SDAccel Programmers Guide

UG1277 (v2019.1) May 22, 2019

See all versions of this document

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/22/2019 Version 2019.1	
SDAccel Build Process	Added kernel description to .xo file.
SDAccel Execution Model	Updated section.
Program	Updated code block and #3 description.
Sub-Devices	Updated section.
Executing Commands in the FPGA Device	Updated description.
Setting Up the Kernels	Updated section. Re-ordered section and removed Execution section.
Buffer Transfer to/from the FPGA Device	Updated section.
Kernel Execution	New section.
Post Processing and FPGA Cleanup	Updated section.
Enabling Host to Kernel Dataflow	Updated section.
Summary	Updated numbered list.
Connecting Kernel Ports to Global Memory	Updated note.
Appendix A: SDAccel Streaming Platform	New appendix.
01/24/2019 Version 2018.3	
Entire document	Editorial Updates.
12/05/2018 Version 2018.3	
Chapter 1: SDAccel Compilation Flow and Execution Model	Updated SDAccel Architecture figure.
Devices	Added sub-device description.
Buffer Transfer to/from the FPGA Device	Updated sections.
Summary	Updated #3 description.
Chapter 3: Programming C/C++ Kernels	Added description.
Memory Data Inputs and Outputs	Updated whole section.
Connecting Kernel Ports to Global Memory	Added note.
10/02/2018 Version 2018.2.xdf	
	No changes for release.
07/02/2018 Version 2018.2	
Connecting Kernel Ports to Global Memory	Updated syntax and requirements for `--sp` option.
06/06/2018 Version 2018.2	
Initial Xilinx® release.	N/A

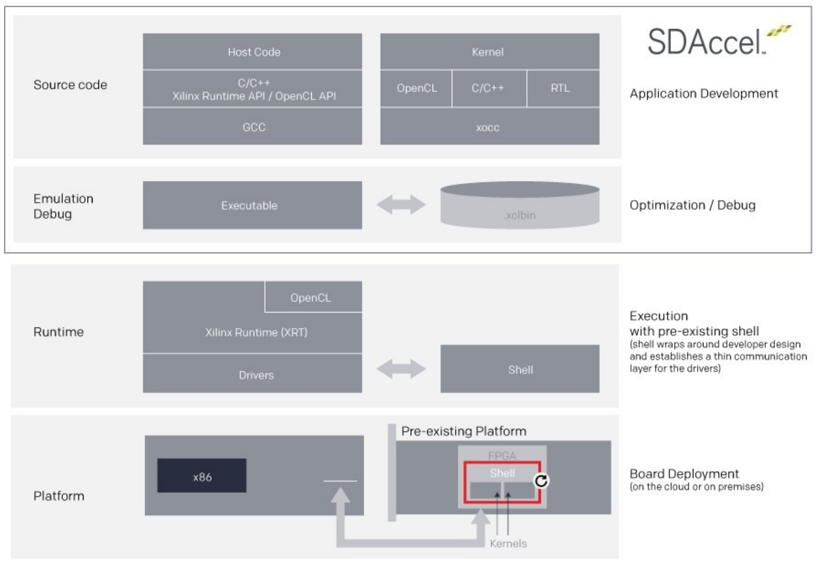
Table of Contents

Chapter 1

SDAccel Compilation Flow and Execution Model

The SDAccel™ development environment is a heterogeneous system architecture platform to accelerate compute intensive tasks using Xilinx® FPGA devices. The SDAccel environment contains a Host x86 machine that is connected to one or more Xilinx FPGA devices through a PCIe® bus, as shown below.

Figure 1: **SDAccel Architecture**



Programming Model

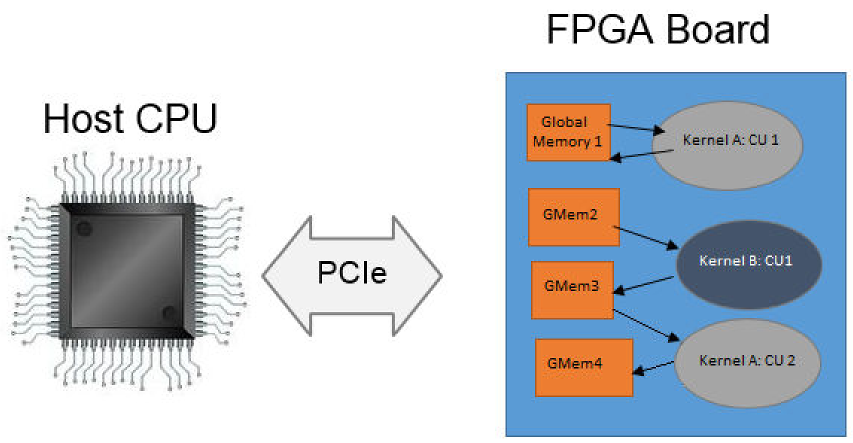
The SDAccel environment supports heterogeneous computing using the industry standard OpenCL protocol (https://www.khronos.org/opencl/). The host program executes on the Host CPU and offloads compute intensive tasks to execute on Xilinx FPGA devices using the OpenCL programming paradigm. Unlike a CPU (or GPU), the FPGA can be thought of as a blank canvas which does not have predefined instruction sets or fixed word size, and is capable of running the same or different instructions in parallel to greatly improve the performance of your applications.

Device Topology

In the SDAccel environment, devices are one or more FPGAs connected to a host x86 machine through a PCIe bus. The FPGA contains a programmable region that implements and executes kernels. The FPGA platform contains one or more global memory banks. The data transfer from the host machine to kernels and from kernels to the host happens through these global memory banks. The kernels running on the FPGA can have one or more memory interfaces. The connection from the global memory banks to those memory interfaces are configurable, as their features are determined by the kernel compilation options.

The programmable logic of Xilinx devices can implement multiple kernels at the same time, allowing for significant task parallelism. A single kernel can also be instantiated multiple times. The number of instances of a kernel is programmable and determined by the kernel compilation options.

Figure 2: **SDAccel Architecture**



The diagram above illustrates the flexible connections from the host application to multiple kernels through the global memory banks. The FPGA board device shown above contains four DDR memory banks. The programmable logic of the FPGA is running two kernels, Kernel A and Kernel B. Each kernel has two memory interfaces one for reading the data and another for writing. Also, note that there are two instances of Kernel A, totaling three simultaneous kernel instances on the FPGA.

In the diagram, the first instance of Kernel A: CU1 uses a single memory interface for reading and writing. Kernel B and the second instance of Kernel A: CU2 use different memory interfaces for reading, and writing, with Kernel B essentially passing data directly to Kernel A: CU2 through the global memory.

RECOMMENDED: *To achieve the best performance, the global memory banks to kernel interface connections should carefully be defined as discussed in Connecting Kernel Ports to Global Memory.*

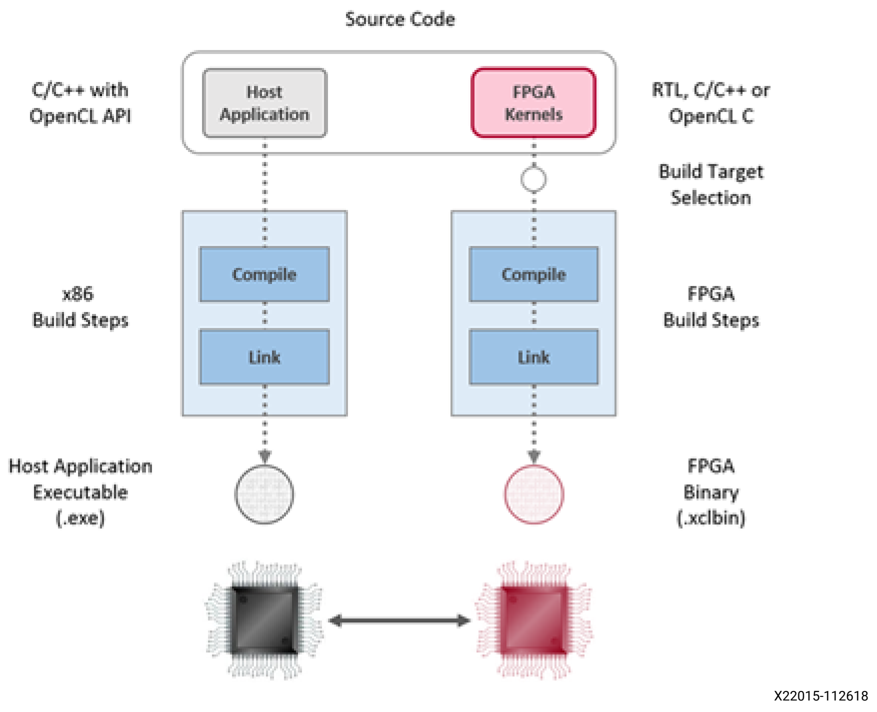
SDAccel Build Process

The SDAccel environment offers all of the features of a standard software development environment:

- Optimized compiler for host applications
- Cross-compilers for the FPGA
- Robust debugging environment to help identify and resolve issues in the code
- Performance profilers to identify bottlenecks and optimize the code

Within this environment, the build process uses a standard compilation and linking process for both the software elements, and the hardware elements of the project. As shown in the following figure, the host application is built through one process using standard GCC compiler, and the FPGA binary is built through a separate process using the Xilinx `xocc` compiler.

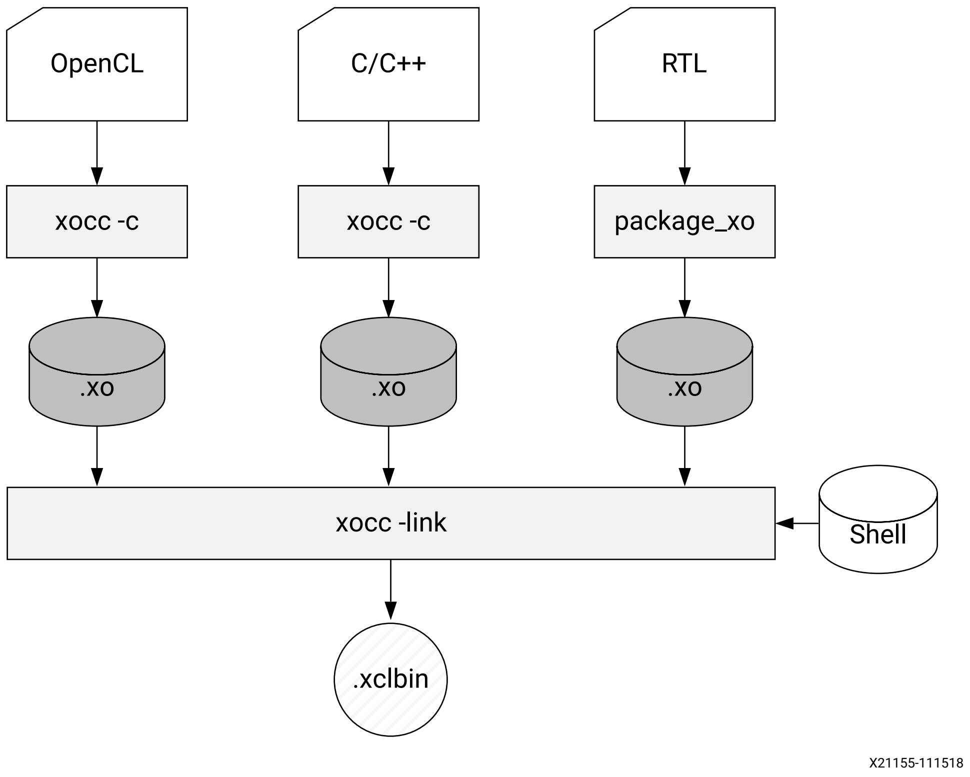
Figure 3: **Software/Hardware Build Process**



1. Host application build process using GCC:
   - Each host application source file is compiled to an object file (`.o`).
   - The object files (`.o`) are linked with the Xilinx SDAccel runtime shared library to create the executable (`.exe`).

2. FPGA build process is highlighted in the following figure:
    - Each kernel is independently compiled to a Xilinx object (`.xo`) file.
        - C/C++ and OpenCL C kernels are compiled for implementation on an FPGA using the `xocc` compiler. This step leverages the Vivado® HLS compiler. Pragmas and attributes supported by Vivado HLS can be used in C/C++ and OpenCL C kernel source code to specify the desired kernel micro-architecture and control the result of the compilation process.
        - RTL kernels are compiled using the `package_xo` utility. The RTL kernel wizard in the SDAccel environment can be used to simplify this process.
    - The kernel `.xo` files are linked with the hardware platform (shell) to create the FPGA binary (`.xclbin`). Important architectural aspects are determined during the link step. In particular, this is where connections from kernel ports to global memory banks are established and where the number of instances for each kernel is specified.
        - When the build target is software or hardware emulation, as described below, `xocc` generates simulation models of the device contents.
        - When the build target is the system (actual hardware), `xocc` generates the FPGA binary for the device leveraging the Vivado Design Suite to run synthesis and implementation.

Figure 4: **FPGA Build Process**



Note: The `xocc` compiler automatically uses the Vivado HLS and Vivado Design Suite tools to build the kernels to run on the FPGA platform. It uses these tools with predefined settings which have proven to provide good quality of results. Using the SDAccel environment and the `xocc` compiler does not require knowledge of these tools; however, hardware-savvy developers can fully leverage these tools and use all their available features to implement kernels.

Build Targets

The SDAccel tool build process generates the host application executable (`.exe`) and the FPGA binary (`.xclbin`). The SDAccel build target defines the nature of FPGA binary generated by the build process.

The SDAccel tool provides three different build targets, two emulation targets used for debug and validation purposes, and the default hardware target used to generate the actual FPGA binary:

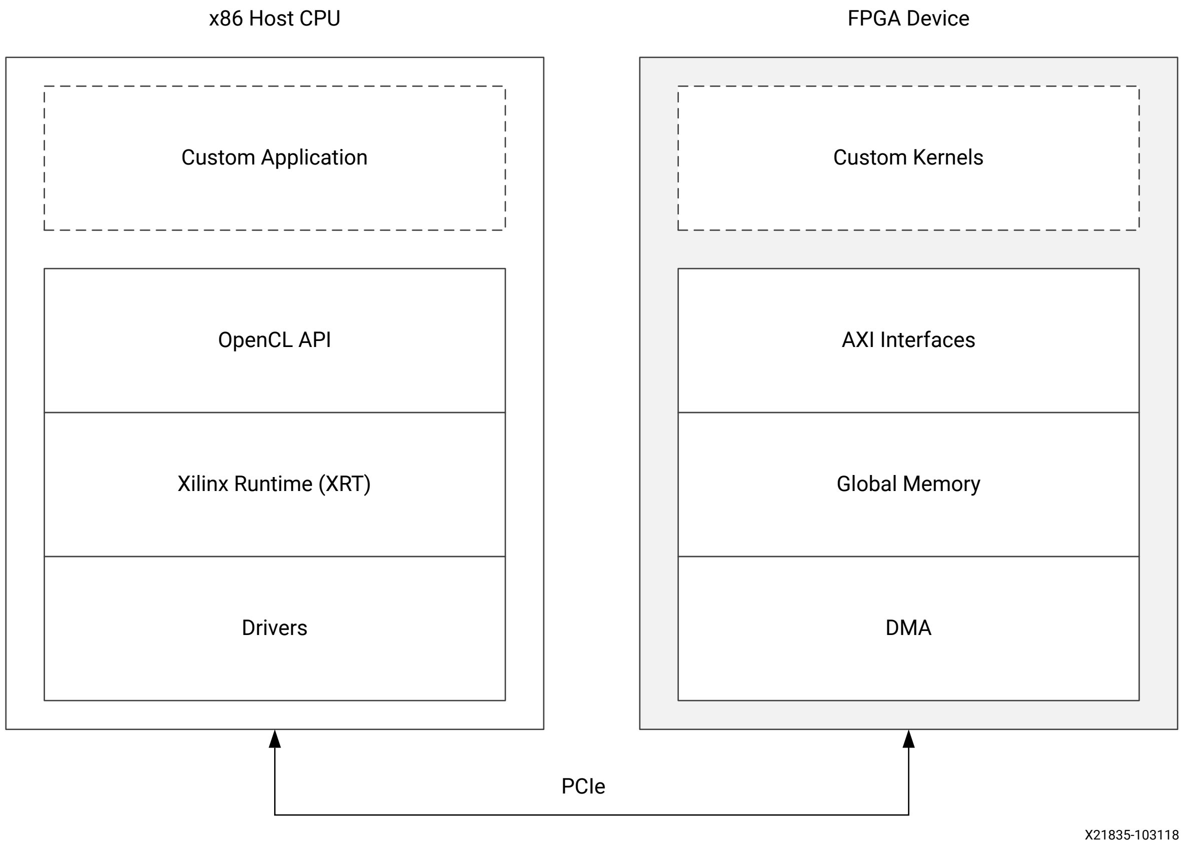
- **Software Emulation (`sw_emu`):** Both the host application code and the kernel code are compiled to run on the x86 processor. This allows iterative algorithm refinement through fast build-and-run loops. This target is useful for identifying syntax errors, performing source-level debugging of the kernel code running together with application, and verifying the behavior of the system.
- **Hardware Emulation (`hw_emu`):** The kernel code is compiled into a hardware model (RTL) which is run in a dedicated simulator. This build and run loop takes longer but provides a detailed, cycle-accurate, view of kernel activity. This target is useful for testing the functionality of the logic that will go in the FPGA and for getting initial performance estimates.
- **System (`hw`):** The kernel code is compiled into a hardware model (RTL) and is then implemented on the FPGA device, resulting in a binary that will run on the actual FPGA.

SDAccel Execution Model

In the SDAccel framework, an application program is split between a host application and hardware accelerated kernels with a communication channel between them. The host application, written in C/C++ and using API abstractions like OpenCL, runs on an x86 server while hardware accelerated kernels run within the Xilinx FPGA. The API calls, managed by the Xilinx Runtime (XRT), are used to communicate with the hardware accelerators. Communication between the host x86 machine and the accelerator board, including control and data transfers, occurs across the PCIe bus. While control information is transferred between specific memory locations in hardware, global memory is used to transfer data between the host application and the kernels. Global memory is accessible by both the host processor and hardware accelerators, while host memory is only accessible by the host application.

For instance, in a typical application, the host will first transfer data, to be operated on by the kernel, from host memory into global memory. The kernel would subsequently operate on the data, storing results back to the global memory. Upon kernel completion, the host would transfer the results back into the host memory. Data transfers between the host and global memory introduce latency which can be costly to the overall acceleration. To achieve acceleration in a real system, the benefits achieved by hardware acceleration kernels must outweigh the extra latency of the data transfers. The general structure of this acceleration platform is shown in the following figure.

Figure 5: **Architecture of an SDAccel Application**



The FPGA hardware platform, on the right-hand side, contains the hardware accelerated kernels, global memory along with the DMA for memory transfers. Kernels can have one or more global memory interfaces and are programmable. The SDAccel execution model can be broken down into these steps:

1. The host application writes the data needed by a kernel into the global memory of the attached device through the PCIe interface.
2. The host application sets up the kernel with its input parameters.
3. The host application triggers the execution of the kernel function on the FPGA.

4. The kernel performs the required computation while reading data from global memory, as necessary.
5. The kernel writes data back to global memory and notifies the host that it has completed its task.
6. The host application reads data back from global memory into the host memory and continues processing as needed.

The FPGA can accommodate multiple kernel instances at one time; this can occur between different types of kernels or multiple instances of the same kernel. The XRT transparently orchestrates the communication between the host application and the kernels in the accelerator. The number of instances of a kernel is determined by compilation options.

SDAccel Emulation Flows

The SDAccel environment development flow can be divided into two distinct steps. The first step is to compile the host and kernel code to generate the executables. The second step is to run the executables in a heterogeneous system comprised of the Host CPU and SDAccel environment accelerator platform. However, the kernel compilation process is long and can take several hours depending on the size of the kernels and the architecture of the target FPGA. Therefore, to save time and shorten the debug cycle before the kernel compilation process, the SDAccel environment provides two other build targets for testing purposes: software emulation and hardware emulation. The compilation and execution of these emulation targets are significantly faster, and do not require the actual FPGA board to be run. These emulation flows abstract the FPGA board, and its connection to the host machine, into software models to validate the combined functionality of the host and kernel code, as well as providing some performance estimates early in the design process. These performance estimates are just estimates, but they can greatly help debugging and identifying performance bottlenecks. Refer to the *SDAccel Environment Debugging Guide* (UG1281) for more information on debugging using software and hardware emulation flows.

Software Emulation Flow

Compilation of the software emulation target is the fastest. It is mainly used for checking the functional correctness when the host and kernel code are running together. The `xocc` compiler does the minimum transformation of the kernel code in order to run it in conjunction with the host code, so this software emulation flow helps to check functional correctness at the very early stage of the final binary creation. The software emulation flow can be used for algorithm refinement, debugging functional issues, and letting developers iterate quickly through the code to make improvements.

Hardware Emulation Flow

In the hardware emulation flow, the `xocc` compiler generates a model of the kernel in a hardware description language (RTL Verilog). The hardware emulation flow helps to check the functional correctness of the final binary creation after synthesis of the RTL from the C, C++, or OpenCL kernel code. The hardware emulation flow offers great debug capability with the waveform viewer if the system does not work as expected.

Table 1: **Comparison of Emulation Flows with Hardware Execution**

Software Emulation	Hardware Emulation	Hardware Execution
Host application runs with a C/C++ or OpenCL model of the kernels.	Host application runs with a simulated RTL model of the kernels.	Host application runs with actual hardware implementation of the kernels.
Confirm functional correctness of the system.	Test the host / kernel integration, get performance estimates.	Confirm that the system runs correctly and with desired performance.
Fastest turnaround time.	Best debug capabilities, moderate compilation time.	Final FPGA implementation and run provides accurate performance result with long build time.

SDAccel Example Designs

SDAccel Examples on GitHub

Xilinx provides many examples for programming with the SDAccel environment in the GitHub repository to help beginning users get familiar with the coding style of host and kernel code, and for more experienced users to use as a source of coding examples. All examples are available with host code, kernel code, and Makefile associated with the compilation flow and runtime flow. The following is one such example to get a basic understanding of the file structure of a standard example.

Inverse Discrete Cosine Transform (IDCT) Example

Look at the IDCT example design that demonstrates the key coding organization required for the SDAccel environment.

The `Readme.md` file discusses in detail how to run this example in both emulation and FPGA execution flows using the provided `Makefile`.

Inside the `./src` directory, you can find host code `idct.cpp`, and kernel code `krnl_idct.cpp`.

In the following chapters you will learn the basic required knowledge to program the host code and kernel code for the SDAccel environment. During this process you might refer to the above design as an example.

Organization of this Guide

The remaining chapters are organized as follows:

- Chapter 2: Programming the Host Application: Describes writing host code in C or C++ using the OpenCL API targeted for Xilinx FPGA devices. This chapter assumes the user has prior knowledge of OpenCL. It discusses coding practices to follow when writing an effective host application interfacing with acceleration kernels running on Xilinx FPGA devices.
- Chapter 3: Programming C/C++ Kernels: Describes different elements of writing high-performance, compute-intensive kernel code to implement on an FPGA device.
- Chapter 4: Configuring the System Architecture: Discusses integrating and connecting the host application to one or more kernel instances during the linking process.

Chapter 2

Programming the Host Application

In the SDAccel™ environment, host code is written in C or C++ language using the industry standard OpenCL™ API. The SDAccel environment provides an OpenCL 1.2 embedded profile conformant runtime API.

Note: The SDAccel environment supports the OpenCL Installable Client Driver (ICD) extension (`cl_khr_icd`). This extension allows multiple implementations of OpenCL to co-exist on the same system. Refer to Appendix B: OpenCL Installable Client Driver Loader for details and installation instructions.

The SDAccel environment consists of a host x86 CPU and compute devices running on a Xilinx® FPGA.

In general, the structure of the host code can be divided into three sections:

1. Setting up the environment.
2. Core command execution including executing one or more kernels.
3. Post processing and FPGA release.

The following sections discuss each of the above topics in detail.

Note: For multithreading the host program, exercise caution when calling a `fork()` system call from an SDAccel environment application. The `fork()` does not duplicate all the runtime threads. Hence the child process cannot run as a complete application in the SDAccel environment. It is advisable to use the `posix_spawn()` system call to launch another process from the SDAccel environment application.

Setting Up the OpenCL Environment

The host code in the SDAccel environment follows OpenCL programming paradigm. To set the environment properly, the host application should identify the standard OpenCL models. They are: platform, devices, context, command queue, and program.

TIP: *The host code examples and API commands used in this document follow the OpenCL C API. The* IDCT example *referred to in* SDAccel Example Designs *is also written with the C API. However, the SDAccel runtime environment also supports the OpenCL C++ wrapper API, and many of the examples in the* GitHub repository *are written using the C++ API. Refer to* https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf *for more information on this C++ wrapper API.*

Platform

From the very beginning the host code should identify the platform composed of Xilinx FPGA as one or more devices. The host code segment below is standard coding to identify the Xilinx device based platform.

```
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];
  }
}
```

The OpenCL API call clGetPlatformIDs is used to discover the set of available OpenCL platforms for a given system. Thereafter, clGetPlatformInfo is used to identify the Xilinx device based platform by matching `cl_platform_vendor` with the string `"Xilinx"`.

RECOMMENDED: *Though it is not explicitly shown in the preceding code, or in other host code examples used throughout this chapter, it is always a good coding practice to use error checking after each of the OpenCL API calls. This can help debugging and improve productivity when you are debugging the host and kernel code in the emulation flow, or during hardware execution. Below is an error checking code example for `clGetPlatformIDs` command:*

```
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);
}
```

Devices

After the platform detection, the Xilinx FPGA devices attached to the platform are identified. The SDAccel environment supports one or more Xilinx FPGA devices working together.

The following code demonstrates finding all the Xilinx devices (with a upper limit of 16) by using API clGetDeviceIDs and printing their names.

```
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);
}
```

IMPORTANT! *The `clGetDeviceIDs` API is called with the `device_type`and `CL_DEVICE_TYPE_ACCELERATOR` to receive all the available Xilinx devices.*

Sub-Devices

In the SDAccel environment, sometimes devices contain multiple kernel instances of a single kernel or of different kernels. The OpenCL API clCreateSubDevices allows the host code to divide the device into multiple sub-devices containing one kernel instance per sub-device. Currently, the SDAccel environment supports equally divided sub-devices each containing only one kernel instance.

The following example shows:

1. The sub-devices are created by equal partition to execute one kernel instance per sub-device.
2. Iterating over the sub-device list and using a separate context and command queue to execute the kernel on each of them.
3. The API related to kernel execution (and corresponding buffer related) code is not shown for the sake of simplicity, but would be described inside the function `run_cu`.

```
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
  });
```

IMPORTANT! *As shown in the above example, create separate context for each and every sub-devices. Though OpenCL allows to create a context that can hold multiple devices and sub-devices, for Xilinx Runtime it is recommended to separate each device and sub-device into a separate context.*

Context

The OpenCL context creation process is straightforward. The API clCreateContext is used to create a context that contains one or more Xilinx devices that will communicate with the host machine.

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

In the code example above, the API `clCreateContext` is used to create a context that contains one Xilinx device. You can create only one context for a device from a host program. However, the host program should use multiple contexts if sub-devices are used; one context for each sub-device.

Command Queues

One or more command queues for each device is created using the clCreateCommandQueue API. The FPGA device can contain multiple kernels. When developing the host application, there are two main programming approaches to execute kernels on a device:

1. Single out-of-order command queue: Multiple kernel executions can be requested through the same command queue. The SDAccel runtime environment dispatches those kernels as soon as possible in any order allowing concurrent kernel execution on the FPGA.
2. Multiple in-order command queue: Each kernel execution will be requested from different in-order command queues. In such cases, the SDAccel runtime environment can dispatch kernels from any command queue with the intention of improving performance by running them concurrently on the FPGA.

The following is an example of standard API calls to create in-order and out-of-order command queues.

```
// 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);
```

Program

As described in the SDAccel Build Process, the host and kernel code are compiled separately to create separate executable files: the host application (`.exe`) and the FPGA binary (`.xclbin`). When the host application is executed it must load the `.xclbin` using the clCreateProgramWithBinary API.

The following code example shows how the standard OpenCL API is used to build the program from the `.xclbin` file:

```
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;
}
```

The above example performs the following steps:

1. The kernel binary file, `.xclbin`, is passed in from the command line argument, `argv[1]`.

TIP: *Passing the `.xclbin` through a command line argument is specific to this example. You can also hardcode the kernel binary file in the application, use an environment variable, read it from a custom initialization file or any other suitable mechanism.*

2. The `load_file_to_memory` function is used to load the file contents in the host machine memory space.
3. The API `clCreateProgramWithBinary` is used to complete the program creation process.

Executing Commands in the FPGA Device

Once the OpenCL environment is initialized, the host application is ready to issue commands to the device and interact with the kernels. Such commands include:

1. Setting up the kernels.
2. Buffer transfer to/from the FPGA.
3. Kernel execution on FPGA.
4. Event synchronization.

Setting Up the Kernels

After the initialization of all the preliminaries such as context, command queues, and program, the host application should identify the kernels required to execute on the device and setting up their arguments.

Kernels Identification

At the beginning, the `clCreateKernel` API should be used to access the kernels present inside the `.xclbin` file. The kernel handle (`cl_kernel` type) denotes a kernel object that now can be used in the rest of the host program.

```
kernel1 = (program, "<kernel_name_1>", &err);
kernel2 = clCreateKernel(program, "<kernel_name_2>", &err);  // etc
```

Setting Kernel Arguments

In the SDAccel environment framework, two types of kernel arguments can be set:

1. The scalar arguments are used for small data transfer, such as constant or configuration type data. These are write-only arguments.

2. The buffer arguments are used for large data transfer.

The kernel arguments can be set using the clSetKernelArg command as shown below. The following example shows setting kernel arguments for two scalar and two buffer arguments.

```
cl_mem dev_buf1 = clCreateBuffer(context, CL_MEM_WRITE, size,
&host_mem_ptr1, NULL);
cl_mem dev_buf2 = clCreateBuffer(context, CL_MEM_READ, size,
&host_mem_ptr2, NULL);

int err = 0;
// Setting up 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);

// Setting up buffer arguments
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_buf1);
err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_buf2);
```

IMPORTANT! *Though OpenCL allows setting kernel arguments any time before the kernel enqueuing, Xilinx recommends setting kernel arguments as early as possible. This helps Xilinx Runtime determine the buffer location on the device. Therefore, set the kernel arguments before performing any enqueue operation (for example,* `clEmqueueMigrateMemObjects`*) on any buffer.*

Buffer Transfer to/from the FPGA Device

Interactions between the host application and kernels rely on transferring data to and from global memory in the device. The method to send data back and forth from the FPGA is using clCreateBuffer, clEnqueueWriteBuffer, and clEnqueueReadBuffer commands.

RECOMMENDED: *Xilinx recommends using* `clEnqueueMigrateMemObjects` *instead of* `clEnqueueReadBuffer` *and* `clEnqueueWriteBuffer`*.*

The following code example demonstrates this:

```
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,
                     sizeof(int) * number_of_words, NULL, NULL);

err = clEnqueueWriteBuffer(commands, dev_mem_ptr, CL_TRUE, 0,
      sizeof(int) * number_of_words, host_mem_ptr, 0, NULL, NULL);
```

IMPORTANT! *A single buffer cannot be bigger than 4 GB.*

Note: To maximize throughput from host to global memory, sending very small buffers is not very effective. Xilinx recommends keeping the buffer size at least 2 MB if possible.

For simple applications the example code above would be sufficient to transfer data from the host to the device memory. However, there are a number of coding practices you should adopt to maximize performance and fine-grain control.

Using clEnqueueMigrateMemObjects

Xilinx recommends using clEnqueueMigrateMemObjects instead of `clEnqueueWriteBuffer` or `clEnqueueReadBuffer` to improve the performance. There are two main parts of a `cl_mem` object: host side pointer and device side pointer. Before the kernel starts its operation, the device side pointer is implicitly allocated on the device side memory (for example, on a specific location inside the device global memory) and the buffer becomes a resident on the device. However, by using `clEnqueueMigrateMemObjects` this allocation and data transfer occur upfront, much ahead of the kernel execution. This especially helps to enable *software pipelining* if the host is executing the same kernel multiple times, because data transfer for the next transaction can happen when kernel is still operating on the previous data set, and thus hide the data transfer latency of successive kernel executions.

The following code example is modified to use `clEnqueueMigrateMemObjects`:

```
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);
```

Allocating Page-Aligned Host Memory

Xilinx Runtime allocates the memory space in 4K boundary for internal memory management. If the host memory pointer is not aligned to a page boundary, the Xilinx Runtime performs extra `memcpy` to make it aligned. Hence you should align the host memory pointer with the 4K boundary to save the extra memory copy operation.

The following is an example of how `posix_memalign` is used instead of `malloc` for the host memory space pointer.

```
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 ,
                     sizeof(int) * number_of_words, host_mem_ptr, NULL);

err = clEnqueueMigrateMemObjects(commands, 1, dev_mem_ptr, 0, 0,
      NULL, NULL);
```

Using clEnqueueMapBuffer

Another approach for creating and managing buffers is to use `clEnqueueMapBuffer`. With this approach, it is not necessary to create a host space pointer aligned to 4K boundary. The `clEnqueueMapBuffer` API maps the specified buffer and returns a pointer created by the Xilinx Runtime to this mapped region. Then, fill the host side pointer with your data, followed by `clEnqueueMigrateMemObject` to transfer the data to and from the device. Below is an example that uses this style. Note `CL_MEM_USE_HOST_PTR` is not used for `clCreateBuffer`.

```
// Two cl_mem buffer, for read and write by kernel
cl_mem dev_mem_read_ptr = clCreateBuffer(context,
                     CL_MEM_READ_ONLY,
                     sizeof(int) * number_of_words, NULL, NULL);

cl_mem dev_mem_write_ptr = clCreateBuffer(context,
                     CL_MEM_WRITE_ONLY,
                     sizeof(int) * number_of_words, NULL, NULL);


// 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
```

Buffer Allocation on the Device

By default, all the memory interfaces from all the kernels are connected to a single default global memory bank when kernels are linked. As a result, only one compute unit (CU) can transfer data to and from the global memory bank at a time, limiting the overall performance of the application. If the FPGA contains only one global memory bank, then this is the only option. However, if the device contains multiple global memory banks, you can customize the global memory bank connections by modifying the default connection. This topic is discussed in greater detail in Connecting Kernel Ports to Global Memory. Overall performance is improved by enabling multiple kernel memory interfaces to concurrently read and write data from separate global memory banks.

As in the SDAccel environment the host code and the kernel code are compiled independently. Xilinx Runtime needs to detect the kernel's memory connection to send the data to the correct memory location from the host code. The latest 2019.1 Xilinx Runtime will automatically find the buffer location from the kernel binary files if `clSetKernelArgs` is used before any enqueue operation on the buffer, for example `clEnqueueMigrateMemObject`.

Before the 2019.1 release, the OpenCL host code required a Xilinx extension (`cl_mem_ext_ptr`) to specify the exact buffer location on the device. Though this method is still supported, it is no longer necessary and is not documented in this version of the guide. For more information on specifying buffer location using `cl_mem_ext_ptr`, see the earlier version of this guide.

Sub-Buffers

Though not very common, using sub-buffer can be very useful in specific situations. The following sections discuss the scenarios where using sub-buffers can be beneficial.

Reading a Specific Portion from the Device Buffer

Consider a kernel that is producing different amounts of data depending on the input to the kernel. A simple example can be a compression engine where the output size varies depending on the input data pattern and similarity. The host can still read the whole output buffer by using `clEnqueueMigrateMemObjects`, but that is a suboptimal approach as more than required memory transfer would take place. Ideally the host only needs to read the exact amount of data that kernel has written. One of the techniques can be adopted by kernel is to write a size information of the output data at the start of the output written data. If using `clEnqueueReadBuffer`, the host code can use `clEnqueueReadBuffer` two times, first for reading the size of the data, and the second to read exact amount of data by using the size information from the first read.

```
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);
```

With `clEnqueueMigrateMemObject`, which is recommended over `clEnqueueReadBuffer` or `clEnqueueWriteBuffer`, you can adopt a similar approach by using sub-buffers. The following sample code is shown below (note that this is not the complete API arguments).

```
//Create a small subbuffer
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 subbuffer into the host space
auto size_info_host_ptr = clEnqueueMapBuffer(queue,size_info,,,, );

// Read only the subbuffer 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 again for required amount 
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
```

Device Buffer Shared by Multiple Memory Ports or Multiple Kernels

Sometimes memory ports of the kernels only require small amounts of data. Managing and sending multiple number of small sized buffers can have potential performance issues. Alternatively, the host can create a large size buffer divided into small sub-buffers. Each sub-buffer assigns a kernel argument for each of the memory ports which requires small amounts of data. This can improve performance as Xilinx Runtime handles a large buffer instead of several small buffers.

Once sub-buffers are created they are used in the host code similar to regular buffers.

Kernel Execution

Assuming the kernel is compiled to a single hardware instance (or CU) on the FPGA, the simplest method of executing the kernel is using `clEnqueueTask` as shown below.

```
err = clEnqueueTask(commands, kernel, 0, NULL, NULL);
```

The Xilinx Runtime schedules the workload (the data passed through OpenCL buffers through the kernel arguments) and schedules the kernel to compute intensive tasks on the FPGA.

IMPORTANT! *Though using `clEnqueueNDRangeKernel` is supported (only for OpenCL kernel), Xilinx recommends using `clEnqueueTask`.*

There are various methods you can execute the kernel, multiple kernels, or multiple instances of the same kernel on the FPGA. Those are discussed in the next section.

Single Kernel Invocation for the Entire Task and Data Parallelism

Often the complete compute intensive task is defined inside a single kernel and the kernel is executed only one time to work on the entire data range. As there is a overhead of multiple kernel executions, this approach certainly helps in many cases. Though the kernel is executed only one time and works on the entire range of the data, the parallelism (and thereby acceleration) is achieved on the FPGA inside the kernel hardware. Most of the time (if properly coded), kernel is capable of achieving parallelism by various technique such as instruction-level parallelism (loop pipeline) and function-level parallelism (dataflow). You will learn about different kernel coding techniques in Chapter 3: Programming C/C++ Kernels.

However, the above mentioned single `clEnqueueTask` is not always feasible due to various practical reasons. For example, the kernel code can become too big and complex to optimize if it attempts to perform all compute intensive task in a single execution. Another possible case is when the host is receiving data over time and not all the data at the same time. Therefore, depending on the situation and application, there are different ways to use `clEnqueueTask` to break the data and the task into multiple `clEnqueueTask` commands as discussed in the next sections.

Task Parallelism by Using Multiple Different Kernels

Sometimes multiple kernels can be designed performing different task on the FPGA in parallel. By using the multiple `clEnqueueTask` command (through a out-of-order command queue), it is possible to allow multiple kernels (performing different task) working in parallel on the FPGA. This enables the task parallelism on the FPGA.

Spatial Data Parallelism: Increase Number of Compute Units

If a single kernel has been compiled into multiple hardware instances (or CUs), `clEnqueueTask` can be called multiple times (using a out-of-order queue) to enable data parallelism. Each call of `clEnqueueTask` would schedule a workload in different CUs working on the different data sets in parallel.

Temporal Data Parallelism: Host to Kernel Dataflow

To understand this approach, assume a kernel has only one CU on the FPGA and the host requires to use the CU multiple times on different sets of data. As shown in Using clEnqueueMigrateMemObjects, by using `clEnqueueMigrateMemObjects` it is possible to send data to the device global memory ahead of time (the data is transferred for the next kernel execution), and thus hiding the data transfer latency by the kernel execution, enabling *software pipelining*.

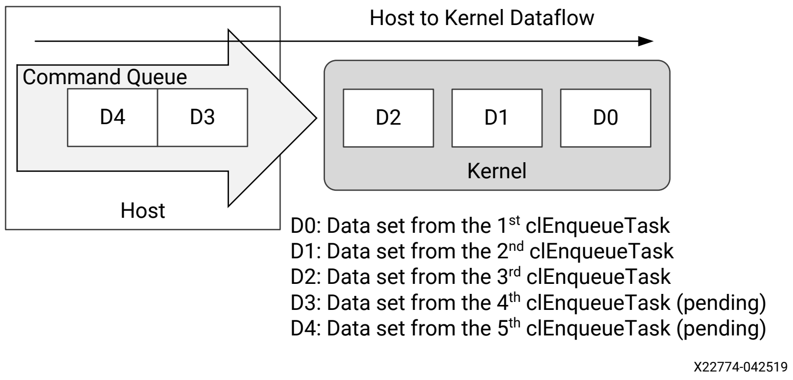
However, by default, the kernel can start operating on the next set of data only when it is finished working on the current set of data. Though `clEnqueueMigrateMemObject` hides the data transfer execution time, the kernel executions still remain sequential.

By enabling the host to kernel dataflow, it is even possible to further improve the performance by restarting the kernel while the kernel is still working on the previous sets of data. If the kernel is optimized in a manner such that it is capable of accepting the new data (for the next kernel operation) even when it is still working on the previous data (to achieve this the kernel has to be compiled in a certain manner, see Enabling Host to Kernel Dataflow), the XRT restarts the kernel as soon as possible, thus overlapping the multiple kernel executions.

This allows *temporal parallelism* between host to kernel where each section of the kernel hardware is working on a specific data set from the different `clEnqueueTask` command in a pipelined manner. However, the host still needs to fill the command queue ahead of the time (by software pipelining) so that kernel can restart as soon as it is ready to accept the new set of data.

The following is a conceptual diagram for the host to kernel dataflow.

Figure 6: **Host to Kernel Dataflow**


Host to Kernel Dataflow
Command Queue
D4
D3
Host
D2
D1
D0
Kernel
D0: Data set from the 1st clEnqueueTask
D1: Data set from the 2nd clEnqueueTask
D2: Data set from the 3rd clEnqueueTask
D3: Data set from the 4th clEnqueueTask (pending)
D4: Data set from the 5th clEnqueueTask (pending)
X22774-042519

For advanced designs, you can effectively use both the spatial parallelism (using more hardware resources or CUs) and software pipeline (`clEnqueueMigrateMemObjects`) combined with temporal parallelism (by host to kernel dataflow, particularly overlapping kernel executions on each CU). If needed, you can potentially combine all the techniques together.

Symmetrical and Asymmetrical Compute Units

During the kernel linking process, a kernel can have multiple CUs on the FPGA.

Symmetrical Compute Units

CUs are considered asymmetrical when they do not have identical connections to global memory (when they do not have exactly the same `--sp` options). As a result, the Xilinx Runtime can use them interchangeably. A call to `clEnqueueTask` can result in the invocation of any one instance in a group of symmetrical CUs.

Asymmetrical Compute Units

CUs are considered asymmetrical when they do not have identical connections to global memory (when they do not have exactly the same `--sp` options). Using the same setup of the input (and output) buffers, it is not possible to execute both of these CUs interchangeably. So these are not execution agnostic from the Xilinx Runtime perspective.

Kernel Handle and Compute Units

The first time `clSetKernelArg` is called for a given kernel object, the Xilinx Runtime selects a group of symmetrical CUs for the subsequent executions of this kernel. When `clEnqueueTask` is called, any of the symmetrical CUs in that group can be used.

If all CUs for a given kernel are symmetrical, a single kernel object is sufficient to access any of these CUs. If there are asymmetrical CUs, the application will need to create as many kernel objects as there are groups of asymmetrical CUs to ensure all of them can be used.

Creating Kernel Objects for Specific Compute Units

From the 2019.1 release, the Xilinx Runtime provides the capability to get kernel handles for specific CUs or a group of CUs. The syntax of this style is shown below:

```
// Create kernel object only for a specific compute unit
kernel1 = clCreateKernel(program, "<kernel_name_1>:{comp_unit_name_1}",
&err);

// Create kernel object for two specific compute units
kernel1 = clCreateKernel(program, "<kernel_name_1>:
{comp_unit_name_1,comp_unit_name_2}", &err);
```

This gives control within the application over which specific CU instance is used. This can be useful in the case of asymmetrical CUs or to perform explicit load and priority management of CUs.

Using Compute Unit Name to Get Handle of All Asymmetrical Compute Units

If a kernel has CUs that are not all symmetrical, the enhanced `clCreateKernel` with the CU name can be used. In this case, the host needs to manage each symmetrical CU group separately with different `cl_kernel` handle. The following shows a hypothetical example.

Assume the kernel `mykernel` has five CUs: K1, K2, K3, K4, and K5. Also consider the CUs K1, K2, and K3 are having symmetrical connection on the device and can be considered as a group of symmetrical CUs. Similarly, CUs named K4 and K5 form another group of symmetrical CU. The code segment below shows how two `cl_kernel` handles are used to manage the two groups of symmetrical CUs.

```
// Kernel handle for Symmetrical compute unit group 1: K1,K2,K3

  cl_kernel kernel_handle1 = 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, kernel_handle1, 0, NULL, NULL);


     //
   }

  // Kernel handle for Symmetrical compute unit group 1: K4, K5

  cl_kernel kernel_handle2 = 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, kernel_handle2, 0, NULL, NULL);
  }
```

Event Synchronization

All OpenCL `clEnqueueXXX` API calls are asynchronous. These commands will return immediately after the command is enqueued in the command queue. To resolve the dependencies among the commands, an API call such as clWaitForEvents or clFinish can be used to pause or block execution of the host program.

Example usage of `clWaitForEvents` and `clFinish` commands are shown below:

```
err = clEnqueueTask(command_queue, kernel, 0, NULL, NULL);

// Execution will wait here until all commands in the command queue are
finished
clFinish(command_queue);

// Read back the results from the device to verify the output
cl_event readevent;
int host_mem_output_ptr[MAX_LENGTH]; // host memory for output vector

clEnqueueReadBuffer(command_queue, dev_mem_ptr, CL_TRUE, 0, sizeof(int) *
number_of_words,
  host_mem_output_ptr, 0, NULL, &readevent );

clWaitForEvents(1, &readevent); // Wait for clEnqueueReadBuffer event to
finish

// Check Results
// Compare Golden values with host_mem_output_ptr
```

Note how the synchronization APIs have been added in the above example.

1. The `clFinish` API has been explicitly used to block the host execution until the Kernel execution is finished. This is necessary otherwise the host can attempt to read back from the FPGA buffer too early and may read garbage data.
2. The data transfer from FPGA memory to the local host machine is done through `clEnqueueReadBuffer`. Here the last argument of `clEnqueueReadBuffer` returns an event object that identifies this particular read command and can be used to query the event, or wait for this particular command to complete. The `clWaitForEvents` specifies that one event, and waits to ensure the data transfer is finished before checking the data from the host side memory.

Post Processing and FPGA Cleanup

At the end of the host code, all the allocated resources should be released by using proper release functions. The SDAccel environment may not able to generate a correct performance related profile and analysis report if resources are not properly released.

```
clReleaseCommandQueue(Command_Queue);
clReleaseContext(Context);
clReleaseDevice(Target_Device_ID);
clReleaseKernel(Kernel);
clReleaseProgram(Program);
free(Platform_IDs);
free(Device_IDs);
```

Summary

As discussed in earlier topics, the recommended coding style for the host application in the SDAccel environment includes the following points:

1. Add error checking after each OpenCL API call for debugging purpose, if required.
2. In the SDAccel environment, one or more kernels are separately pre-compiled to the `.xclbin` file. The API `clCreateProgramWithBinary` is used to build the program from the kernel binary.
3. Use buffer for setting the kernel argument (`clSetKernelArg`) before any enqueue operation on the buffer.
4. Transfer data back and forth from the host code to the FPGAs by using `clEnqueueMigrateMemObjects`.
5. Use `posix_memalign` to align the host memory pointer at 4K boundary.
6. Preferably use the out-of-order command queue for concurrent command execution on the FPGA.
7. Use synchronization commands to resolve dependencies of the asynchronous OpenCL API calls.

Chapter 3

Programming C/C++ Kernels

In the SDAccel™ environment, the kernel code is generally a compute-intensive part of the algorithm and meant to be accelerated on the FPGA. The SDAccel environment supports the kernel code written in C, OpenCL™, and also in RTL. This guide mainly focuses on the C kernel coding style.

During the runtime, the C/C++ kernel executable is called through the host code executable. As the host code and the kernel code are developed and compiled independently, there could be a name mangling issue if one of the code is written in C and another in C++. To avoid this issue, it is good practice to put the kernel function declaration inside a header file wrapped around the `extern "C"` linkage.

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

Data Types

As it is faster to write and verify the code by using native C data types such as `int`, `float`, or `double`, it is a common practice to use these data types when coding for the first time. However, the code is implemented in hardware, and all the operator sizes used in the hardware are dependent on the data types used in the accelerator code. The default native C/C++ data types can result in larger and slower hardware resources that can limit the performance of the kernel. Instead, consider using bit-accurate data types to ensure the code is optimized for implementation in hardware. Using bit-accurate, or arbitrary precision data types, results in hardware operators which are smaller and faster. This allows more logic to be placed into the programmable logic and also allows the logic to execute at higher clock frequencies while using less power.

Consider using bit-accurate data types instead of native C/C++ data types in your code.

RECOMMENDED: *Consider using bit-accurate data types instead of native C/C++ data types in your code.*

In the following sections, the two most common arbitrary precision data types (arbitrary precision integer type and arbitrary precision fixed-point type) supported by the `xocc` compiler are discussed. Note that these data types should be used for C/C++ kernels only, not for OpenCL kernel (or inside the host code).

Arbitrary Precision Integer Types

Arbitrary precision integer data types are defined by `ap_int` or `ap_uint` for signed and unsigned integer respectively inside the header file `ap_int.h`. To use arbitrary precision integer data type:

- Add header file `ap_int.h` to the source code.
- Change the bit types to `ap_int<N>` or `ap_uint<N>`, where N is a bit-size from 1 to 1024.

The following example shows how the header file is added and the two variables are implemented to use 9-bit integer and 10-bit unsigned integer.

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

Arbitrary Precision Fixed-Point Data Types

Some existing applications use floating point data types as they are written for other hardware architectures. However, fixed-point data types are a useful replacement for floating point types which require many clock cycles to complete. Carefully evaluate trade-offs in power, cost, productivity, and precision when choosing to implement floating-point vs. fixed-point arithmetic for your application and accelerators.

As discussed in *Deep Learning with INT8 Optimization on Xilinx Devices* (WP486), using fixed-point arithmetic instead of floating point for applications like machine learning can increase power efficiency, and lower the total power required. Unless the entire range of the floating-point type is required, the same accuracy can often be implemented with a fixed-point type resulting in the same accuracy with smaller and faster hardware. The paper *Reduce Power and Cost by Converting from Floating Point to Fixed Point* (WP491) provides some examples of this conversion.

Fixed-point data types model the data as an integer and fraction bits. The fixed-point data type requires the `ap_fixed` header, and supports both a signed and unsigned form as follows:

- Header file: `ap_fixed.h`
- Signed fixed point: `ap_fixed<W,I,Q,O,N>`
- Unsigned fixed point: `ap_ufixed<W,I,Q,O,N>`
    - W = Total width < 1024 bits

- I = Integer bit width. The value of I must be less than or equal to the width (W). The number of bits to represent the fractional part is W minus I. Only a constant integer expression can be used to specify the integer width.
- Q = Quantization mode. Only predefined enumerated values can be used to specify Q. The accepted values are:
  - `AP_RND`: Rounding to plus infinity.
  - `AP_RND_ZERO`: Rounding to zero.
  - `AP_RND_MIN_INF`: Rounding to minus infinity.
  - `AP_RND_INF`: Rounding to infinity.
  - `AP_RND_CONV`: Convergent rounding.
  - `AP_TRN`: Truncation. This is the default value when Q is not specified.
  - `AP_TRN_ZERO`: Truncation to zero.
- O = Overflow mode. Only predefined enumerated values can be used to specify O. The accepted values are:
  - `AP_SAT`: Saturation.
  - `AP_SAT_ZERO`: Saturation to zero.
  - `AP_SAT_SYM`: Symmetrical saturation.
  - `AP_WRAP`: Wrap-around. This is the default value when O is not specified.
  - `AP_WRAP_SM`: Sign magnitude wrap-around.
- N = The number of saturation bits in the overflow WRAP modes. Only a constant integer expression can be used as the parameter value. The default value is zero.

TIP: *The `ap_fixed` and `ap_ufixed` data types permit shorthand definition, with only W and I being required, and other parameters assigned default values. However, to define Q or N, you must also specify the parameters before those, even if you just specify the default values.*

In the example code below, the `ap_fixed` type is used to define a signed 18-bit variable with 6 bits representing the integer value above the binary point, and by implication, 12 bits representing the fractional value below the binary point. The quantization mode is set to round to plus infinity (`AP_RND`). Because the overflow mode and saturation bits are not specified, the defaults `AP_WRAP` and 0 are used.

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

When performing calculations where the variables have different numbers of bits (W), or different precision (I), the binary point is automatically aligned. See the "C++ Arbitrary Precision Fixed-Point Types" in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information on using fixed-point data types.

Interfaces

Two types of data transfer occur from the host machine to and from the kernels on the FPGA device: data transferred through the global memory memory banks, and scalar data.

Memory Data Inputs and Outputs

The main data processed by the kernel computation, often in a large volume, should be transferred through the global memory banks on the FPGA board. The host machine transfers a large chunk of data to one or more global memory bank(s). The kernel accesses the data from those global memory banks, preferably in burst. After the kernel finishes the computation, the resulting data is transferred back to the host machine through the global memory banks.

When writing the kernel interface description, pragmas are used to denote the interfaces coming to and from the global memory banks.

Memory Data

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

In the example above, there are three large data interfaces. The two inputs are `pixel` and `weights` and one output, `out`. These inputs and outputs connected to the global memory bank are specified in C code by using `HLS INTERFACE m_axi` pragmas as shown above.

The `bundle` keyword specifies the name of the port. The compiler will create a port for each unique bundle name. When the same name is used for different interfaces, this results in these interfaces being mapped to same port.

Sharing ports helps saves FPGA resources, but can limit the performance of the kernel because all the memory transfers have to go through a single port. The bandwidth and throughput of the kernel can be increased by creating multiple ports (using different bundle names).

```
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
```

In the above example, the `bundle` attribute is used to create two distinct ports: `gmem` and `gmem1`. The kernel will access `pixel` and `out` through the `gmem` port while `weights` will be accessed through the `gmem1` port. As a result, the kernel will be able to make parallel accesses to `pixel` and `weights`, potentially improving the throughput of the kernel.

For this performance potential to be fully realized, it is also necessary to connect these different ports to different global memory banks. This is done during the `xocc` linking stage using the `--sp` switch. For more details about this option, see the Configuring the System Architecture.

Memory Interface Data Width Considerations

In the SDAccel environment, the maximum data width from the global memory to and from the kernel is 512 bits. To maximize the data transfer rate, using this full data width is recommended. The kernel code should be modified to take advantage of the full bit width.

Because a native integer type is used in the prior example, the full data transfer bandwidth is not used. As discussed previously in Data Types, arbitrary precision data types `ap_int` or `ap_uint` can be used to achieve bit-accurate data width for this purpose.

```
void cnn( ap_uint<512> *pixel, // Input pixel
  int *weights, // Input Weight Matrix
  ap_uint<512> *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=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

The example above shows the output (`out`) interface using the `ap_uint` data type to make use of the full transfer width of 512 bits.

The data width of the memory interfaces should be a power of 2. For data width less than 32, use native C data types. Use `ap_int/ap_uint` for data widths greater than 32 and with power of 2 increment.

Reading and Writing by Burst

Accessing the global memory bank interface from the kernel has a large latency. So global memory data transfer should be done in burst. To infer the burst, a pipelined loop coding style is recommended as shown below:

```
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
}
```

In the above code example, a pipelined `for` loop is used to read data from the input memory interface, and writes to an internal `hls::stream` variable. The above coding style reads from the global memory bank in burst.

It is a recommended coding style to implement the `for` loop operation in the example above inside a separate function, and apply the `dataflow` optimization from the top level as shown below:

```
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); // 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
}
```

TIP: *The* Dataflow Optimization *is discussed in a later topic.*

Scalar Data Inputs

Scalar inputs are typically control variables that are directly loaded from the host machine. They can be thought of as programming data or parameters under which the main kernel computation takes place. These kernel inputs are write-only from the host side. These interfaces are specified in the kernel code as shown below:

```
void process_image(int *input, int *output, int width, int height) {
  #pragma HLS INTERFACE s_axilite port=width bundle=control
  #pragma HLS INTERFACE s_axilite port=height bundle=control
```

In the example above, there are two scalar inputs specify the image `width` and `height`. These inputs are specified using the `#pragma HLS INTERFACE s_axilite`. These data inputs come to the kernel directly from the host machine and not using global memory bank.

IMPORTANT! *Currently, the SDAccel environment supports one and only one control interface bundle for each kernel. Hence the* `bundle` *name should be same for all scalar data inputs. In the preceding example the same* `bundle` *name,* `control`*, is used for all control inputs.*

Enabling Host to Kernel Dataflow

As discussed in the Kernel Execution, if a kernel is capable of accepting more data while it is still operating on data from the previous transactions, the SDAccel runtime can send the next batch of data. The kernel then works on multiple data sets at the same time at the different portion of the hardware, and thus improve the latency and the performance. However, the kernel has to be compiled with the following `ap_ctrl_chain` pragma:

```
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
```

IMPORTANT! *To take the advantage of the host to kernel pipeline, the kernel has to be pipelined at the loop-level and task-level (more information about the kernel pipelining technique such as loop pipeline and dataflow are discussed later in the chapter).*

Loops

Loops are an important aspect for a high performance accelerator. Generally, loops are either pipelined or unrolled to take advantage of the highly distributed and parallel FPGA architecture to provide a performance boost compared to running on a CPU.

By default, loops are neither pipelined nor unrolled. Each iteration of the loop takes at least one clock cycle to execute in hardware. Thinking from the hardware perspective, there is an implicit *wait until clock* for the loop body. The next iteration of a loop only starts when the previous iteration is finished.

Loop Pipelining

By default, every iteration of a loop only starts when the previous iteration has finished. In the loop example below, a single iteration of the loop adds two variables and stores the result in a third variable. Assume that in hardware this loop takes three cycles to finish one iteration. Also, assume that the loop variable `len` is 20, that is, the `vadd` loop runs for 20 iterations in the kernel. Therefore, it requires a total of 60 clock cycles (20 iterations * 3 cycles) to complete all the operations of this loop.

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

TIP: *It is good practice to always label a loop as shown in the above code example (`vadd:…`). This practice helps with debugging when working in the SDAccel environment. Note that the labels generate warnings during compilation, which can be safely ignored.*

Pipelining the loop executes subsequent iterations in a pipelined manner. This means that subsequent iterations of the loop overlap and run concurrently, executing at different sections of the loop-body. Pipelining a loop can be enabled by the pragma `HLS PIPELINE`. Note that the pragma is placed inside the body of the loop.

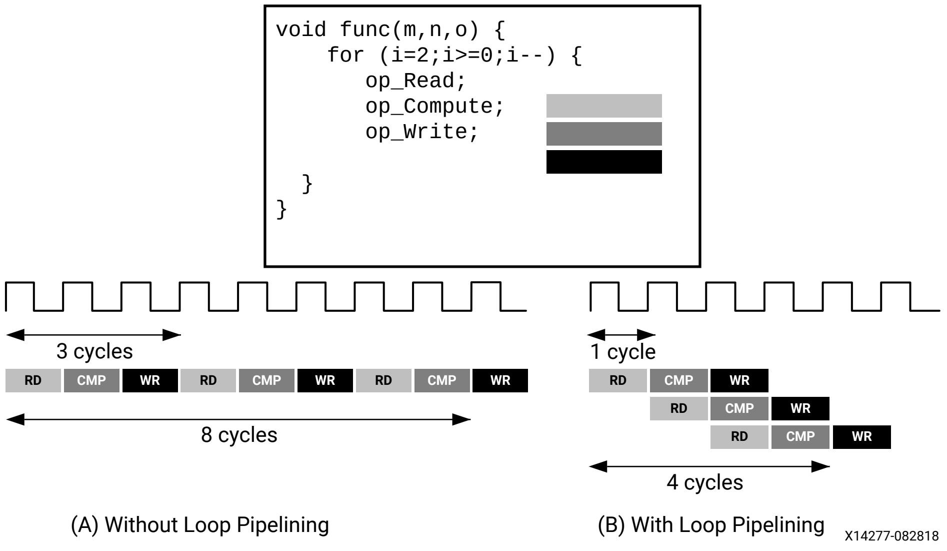
```
vadd: for(int i = 0; i < len; i++) {
  #pragma HLS PIPELINE
  c[i] = a[i] + b[i];
}
```

In the example above, it is assumed that every iteration of the loop takes three cycles: read, add, and write. Without pipelining, each successive iteration of the loop starts in every third cycle. With pipelining the loop can start subsequent iterations of the loop in fewer than three cycles, such as in every second cycle, or in every cycle.

The number of cycles it takes to start the next iteration of a loop is called the initiation interval (II) of the pipelined loop. So II = 2 means each successive iteration of the loop starts every two cycles. An II = 1 is the ideal case, where each iteration of the loop starts in the very next cycle. When you use the `pragma HLS PIPELINE` the compiler always tries to achieve II = 1 performance.

The following figure illustrates the difference in execution between pipelined and non-pipelined loops. In this figure, (A) shows the default sequential operation where there are three clock cycles between each input read (II = 3), and it requires eight clock cycles before the last output write is performed.

Figure 7: **Loop Pipelining**



In the pipelined version of the loop shown in (B), a new input sample is read every cycle (II = 1) and the final output is written after only four clock cycles: substantially improving both the II and latency while using the same hardware resources.

IMPORTANT! *Pipelining a loop causes any loops nested inside the pipelined loop to get unrolled.*

If there are data dependencies inside a loop it might not be possible to achieve II = 1, and a larger initiation interval might be the result. Loop dependencies are discussed in Loop Dependencies.

Loop Unrolling

The compiler can also unroll a loop, either partially or completely to perform multiple loop iterations in parallel. This is done using the `HLS UNROLL` pragma. Unrolling a loop can lead to a very fast design, with significant parallelism. However, because all the operations of the loop iterations are executed in parallel, a large amount of programmable logic resource are required to implement the hardware. As a result, the compiler can face challenges dealing with such a large number of resources and can face capacity problems that slow down the kernel compilation process. It is a good guideline to unroll loops that have a small loop body, or a small number of iterations.

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

In the preceding example, you can see `pragma HLS UNROLL` has been inserted into the body of the loop to instruct the compiler to unroll the loop completely. All 20 iterations of the loop are executed in parallel if that is permitted by any data dependency.

Completely unrolling a loop can consume significant device resources, while partially unrolling the loop provides some performance improvement without causing a significant impact on hardware resources.

Partially Unrolled Loop

To completely unroll a loop, the loop must have a constant bound (20 in the example above). However, partial unrolling is possible for loops with a variable bound. A partially unrolled loop means that only a certain number of loop iterations can be executed in parallel.

The following code examples illustrates how partially unrolled loops work:

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

In the above example the `UNROLL` pragma is given a factor of 2. This is the equivalent of manually duplicating the loop body and running the two loops concurrently for half as many iterations. The following code shows how this would be written. This transformation allows two iterations of the above loop to execute in parallel.

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

Just like data dependencies inside a loop impact the initiation interval of a pipelined loop, an unrolled loop performs operations in parallel only if data dependencies allow it. If operations in one iteration of the loop require the result from a previous iteration, they cannot execute in parallel, but execute as soon as the data from one iteration is available to the next.

RECOMMENDED: *A good methodology is to `PIPELINE` loops first, and then `UNROLL` loops with small loop bodies and limited iterations to improve performance further.*

Loop Dependencies

Data dependencies in loops can impact the results of loop pipelining or unrolling. These loop dependencies can be within a single iteration of a loop or between different iterations of a loop. The straightforward method to understand loop dependencies is to examine an extreme example. In the following code example, the result of the loop is used as the loop continuation or exit condition. Each iteration of the loop must finish before the next can start.

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

This loop cannot be pipelined. The next iteration of the loop cannot begin until the previous iteration ends.

Dealing with various types of dependencies with the `xocc` compiler is an extensive topic requiring a detailed understanding of the high-level synthesis procedures underlying the compiler. Refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information on "Dependencies with Vivado HLS."

Nested Loops

Coding with nested loops is a common practice. Understanding how loops are pipelined in a nested loop structure is key to achieving the desired performance.

If the pragma `HLS PIPELINE` is applied to a loop nested inside another loop, the xocc compiler attempts to flatten the loops to create a single loop, and apply the `PIPELINE` pragma to the constructed loop. The loop flattening helps in improving the performance of the kernel.

The compiler is able to flatten the following types of nested loops:

1. Perfect nested loop:
    - Only the inner loop has a loop body.
    - There is no logic or operations specified between the loop declarations.
    - All the loop bounds are constant.
2. Semi-perfect nested loop:
    - Only the inner loop has a loop body.
    - There is no logic or operations specified between the loop declarations.
    - The inner loop bound must be a constant, but the outer loop bound can be a variable.

The following code example illustrates the structure of a perfect nested loop:

```
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
  }
}
```

The above example shows a nested loop structure with two loops that performs some computation on incoming pixel data. In most cases, you want to process a pixel in every cycle, hence `PIPELINE` is applied to the nested loop body structure. The compiler is able to flatten the nested loop structure in the example because it is a perfect nested loop.

The nested loop in the preceding example contains no logic between the two loop declarations. No logic is placed between the `ROW_LOOP` and `COL_LOOP`; all of the processing logic is inside the `COL_LOOP`. Also, both the loops have a fixed number of iterations. These two criteria help the xocc compiler flatten the loops and apply the `PIPELINE` constraint.

RECOMMENDED: *If the outer loop has a variable boundary, then the compiler can still flatten the loop. You should always try to have a constant boundary for the inner loop.*

Sequential Loops

If there are multiple loops in the design, by default they do not overlap, and execute sequentially. This section introduces the concept of dataflow optimization for sequential loops. Consider the following code example:

```
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];
  }
}
```

In the previous example, three sequential loops are shown: `mem_rd`, `compute`, and `mem_wr`.

- The `mem_rd` loop reads input vector data from the memory interface and stores it in internal storage.

- The main `compute` loop reads from the internal storage and performs an increment operation and saves the result to another internal storage.
- The `mem_wr` loop writes the data back to memory from the internal storage.

As described in Memory Data Inputs and Outputs, this code example is using two separate loops for reading and writing from/to the memory input/output interfaces to infer burst read/write.

By default, these loops are executed sequentially without any overlap. First, the `mem_rd` loop finishes reading all the input data before the `compute` loop starts its operation. Similarly, the `compute` loop finishes processing the data before the `mem_wr` loop starts to write the data. However, the execution of these loops can be overlapped, allowing the `compute` (or `mem_wr`) loop to start as soon as there is enough data available to feed its operation, before the `mem_rd` (or `compute`) loop has finished processing its data.

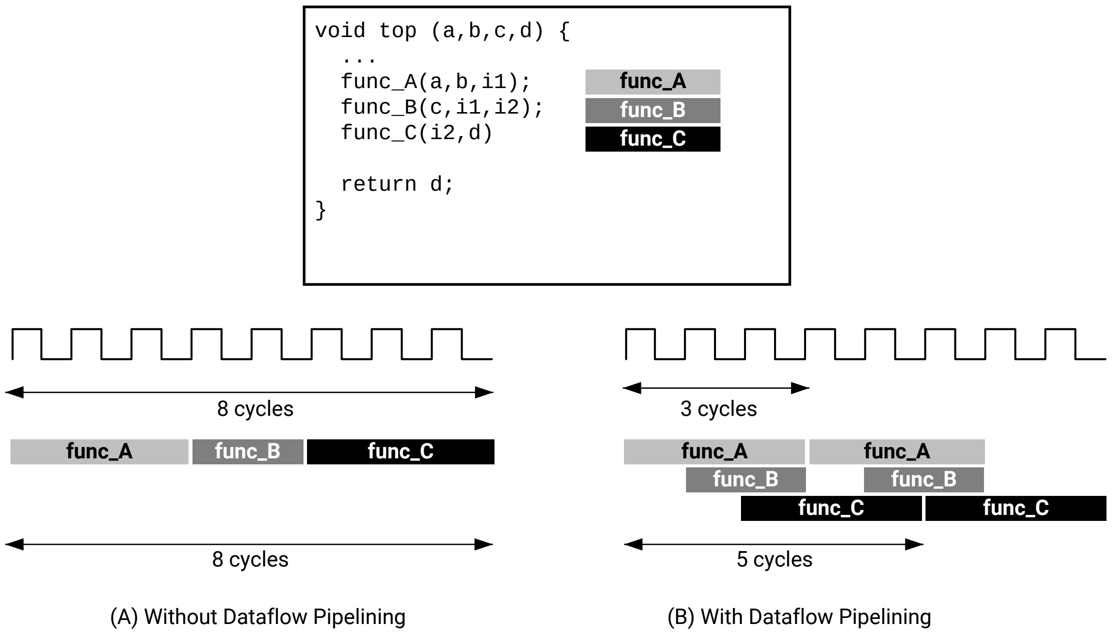
The loop execution can be overlapped using dataflow optimization as described in Dataflow Optimization.

Dataflow Optimization

Dataflow optimization is a powerful technique to improve the kernel performance by enabling tasklevel pipelining and parallelism inside the kernel. It allows the xocc compiler to schedule multiple functions of the kernel to run concurrently to achieve higher throughput and lower latency. This is also known as task-level parallelism.

The following figure shows a conceptual view of dataflow pipelining. The default behavior is to execute and complete `func_A`, then `func_B`, and finally `func_C`. With the `HLS DATAFLOW` pragma enabled, the compiler can schedule each function to execute as soon as data is available. In this example, the original `top` function has a latency and interval of eight clock cycles. With `DATAFLOW` optimization, the interval is reduced to only three clock cycles.

Figure 8: **Dataflow Optimization**



X14266-083118

Dataflow Coding Example

In the dataflow coding example you should notice the following:

1. The `HLS DATAFLOW` pragma is applied to instruct the compiler to enable dataflow optimization. This is not a data mover, which deals with interfacing between the PS and PL, but how the data flows through the accelerator.
2. The `stream` class is used as a data transferring channel between each of the functions in the dataflow region.

TIP: *The `stream` class infers a first-in first-out (FIFO) memory circuit in the programmable logic. This memory circuit, which acts as a queue in software programming, provides data-level synchronization between the functions and achieves better performance. For additional details on the `hls::stream` class, see the Vivado Design Suite User Guide: High-Level Synthesis (UG902).*

```
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;
}
```

Canonical Forms of Dataflow Optimization

Xilinx recommends writing the code inside a dataflow region using canonical forms. There are canonical forms for dataflow optimizations for both functions and loops.

- Functions: The canonical form coding guideline for dataflow inside a function specifies:

  1. Use only the following types of variables inside the dataflow region:

     a. Local non-static scalar/array/pointer variables.

     b. Local static `hls::stream` variables.

  2. Function calls transfer data only in the forward direction.

  3. Array or `hls::stream` should have only one producer function and one consumer function.

  4. The function arguments (variables coming from outside the dataflow region) should only be read, or written, not both. If performing both read and write on the same function argument then read should happen before write.

  5. The local variables (those that are transferring data in forward direction) should be written before being read.

  The following code example illustrates the canonical form for dataflow within a function. Note that the first function (`func1`) reads the inputs and the last function (`func3`) writes the outputs. Also note that one function creates output values that are passed to the next function as input parameters.

```
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);
}
```

- Loop: The canonical form coding guideline for dataflow inside a loop body includes the coding guidelines for a function defined above, and also specifies the following:

  1. Initial value 0.

  2. The loop condition is formed by a comparison of the loop variable with a numerical constant or variable that does not vary inside the loop body.

  3. Increment by 1.

The following code example illustrates the canonical form for dataflow within a loop.

```
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);
               }
}
```

Troubleshooting Dataflow

The following behaviors can prevent the xocc compiler from performing `DATAFLOW` optimizations:

1. Single producer-consumer violations.
2. Bypassing tasks.
3. Feedback between tasks.
4. Conditional execution of tasks.
5. Loops with multiple exit conditions or conditions defined within the loop.

If any of the above conditions occur inside the dataflow region, you might need to re-architect the code to successfully achieve dataflow optimization.

Array Configuration

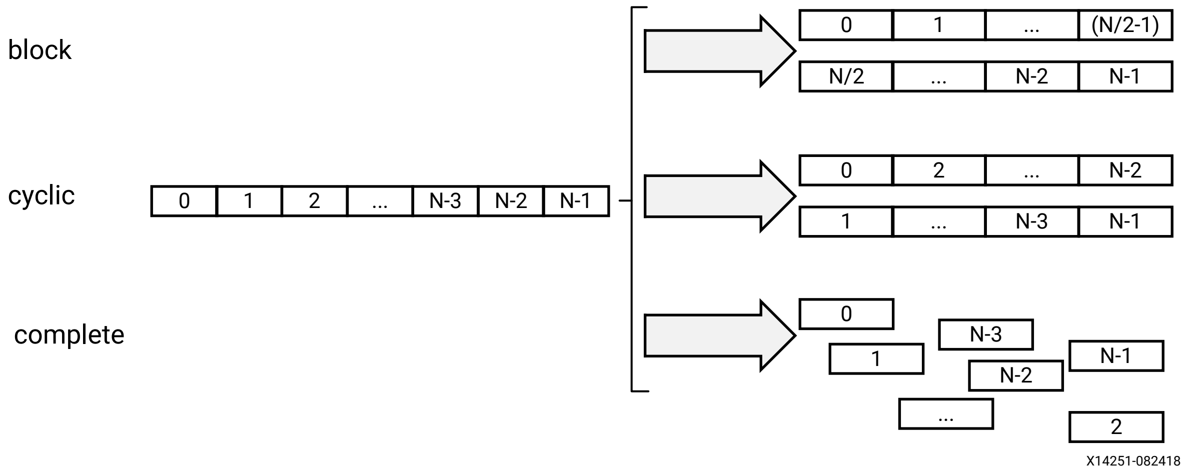
The SDAccel compiler maps large arrays to the block Ram (BRAM) memory in the PL region. These BRAM can have a maximum of two access points or ports. This can limit the performance of the application as all the elements of an array cannot be accessed in parallel when implemented in hardware.

Depending on the performance requirements, you might need to access some or all of the elements of an array in the same clock cycle. To achieve this, the `#pragma HLS ARRAY_PARTITION` can be used to instruct the compiler to split the elements of an array and map it to smaller arrays, or to individual registers. The compiler provides three types of array partitioning, as shown in the following figure. The three types of partitioning are:

- `block`: The original array is split into equally sized blocks of consecutive elements of the original array.
- `cyclic`: The original array is split into equally sized blocks interleaving the elements of the original array.

- `complete`: Split the array into its individual elements. This corresponds to resolving a memory into individual registers. This is the default for the `ARRAY_PARTITION` pragma.

Figure 9: **Partitioning Arrays**

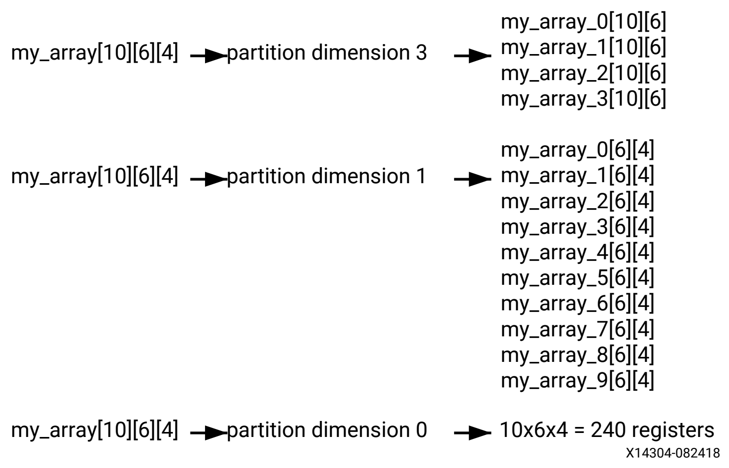


For block and cyclic partitioning, the `factor` option specifies the number of arrays that are created. In the preceding figure, a factor of 2 is used to split the array into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the later arrays will have fewer elements.

When partitioning multi-dimensional arrays, the `dimension` option is used to specify which dimension is partitioned. The following figure shows how the `dimension` option is used to partition the following example code in three different ways:

```
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];
  ...
}
```

Figure 10: **Partitioning the Dimensions of an Array**



The examples in the figure demonstrate how partitioning dimension 3 results in four separate arrays and partitioning dimension 1 results in 10 separate arrays. If 0 is specified as the dimension, all dimensions are partitioned.

The Importance of Careful Partitioning

A complete partition of the array maps all the array elements to the individual registers. This helps in improving the kernel performance because all of these registers can be accessed concurrently in a same cycle.

CAUTION! *Complete partitioning of the large arrays consumes a lot of PL region. It could even cause the compilation process to slow down and face capacity issue. Partition the array only when it is needed. Consider selectively partitioning a particular dimension or performing a block or cycle partitioning.*

Choosing a Specific Dimension to Partition

Suppose A and B are two-dimensional arrays representing two matrices. Consider the following Matrix Multiplication algorithm:

```
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;
  }
}
```

Due to the PIPELINE pragma, the ROW_WISE and COL_WISE loop is flattened together and COMPUTE_LOOP is fully unrolled. To concurrently execute each iteration (k) of the COMPUTE_LOOP, the code must access each column of matrix A and each row of matrix B in parallel. Therefore, the matrix A should be split in the second dimension, and matrix B should be split in the first dimension.

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

Choosing Between Cyclic and Block Partitions

Here the same matrix multiplication algorithm is used to demonstrate choosing between cyclic and block partitioning and determining the appropriate factor, by understanding the array access pattern of the underlying algorithm.

```
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;
  }
}
```

In this version of the code, A and B are now one-dimensional arrays. To access each column of matrix A and each row of matrix B in parallel, cyclic and block partitions are used as shown in the above example. To access each column of matrix A in parallel, cyclic partitioning is applied with the factor specified as the row size, in this case 64. Similarly, to access each row of matrix B in parallel, block partitioning is applied with the factor specified as the column size, or 64.

Minimizing Array Accesses with Caching

As arrays are mapped to BRAM with limited number of access ports, repeated array accesses can limit the performance of the accelerator. You should have a good understanding of the array access pattern of the algorithm, and limit the array accesses by locally caching the data to improve the performance of the kernel.

The following code example shows a case in which accesses to an array can limit performance in the final implementation. In this example, there are three accesses to the array `mem[N]` to create a summed result.

```
#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;
}
```

The code in the preceding example can be rewritten as shown in the following example to allow the code to be pipelined with a II = 1. By performing pre-reads and manually pipelining the data accesses, there is only one array read specified inside each iteration of the loop. This ensures that only a single-port BRAM is needed to achieve the performance.

```
#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;
}
```

RECOMMENDED: *Consider minimizing the array access by caching to local registers to improve the pipelining performance depending on the algorithm.*

For more detailed information related to the configuration of arrays, see the "Arrays" section in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Function Inlining

C code generally consists of several functions. By default, each function is compiled, and optimized separately by the `xocc` compiler. A unique hardware module will be generated for the function body and reused as needed.

From performance perspective, in general it is better to inline the function, or dissolve the function hierarchy. This helps `xocc` compiler to do optimization more globally across the function boundary. For example, if a function is called inside a pipelined loop, then inlining the function helps the compiler to do more aggressive optimization and results in a better pipeline performance of the loop (lower initiation interval or II number).

The following `INLINE` pragma placed inside the function body instruct the compiler to inline the function.

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

However, if the function body is very big and called several times inside the main kernel function, then inlining the function may cause capacity issues due to too many resources. In cases like that you might not want to inline such functions, and let the `xocc` compiler optimize the function separately in its local context.

Summary

As discussed in earlier topics, several important aspects of coding the kernel for FPGA acceleration using C/C++ include the following points:

1. Consider using arbitrary precision data types, `ap_int`, and `ap_fixed`.
2. Understand kernel interfaces to determine scalar and memory interfaces. Use `bundle` switch with different names if separate DDR memory banks will be specified in the linking stage.
3. Use Burst read and write coding style from and to the memory interface.
4. Consider exploiting the full width of DDR banks during the data transfer when selecting width of memory data inputs and outputs.
5. Get the greatest performance boost using pipelining and dataflow.
6. Write perfect or semi-perfect nested loop structure so that the `xocc` compiler can flatten and apply pipeline effectively.
7. Unroll loops with a small number of iterations and low operation count inside the loop body.
8. Consider understanding the array access pattern and apply `complete` partition to specific dimensions or apply `block` or `cyclic` partitioning instead of a `complete` partition of the whole array.
9. Minimize the array access by using local cache to improve kernel performance.
10. Consider inlining the function, specifically inside the pipelined region. Functions inside the dataflow should not be inlined.

Chapter 4

Configuring the System Architecture

In Chapter 1: SDAccel Compilation Flow and Execution Model, you learned of the two distinct phases in the SDAccel™ environment kernel build process:

1. Compilation stage: The compilation process is controlled by the `xocc -c` option. At the end of the compilation stage one or more kernel functions are compiled into separate `.xo` files. At this stage, the `xocc` compiler extracts the hardware intent from the C/C++ code and associated pragmas. Refer to the *SDx Command and Utility Reference Guide* (UG1279) for more information on the `xocc` compiler.
2. Linking stage: The linking stage is controlled by the `xocc -l` option. During the linking process all the `.xo` files are integrated into the FPGA hardware.

If needed, the kernel linking process can be customized to improve the SDAccel environment runtime performance. This chapter introduces a few such techniques.

Multiple Instances of a Kernel

By default, a single hardware instance is implemented from a kernel. If the host intends to execute the same kernel multiple times, then multiple such kernel executions take place on the same hardware instance sequentially. However, you can customize the kernel compilation (linking stage) to create multiple hardware instances from a single kernel. This can improve execution performance as the multiple kernel calls can now run concurrently, overlapping their execution while running on separate hardware instances.

Multiple instances of the kernel can be created by using the `xocc --nk` switch during linking.

For example, for a kernel name `foo`, two hardware instances can be implemented as follows:

```
# xocc --nk <kernel name>:<number of instance>
xocc --nk foo:2
```

By default, the implemented instance names are `<kernel_name>_1` and `<kernel_name>_2`. However, you can optionally change the default instance names as shown below:

```
# xocc --nk <kernel name>:<no of instance>:<name 1>.<name 2>…<name N>
xocc --nk foo:3:fooA.fooB.fooC
```

This example implements three identical copies, or hardware instances of kernel `foo`, named `fooA`, `fooB`, and `fooC` on the FPGA programmable logic.

Connecting Kernel Ports to Global Memory

By default, all kernel memory interfaces are connected to the same global memory bank. As a result, only one kernel port at a time can transfer data to and from the memory bank, limiting the performance of the application.

However, all off-the-shelf SDAccel platforms contain multiple global memory banks. During the linking stage, it is possible to specify for each kernel port (or interface) which global memory bank it should be connected to.

Proper configuration of kernel to memory connectivity is important to maximize bandwidth, optimize data transfers, and improve overall performance.

Consider the following example:

```
void cnn( int *image, // Read-Only Image
  int *weights, // Read-Only Weight Matrix
  int *out, // Output Filters/Images
  ... // Other input or Output ports

  #pragma HLS INTERFACE m_axi port=image 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
```

The example shows two memory interface inputs for the kernel: `image` and `weights`. If both are connected to the same memory bank, a concurrent transfer of both of these inputs into the kernel is not possible.

The following steps are needed to implement separate memory bank connections for the `image` and `weights` inputs:

1. Specify separate bundle names for these inputs. This is discussed in Memory Data Inputs and Outputs. However, for reference the code is shown here again.

```
void cnn( int *image, // Read-Only Image
 int *weights, // Read-Only Weight Matrix
 int *out, // Output Filters/Images
 ... // Other input or Output ports

 #pragma HLS INTERFACE m_axi port=image 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
```

IMPORTANT! *When specifying a `bundle=` name, you should use all lowercase characters to be able to assign it to a specific memory bank using the `--sp` option.*

The memory interface inputs `image` and `weights` are assigned different bundle names in the example above.

2. Specify kernel port to global memory connection during the linking stage with the `--sp` switch:

```
--sp <kernel_instance_name>.<interface_name>:<bank name>
```

Where:

- `<kernel_instance_name>` is the instance name of the kernel as specified by the `--nk` option, described in Multiple Instances of a Kernel.
- `<interface_name>` is the name of the interface bundle defined by the HLS INTERFACE pragma, including `m_axi_` as a prefix, and the `bundle=` name when specified.

  Note: It is also possible to use the `<interface_name>` (image, weights).

TIP: *If the port is not specified as part of a bundle, then the `<interface_name>` is the specified `port=` name, without the `m_axi_` prefix.*

- `<bank_name>` is denoted as `DDR[0]`, `DDR[1]`, etc. For a platform with four DDR banks, the bank names are `DDR[0]`, `DDR[1]`, `DDR[2]`, and `DDR[3]`. Some platforms also provide support for PLRAM and HBM memory banks.

For the above example, considering a single instance of the `cnn` kernel, the `--sp` switch can be specified as follows:

```
--sp cnn_1.m_axi_gmem:DDR[0]  \
--sp cnn_1.m_axi_gmem1:DDR[1]
```

Note: Up to 15 kernel ports can be connected to a given memory bank. Therefore, if there are more than 15 ports in the entire design, it is not possible to rely on the default mapping and the `--sp` option must be used to distribute connections across different banks.

Summary

This section discusses two powerful ways to customize the kernel compilation to improve the system performance during execution.

1. Consider creating multiple instances of a kernel on the fabric of the FPGA by specifying the `xocc --nk` if the kernel is called multiple times from the host code.
2. Consider using the `xocc --sp` switch to customize the global memory connection to kernel memory interfaces to achieve concurrent access.

Depending on the host and kernel design, these options can be exploited to improve the kernel acceleration on Xilinx® FPGAs.

Appendix A

SDAccel Streaming Platform

Streaming Data Transfers Between Host and Kernel

Starting from the SDAccel™ 2019.1 release, SDAccel provides a new programming model which supports the direct streaming of data from host to kernel and kernel to host without having to go through global memory. This feature is an addition to the existing host to kernel and kernel to host data transfer using global memories. By using streams, you can get some of the advantages such as:

- The host application does not necessarily need to know the size of the data coming from the kernel.
- Data resides on the host memory can be transferred to the kernel as soon as it is needed. Similarly, the processed data can be transferred back when it is required.

This programming model uses minimal storage compared to the larger and slower global memory bank, and thus improving the performance and power.

Host Coding Guidelines

Xilinx® provides new OpenCL™ APIs for streaming operation as extension APIs.

- `clCreateStream()`: Creates a read or write stream.
- `clReleaseStream()`: Frees the created stream and its associated memory.
- `clWriteStream()`: Writes data to stream.
- `clReadStream()`: Gets data from stream.
- `clPollStreams()`: Polls for any stream on the device to finish. Required only for non-blocking stream operation.

The typical API flow is described below:

- Create the required number of the read/write streams by `clCreateStream`.

- Streams should be directly attached to the OpenCL device object because it does not use any command queue. A stream itself is a command queue that only passes the data to a particular direction, either from host to kernel or from kernel to host.
- An appropriate flag should be used to denote stream write/read operation (from the host perspective).
- To specify how the stream is connected to the device, a predefined extension pointer (`cl_mem_ext_ptr_t`) should be used to denote the kernel and its argument the stream is associated with.

  In the code block below, a Read Stream (named `read_stream`) and a Write Stream (named `write_stream`) are created.

```
#include <CL/cl_ext_xilinx.h> // Required for Xilinx Extension

// Device connection specification of the stream through extension
pointer
cl_mem_ext_ptr_t  ext;  // Extension pointer
ext.param = kernel;     // The .param should be set to kernel
                          (cl_kernel type)
ext.obj = nullptr;

// The .flag should be used to denote the kernel argument
// Create write stream for argument 3 of kernel
ext.flags = 3;
cl_stream write_stream = clCreateStream(device_id,
CL_STREAM_WRITE_ONLY, CL_STREAM, &ext, &ret);

// Create read stream for argument 4 of kernel
ext.flags = 4;
cl_stream read_stream = clCreateStream(device_id, CL_STREAM_READ_ONLY,
CL_STREAM, &ext,&ret);
```

- Set the remaining non-stream kernel arguments and enqueue the kernel. The following code block shows typical kernel argument (non-stream arguments such as buffer and/or scalar) setting and kernel enqueuing.

```
// Set kernel non-stream argument (if any)
clSetKernelArg(kernel, 0,...,...);
clSetKernelArg(kernel, 1,...,...);
clSetKernelArg(kernel, 2,...,...);
// Argument 3 and 4 are not set as those are already specified during
    the clCreateStream through extension pointer

// Schedule kernel enqueue
clEnqueueTask(commands, kernel, . .. . );
```

- Initiate Read and Write transfer by `clReadStream` and `clWriteStream`.
  - Note the usage of attribute `cl_stream_xfer_req` associated with read and write request.
  - The `.flag` is used to denote transfer mechanism.
    - **CL_STREAM_EOT:** Currently, successful stream transfer mechanism depends on identifying the end of the transfer by an *End of Transfer* signal. This flag is mandatory in the current release.

        - **CL_STREAM_NONBLOCKING:** By default the Read and Write transfers are blocking. For non-blocking transfer, CL_STREAM_NONBLOCKING has to be set.

    - The `.priv_data` is used to specify a string (as a name for tagging purpose) associated with the transfer. This will help identify specific transfer completion when polling the stream completion. It is required when using the non-blocking version of the API.

      In the following code block, the stream read and write transfers are executed with the non-blocking approach.

      ```
      // Initiate the READ transfer
      cl_stream_xfer_req rd_req {0};

      rd_req.flags = CL_STREAM_EOT | CL_STREAM_NONBLOCKING;
      rd_req.priv_data = (void*)"read"; // You can think this as tagging the
                                           transfer with a name

      clReadStream(read_stream, host_read_ptr, max_read_size, &rd_req, &ret);

      // Initiating the WRITE transfer
      cl_stream_xfer_req wr_req {0};

      wr_req.flags = CL_STREAM_EOT | CL_STREAM_NONBLOCKING;
      wr_req.priv_data = (void*)"write";

      clWriteStream(write_stream, host_write_ptr, write_size, &wr_req , &ret);
      ```

- Poll all the streams for completion. For the non-blocking transfer, a polling API is provided to ensure the read/write transfers are completed. For the blocking version of the API, polling is not required.

    - The number of poll requests should be used through `cl_streams_poll_req_completions`.

    - The `clPollStreams` is a blocking API. It returns the execution to the host code as soon as it receives the notification that all stream requests have been completed, or until you specify the timeout.

      ```
      // Checking the request completion
         cl_streams_poll_req_completions poll_req[2] {0, 0}; // 2 Requests

         auto num_compl = 2;
         clPollStreams(device_id, poll_req, 2, 2, &num_compl, 5000, &ret);
         // Blocking API, waits for 2 poll request completion or 5000ms,
            whichever occurs first
      ```

- Read and use the stream data in host.

    - After the successful poll request is completed, the host can read the data from the host pointer.

- Also, the host can check the size of the data transferred to the host. For this purpose, the host needs to find the correct poll request by matching `priv_data` and then fetching nbytes (the number of bytes transferred) from the `cl_streams_poll_req_completions structure.`

```
for (auto i=0; i<2; ++i) {
    if(rd_req.priv_data == poll_req[i].priv_data) { // Identifying the
                                                       read transfer
        // Getting read size, data size from kernel is unknown
        ssize_t result_size=poll_req[i].nbytes;
        }
    }
```

The header file containing function prototype and argument description is available in the Xilinx Runtime GitHub repository.

IMPORTANT! *If the streaming kernel has multiple CUs, the host code needs to use a unique `cl_kernel` object for each CU. The host code must use `clCreateKernel` with `<kernel_name>:{compute_unit_name}` to get each CU, creating streams for them, and enqueuing them individually.*

Kernel Coding Guidelines

The basic guidelines to develop stream-based C kernel is as follows:

- Use `hls::stream` with the `qdma_axis<D,0,0,0>` data type. The `qdma_axis` data type needs the header file `ap_axi_sdata.h`.
- The `qdma_axis<D,0,0,0>` is a special class used for data transfer between host and kernel when using the streaming platform. This is only used in the streaming kernel interface interacting with the host, not with another kernel. The template parameter <D> denotes data width. The remaining three parameters should be set to 0 (not to be used in the current release).
- The following code block shows a simple kernel interface with one input stream and one output stream.

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

//qdma_axis is the HLS class for stream data transfer between host and
kernel for streaming platform
//It contains "data" and two sideband signals (last and keep) exposed to
the user via class member function.
typedef qdma_axis<64,0,0,0> datap;

void kernel_top (
             hls::stream<datap> &input,
             hls::stream<datap> &output,
             ..... , // Other Inputs/Outputs if any
             )
{
    #pragma HLS INTERFACE axis port=input
    #pragma HLS INTERFACE axis port=output
}
```

- The `qdma_axis` data type contains three variables which should be used inside the kernel code:
  - **data:** Internally `qdma_axis` contains an `ap_uint`<D> that should be accessed by the `.get_data()` and `.set_data()` method.
    - The D must be 8, 16, 32, 64, 128, 256, or 512 bits wide.
  - **last:** The `last` variable is used to indicate the last value of an incoming and outgoing stream. When reading from the input stream, `last` is used to detect the end of the stream. Similarly when kernel writes to an output stream transferred to the host, the `last` must be set to indicate the end of stream.
    - `get_last/set_last`: Accesses/sets the `last` variable used to denote the last data in the stream.
  - **keep:** In some special situation, `keep` signal can be used to truncate the last data to the fewer number of bytes. However, `keep` should not be used to any data other than the last data from the stream. So, in most of the cases, you should set `keep` to -1 for all the outgoing data from the kernel.
    - `get_keep/set_keep`: Accesses/sets the `keep` variable.
    - For all the data before the last data, `keep` must be set to -1 to denote all bytes of the data are valid.
    - For the last data, the kernel has the flexibility to send fewer bytes. For example, for the four bytes data transfer, the kernel can truncate the last data by sending one byte, two bytes, or three bytes by using `set_keep()` function as below.
      - If the last data is one byte => `.set_keep(1)`
      - If the last data is two bytes => `.set_keep(3)`
      - If the last data is three bytes => `.set_keep(7)`
      - If the last data is all four bytes (similar to all non-last data) => `.set_keep(-1)`
- The following code block shows how the stream `input` is read. Note the usage of `.last` to determine the last data.

```
// Stream Read
// Using "last" flag to determine the end of input-stream
// when kernel does not know the length of the input data
 hls::stream<ap_uint<64> >   internal_stream;
 while(true) {
        datap temp = input.read(); // "input" -> Input stream
        internal_stream << temp.get_data();  // Getting data from the
        stream
        if(temp.get_last())  // Getting last signal to determine the
        EOT (end of transfer).
            break;
 }
```

- The following code block shows how the stream `output` is written. The `set_keep` is setting -1 for all data (general case). Also, the kernel uses the `set_last()` to specify the last data of the stream.

IMPORTANT! *For the proper functionality of the host and kernel system, it is very important to set the* `last` *bit setting.*

```
// Stream Write
for(int j = 0; j <....; j++) {
       datap t;
       t.set_data(...);
       t.set_keep(-1);         // keep flag -1 , all bytes are valid
       if(... )                // check if this is last data to be write
          t.set_last(1);       // Setting last data of the stream
       else
          t.set_last(0);
       output.write(t);        // output stream from the kernel
}
```

Streaming Data Transfers Between the Kernels

The SDAccel environment also supports streaming data transfer between two kernels. Consider the situation where one kernel is performing some part of the computation and the second kernel is operating the rest after receiving the output data from the first kernel. Before SDx™ 2019.1 version, the only method to transfer data from one kernel to another was through the global memory. Now with kernel to kernel streaming support, data can move directly from one kernel to another without having to transmit through global memory, improving performance.

Host Coding Guidelines

There is only one consideration from the host coding perspective for kernel to kernel streaming data transfer, the kernel ports involved in kernel to kernel data transfer does not need `clSetKernelArg` from the host code. The host code should set other kernel port arguments that are directly interacting with the host with the `clSetKernelArg` command.

Kernel Coding Guidelines

The kernel streaming interface directly sending or receiving data to another kernel streaming interface should be defined by `hls::stream` with the `ap_axiu<D,0,0,0>` data type. The `ap_axiu<D,0,0,0>` data type needs the header file `ap_axi_sdata.h`.

IMPORTANT! *Xilinx requires using the* `qdma_axis` *data type for host to kernel and kernel to host as described in the previous section. On the other hand, the* `ap_axiu` *data type should be used for intra-kernel streaming data transfer. Both of these data types are defined inside* `ap_axi_sdata.h` *file distributed with the SDAccel release.*

The following example shows the streaming interfaces of the producer and consumer kernels.

```
// Producer kernel
// Producing stream output to another kernel on the FPGA
// The below code segment ignores all other inputs and outputs, if any

void kernel1 (.... , hls::stream<ap_axiu<32, 0, 0, 0> >& stream_out)    {
#pragma HLS interface axis port=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
// Consuming stream input from another kernel on the FPGA
// The below code segment ignores all other inputs and outputs, if any
void kernel2 (hls::stream<ap_axiu<32, 0, 0, 0> >& stream_in, .... )    {
#pragma HLS interface axis port=stream_in

        for(int i = 0; i < ....; i++) {
            ap_axiu<32, 0, 0, 0> v = stream_in.read(); // Reading from the
            Input stream
            int a = v.data; // Extract the data

            // Do further processing
        }
 }
```

Linking the Kernels

Additionally, connect the streaming output port of the producer kernel to the streaming input port of the consumer kernel by the `--sc` switch applied during the `xocc` link (-l) stage.

```
#Syntax:: xocc -l --sc <Source streaming port>:<Destination streaming port>
xocc -l --sc <kernel1 instance name>.stream_in:<kernel2 instance
name>.stream_out
```

OpenCL Installable Client Driver Loader

A system can have multiple OpenCL™ platforms, each with its own driver and OpenCL version. The SDAccel™ environment supports the OpenCL Installable Client Driver (ICD) extension (`cl_khr_icd`). This extension allows multiple implementations of OpenCL to co-exist on the same system. The ICD Loader acts as a supervisor for all installed platforms, and provides a standard handler for all API calls.

Applications can choose an OpenCL platform from the list of installed platforms. Based on the platform ID specified by the application, the ICD dispatches the OpenCL host calls to the right runtime.

Xilinx does not provide the OpenCL ICD library, so the following should be used to install the library on your preferred system.

Ubuntu

On Ubuntu the ICD library is packaged with the distribution. Install the following packages:

- ocl-icd-libopencl1
- opencl-headers
- ocl-icd-opencl-dev

Linux

For RHEL/CentOS 7.X use EPEL 7, install the following packages:

- ocl-icd
- ocl-icd-devel
- opencl-headers

Appendix C

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. DocNav is installed with the SDSoC™ and SDAccel™ development environments. To open it:

- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

1. *SDAccel Environment Release Notes, Installation, and Licensing Guide* (UG1238)

2. *SDAccel Environment Profiling and Optimization Guide* (UG1207)
3. *SDAccel Environment Getting Started Tutorial* (UG1021)
4. SDAccel™ Development Environment web page
5. Vivado® Design Suite Documentation
6. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
7. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
8. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
9. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
10. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
11. *Vivado Design Suite Properties Reference Guide* (UG912)
12. Khronos Group web page: Documentation for the OpenCL standard
13. Xilinx® Virtex® UltraScale+™ FPGA VCU1525 Acceleration Development Kit
14. Xilinx® Kintex® UltraScale™ FPGA KCU1500 Acceleration Development Kit
15. Xilinx® Alveo™ web page

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright