧写分区。此外，它还可返回其它信息，包括 SC 版本、BDF、序列号和 MAC 地址。

它采用如下命令行格式。

```
xbmgmt flash --scan [--verbose | --json]
```

表 92：**xbmgmt flash --scan Sub-command Options**

选项	描述	是否必需
`--verbose`	返回详细输出，包含额外字段。 `--verbose` 选项与 `--json` 选项互斥。	否
`--json`	返回随 `--verbose` 选项提供的所有字段（采用 JSON 格式）。 `--verbose` 选项与 `--json` 选项互斥。	否

使用不含任何选项的 `flash --scan` 子命令（如下所示）将返回下表中所列的字段。

```
xbmgmt flash --scan
```

表 93：**xbmgmt flash --scan 字段定义**

字段	描述
卡	为卡提供枚举的卡总线器件功能 (BDF)，格式如下： [Bus : Device : Function] **提示：** `xbmgmt` 命令返回的 BDF 中包含卡上的管理功能。`xbutil scan` 命令返回的 BDF 中包含卡上的用户功能。
Card type	赛灵思卡类型
Flash type	返回卡上物理安装的烧写类型。烧写类型包括： · Dual QSPI：双 x4 SPI · SPI：单 x4 SPI · OSPI：单 x8 SPI
Flashable partition running on FPGA	返回有关卡上安装的可烧写分区的详细信息： · FPGA 上运行的目标平台的名称 · 平台比特流的 ID 唯一标识 · 卫星控制器 (SC) 版本号 **重要提示！** FPGA 的 ID 上运行的可烧写分区必须与系统内安装的可烧写分区匹配，否则堆栈将无法正确运行。
Flashable partitions installed in system	返回有关主机系统上安装的可烧写分区的详细信息： · 主机系统上已安装并正在运行的目标平台的名称 · 表示目标平台的时间戳的 ID · SC 版本号

以下显示了针对安装有 2 张不同的卡的系统执行 `xbmgmt flash --scan` 的输出示例：

```
Card [0000:a6:00.0]
    Card type:          u280
    Flash type:         SPI
    Flashable partition running on FPGA:
        xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]
    Flashable partitions installed in system:
        xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]

Card [0000:73:00.0]
    Card type:          u250
    Flash type:         SPI
    Flashable partition running on FPGA:
        xilinx_u250_xdma_201830_2,[ID=0x5d14fbe6],[SC=4.3.7]
    Flashable partitions installed in system:
        xilinx_u250_xdma_201830_2,[ID=0x5d14fbe6],[SC=4.3.7]
```

如果卡是首次安装或者先前未曾进行烧写，或者如果卡已重置为出厂状态，那么 FPGA 上运行的可烧写分区将在其名称中以 GOLDEN 一词来表示此状态，如 --factory_reset 中所示。

使用 `--verbose` 选项（如以下示例所示）会返回下表中指定的额外字段。

注释：平台必须安装在系统上才能获取卡的完整信息。

```
xbmgmt flash --scan --verbose
```

表 94：**xbmgmt flash --scan --verbose 字段定义**

字段	描述
Card Name	赛灵思提供的卡名称
Card serial number (S/N)	卡的唯一序列号
Configuration mode	返回 FPGA 从冷复位启动时采用的配置模式。配置模式包括： · Dual QSPI：双 x4 SPI · QSPI：单 x4 SPI · OSPI：单 x8 SPI
Fan presence	表示卡上是否存在风扇。 A - 主动散热。卡上存在风扇。 P - 被动散热。卡上不存在风扇，必须由主机服务器进行散热。
Max power level	从 PCIe 和已连接的辅助电源端口提供的卡的最大可用电源功耗（瓦）。并非所有卡都有辅助电源端口。
MAC address	返回赛灵思为卡分配的 MAC 地址列表。 您可自由使用赛灵思分配的 MAC 地址或提供您自己的 MAC 地址。 地址 FF:FF:FF:FF:FF:FF 暗示尚未为此 MAC 插槽分配地址。

以下提供了含单卡的系统的 `xbmgmt flash --scan --verbose` 输出示例：

```
Card [0000:a6:00.0]
    Card type:          u280
    Flash type:         SPI
    Flashable partition running on FPGA:
        xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]
    Flashable partitions installed in system:
        xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]
    Card name                   ALVEO U280 PQ
    Card S/N:                   21760394R01L
    Config mode:                7
    Fan presence:               A
    Max power level:            225W
    MAC address0:               00:0A:35:06:00:0A
    MAC address1:               00:0A:35:06:00:0B
    MAC address2:               FF:FF:FF:FF:FF:FF
    MAC address3:               FF:FF:FF:FF:FF:FF
```

使用 `--json` 子选项会返回类似于 `xbmgmt flash --scan` 的信息，但采用的是 JSON 格式。以下显示了针对含单卡的系统生成的 JSON 输出示例。

```
{
    "card0": {
        "shellpackage": "xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],
[SC=4.3.4]; ",
        "name": "ALVEO U280 PQ",
        "serial": "21760394R01L",
        "config_mode": "7",
        "fan_presence": "A",
        "max_power": "225W",
        "mac0": "00:0A:35:06:00:0A",
```

```
            "mac1": "00:0A:35:06:00:0B",
            "mac2": "FF:FF:FF:FF:FF:FF",
            "mac3": "FF:FF:FF:FF:FF:FF"
        }
}
```

--update

使用 `flash --update` 子命令来更改卡上的可烧写分区（目标平台）。方法是将指定目标平台和关联的卫星控制器烧写到卡上的可配置 ROM 中。它采用如下命令行格式：

```
xbmgmt flash --update [--shell <target_platform_name>
[--id <target_platform_id>]] [--card <bdf] [-force]
```

下表中列出了可用选项。

表 95：**xbmgmt --update 子命令选项**

选项	描述	是否必需

`--shell <target_platform_name> [--id]`	要烧写到卡上的目标平台（可烧写分区）的名称。目标平台必须先安装到主机系统上，然后才能通过目标平台安装包来指定。请参阅 Alveo™ 卡安装指南，以获取有关下载和安装部署目标平台的详细信息。`xbmgmt flash --scan` 可用于列出主机系统上已安装的可用目标平台。
	如果 `--shell` 和 `--card` 均未指定，那么将以主机系统上可用的兼容目标平台来烧写所有卡。
	例如，如果系统安装有 U200 和 U250 卡，且 U200 和 U250 目标平台在主机系统上都存在，那么将以这两张卡的对应目标平台来烧写这两张卡。如果卡的目标平台在系统上不存在，那么将不会烧写该卡。
	如果已指定 `--shell` 选项，但未指定 `--card` 选项，那么将烧写与指定 `target_platform_name` 兼容的所有卡。例如，如果系统安装有 2 张 U200 卡，那么将以指定 `target_platform_name` 来烧写这 2 张卡。如果卡的目标平台在系统上不存在，那么将不会烧写该卡。
	如果未指定 `--shell` 选项，但指定了 `--card` 选项，那么将以主机系统上可用的兼容目标平台来烧写指定的卡。
	如果 `--shell` 和 `--card` 选项均已指定，那么将以主机系统上可用的指定目标平台来烧写指定的卡。
	如果卡上可烧写的分区与系统上可烧写的分区相匹配，那么将显示如下消息，且该卡将不进行更新。但您可通过使用 `--force` 选项来强制对该卡进行烧写，如下所述。
	`Card [0000:65:00.0]:`
	`Status: shell is up-to-date`
	`Card(s) up-to-date and do not need to be flashed.`
	如果在主机系统上未安装指定的 `target_platform_name`，那么该卡将不会进行更新，您将收到如下消息：
	`Specified shell not found.`
	此 `--id` 子选项可指定目标平台的 ID。
	`xbmgmt flash --scan` 可用于获取可烧写分区的 ID。
	如果未指定 `--id` 选项，那么将以主机系统上可用的最新版本的目标平台来更新该卡。
		否

表 95：**xbmgmt --update 子命令选项** (续)

选项	描述	是否必需
`--card <bdf>`	指定按加速器卡的 Bus:Device:Function (BDF) 标签所标识的方式来更新该卡。`xbmgmt flash --scan` 可用于获取卡的 BDF。 如果未指定 `--card`，那么系统中与指定 `target_platform` 兼容的所有加速器卡都将进行更新。 如果 `--shell` 和 `--card` 均未指定，那么将以主机系统上可用的兼容目标平台来烧写所有卡。例如，如果系统安装有 U200 和 U250 卡，且 U200 和 U250 目标平台在主机系统上都存在，那么将以这两张卡的对应目标平台来烧写这两张卡。如果卡的目标平台在系统上不存在，那么将不会烧写该卡。 如果已指定 `--shell` 选项，但未指定 `--card` 选项，那么将烧写与指定 `target_platform_name` 兼容的所有卡。 例如，如果系统安装有 2 张 U200 卡，那么将以指定 `target_platform_name` 来烧写这 2 张卡。如果卡的目标平台在系统上不存在，那么将不会烧写该卡。 如果未指定 `--shell` 选项，但指定了 `--card` 选项，那么将以主机系统上可用的兼容目标平台来烧写指定的卡。 如果 `--shell` 和 `--card` 选项均已指定，那么将以主机系统上可用的指定目标平台来烧写指定的卡。 如果未找到 BDF，那么您将收到如下消息，其中 <bdf> 表示输入的 BDF 值。`xbmgmt flash --scan` 可用于列出已安装的卡的 BDF 值。 `ERROR: No mgmt PF found for <bdf>`	否
`--force`	force 选项表示对于来自 `xbmgmt flash --update` 命令的所有提示都选择“是”。 例如，通过 `xbmgmt flash --update` 烧写卡时，它将显示如下提示，要求您确认是否要执行更新： `Are you sure you wish to proceed? [y/n]:` `--force` 将自动把答案设为“y”（表示“是”），然后继续执行操作。	否

--factory_reset

`flash --factory_reset` 子命令可用于将 FPGA 上运行的可烧写分区还原为原始黄金镜像。此命令将不会更改卫星控制器的版本。它采用如下命令行格式：

```
xbmgmt flash --factory_reset [--card <bdf>]
```

可用选项是唯一的，如下表所示。

表 96：**xbmgmt --factory_reset 子命令选项**

选项	描述	是否必需
`--card <bdf>`	指定按加速器卡的 Bus:Device:Function (BDF) 标签所标识的方式来将该卡重置为出厂状态。 `xbmgmt flash --scan` 可用于获取卡的 BDF。 如果未指定 `--card`，那么将重置卡 ID 0。卡 ID 并非固定，可在冷重启或热重启后发生更改。	否

运行 `xbmgmt flash --factory_reset` 命令后，需要冷重启系统以便将卡复原为原始黄金镜像。

完成出厂状态重置和冷启动后，请使用 `xbmgmt flash --scan` 来确认 FPGA 上运行的可烧写分区是否已还原。此分区将在其名称中包含 GOLDEN 一词，如下所示：

```
Card [0000:a6:00.0]
    Card type:          u280
    Flash type:         SPI
    Flashable partition running on FPGA:
        xilinx_u280_GOLDEN_8,[SC=4.3]
    Flashable partitions installed in system:
        xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]
```

partition

`xbmgmt partition` 命令可显示 shell 分区，并将其编程到 Alveo 卡上。

`partition` 命令旨在搭配 DFX-2RP 平台来使用。运行应用（包括确认应用）前，需要首先使用以下命令将 shell 分区编程到卡上：

```
sudo /opt/xilinx/xrt/bin/xbmgmt partition --program --name <shell_name> --
card <card_bdf>
```

完成 shell 分区编程后，无需对加速器卡重新编程，除非对系统进行热重启或冷重启，或者加载其它 shell。

提示：平台库会保持处于已加载状态，即使在加速器卡上掉电并重新上电也是如此。

`xbmgmt partition --scan` 命令用于报告是否已加载平台 shell。

表 97：**xbmgmt partition 子命令**

子命令	描述
`--scan`	扫描并显示 FPGA 上正在运行以及系统中已安装的库和 shell 分区。
`--program`	此命令用于通过提供 shell 名称来对 shell 分区进行编程。此命令还支持从给定 interface-uuid 对 shell 进行编程的选项。

Scan

`--scan` 子命令用于返回每个卡上安装的可烧写分区以及主机系统上安装的可烧写分区。此外，它还可返回其它信息，包括 SC 版本、BDF、序列号和 MAC 地址。

它采用如下命令行格式。

```
xbmgmt partition -–scan
```

以下是此命令的输出示例：

```
Card [0000:d8:00.0]
    Partitions running on FPGA:
        xilinx_u200_gen3x16_base_1
            logic-uuid:
            8892e9a0478feaa2699f5df1f696470d
            interface-uuid:
            1641962866f4b5e579cec90a6bdabcbf
    Partitions installed in system:
```

```
        xilinx_u200_gen3x16_xdma_shell_1_1
            logic-uuid:
            a21db155d2fbd60ccc95eea0ea8144e1
            interface-uuid:
            9437e0f859a4d9bd9e226d7edf5e6be8
```

如果分区已编程，则输出如下所示。

```
alveo@alveo:~$ sudo /opt/xilinx/xrt/bin/xbmgmt partition --scan
Card [0000:65:00.0]
    Partitions running on FPGA:
        xilinx_u200_gen3x16_base_1
            logic-uuid:
            3d40702f37777396cc82e0df89bafde2
            interface-uuid:
            19f21ba41b9c38dffefc1ee68910b8bb
        xilinx_u200_gen3x16_xdma_shell_1_1
            logic-uuid:
            fd19b2fde5a10b8cb89e35b0be02f274
            interface-uuid:
            995b41d8c729d658d6700a027f412f78
    Partitions installed in system:
        xilinx_u200_gen3x16_xdma_shell_1_1
            logic-uuid:
            fd19b2fde5a10b8cb89e35b0be02f274
            interface-uuid:
            995b41d8c729d658d6700a027f412f78
```

Program

`--program` 子命令允许您对指定 shell 分区进行编程。

它采用如下命令行格式。

```
xbmgmt partition --program --name name [--id interface-uuid] [--card bdf]
```

表 98：**xbmgmt --program 子命令选项**

选项	描述	是否必需
--name <name>	指定要编程的 shell 分区名称。 `xbmgmt partition --scan` 用于显示可编程分区的名称。	是
`--id`	`--id` 子选项用于指定分区的 interface-uuid。 `xbmgmt partition --scan` 可用于获取可烧写分区的 ID。	否
`--card <bdf>`	指定按加速器卡的 Bus:Device:Function (BDF) 标签所标识的方式来对该卡进行编程。 `xbmgmt partition --scan` 可用于获取卡的 BDF。 如果未找到 BDF，那么您将收到如下消息，其中 <bdf> 表示输入的 BDF 值： `ERROR: No mgmt PF found for <bdf>`	否

scan (xbmgmt)

重要提示！该选项不适用于嵌入式处理器平台。

`xbmgmt scan` 命令可返回所有已检测到的 PCIe 管理函数的列表。列表中的每个项都包含卡 BDF、目标平台名称、目标平台 ID 和管理驱动程序实例编号。

提示：如需更多详细信息，请使用 `xbmgmt flash --scan --verbose` 命令。

它采用如下命令行格式。没有可用选项。

```
xbmgmt scan
```

下表列出了 `xbmgmt scan` 命令返回的字段。

表 99：**xbmgmt scan 字段定义**

字段	描述
BDF	为卡提供枚举的总线:器件:功能 (BDF) 标识，格式如下：[Bus:Device:Function]
Flashable partition running on FPGA	有关可烧写分区的详细信息包括： · FPGA 上烧写的目标平台的名称 · 与目标平台关联的唯一 ID。
mgmt	返回已分配的管理驱动程序实例。 实例编号可便于查找每个函数的器件节点。 在受支持的 Linux 分区上，可在以下位置找到器件节点：`/dev/xclmgmt<inst>`。 此外，将 `dmesg` 信息映射到指定卡时，实例也很有用。

以下显示了针对安装有 2 张卡的系统执行 `xbmgmt scan` 的输出示例。每张卡的详细信息各占一行：

```
0000:d8:00.0 xilinx_u200_gen3x16_xdma_shell_1_1 mgmt(inst=55296)
 0000:af:00.0 xilinx_u250_gen3x16_base_3 mgmt(inst=44800)
```

version

重要提示！该选项不适用于嵌入式处理器平台。

`version` 命令用于返回 XRT 构建版本详细信息。其作用与 xbutil version 命令相同。它采用如下命令行格式。没有可用选项。

```
xbmgmt version
```

下表列出了 `xbmgmt version` 命令返回的字段。

表 100：**版本字段定义**

字段	描述
XRT Build Version	XRT 构建版本
Build Version Branch	构建版本分支
Build Version Hash	构建版本散列
Build Version Hash Date	构建版本分支日期

表 100：**版本字段定义** (续)

字段	描述
Build Version Date	构建版本日期
XOCL	XOCL 版本
XCLMGMT	XCLMGMT 版本

以下是 `xbmgmt version` 的输出示例。

```
      XRT Build Version: 2.3.1301
   Build Version Branch: 2019.2
     Build Version Hash: 192e706aea53163a04c574f9b3fe9ed76b6ca471
Build Version Hash Date: Thu, 24 Oct 2019 19:27:30 -0700
     Build Version Date: Thu, 24 Oct 2019 20:04:29 -0700
                  XOCL: 2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471
                XCLMGMT: 2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471
```

XILINX

第 54 章

数据流传输

内核之间 (K2K) 的数据流传输

Vitis™ 核开发套件同样支持 2 个内核之间的数据流传输。假设某个内核正在执行某部分计算，另一个内核在收到来自第一个内核的输出数据后即完成操作。借助内核至内核数据流传输支持，数据即可从某一个内核直接移至另一个内核，而无需通过全局存储器发回数据。这样即可使性能得到显著提升。

主机编码准则

内核至内核数据流传输中涉及的内核端口无需在主机代码中使用 `clSetKernelArg` 进行设置。数据流传输连接中未涉及的所有内核实参都应使用 `clSetKernelArg` 进行设置，如 设置内核参数 中所述。但是，数据流传输中所涉及的内核端口都将在内核自身中定义，主机程序不会对其进行寻址。

数据流传输内核编码指南

在内核中，通过 `hls::stream` 采用 `ap_axiu<D,0,0,0>` 作为数据类型，来定义两个内核数据流传输接口之间直接执行的数据发送或接收操作。`ap_axiu<D,0,0,0>` 数据类型需要使用 `ap_axi_sdata.h` 报头文件。

重要提示！ 主机到内核以及内核到主机之间的数据流传输需要使用 `qdma_axis` 数据类型。在随 Vitis 软件平台安装而分发的 `ap_axi_sdata.h` 报头文件内已定义 `ap_axiu` 和 `qdma_axis` 数据类型。

以下示例显示了生产者和使用者内核的数据流传输接口。

```
// Producer kernel - provides output as a data stream
// The example kernel code does not show any other inputs or outputs.

void kernel1 (.... , hls::stream<ap_axiu<32, 0, 0, 0> >& stream_out) {

  for(int i = 0; i < ...; i++) {
    int a = ..... ;          // Internally generated data
    ap_axiu<32, 0, 0, 0> v;  // temporary storage for ap_axiu
    v.data = a;              // Writing the data
    stream_out.write(v);          // Writing to the output stream.
  }
}

// Consumer kernel - reads data stream as input
// The example kernel code does not show any other inputs or outputs.

void kernel2 (hls::stream<ap_axiu<32, 0, 0, 0> >& stream_in, .... ) {

  for(int i = 0; i < ....; i++) {
    ap_axiu<32, 0, 0, 0> v = stream_in.read(); // Reading the input stream
```

```
    int a = v.data; // Extract the data

    // Do further processing
  }
}
```

鉴于已定义 `hls::stream` 数据类型，Vitis HLS 工具可推断 `axis` 接口。以下 INTERFACE 编译指示作为示例显示，而并非旨在添加到代码中。

```
#pragma HLS INTERFACE axis port=stream_out
#pragma HLS INTERFACE axis port=stream_in
```

提示：这些示例内核显示了内核特征符内的数据流传输输入/输出端口的定义，以及内核代码中输入/输出数据流传输的处理方式。在内核链接进程期间，必须定义 `kernel1` 到 `kernel2` 的连接，如 在计算单元之间指定数据流传输连接 中所述。

如需了解有关映射数据流传输连接的更多信息，请参阅 第三部分：构建和运行应用。

自由运行的内核

Vitis 核开发套件可以为一个或多个自由运行的内核提供支持。自由运行的内核没有控制信号端口，并且无法手动启动或停止。自由运行的内核的无控制信号功能会产生如下特征：

- 自由运行的内核没有存储器输入或输出端口，因此它只能通过数据流传输来与主机或其它内核（包括常规内核或其它自由运行的内核）进行交互。
- 如果通过二进制容器 (`xclbin`) 来对 FPGA 进行编程，那么自由运行的内核会在 FPGA 上开始运行，因此无需来自主机代码的 `clEnqueueTask` 命令。
- 当内核一旦接收到来自主机或其它内核的流传输数据后，就会立即开始处理这些数据，一旦这些数据不可用，内核就会停止处理。
- 自由运行的内核在其内核主体内需要特殊的接口编译指示 `ap_ctrl_none`。

适用于自由运行的内核的主机编码

如果自由运行的内核与主机交互，主机代码应通过 `clCreateStream/clReadStream/clWriteStream` 来管理数据流传输操作，如 主机编码准则 中所述。由于自由运行的内核没有任何其它类型的输入或输出（例如，存储器端口或控制端口），因此无需指定 `clSetKernelArg`。由于当内核一旦接收到来自主机或其它内核的流传输数据后，就会立即开始处理这些数据，一旦这些数据不可用，内核就会停止处理，因此不使用 `clEnqueueTask`。

自由运行的内核的编码指南

如前文所述，自由运行的内核仅包含 `hls::stream` 输入和输出。编码指南建议包括：

- 如果端口正在与来自内核的另一个数据流传输进行交互，请使用 `hls::stream<ap_axiu<D,0,0,0> >`。
- 如果端口正在与主机交互，请使用 `hls::stream<qdma_axis<D,0,0,0> >`。
- 针对函数参数使用 `hls::stream` 数据类型会导致 Vitis HLS 推断接口的 AXI4-Stream 端口 (`axis`)。

- 自由运行的内核还必须指定以下特殊 INTERFACE 编译指示。

```
#pragma HLS interface ap_ctrl_none port=return
```

提示： `ap_ctrl_none` 意味着内核没有控制接口，因此通常不生成 `s_axilite` 接口。但是无论存在标量实参还是 `m_axi` 接口，都需要使用 `s_axilite` 接口。

以下代码示例显示的是包含一项输入和一项输出的自由运行的内核，该内核与另一个内核进行通信。`while(1)` 循环结构包含内核代码的一部分，只要内核运行，就会重复执行这部分结构。

```
void kernel_top(hls::stream<ap_axiu<32, 0, 0, 0> >& input,
   hls::stream<ap_axiu<32, 0, 0, 0> >& output) {
#pragma HLS interface ap_ctrl_none port=return  // Special pragma for free-
running kernel

#pragma HLS DATAFLOW // The kernel is using DATAFLOW optimization
    while(1) {
        ...
    }
}
```

提示： 此示例显示了自由运行的内核中数据流传输输入/输出端口的定义。但在内核链接进程中，必须定义自由运行的内核与其它内核之间的往来数据流传输连接，如 在计算单元之间指定数据流传输连接 中所述。

第十部分

附加资源与法律声明

赛灵思资源

如需获取答复记录、技术文档、下载以及论坛等支持性资源，请参阅赛灵思技术支持。

Documentation Navigator 与设计中心

赛灵思 Documentation Navigator (DocNav) 提供了访问赛灵思文档、视频和支持资源的渠道，您可以在其中筛选搜索信息。打开 DocNav 的方法：

- 在 Vivado® IDE 中，选择“Help” → “Documentation and Tutorials”。
- 在 Windows 上，选择“Start” → “All Programs” → “Xilinx Design Tools” → “DocNav”。
- 在 Linux 命令提示中输入 `docnav`。

赛灵思设计中心提供了根据设计任务和其它主题整理的文档链接，可供您用于了解关键概念以及常见问题解答。要访问设计中心，请执行以下操作：

- 在 DocNav 中，单击“Design Hubs View”选项卡。
- 在赛灵思网站上，查看设计中心页面。

注释：如需了解有关 DocNav 的更多信息，请参阅赛灵思网站上的 Documentation Navigotor。

修订历史

Vitis 入门指南修订历史

下表显示了 第一部分：Vitis 入门 的修订历史。

章节	修订综述
2021 年 7 月 19 日 2021.1 版	
不适用	本节无更改。

章节	修订综述
2021 年 6 月 16 日 2021.1 版	
安装要求	更新至 2021.1。
安装 Vitis 软件平台	更新至 2021.1。
设置用于运行 Vitis 软件平台的环境	更新至 2021.1。
数据中心应用加速开发流程	更新“运行应用”描述和图示。
嵌入式处理器应用加速开发流程	更新“运行应用”描述和图示。
C/C++ 内核的开发方法论	添加描述和 UG1399 链接。
关于高层次综合编译器	更新 UG1399 链接。

开发应用修订历史

下表显示了 第二部分：开发应用 的修订历史。

章节	修订综述
2021 年 7 月 19 日 2021.1 版	
不适用	本节无更改。
2021 年 6 月 16 日 2021.1 版	
模型编程	更新章节。
器件拓扑结构	更新 CPU 描述。
内核属性	更新章节。
时钟和复位要求	更新表格。
命令队列	添加注释。
主机编程	添加章节。
C/C++ 内核	更新链接。
进程执行模式	更新章节。
接口	添加吞吐量和链接。
数据流最优化	添加最佳实践和链接。
在用户管理的永续内核中进行数据流传输	添加子主题。
内核接口要求	更新章节。
RTL 内核	更新章节。
创建用户管理的 RTL 内核	添加子主题。

构建和运行应用修订历史

下表显示了 第三部分：构建和运行应用 的修订历史。

章节	修订综述
2021 年 7 月 19 日 2021.1 版	
不适用	本节无更改。
2021 年 6 月 16 日 2021.1 版	
构建器件二进制文件	更新 Vivado IP 封装器。
使用 Vitis HLS 编译内核	更新描述。
将计算单元分配给 SLR	添加重要注释。

章节	修订综述
管理时钟频率	更新章节。
运行多项实现策略来实现时序收敛	次要更新。
v++ 命令的输出目录	更新 profile_summary.csv 和 timeline_trace.csv
在嵌入式处理器平台上运行仿真	更新 profile_summary.csv 和 timeline_trace.csv
处理 SystemC 模型	新增。

应用剖析、最优化和调试修订历史

下表显示了 第四部分：对应用进行剖析、最优化和调试 的修订历史。

章节	修订综述
2021 年 7 月 19 日 2021.1 版	
启用内核以利用 Chipscope 进行调试	在 System ILA 添加了重要信息和提示注释，并简化了设计描述。
2021 年 6 月 16 日 2021.1 版	
在应用中启用剖析	更新 power_profile。
连续追踪捕获	新增。
主机应用的定制剖析	更新章节。
C++ 代码剖析	更新 user_range 代码并移除注释。
生成和打开剖析汇总报告	更新第 2 项中的代码和 profile_summary。
时间线轨迹	更新标题和描述。
生成并打开时间线轨迹	更新标题和描述。
解读时间线轨迹	更新标题和描述。
生成并打开波形报告	更新第 2 项中的代码。
解读“Waveform”视图中数据	更新描述。
多个有序命令队列	添加注释。
基于 GDB 的调试	移除硬件。
基于 GDB 内核的调试	移除硬件。
启动主机和内核调试	移除硬件。
在硬件仿真中调试	更新图示。
硬件仿真中基于 GDB 的调试	已移除相关章节。
利用 Vitis 编译器命令启用波形调试	更新第 2 项中的代码。
使用赛灵思 xbutil 实用工具	更新 Performance Monitor 描述。
由于 AXI 违例导致内核挂起	更新第 4 项中的提示注释。
命令行调试示例	更新第 8 项。

Vitis 环境参考资料修订历史

下表显示了 第五部分：Vitis 环境参考资料 的修订历史。

章节	修订综述
2021 年 7 月 19 日 2021.1 版	
不适用	本节无更改。

章节	修订综述
2021 年 6 月 16 日 2021.1 版	
Vitis 编译器常规选项	更新描述。
--advanced 选项	移除表中的 gdb 选项，并更新 --advanced.param 表。
--clock 选项	更新章节。
--connectivity 选项	添加 --connectivity.connect 并重新组织。
--hls 选项	添加 --hls.export_mode。
--package 选项	在 --package.boot_mode 中添加提示注释并更新 --package.ps_elf。
Vitis 编译器配置文件	更新表格。
launch_emulator 实用工具	更新章节。
package_xo 命令	更新 -kernel_files 描述。
RTL 内核 XML 文件	更新 hwControlProtocol 描述。
xbutil 实用工具	添加脚本描述。
xbmgmt 实用工具	添加提示注释。
xclbinutil 实用工具	更新表格。
xrt.ini 文件	更新“调试和仿真”表。

使用 Vitis 分析器修订历史

下表显示了 第六部分：使用 Vitis 分析器 的修订历史。

章节	修订综述
2021 年 7 月 19 日 2021.1 版	
添加硬件接口	在“一般要求”中添加了重要注释。
为可扩展 XSA 启用硬件仿真	添加了重要注释 #3。
验证嵌入式平台	新增。
2021 年 6 月 16 日 2021.1 版	
配置 Vitis 分析器	更新“运行汇总”。
比对两份时间线轨迹报告	更新图示。
平台框图和系统框图	更新 profile_summary.csv 并添加“器件映射”。
链接汇总：多种策略和时序报告	新增。
创建存档文件	更新 profile_summary.csv、timeline_trace.csv 和注释。

使用 Vitis IDE 修订历史

下表显示了 第七部分：使用 Vitis IDE 的修订历史。

章节	修订综述
2021 年 7 月 19 日 2021.1 版	
不适用	本节无更改。
2021 年 6 月 16 日 2021.1 版	
Vitis IDE 的输出目录	更新 profile_summary.csv 和 timeline_trace.csv
vitis -debug 命令行	更新 -kernels。

章节	修订综述
Vitis 二进制容器设置	更新图示和描述。
Vitis IDE 调试流程	移除硬件和末尾的注释。

使用 Vitis 嵌入式平台修订历史

下表显示了 第八部分：使用 Vitis 嵌入式平台 的修订历史。

章节	修订综述
2021 年 7 月 19 日 2021.1 版	
平台类型	次要更新。
2021 年 6 月 16 日 2021.1 版	
平台类型	次要更新。
为可扩展 XSA 启用硬件仿真	更新 2b 和 2c。

附加信息

下表显示了 第九部分：附加信息 的修订历史。

章节	修订综述
2021 年 7 月 19 日 2021.1 版	
不适用	本节无更改。
2021 年 6 月 16 日 2021.1 版	
OpenCL 编程	新增。
理解 FPGA 架构	次要更新。
旧版本参考信息	新增。
自由运行的内核的编码指南	添加提示注释。

请阅读：重要法律提示

关于与汽车相关用途的免责声明

如将汽车产品（部件编号中含“XA”字样）用于部署安全气囊或用于影响车辆控制的应用（“安全应用”），除非有符合 ISO 26262 汽车安全标准的安全概念或冗余特性（“安全设计”），否则不在质保范围内。客户应在使用或分销任何包含产品的系统之前为了安全的目的全面地测试此类系统。在未采用安全设计的条件下将产品用于安全应用的所有风险，由客户自行承担，并且仅在适用的法律法规对产品责任另有规定的情况下，适用该等法律法规的规定。

版权声明