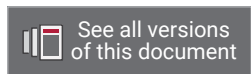# SDSoC Profiling and Optimization Guide

UG1235 (v2019.1) May 22, 2019

XILINX®

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/22/2019 Version 2019.1	
General Updates	Editorial updates only. No technical changes.

Table of Contents

Introduction

The Zynq®-7000 SoC processor and the Zynq® UltraScale+™ MPSoC processor integrate the software programmability of an Arm®-based processor with the hardware programmability of an FPGA, enabling key analytics and hardware acceleration while integrating CPU, DSP, ASSP, and mixed-signal functionality on a single device. The SDSoC™ environment is a tool suite that includes an Eclipse-based integrated development environment (IDE) for implementing heterogeneous embedded systems using the Zynq platforms.

The SDSoC environment includes system compilers that transform C/C++ programs into complete hardware/software systems with selected functions compiled into programmable logic to enable hardware acceleration of the select functions. This guide provides software programmers with an understanding of the underlying hardware used to provide the acceleration, including factors that might limit or improve performance, and a methodology to help achieve the highest system performance.

A detailed knowledge of hardware is not required to get started with the SDSoC environment. However, an understanding of the hardware resources available on the device and how a hardware function achieves very high performance through increased parallelism helps you select the appropriate compiler optimization directives to meet your performance needs.

The methodology can be summarized as follows:

1. Identify functions to be accelerated in hardware. Profiling your software can help identify compute intensive regions.

2. Optimize the hardware function code to achieve performance targets using the Vivado® High-Level Synthesis (HLS) tool coding guidelines.

3. Optimize data transfers between the CPU processor system (PS) and hardware functions programmable logic (PL). This step can involve restructuring data accesses to be burst-friendly, and to select data movers as described in this document.

After providing an understanding of how hardware acceleration is achieved, this guide concludes with a number of real-world examples demonstrating the methodology for use in your own SDSoC-based applications.

Software Acceleration with SDSoC

When compared with processor architectures, the structures that comprise the programmable logic (PL) in a Xilinx device enable a high degree of parallelism in application execution. The custom processing architecture generated by the `sds++/sdscc` (referred to as `sds++`) system compiler for a hardware function in an accelerator presents a different execution paradigm from CPU execution, and provides an opportunity for significant performance gains. While you can re-target an existing embedded processor application for acceleration in PL, writing your application to use the source code libraries of existing hardware functions, such as the Xilinx xfOpenCV library, or modifying your code to better use the PL device architecture, yields significant performance gains and power reduction.

CPUs have fixed resources and offer limited opportunities for parallelization of tasks or operations. A processor, regardless of its type, executes a program as a sequence of instructions generated by processor compiler tools, which transform an algorithm expressed in C/C++ into assembly language constructs that are native to the target processor. Even a simple operation, such as the multiplication of two values, results in multiple assembly instructions that must be executed across multiple clock cycles.

An FPGA is an inherently parallel processing device capable of implementing any function that can run on a processor. Xilinx devices have an abundance of resources that can be programmed and configured to implement any custom architecture and achieve virtually any level of parallelism. Unlike a processor, where all computations share the same ALU, the FPGA programming logic acts as a blank canvas to define and implement your acceleration functions. The FPGA compiler creates a unique circuit optimized for each application or algorithm; for example, only implementing multiply and accumulate hardware for a neural net—not a whole ALU.

The `sds++` system compiler invoked with the `-c` option compiles a file into a hardware IP by invoking the Vivado High-Level Synthesis (HLS) tool on the desired function definition. Before calling the HLS tool, the `sds++` compiler translates `#pragma SDS` into pragmas understood by the HLS tool. The HLS tool performs hardware-oriented transformations and optimizations, including scheduling, pipelining, and dataflow operations to increase concurrency.

The `sds++` linker analyzes program dataflow involving calls into and between hardware functions, mapping into a system hardware data motion network, and software control code (called stubs) to orchestrate accelerators and data transfers through data movers. As described in the following section, the `sds++` linker performs data transfer scheduling to identify operations that can be shared, and to insert wait barrier API calls into stubs to ensure program semantics are preserved.

Execution Model of an SDSoC Application

The execution model for an SDSoC environment application can be understood in terms of the normal execution of a C++ program running on the target CPU after the platform has booted. It is useful to understand how a C++ binary executable interfaces to hardware.

The set of declared hardware functions within a program is compiled into hardware accelerators that are accessed with the standard C runtime through calls into these functions. Each hardware function call in effect invokes the accelerator as a task and each of the arguments to the function is transferred between the CPU and the accelerator, accessible by the program after accelerator task completion. Data transfers between memory and accelerators are accomplished through data movers, such as a DMA engine, automatically inserted into the system by the `sds++` system compiler taking into account user data mover pragmas such as `zero_copy`.

Figure 1: **Architecture of an SDSoC System**

X21358-082418

Send Feedback
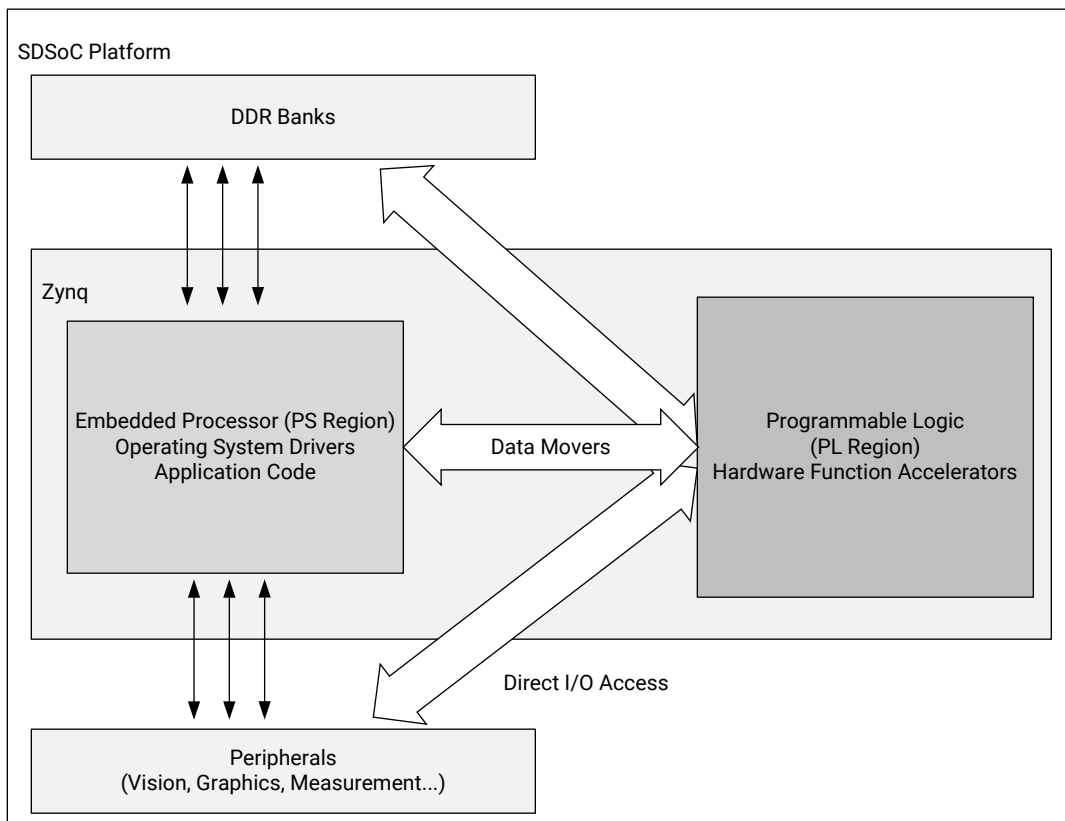
To ensure program correctness, the system compiler intercepts each call to a hardware function, and replaces it with a call to a generated stub function that has an identical signature but with a derived name. The stub function orchestrates all data movement and accelerator operation, synchronizing software and accelerator hardware at the exit of the hardware function call. Within the stub, all accelerator and data mover control is realized through a set of send and receive APIs provided by the `sds_lib` library.

When program dataflow between hardware function calls involves array arguments that are not accessed after the function calls have been invoked within the program (other than destructors or `free()` calls), and when the hardware accelerators can be connected using streams, the system compiler transfers data from one hardware accelerator to the next through direct hardware stream connections, rather than implementing a round trip to and from memory. This optimization can result in significant performance gains and reduction in hardware resources.

At a high-level, the SDSoC program execution model includes the following steps:

1. Initialization of the `sds_lib` library occurs during the program constructor before entering `main()`.

2. Within a program, every call to a hardware function is intercepted by a function call into a stub function with the same function signature (other than name) as the original function. Within the stub function, the following steps occur:

   a. A synchronous accelerator task control command is sent to the hardware.

   b. For each argument to the hardware function, an asynchronous data transfer request is sent to the appropriate data mover, with an associated `wait()` handle. A non-void return value is treated as an implicit output scalar argument.

   c. A barrier `wait()` is issued for each transfer request. If a data transfer between accelerators is implemented as a direct hardware stream, the barrier `wait()` for this transfer occurs in the stub function for the last in the chain of accelerator functions for this argument.

3. Clean up of the `sds_lib` library occurs during the program destructor, upon exiting `main()`.

💡 **TIP:** *Steps 2a–2c ensure that program correctness is preserved at the entrance and exit of accelerator pipelines while enabling concurrent execution within the pipelines.*

Sometimes, the programmer has insight of the potential concurrent execution of accelerator tasks that cannot be automatically inferred by the system compiler. In this case, the `sds++` system compiler supports a `#pragma SDS async(ID)` that can be inserted immediately preceding a call to a hardware function. This pragma instructs the compiler to generate a stub function without any barrier `wait()` calls for data transfers. As a result, after issuing all data transfer requests, control returns to the program, enabling concurrent execution of the program while the accelerator is running. In this case, it is your responsibility to insert a `#pragma SDS wait(ID)` within the program at appropriate synchronization points, which are resolved into `sds_wait(ID)` API calls to correctly synchronize hardware accelerators, their implicit data movers, and the CPU.

> ⭐ **IMPORTANT!** *Every* `async(ID)` *pragma requires a matching* `wait(ID)` *pragma.*

SDSoC Build Process

The SDSoC build process uses a standard compilation and linking process. Similar to `g++`, the `sds++` system compiler invokes sub-processes to accomplish compilation and linking.

As shown in the following figure, compilation is extended not only to object code that runs on the CPU, but it also includes compilation and linking of hardware functions into IP blocks using the Vivado High-Level Synthesis (HLS) tool, and creating standard object files (`.o`) using the target CPU toolchain. System linking consists of program analysis of caller/callee relationships for all hardware functions, and the generation of an application-specific hardware/software network to implement every hardware function call. The `sds++` system compiler invokes all necessary tools, including Vivado HLS (function compiler), the Vivado Design Suite to implement the generated hardware system, and the Arm compiler and `sds++` linker to create the application binaries that run on the CPU invoking the accelerator (stubs) for each hardware function by outputting a complete bootable system for an SD card.

Figure 2: **SDSoC Build Process**
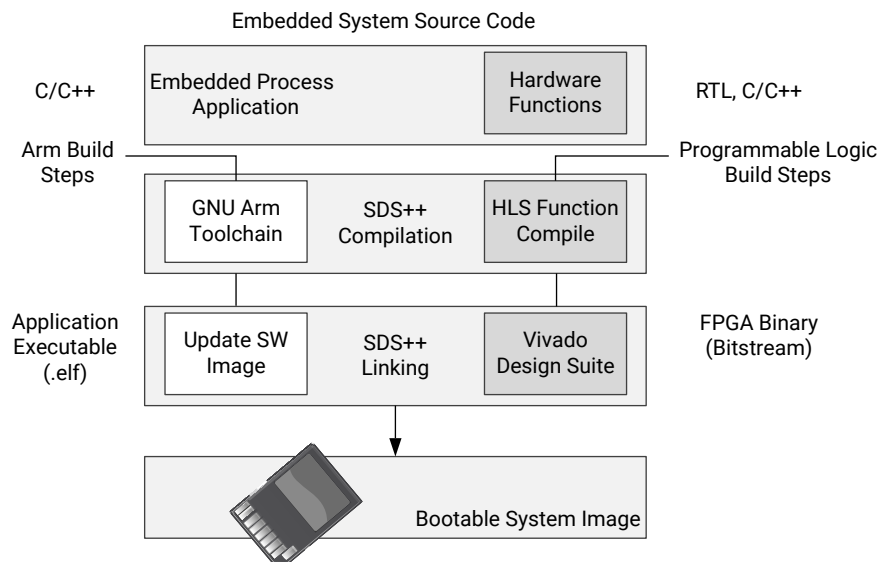
The compilation process includes the following tasks:

1. Analyzing the code and running a compilation for the main application on the Arm core, as well as a separate compilation for each of the hardware accelerators.

2. Compiling the application code through standard GNU Arm compilation tools with an object (`.o`) file produced as final output.

3. Running the hardware accelerated functions through the HLS tool to start the process of custom hardware creation with an object (.o) file as output.

After compilation, the linking process includes the following tasks:

1. Analyzing the data movement through the design and modifying the hardware platform to accept the accelerators.

2. Implementing the hardware accelerators into the programmable logic (PL) region using the Vivado Design Suite to run synthesis and implementation, and generate the bitstream for the device.

3. Updating the software images with hardware access APIs to call the hardware functions from the embedded processor application.

4. Producing an integrated SD card image that can boot the board with the application in an Executable and Linkable Format (ELF) file.

Estimating Performance

Profiling and Instrumenting Code to Measure Performance

The first major task in profiling and instrumenting code is to identify portions of application code that are suitable for implementation in hardware, and that significantly improve overall performance when run in hardware. Compute intensive regions of code are good candidates for hardware acceleration, especially when it is possible to stream data between hardware, the CPU, and memory to overlap the computation with the communication. Software profiling is a standard way to identify the most CPU-intensive portions of your program. An example of a function that would not do well for acceleration is one that takes more time to transfer data to/ from the accelerator than to compute the result. The SDSoC™ environment includes all performance and profiling capabilities that are included in the Xilinx® SDK tool, including `gprof`, the non-intrusive Target Communication Framework (TCF) profiler, and the Performance Analysis perspective within Eclipse.

To run the TCF Profiler for a standalone application, use the following steps:

1. Set the active build configuration to **Debug** by right-clicking the project in the Project Explorer and selecting **Build Configurations → Set Active → Debug**.

2. Launch the debugger by right-clicking the project name in the Project Explorer and selecting **Debug As → Launch on hardware (SDx Application Debugger)**.

   Note: The board must be connected to your computer and powered on. The application automatically breaks at the entry to `main()`.

3. Launch the TCF Profiler by selecting **Window → Show View → Other**. In the window that is produced, expand **Debug**, and select **TCF profiler**.

4. To start the TCF Profiler, click the green **Start** button at the top of the TCF Profiler tab.

5. Enable **Aggregate per function** in the Profiler Configuration dialog box.

6. To start the profiling, click the **Resume** button or press **F8**. The program runs to completion and breaks at the `exit()` function.

7. View the results in the TCF Profiler tab.

Profiling provides a statistical method for finding highly used regions of code based on sampling the CPU program counter and correlating to the program in execution. Another way to measure program performance is to instrument the application to determine the actual duration between different parts of a program in execution.

Using the TCF Profiler provides more in-depth information related to either a standalone or a Linux OS application. As seen in the previous steps, no additional compilation flags were needed to use the Profiler.

Note: This type of profiling for hardware requires a JTAG connection.

The `sds_lib` library included in the SDSoC environment provides a simple, source code annotation-based, time-stamping API that can be used to measure application performance, as shown in the following example:

```
/*
 * @return value of free-running 64-bit Zynq(TM) global counter
 */
unsigned long long sds_clock_counter(void);
```

Using this API to collect timestamps and differences between them, you can determine duration of key parts of your program. For example, you can measure data transfer or overall round trip execution time for hardware functions, as shown in the following code snippet:

```
class perf_counter
{
public:
    uint64_t tot, cnt, calls;
    perf_counter() : tot(0), cnt(0), calls(0) {};
    inline void reset() { tot = cnt = calls = 0; }
    inline void start() { cnt = sds_clock_counter(); calls++; };
    inline void stop() { tot += (sds_clock_counter() - cnt); };
    inline uint64_t avg_cpu_cycles() { return (tot / calls); };
};

extern void f();
void measure_f_runtime()
{
    perf_counter f_ctr;
    f_ctr.start();
    f()
    f_ctr.stop();
    std::cout << "Cpu cycles f(): " << f_ctr.avg_cpu_cycles()
              << std::endl;
}
```

The performance estimation feature within the SDSoC environment employs this API by automatically instrumenting functions selected for hardware implementation, measuring actual runtimes by running the application on the target, and then comparing actual times with estimated times for the hardware functions.

Note: While off-loading CPU-intensive functions is one of the most reliable heuristics to partition your application, it is not guaranteed to improve system performance without algorithmic modification to optimize memory accesses. A CPU almost always has much faster random access to external memory than you can achieve from programmable logic, due to multi-level caching and a faster clock speed (typically 2x to 8x faster than programmable logic). Extensive manipulation of pointer variables over a large address range, for example, a sort routine that sorts indices over a large index set, while very well-suited for a CPU, could become a liability when moving a function into programmable logic. This does not mean that such compute functions are not good candidates for hardware, only that code or algorithm restructuring could be required. This is a known issue for DSP and GPU coprocessors.

SDSCC/SDS++ Performance Estimation Flow Options

A full bitstream compile can take much more time than a software compile, so the `sds++/sdscc` (referred to as `sds++`) applications provide performance estimation options to compute the estimated runtime improvement for a set of hardware function calls.

In the Application Project Settings pane, to invoke the estimator, select the **Estimate Performance** check box. This enables performance estimation for the current build configuration and builds the project.

Figure 3: **Setting Estimate Performance in Application Project Settings**
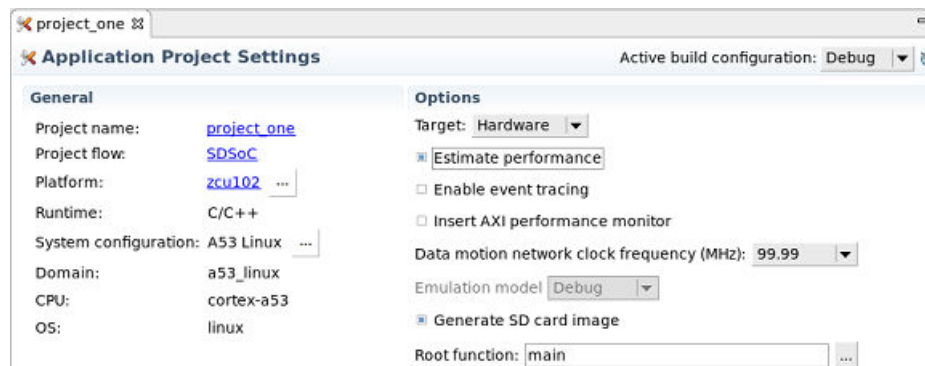
Estimating the speed-up is a two phase process:

1. The SDSoC environment compiles the hardware functions and generates the system. Instead of synthesizing the system to bitstream, the `sds++` computes an estimate of the performance based on estimated latencies for the hardware functions and data transfer time estimates for the callers of hardware functions.

2. In the generated Performance Report, to determine a performance baseline and the performance estimate, select **Click Here** to run an instrumented version of the software on the target.

Send Feedback

See the *SDSoC Environment Getting Started Tutorial* ([UG1028](#)) for a tutorial on how to use the Performance Report.

You can also generate a performance estimate from the command line. As a first pass to gather data about software runtime, use the `-perf-funcs` option to specify functions to profile and `-perf-root` to specify the root function encompassing calls to the profiled functions.

The `sds++` system compiler then automatically instruments these functions to collect runtime data when the application is run on a board. When you run an instrumented application on the target, the program creates a file on the SD card called `swdata.xml`, which contains the runtime performance data for the run.

Copy the `swdata.xml` to the host, and run a build that estimates the performance gain on a per hardware function caller basis and for the top-level function specified by the `-perf-root` function in the first pass run. Use the `-perf-est` option to specify `swdata.xml` as input data for this build.

The following table specifies the `sds++` system compiler options normally used to build an application.

Table 1: **Commonly used sds++ options**

Option	Description
`-perf-funcs function_name_list`	Specifies a comma separated list of all functions to be profiled in the instrumented software application.
`-perf-root function_name`	Specifies the root function encompassing all calls to the profiled functions. The default is the function main.
`-perf-est data_file`	Specifies the file containing runtime data generated by the instrumented software application when run on the target. Estimate performance gains for hardware accelerated functions. The default name for this file is `swdata.xml`.
`-perf-est-hw-only`	Runs the estimation flow without running the first pass to collect software run data. Using this option provides hardware latency and resource estimates without providing a comparison against baseline.

⚠️ **CAUTION!** *After running the* `sd_card` *image on the board for collecting profile data, type* `cd /; sync; umount /mnt;`. *This ensures that the* `swdata.xml` *file is written out to the SD card.*

Improving System Performance

This chapter describes underlying principles and inference rules within the SDSoC™ system compiler to assist the programmer to improve overall system performance through the following:

- Increased parallelism in the hardware function.

- Increased system parallelism and concurrency.

- Improved access to external memory from programmable logic.

- An understanding of the data motion network (default behavior and user specification).

There are many factors that affect overall system performance. A well-designed system generally balances computation and communication, so that all hardware components remain occupied doing meaningful work.

- Some applications are compute-bound; for these applications, concentrate on maximizing throughput and minimizing latency in hardware accelerators.

- Other applications might be memory-bound, in which case, you might need to restructure algorithms to increase temporal and spatial locality in the hardware; for example, adding copy-loops or `memcopy` to pull blocks of data into hardware rather than making random array accesses to external memory.

Control over the various aspects of optimization is provided through the use of pragmas in the code. For a complete description of the available pragmas, refer to the *SDx Pragma Reference Guide* (UG1253).

Related Information
Improving Hardware Function Parallelism
Increasing System Parallelism and Concurrency
Memory Access Optimizations
Data Motion Network Generation in SDSoC

Improving Hardware Function Parallelism

This section provides a concise introduction to writing efficient code that can be cross-compiled into programmable logic.

The SDSoC environment employs the Vivado® High-Level Synthesis (HLS) tool as a programmable logic cross-compiler to transform C/C++ functions into hardware.

By applying the principles described in this section, you can dramatically increase the performance of the synthesized functions, which can lead to significant increases in overall system performance for your application.

Top-Level Hardware Function Guidelines

This section describes coding guidelines to ensure that the Vivado HLS tool hardware function has a consistent interface with object code generated by the Arm® core GNU toolchain.

Use Standard C99 Data Types for Top-Level Hardware Function Arguments

1. Avoid using arrays of `bool`. An array of `bool` has a different memory layout between the GNU Arm cross compiler and the HLS tool.

2. Avoid using `hls::stream` at the hardware function top-level interface. This data type helps the HLS tool compiler synthesize efficient logic within a hardware function but does not apply to application software.

Omit HLS Interface Directives for Top-Level Hardware Function Arguments

Although supported, a top-level hardware function should not, in general, contain HLS interface pragmas. The `sdcc/sds++` (referred to as `sds++`) system compiler automatically generates the appropriate HLS tool interface directives.

There are two SDSoC environment pragmas you can specify for a top-level hardware function to guide the `sds++` system compiler to generate the required HLS tool interface directives:

- `#pragma SDS data zero_copy():` Use to generate a shared memory interface implemented as an AXI master interface in hardware.

- `#pragma SDS data access_pattern(argument:SEQUENTIAL):` Use to generate a streaming interface implemented as a FIFO interface in hardware.

If you specify the interface using `#pragma HLS interface` for a top-level function argument, the SDSoC environment does not generate the HLS tool interface directive for that argument; you should ensure that the generated hardware interface is consistent with all other function argument hardware interfaces.

RECOMMENDED: *Because a function with the incompatible HLS tool interface types can result in cryptic `sds++` system compiler error messages, it is strongly recommended (though not absolutely mandatory) that you omit HLS interface pragmas.*

Send Feedback

Using Vivado Design Suite HLS Libraries

This section describes how to use the Vivado HLS tool libraries with the SDSoC environment.

The HLS tool libraries are provided as source code with the HLS installation tool in the SDSoC environment. Consequently, you can use these libraries as you would any other source code that you plan to cross-compile for programmable logic using the HLS tool. In particular, ensure that the source code conforms to the rules described in Hardware Function Argument Types in the *SDSoC Environment Programmers Guide* (UG1278), which might require you to provide a C/C++ wrapper function to ensure the functions export a software interface to your application.

In the SDSoC IDE, the synthesizeable finite impulse response (FIR) example template for all basic platforms provides an example that uses an HLS tool library. You can find several additional code examples that employ the HLS tool libraries in the `samples/hls_lib` directory. For example, `samples/hls_lib/hls_math` contains an example to implement and uses a square root function.

The file `my_sqrt.h` contains:

```
#ifndef _MY_SQRT_H_
#define _MY_SQRT_H_

#ifdef __SDSVHLS__
#include "hls_math.h"
#else
// The hls_math.h file includes hdl_fpo.h which contains actual code and
// will cause linker error in the ARM compiler, hence we add the function
// prototypes here
static float sqrtf(float x);
#endif

void my_sqrt(float x, float *ret);

#endif // _SQRT_H_
```

The file `my_sqrt.cpp` contains:

```
#include "my_sqrt.h"

void my_sqrt(float x, float *ret)
{
    *ret = sqrtf(x);
}
```

The makefile has the commands to compile these files:

```
sds++ -c -hw my_sqrt –sds-pf zc702 my_sqrt.cpp
sds++ -c my_sqrt_test.cpp
sds++ my_sqrt.o my_sqrt_test.o -o my_sqrt_test.elf
```

Send Feedback

Increasing System Parallelism and Concurrency

Increasing the level of concurrent execution is a standard way to increase overall system performance, and increasing the level of parallel execution is a standard way to increase concurrency. Programmable logic is well-suited to implement architectures with application-specific accelerators that run concurrently, especially communicating through flow-controlled streams that synchronize between data producers and consumers.

In the SDSoC environment, you influence the macro-architecture parallelism at the function and data mover level, and the micro-architecture parallelism within hardware accelerators. By understanding how the `sds++/sdscc` (referred to as `sds++`) compiler infers system connectivity and data movers, you can structure application code and apply pragmas, as needed, to control hardware connectivity between accelerators and software, data mover selection, number of accelerator instances for a given hardware function, and task level software control.

You can control the micro-architecture parallelism, concurrency, and throughput for hardware functions within the Vivado HLS tool, or within the IP, you incorporate as C-callable and linkable libraries.

At the system level, the `sds++` compiler chains together hardware functions when the data flow between them does not require transferring arguments out of programmable logic and back to system memory.

For example, consider the code in the following figure, where `mmult` and `madd` functions have been selected for hardware.

*Figure 4:* **Hardware/Software Connectivity with Direct Connection**

Send Feedback
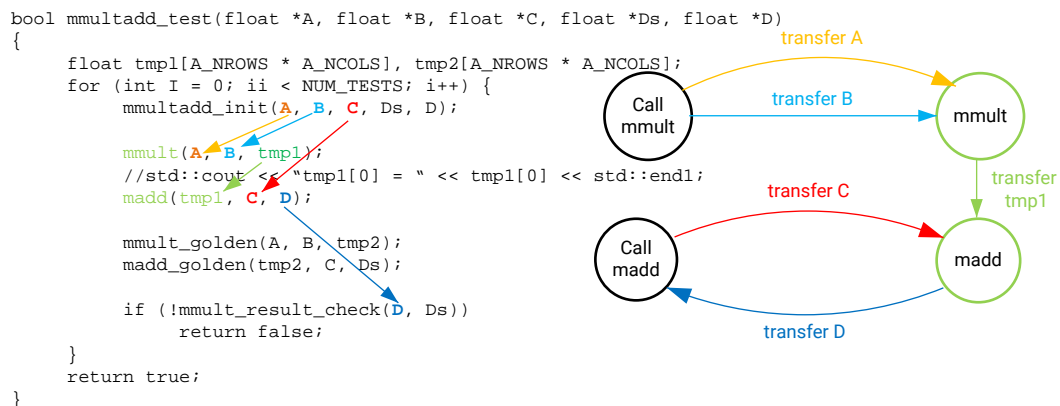
Because the intermediate array variable `tmp1` is used only to pass data between the two hardware functions, the `sds++` compiler chains the two functions together in hardware with a direct connection between them.

It is instructive to consider a timeline for the calls to hardware, as shown in the following figure:

*Figure 5:* **Timeline for mmult/madd Function Calls**
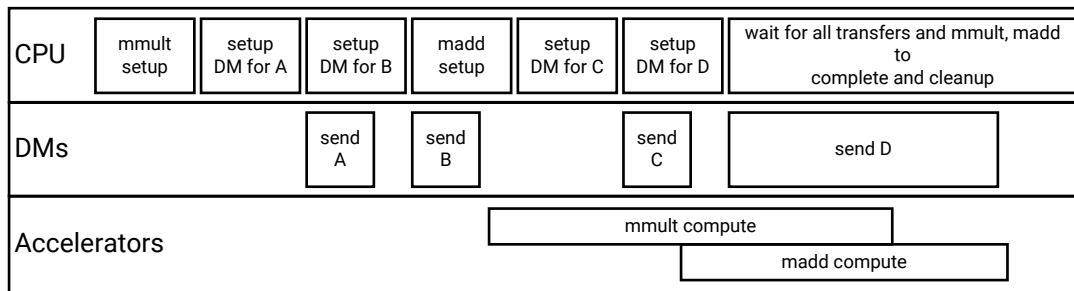
X14764-121417

The program preserves the original program semantics, but instead of the standard Arm core procedure calling sequence, each hardware function call is broken into multiple phases involving setup, execution, and cleanup, both for the data movers (DM) and the accelerators. The CPU, in turn, sets up each hardware function (that is, the underlying IP control interface), and the data transfers for the function call with non-blocking APIs, and then waits for all calls and transfers to complete.

In the example shown in the following figure, the `mmult` and `madd` functions run concurrently whenever their inputs become available. The ensemble of function calls is orchestrated in the compiled program by control code automatically generated by the `sds++` system compiler according to the program, data mover, and accelerator structure.

In general, it is impossible for the `sds++` system compiler to determine side effects of function calls in your application code (for example, `sds++` might have no access to source code for functions within linked libraries), so any intermediate access of a variable occurring lexically between hardware function calls requires the compiler to transfer data back to memory.

For example, an injudicious simple change to uncomment the debug print statement (in the "wrong place"), as shown in the following figure, can result in a significantly different data transfer graph and consequently, an entirely different generated system and application performance.

Send Feedback

Figure 6: **Hardware/Software Connectivity with Broken Direct Connection**
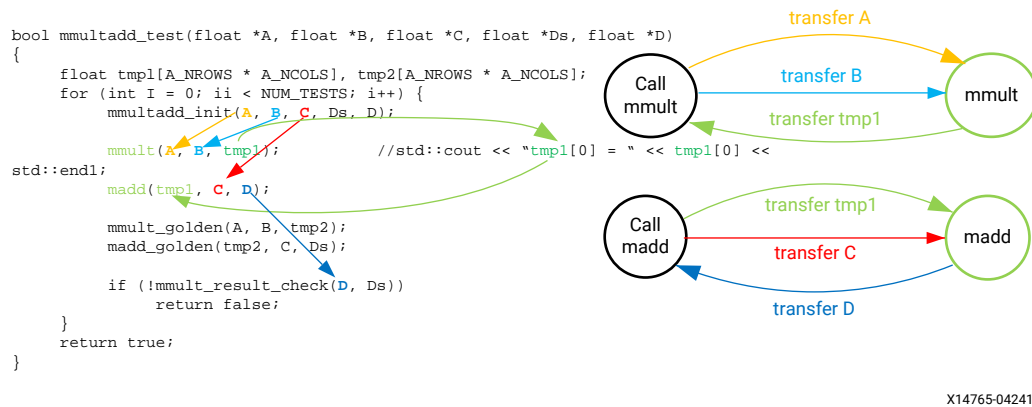
A program can invoke a single hardware function from multiple call sites. In this case, the `sds++` system compiler behaves as follows. If any of the function calls results in "direct connection" data flow, then the `sds++` system compiler creates an instance of the hardware function that services every similar direct connection, and an instance of the hardware function that services the remaining calls between memory (software) and PL.

To achieve high performance in PL, one of the best methods is structuring your application code with "direct connection" data flow between hardware functions. You can create deep pipelines of accelerators connected with data streams, increasing the opportunity for concurrent execution.

There is another way in which you can increase parallelism and concurrency using the `sds++` system compiler. You can direct the system compiler to create multiple instances of a hardware function by inserting the following pragma immediately preceding a call to the function.

```
#pragma SDS resource(<id>) // <id> a non-negative integer
```

This pragma creates a hardware instance that is referenced by `<id>`.

A simple code snippet that creates two instances of a hardware function `mmult` is as follows.

```
{
#pragma SDS resource(1)
 mmult(A, B, C); // instance 1
#pragma SDS resource(2)
 mmult(D, E, F); // instance 2
}
```

If creating multiple instances of an accelerator is not what you want, using the `sds_async` mechanism gives the programmer ability to handle the "hardware threads" explicitly to achieve very high levels of parallelism and concurrency. However, like any explicit multi-threaded programming model, it requires careful attention to synchronization details to avoid non-deterministic behavior or deadlocks. For more information , refer to the *SDSoC Environment Programmers Guide* (UG1278).

Send Feedback

Data Motion Network Generation in SDSoC

This section describes:

- Components that make up the data motion network in the SDSoC environment, and also provides guidelines to help you understand the data motion network generated by the SDSoC compiler.

- Guidelines to help you guide the data motion network generation by using appropriate SDSoC pragmas.

Every transfer between the software program and a hardware function requires a data mover, which consists of a hardware component that moves the data, and an operating system-specific library function. The following table lists supported data movers and various properties for each:

Table 2: **SDSoC Data Movers Table**

SDSoC Data Mover	Vivado® IP Data Mover	Accelerator IP Port Types	Transfer Size	Contiguous Memory Only
axi_dma_simple	axi_dma	bram, ap_fifo, axis	≤ 32 MB	Yes
axi_dma_sg	axi_dma	bram, ap_fifo, axis	N/A	No (but recommended)
axi_fifo	axi_fifo_mm_s	bram, ap_fifo, axis	≤ 300 B	No
zero_copy	accelerator IP	aximm master	N/A	Yes

- For array arguments, the data mover inference is based on transfer size, hardware function port mapping, and function call site information. The selection of data movers is a trade off between performance and resource, for example:

  - The `axi_dma_simple` data mover is the most efficient bulk transfer engine and supports up to 32 MB transfers, so it is best for transfers under that limit.

  - The `axi_fifo` data mover does not require as many hardware resources as the DMA, but due to its slower transfer rates, is preferred only for payloads of up to 300 bytes.

  - The `axi_dma_sg` (scatter-gather DMA) data mover provides slower DMA performance and consumes more hardware resources but has fewer limitations, and in the absence of any pragma directives, is often the best default data mover.

You can specify the data mover selection by inserting a pragma into program source immediately before the function declaration; for example:

```
#pragma SDS data data_mover(A:AXI_DMA_SIMPLE)
```

*Note:* `#pragma SDS` is always treated as a rule, not a hint, so you must ensure that their use conforms with the data mover requirements in the previous table.

The data motion network in the SDSoC environment is made up of three components:

- The memory system ports on the PS (A)

- Data movers between the PS and accelerators as well as among accelerators (B)

- The hardware interface on an accelerator (C)

The following figure illustrates these three components.

*Figure 7:* **Data Motion Network Components**



X22627-051019
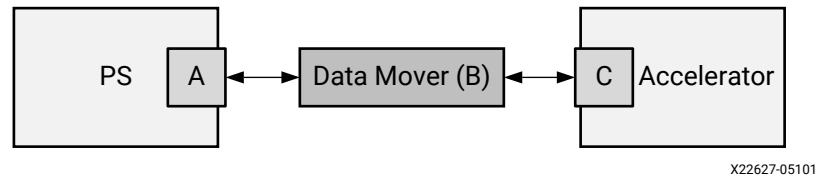
Without any SDS pragma, the SDSoC environment automatically generates the data motion network based on an analysis of the source code; however, the SDSoC environment also provides pragmas for you to guide the data motion network generation. See the *SDx Pragma Reference Guide* (UG1253).

# System Port

A system port connects a data mover to the PS. It can be an acceptance filter ID (AFI)—which corresponds to high-performance ports, memory interface generator (MIG)—which is a PL-based DDR memory controller, or a stream port on the Zynq®-7000 SoC or Zynq® UltraScale+™ MPSoC processors.

The AFI port is a non-cache-coherent port. If needed, cache coherency, such as cache flushing and cache invalidation, is maintained by software.

The AFI port depends on the cache requirement of the transferred data, the cache attribute of the data, and the data size. If the data is allocated with `sds_alloc_non_cacheable()` or `sds_register_dmabuf()`, it is better to connect to the AFI port to avoid cache flushing/invalidation.

*Note:* These functions can be found in the `sds_lib.h` and described in the environment APIs in the *SDSoC Environment Programmers Guide* (UG1278).

The SDSoC system compiler analyzes these memory attributes for the data transactions with the accelerator, and connects data movers to the appropriate system port.

To override the compiler decision, or in some cases where the compiler is not able to do such analysis, you can use the following pragma to specify the system port:

```
#pragma SDS data sys_port(arg:ip_port)
```

For example, the following function directly connects to a FIFO AXI interface. This is where `ip_port` can be either `AFI` or `MIG`:

```
#pragma SDS data sys_port:(A:fifo_S_AXI)*
void foo(int* A, int* B, int* C);
```

This function can also be used for a streaming interface:

```
#pragma SDS data sys_port:(A:stream_fifo_S_AXIS)*
void foo(int* A, int* B, int* C)
```

Note: For more information about AXI functionality in the Vivado® High Level Synthesis (HLS) tool, see the *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#)).

Use the following `sds++` system compiler command to see the list of system ports for the platform:

```
sds++ -sds-pf-info <platform> -verbose
```

Data Mover

The data mover transfers data between the PS and accelerators and among accelerators. The SDSoC environment can generate various types of data movers based on the properties and size of the data being transferred.

- **Scalar:** Scalar data is always transferred by the `AXI_LITE` data mover.

- **Array:** The `sds++` system compiler can generate the following data movers:

  - `AXI_DMA_SG`

  - `AXI_DMA_SIMPLE`

  - `AXI_FIFO`

  - `zero_copy` (accelerator-mastered AXI4 bus)

  - `AXI_LITE` (depending on the memory attributes and data size of the array)

  For example, if the array is allocated using `malloc()`, the memory is not physically contiguous, and SDSoC environment generates a scatter-gather DMA (`AXI_DMA_SG`); however, if the data size is less than 300 bytes, `AXI_FIFO` is generated instead because the data transfer time is less than `AXI_DMA_SG`, and it occupies much less PL resource.

- **Struct or Class:** The implementation of a `struct` depends on how the struct is passed to the hardware —passed by value, passed by reference, or as an array of `structs`—and the type of data mover selected. The following table shows the various implementations..

Table 3: **Struct Implementations**

Struct Pass Method	Default (no pragma)	#pragma SDS data zero_copy (arg)	#pragma SDS data zero_copy (arg[0:SIZE])	#pragma SDS data copy (arg)	#pragma SDS data copy (arg[0:SIZE])
pass by value (`struct RGB arg`)	Each field is flattened and passed individually as a scalar or an array.	This is not supported and will result in an error.	This is not supported and will result in an error.	The `struct` is packed into a single wide scalar.	Each field is flattened and passed individually as a scalar or an array. The value of `SIZE` is ignored.
pass by pointer (`struct RGB *arg`) or reference (`struct RGB &arg`)	Each field is flattened and passed individually as a scalar or an array.	The `struct` is packed into a single wide scalar and transferred as a single value. The data is transferred to the hardware accelerator through an AXI4 bus.	The `struct` is packed into a single wide scalar. The number of data values transferred to the hardware accelerator through an AXI4 bus is defined by the value of `SIZE`.	The `struct` is packed into a single wide scalar.	The `struct` is packed into a single wide scalar. The number of data values transferred to the hardware accelerator using an `AXIDMA_SG` or `AXIDMA_SIMPLE` is defined by the value of `SIZE`.
array of `struct` (`struct RGB arg[1024]`)	Each `struct` element of the array is packed into a single wide scalar.	Each `struct` element of the array is packed into a single wide scalar. The data is transferred to the hardware accelerator using an AXI4 bus.	Each `struct` element of the array is packed into a single wide scalar. The data is transferred to the hardware accelerator using an AXI4 bus. The value of `SIZE` overrides the array size and determines the number of data values transferred to the accelerator.	Each `struct` element of the array is packed into a single wide scalar. The data is transferred to the hardware accelerator using a data mover such as `AXI_DMA_SG` or `AXI_DMA_SIMPLE`.	Each `struct` element of the array is packed into a single wide scalar. The data is transferred to the hardware accelerator using a data mover such as `AXI_DMA_SG` or `AXI_DMA_SIMPLE`. The value of `SIZE` overrides the array size and determines the number of data values transferred to the accelerator.

Determining which data mover to use for transferring an array depends on two attributes of the array, data size and physical memory contiguity. For example, if the memory size is one MB and not physically contiguous (allocated by `malloc()`), use `AXI_DMA_SG`. The following table shows the applicability of these data movers.

Table 4: **Data Mover Selection**

Data Mover	Physical Memory Contiguity	Data Size (bytes)
`AXI_DMA_SG`	Either	> 300
`AXI_DMA_Simple`	Contiguous	< 32M
`AXI_FIFO`	Non-contiguous	< 300

Send Feedback

Normally, the SDSoC cross-compiler analyzes the array that is transferred to the hardware accelerator for these two attributes, and selects the appropriate data mover accordingly. However, there are cases where such analysis is not possible. At that time, the SDSoC cross-compiler issues a warning message, as shown in the following example, that states that it is unable to determine the memory attributes using SDS pragmas.

```
WARNING: [DMAnalysis 83-4492] Unable to determine the memory attributes
passed to rgb_data_in of function img_process at
C:/simple_sobel/src/main_app.c:84
```

The following pragma specifies the memory attributes:

```
#pragma SDS data mem_attribute(function_argument:contiguity)
```

The `contiguity` can be either `PHYSICAL_CONTIGUOUS` or `NON_PHYSICAL_CONTIGUOUS`. Use the following pragma to specify the data size:

```
#pragma SDS data copy(function_argument[offset:size])
```

The `size` can be a number or an arbitrary expression.

Zero Copy Data Mover

The zero copy data mover is unique because it covers both the accelerator interface and the data mover. The syntax of this pragma is:

```
#pragma SDS data zero_copy(arg[offset:size])
```

The `[offset:size]` is optional, and only needed if the data transfer size for an array cannot be determined at compile time.

By default, the SDSoC environment assumes `copy` semantics for an array argument, meaning the data is explicitly copied from the PS to the accelerator through a data mover. When this ZERO_COPY pragma is specified, SDSoC environment generates an `AXI-Master` interface for the specified argument on the accelerator, which grabs the data from the PS as specified in the accelerator code.

To use the ZERO_COPY pragma, the memory corresponding to the array must be physically contiguous, that is allocated with `sds_alloc`.

Accelerator Interface

The accelerator interface generated in depends on the data type of the argument.

- **Scalar:** For a scalar argument, the register interface is generated to pass in and/or out of the accelerator.

Send Feedback

- **Arrays:** The hardware interface on an accelerator for transferring an array can be either a RAM interface or a streaming interface, depending on how the accelerator accesses the data in the array.

  The RAM interface allows the data to be accessed randomly within the accelerator; however, it requires the entire array to be transferred to the accelerator before any memory accesses can happen within the accelerator. Moreover, the use of this interface requires block RAM resources on the accelerator side to store the array.

  The streaming interface, on the other hand, does not require memory to store the whole array, it allows the accelerator to pipeline the processing of array elements; for example, the accelerator can start processing a new array element while the previous ones are still being processed. However, the streaming interface requires the accelerator to access the array in a strict sequential order, and the amount of data transferred must be the same as the accelerator expects.

  The SDSoC environment, by default, generates the RAM interface for an array; however, the SDSoC environment provides pragmas to direct it to generate the streaming interface.

- **struct or class:** The implementation of a `struct` depends on how the `struct` is passed to the hardware—passed by value, passed by reference, or as an arrays of `structs`—and the type of data mover selected. The previous table shows the various implementations.

The following SDS pragma can be used to guide the interface generation for the accelerator.

```
#pragma SDS data access_pattern(function_argument:pattern)
```

Where `pattern` can either be RANDOM or SEQUENTIAL, and `arg` can be an array argument name of the accelerator function.

If an array argument's access pattern is specified as RANDOM, a RAM interface is generated. If it is specified as SEQUENTIAL, a streaming interface is generated.

Note:

- The default access pattern for an array argument is RANDOM.

- The specified access pattern must be consistent with the behavior of the accelerator function. For SEQUENTIAL access patterns, the function must access every array element in a strict sequential order.

- This pragma only applies to arguments without the `zero_copy` pragma.

Related Information
Zero Copy Data Mover

Optimizing the Hardware Function

The SDSoC™ environment employs heterogeneous cross-compilation, with Arm® CPU-specific compilers for the Zynq®-7000 and Zynq® UltraScale+™ MPSoC processor CPUs, and the Vivado® High-Level Synthesis (HLS) tool as a programmable logic (PL) cross-compiler for hardware functions. This section explains the default behavior and optimization directives associated with the HLS cross-compiler.

The default behavior of the HLS tool is to execute functions and loops in a sequential manner such that the hardware is an accurate reflection of the C/C++ code. Optimization directives can be used to enhance the performance of the hardware function, allowing pipelining which substantially increases the performance of the functions. This chapter outlines a general methodology for optimizing your design for high performance.

There are many possible goals when trying to optimize a design using the HLS tool. The methodology assumes you want to create a design with the highest possible performance, processing one sample of new input data every clock cycle, and so addresses those optimizations before the ones used for reducing latency or resources.

Detailed explanations of the optimizations discussed here are provided in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

RECOMMENDED: *Xilinx® recommends reviewing the methodology and obtaining a global perspective of hardware function optimization before reviewing the details of specific optimization.*

Related Information
Real-World Examples

Understanding the Hardware Function Optimization Methodology

Hardware functions are synthesized in the PL by the VivadoHLS tool compiler. This compiler automatically translates C/C++ code into an FPGA hardware implementation, and as with all compilers, does so using compiler defaults.

In addition to the compiler defaults, the HLS tool provides a number of optimizations that are applied to the C/C++ code through the use of pragmas in the code. This chapter explains the optimizations that can be applied and a recommended methodology for applying them.

There are two flows for optimizing the hardware functions:

- **Top-down flow:** In this flow, program decomposition into hardware functions proceeds top-down within the SDSoC environment, letting the system cross-compiler create pipelines of functions that automatically operate in dataflow mode. The microarchitecture for each hardware function is optimized using the HLS tool.

- **Bottom-up flow:** In this flow, the hardware functions are optimized in isolation from the system using the HLS tool compiler provided in the Vivado Design Suite. The hardware functions are analyzed, optimizations directives can be applied to create an implementation other than the default, and the resulting optimized hardware functions are then incorporated into the SDSoC environment.

The bottom-up flow is often used in organizations where the software and hardware are optimized by different teams and can be used by software programmers who wish to take advantage of existing hardware implementations from within their organization or from partners. Both flows are supported, and the same optimization methodology is used in either case. Both workflows result in the same high-performance system. Xilinx sees the choice as a workflow decision made by individual teams and organizations and provides no recommendation on which flow to use.

The optimization methodology for hardware functions is shown in the following figure:

Figure 8: **Hardware Function Optimization Methodology**

Simulate Design	- Validate The C function
Synthesize Design	- Baseline design
1: Initial Optimizations	- Define interfaces (and data packing) - Define loop trip counts
2: Pipeline for Performance	- Pipeline and dataflow
3: Optimize Structures for Performance	- Partition memories and ports - Remove false dependencies
4: Reduce Latency	- Optionally specify latency requirements
5: Improve Area	- Optionally recover resources through sharing

X15638-110617

This figure details all the steps in the methodology and the subsequent sections in this chapter explain the optimizations in detail.

Note: When optimizing code, the build should be done in Release configuration and not Debug configuration due to the -O optimization. Debug should be used to confirm that the functions and system are working, and Release should be used to confirm performance optimizations.

IMPORTANT! *Designs will reach the optimum performance after Step 3.*

- Step 1: See Optimizing Metrics, and review the topics in this chapter prior to attempting to optimize.

- Step 2: See Pipelining for Performance

- Step 3: See Optimizing Structures for Performance

- Step 4: See Reducing Latency. This step is used to minimize or specifically control the latency through the design and is required only for applications where this is of concern.

- Step 5: See Reducing Area. This topic explains how to reduce the resources required for hardware implementation and is typically applied only when larger hardware functions fail to implement in the available resources. The FPGA has a fixed number of resources, and there is typically no benefit in creating a smaller implementation if the performance goals have been met.

Baselining Hardware Functions

Before you perform any hardware function optimization, it is important to understand the performance achieved with the existing code and compiler defaults, and appreciate how performance is measured. Select the functions to implement hardware and build the project.

After you build a project, a report is available in the Hardware Reports section of the IDE. The report is also available from `<project name>/<build_config>/_sds/vhls/ <hw_function>/solution/syn/report/<hw_function>.rpt`. This report details the performance estimates and usage estimates.

The key factors in the performance estimates are ordered by the timing, interval (which includes loop initiation interval), and latency.

- The timing summary shows the target and estimated clock period. If the estimated clock period is greater than the target, *the hardware will not function at this clock period*. Reduce the clock period by using the **Project Settings → Data Motion Network Clock Frequency** option. Alternatively, because this is only an estimate at this point in the flow, it might be possible to proceed through the remainder of the flow if the estimate only exceeds the target by 20%. Further optimizations are applied when the bitstream is generated, and it might still be possible to satisfy the timing requirements. However, this is an indication that the hardware function is not guaranteed to meet timing.

- The function initiation interval (II) is the number of clock cycles before the function can accept new inputs and is generally *the most critical performance metric in any system*. In an ideal hardware function, the hardware processes data at the rate of one sample per clock cycle. If the largest data set passed into the hardware is size <N> (for example: `my_array[<N>]`), the most optimal II is `<N> + 1`. This means the hardware function processes <N> data samples in <N> clock cycles and can accept new data one clock cycle after all <N> samples are processed. It is possible to create a hardware function with an `II <N>`; however, this requires greater resources in the programmable logic (PL) with typically little benefit. Often, this hardware function is ideal because it consumes and produces data at a rate faster than the rest of the system.

- The loop initiation interval is the number of clock cycles before the next iteration of a loop starts to process data. This metric becomes important as you delve deeper into the analysis to locate and remove performance bottlenecks.

- The latency is the number of clock cycles required for the function to compute all output values. This is simply the lag from when data is applied until when it is ready. For most applications this is of little concern, especially when the latency of the hardware function vastly exceeds that of the software or system functions, such as DMA; however, it is a performance metric that you should review and confirm is not an issue for your application.

- The loop iteration latency is the number of clock cycles it takes to complete one iteration of a loop, and the loop latency is the number of cycles to execute all iterations of the loop. See Optimizing Metrics.

The Area Estimates section of the report details how many resources are required in the PL to implement the hardware function and how many are available. The key metric here is the Utilization (%). *The utilization (%) should not exceed 100% for any of the resources*. A figure greater than 100% means there are not enough resources to implement the hardware function, and a larger FPGA device might be required. As with the timing, at this point in the flow, this is an estimate. If the numbers are only slightly over 100%, it might be possible for the hardware to be optimized during bitstream creation.

You should already have an understanding of the required performance of your system and what metrics are required from the hardware functions; however, even if you are unfamiliar with hardware concepts such as clock cycles, you are now aware that the highest performing hardware functions have an `II = <N> + 1`, where <N> is the largest data set processed by the function. With an understanding of the current design performance and a set of baseline performance metrics, you can now proceed to apply optimization directives to the hardware functions.

Optimizing Metrics

The following table shows the first directive for you to consider adding to your design.

Table 5: **Optimization Strategy Step 1: Optimization For Metrics**

Directives and Configurations	Description
LOOP_TRIPCOUNT	Used for loops that have variable bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.

A common issue when hardware functions are first compiled is report files showing the latency and interval as a question mark "?" rather than as numerical values. If the design has loops with variable loop bounds, the compiler cannot determine the latency or II and uses the "?" to indicate this condition. Variable loop bounds are where the loop iteration limit cannot be resolved at compile time, as when the loop iteration limit is an input argument to the hardware function, such as variable height, width, or depth parameters.

To resolve this condition, use the hardware function report to locate the lowest level loop, which fails to report a numerical value, and use the `LOOP_TRIPCOUNT` directive to apply an estimated tripcount. The tripcount is the minimum, average, and/or maximum number of expected iterations. This allows values for latency and interval to be reported and allows implementations with different optimizations to be compared.

Because the `LOOP_TRIPCOUNT` value is only used for reporting and has no impact on the resulting hardware implementation, any value can be used. However, an accurate expected value results in more useful reports.

Pipelining for Performance

The next stage in creating a high-performance design is to pipeline the functions, loops, and operations. Pipelining results in the greatest level of concurrency and a very high level of performance. The following table shows the directives you can use for pipelining.

Table 6: **Optimization Strategy Step 2: Pipeline for Performance**

Directives and Configurations	Description
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
DATAFLOW	Enables task-level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.
RESOURCE	Specifies pipelining on the hardware resource used to implement a variable (array, arithmetic operation).
Config Compile	Allows loops to be automatically pipelined based on their iteration count when using the bottom-up flow.

At this stage of the optimization process, you want to create as much concurrent operation as possible. You can apply the PIPELINE directive to functions and loops. You can use the DATAFLOW directive at the level that contains the functions and loops to make them work in parallel. Although rarely required, the RESOURCE directive can be used to squeeze out the highest levels of performance.

A recommended strategy is to work from the bottom up and be aware of the following:

- Some functions and loops contain sub-functions. If the sub-function is not pipelined, the function above it might show limited improvement when it is pipelined. The non-pipelined sub-function will be the limiting factor.

- Some functions and loops contain sub-loops. When you use the PIPELINE directive, the directive automatically unrolls all loops in the hierarchy below. This can create a great deal of logic. It might make more sense to pipeline the loops in the hierarchy below.

- For cases where it does make sense to pipeline the upper hierarchy and unroll any loops lower in the hierarchy, loops with variable bounds cannot be unrolled, and any loops and functions in the hierarchy above these loops cannot be pipelined. To address this issue, pipeline these loops with variable bounds, and use the DATAFLOW optimization to ensure the pipelined loops operate concurrently to maximize the performance of the tasks that contains the loops. Alternatively, rewrite the loop to remove the variable bound. Apply a maximum upper bound with a conditional break.

The basic strategy at this point in the optimization process is to pipeline the tasks (functions and loops) as much as possible. For detailed information on which functions and loops to pipeline, see Hardware Function Pipeline Strategies.

Although not commonly used, you can also apply pipelining at the operator level. For example, wire routing in the FPGA can introduce large and unanticipated delays that make it difficult for the design to be implemented at the required clock frequency. In this case, you can use the RESOURCE directive to pipeline specific operations such as multipliers, adders, and block RAM to add additional pipeline register stages at the logic level and allow the hardware function to process data at the highest possible performance level without the need for recursion.

*Note***:** The configuration commands are used to change the optimization default settings and are only available from within the Vivado® HLS tool when using a bottom-up flow. For more details, see the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Hardware Function Pipeline Strategies

The key optimization directives for obtaining a high-performance design are the PIPELINE and DATAFLOW directives. This section discusses in detail how to apply these directives for various C code architectures.

There are two types of C/C++ functions: those that are frame-based and those that are sampled-based. No matter which coding style is used, the hardware function can be implemented with the same performance in both cases. The difference is only in how the optimization directives are applied.

Frame-Based C Code

The primary characteristic of a frame-based coding style is that the function processes multiple data samples—a frame of data—typically supplied as an array or pointer with data accessed through pointer arithmetic during each transaction (a transaction is considered to be one complete execution of the C function). In this coding style, the data is typically processed through a series of loops or nested loops.

The following is an example outline of frame-based C code:

```
void foo(
  data_t in1[HEIGHT][WIDTH],
  data_t in2[HEIGHT][WIDTH],
  data_t out[HEIGHT][WIDTH] {
  Loop1: for(int i = 0; i < HEIGHT; i++) {
    Loop2: for(int j = 0; j < WIDTH; j++) {
      out[i][j] = in1[i][j] * in2[i][j];
        Loop3: for(int k = 0; k < NUM_BITS; k++) {
          . . . .
        }
    }
  }
}
```

When seeking to pipeline any C/C++ code for maximum performance in hardware, you want to place the pipeline optimization directive at the level where a sample of data is processed.

The above example is representative of code used to process an image or video frame and can be used to highlight how to effectively pipeline hardware functions. Two sets of input are provided as frames of data to the function, and the output is also a frame of data. There are multiple locations where this function can be pipelined:

- At the level of function foo.

- At the level of loop Loop1.

- At the level of loop Loop2.

- At the level of loop Loop3.

There are advantages and disadvantages for placing the PIPELINE directive at various locations. Understanding them helps guide you to the best location to place the pipeline directive in your code.

- **Function Level:** The function accepts a frame of data as input (`in1` and `in2`). If the function is pipelined with `II = 1`—read a new set of inputs every clock cycle—this informs the compiler to read all `HEIGHT*WIDTH` values of `in1` and `in2` in a single clock cycle. This is a lot of data to read in one cycle and is unlikely the design you want.

  If the PIPELINE directive is applied to function foo, all loops in the hierarchy below this level must be unrolled. This is a requirement for pipelining, namely, there cannot be sequential logic inside the pipeline. This would create `HEIGHT*WIDTH*NUM_ELEMENT` copies of the logic, which would lead to a large design.

  Because the data is accessed in a sequential manner, the arrays on the interface to the hardware function can be implemented as multiple types of hardware interface:

  - Block RAM interface

  - AXI4 interface

  - AXI4-Lite interface

Send Feedback

- AXI4-Stream interface

- FIFO interface

A block RAM interface can be implemented as a dual-port interface supplying two samples per clock. The other interface types can only supply one sample per clock. This would result in a bottleneck; there would be a highly parallel but large hardware design unable to process all the data in parallel, resulting in a waste of hardware resources.

- **Loop1 Level:** The logic in Loop1 processes an entire row of the two-dimensional matrix. Placing the PIPELINE directive here would create a design which seeks to process one row in each clock cycle. Again, this would unroll the loops below and create additional logic. To make use of the additional hardware, transfer an entire row of data each clock cycle: an array of `HEIGHT` data words, with each word being `WIDTH*` <number of bits in data_t> bits wide.

  Because it is unlikely the host code running on the PS can process such large data words, this would again be a case where there are many highly parallel hardware resources that cannot operate in parallel due to bandwidth limitations.

- **Loop2 Level:** The logic in Loop2 seeks to process one sample from the arrays. In an image algorithm, this is the level of a single pixel. This is the level to pipeline if the design is to process one sample per clock cycle. This is also the rate at which the interfaces consume and produce data to and from the PS.

  This causes Loop3 to be completely unrolled and process one sample per clock. It is a requirement that all the operations in Loop3 execute in parallel. In a typical design, the logic in Loop3 is a shift register or is processing bits within a word. To execute at one sample per clock, you want these processes to occur in parallel and hence you want to unroll the loop. The hardware function created by pipelining Loop2 processes one data sample per clock and creates parallel logic only where needed to achieve the required level of data throughput.

- **Loop3 Level:** As stated above, given that Loop2 operates on each data sample or pixel, Loop3 will typically be doing bit-level or data shifting tasks, so this level is doing multiple operations per pixel. Pipelining this level would mean performing each operation in this loop once per clock and thus `NUM_BITS` clocks per pixel: processing at the rate of multiple clocks per pixel or data sample.

  For example, Loop3 might contain a shift register holding the previous pixels required for a windowing or convolution algorithm. Adding the PIPELINE directive at this level informs the compiler to shift one data value every clock cycle. The design would only return to the logic in Loop2 and read the next inputs after `NUM_BITS` iterations resulting in a very slow data processing rate.

  The ideal location to pipeline in this example is Loop2.

  When dealing with frame-based code you will want to pipeline at the loop level and typically pipeline the loop that operates at the level of a sample. If in doubt, place a print command into the C code and to confirm this is the level you wish to execute on each clock cycle.

For cases where there are multiple loops at the same level of hierarchy—the example above shows only a set of nested loops—the best location to place the PIPELINE directive can be determined for each loop and then the DATAFLOW directive applied to the function to ensure each of the loops executes in a concurrent manner.

Sample-Based C Code

An example outline of sample-based C code is shown below. The primary characteristic of this coding style is that the function processes a single data sample during each transaction.

```
void foo (data_t *in, data_t *out) {
  static data_t acc;
  Loop1: for (int i=N-1;i>=0;i--) {
    acc+= ..some calculation..;
  }
  *out=acc>>N;
}
```

Another characteristic of sample-based coding style is that the function often contains a static variable: a variable whose value must be remembered between invocations of the function, such as an accumulator or sample counter.

With sample-based code, the location of the PIPELINE directive is clear, namely, to achieve an `II = 1` and process one data value each clock cycle, for which the function must be pipelined.

This unrolls any loops inside the function and creates additional hardware logic, but there is no way around this. If `Loop1` is not pipelined, it takes a minimum of `N` clock cycles to complete. Only then can the function read the next `x` input value.

When dealing with C code that processes at the sample level, the strategy is always to pipeline the function.

In this type of coding style, the loops are typically operating on arrays and performing a shift register or line buffer functions. It is not uncommon to partition these arrays into individual elements as discussed in Optimizing Structures for Performance to ensure all samples are shifted in a single clock cycle. If the array is implemented in a block RAM, only a maximum of two samples can be read or written in each clock cycle, creating a data processing bottleneck.

The solution here is to pipeline function `foo`. Doing so results in a design that processes one sample per clock.

Optimizing Structures for Performance

C code can contain descriptions that prevent a function or loop from being pipelined with the required performance. This is often implied by the structure of the C code or the default logic structures used to implement the PL. In some cases, this might require a code modification, but in most cases these issues can be addressed using additional optimization directives.

The following example shows a case where an optimization directive is used to improve the structure of the implementation and the performance of pipelining. In this initial example, the PIPELINE directive is added to a loop to improve the performance of the loop. This example code shows a loop being used inside a function.

```
#include "bottleneck.h"
dout_t bottleneck(...) {
  ...
  SUM_LOOP: for(i=3;i<N;i=i+4) {
#pragma HLS PIPELINE
    sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];
  }
  ...
}
```

When the code above is compiled into hardware, the following message appears as output:

```
INFO: [SCHED 61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
I
```

The issue in this example is that arrays are implemented using the efficient block RAM resources in the PL. This results in a small, cost-efficient, fast design. The disadvantage of block RAM is that, like other memories, such as DDR or SRAM, they have a limited number of data ports, typically a maximum of two.

In the code above, four data values from mem are required to compute the value of sum. Because mem is an array and implemented in a block RAM that only has two data ports, only two values can be read (or written) in each clock cycle. With this configuration, it is impossible to compute the value of sum in one clock cycle and thus consume or produce data with an II of 1 (process one data sample per clock).

The memory port limitation issue can be solved by using the ARRAY_PARTITION directive on the mem array. This directive partitions arrays into smaller arrays, improving the data structure by providing more data ports and allowing a higher performance pipeline.

With the additional directive shown below, array `mem` is partitioned into two dual-port memories so that all four reads can occur in one clock cycle. There are multiple options to partitioning an array. In this case, cyclic partitioning with a factor of two ensures the first partition contains elements 0, 2, 4, and so forth, from the original array and the second partition contains elements 1, 3, 5, and so forth. Because the partitioning ensures there are now two dual-port block RAMs (with a total of four data ports), this allows elements 0, 1, 2, and 3 to be read in a single clock cycle.

Note: The ARRAY_PARTITION directive cannot be used on arrays which are arguments of the function selected as an accelerator.

```
#include "bottleneck.h"
dout_t bottleneck(...) {
#pragma HLS ARRAY_PARTITION variable=mem cyclic factor=2 dim=1
  ...
  SUM_LOOP: for(i=3;i<N;i=i+4) {
#pragma HLS PIPELINE
    sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];
  }
  ...
}
```

Other such issues might be encountered when trying to pipeline loops and functions. The following table lists the directives that are likely to address these issues by helping to reduce bottlenecks in data structures.

Table 7: **Optimization Strategy Step 3: Optimize Structures for Performance**

Directives and Configurations	Description
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers to improve access to data and remove block RAM bottlenecks.
DEPENDENCE	Provides additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).
INLINE	Inlines a function, removing all function hierarchy. Enables logic optimization across function boundaries and improves latency/interval by reducing function call overhead.
UNROLL	Unrolls for-loops to create multiple independent operations rather than a single collection of operations, allowing greater hardware parallelism. This also allows for partial unrolling of loops.
Config Array Partition	This configuration determines how arrays are automatically partitioned, including global arrays, and if the partitioning impacts array ports.
Config Compile	Controls synthesis specific optimizations such as the automatic loop pipelining and floating point math optimizations.
Config Schedule	Determines the effort level to use during the synthesis scheduling phase, the verbosity of the output messages, and to specify if II should be relaxed in pipelined tasks to achieve timing.
Config Unroll	Allows all loops below the specified number of loop iterations to be automatically unrolled.

In addition to the ARRAY_PARTITION directive, the configuration for array partitioning can be used to automatically partition arrays.

Send Feedback

The DEPENDENCE directive might be required to remove implied dependencies when pipelining loops. Such dependencies are reported by message `SCHED-68`.

```
@W [SCHED-68] Target II not met due to carried dependence(s)
```

The INLINE directive removes function boundaries. This can be used to bring logic or loops up one level of hierarchy. It might be more efficient to pipeline the logic in a function by including it in the function above it, and merging loops into the function above them where the DATAFLOW optimization can be used to execute all the loops concurrently without the overhead of the intermediate sub-function call. This might lead to a higher performing design.

The UNROLL directive might be required for cases where a loop cannot be pipelined with the required II. If a loop can only be pipelined with `II = 4`, it will constrain the other loops and functions in the system to be limited to `II = 4`. In some cases, it might be worth unrolling or partially unrolling the loop to create more logic, and remove a potential bottleneck. If the loop can only achieve `II = 4`, unrolling the loop by a factor of 4 creates logic that can process four iterations of the loop in parallel and achieve `II = 1`.

The configuration commands are used to change the optimization default settings and are only available from within the Vivado HLS tool when using a bottom-up flow. For more details, see the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902). If optimization directives cannot be used to improve the initiation interval, it might require changes to the code. See examples of such in the same guide.

Reducing Latency

When the compiler finishes minimizing the initiation interval (II), it automatically seeks to minimize the latency. The optimization directives listed in the following table can help specify a particular latency or inform the compiler to achieve a latency lower than the one produced, namely, instruct the compiler to satisfy the latency directive even if it results in a higher II. This could result in a lower performance design.

Latency directive are generally not required because most applications have a required throughput but no required latency. When hardware functions are integrated with a processor, the latency of the processor is generally the limiting factor in the system.

If the loops and functions are not pipelined, the throughput is limited by the latency because the task does not start reading the next set of inputs until the current task has completed.

Table 8:   **Optimization Strategy Step 4: Reduce Latency**

Directive	Description
LATENCY	Allows a minimum and maximum latency constraint to be specified.

Send Feedback

Table 8: **Optimization Strategy Step 4: Reduce Latency** *(cont'd)*

Directive	Description
LOOP_FLATTEN	Allows nested loops to be collapsed into a single loop. This removes the loop transition overhead and improves the latency. Nested loops are automatically flattened when the PIPELINE directive is applied.
LOOP_MERGE	Merges consecutive loops to reduce overall latency, increase logic resource sharing, and improve logic optimization.

The loop optimization directives can be used to flatten a loop hierarchy or merge consecutive loops together. The benefit to the latency is due to the fact that it typically costs a clock cycle in the control logic to enter and leave the logic created by a loop. The fewer the number of transitions between loops, the lesser number of clock cycles a design takes to complete.

Reducing Area

In hardware, the number of resources required to implement a logic function is referred to as the design area. Design area also refers to how much area the resource used on the fixed-size PL fabric. The area is important when the hardware is too large to be implemented in the target device, and when the hardware function consumes a very high percentage (> 90%) of the available area. This can result in difficulties when trying to wire the hardware logic together because the wires themselves require resources.

After meeting the required performance target or initiation interval (II), the next step might be to reduce the area while maintaining the same performance. This step can be optimal because there is no advantage to reducing the area if the hardware function is operating at the required performance, and no other hardware functions are to be implemented in the remaining space in the PL.

The most common area optimization is the optimization of dataflow memory channels to reduce the number of block RAM resources required to implement the hardware function. Each device has a limited number of block RAM resources.

If you used the DATAFLOW optimization, and the compiler cannot determine whether the tasks in the design are streaming data, it implements the memory channels between dataflow tasks using ping-pong buffers. These require two block RAMs each of size <N>, where <N> is the number of samples to be transferred between the tasks (typically the size of the array passed between tasks). If the design is pipelined and the data is streaming from one task to the next with values produced and consumed in a sequential manner, you can greatly reduce the area by using the STREAM directive to specify that the arrays are to be implemented in a streaming manner that uses a simple FIFO for which you can specify the depth. FIFOs with a small depth are implemented using registers and the PL fabric has many registers.

Send Feedback

For most applications, the depth can be specified as 1, which results in the memory channel being implemented as a simple register. If the algorithm implements data compression or extrapolation, where some tasks consume more data than they produce or produce more data than they consume, some arrays must be specified with a higher depth:

- For tasks which produce and consume data at the same rate, specify the array between them to stream with a depth of 1.

- For tasks which reduce the data rate by a factor of X-to-1, specify arrays at the input of the task to stream with a depth of X. All arrays prior to this in the function should also have a depth of X to ensure the hardware function does not stall because the FIFOs are full.

- For tasks which increase the data rate by a factor of 1-to-Y, specify arrays at the output of the task to stream with a depth of Y. All arrays after this in the function should also have a depth of Y to ensure the hardware function does not stall because the FIFOs are full.

Note: If the depth is set too small, the hardware function will stall (hang) during hardware emulation, resulting in lower performance or even deadlock in some cases, due to full FIFOs causing the rest of the system to wait.

The following table lists the other directives and configurations to consider when attempting to minimize the resources used to implement the design.

Table 9: **Optimization Strategy Step 5: Reduce Area**

Directives and Configurations	Description
ALLOCATION	Specifies a limit for the number of operations, hardware resources, or functions used. This can force the sharing of hardware resources but might increase latency.
ARRAY_MAP	Combines multiple smaller arrays into a single large array to help reduce the number of block RAM resources.
ARRAY_RESHAPE	Reshapes an array from one with many elements to one with greater word width. Useful for improving block RAM accesses without increasing the number of block RAM.
DATA_PACK	Packs the data fields of an internal struct into a single scalar with a wider word width, allowing a single control signal to control all fields.
LOOP_MERGE	Merges consecutive loops to reduce overall latency, increase sharing, and improve logic optimization.
OCCURRENCE	Used when pipelining functions or loops to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop.
RESOURCE	Specifies that a specific hardware resource (core) is used to implement a variable (array, arithmetic operation).
STREAM	Specifies that a specific memory channel is to be implemented as a FIFO with an optional specific depth.
Config Bind	Determines the effort level to use during the synthesis binding phase and can be used to globally minimize the number of operations used.
Config Dataflow	This configuration specifies the default memory channel and FIFO depth in dataflow optimization.

The ALLOCATION and RESOURCE directives are used to limit the number of operations and to select which cores (hardware resources) are used to implement the operations. For example, you could limit the function or loop to using only one multiplier, and specify it to be implemented using a pipelined multiplier.

If the ARRAY_PARITION directive is used to improve the initiation interval you might want to consider using the ARRAY_RESHAPE directive instead. The ARRAY_RESHAPE optimization performs a similar task to array partitioning, however, the reshape optimization recombines the elements created by partitioning into a single block RAM with wider data ports. This might prevent an increase in the number of block RAM resources required.

If the C code contains a series of loops with similar indexing, merging the loops with the LOOP_MERGE directive might allow some optimizations to occur. Finally, in cases where a section of code in a pipeline region is only required to operate at an initiation interval lower than the rest of the region, the OCCURENCE directive is used to indicate that this logic can be optimized to execute at a lower rate.

Note: The configuration commands are used to change the optimization default settings and are only available from within the Vivado HLS tool when using a bottom-up flow. For more information, see the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Design Optimization Workflow

Before performing any optimizations, it is recommended to create a new build configuration within the project. Using different build configurations allows one set of results to be compared against a different set of results. In addition to the standard Debug and Release configurations, custom configurations with more useful names (for example, `Opt_ver1` and `UnOpt_ver`) might be created in the window using the **Project Settings→ Manage Build Configurations for the Project** toolbar button.

Different build configurations allow you to compare not only the results, but also the log files and even output RTL files used to implement the FPGA (the RTL files are only recommended for users very familiar with hardware design).

The basic optimization strategy for a high-performance design is:

- Create an initial or baseline design.

- Pipeline the loops and functions. Apply the `DATAFLOW` optimization to execute loops and functions concurrently.

- Address any issues that limit pipelining, such as array bottlenecks and loop dependencies (with ARRAY_PARTITION and DEPENDENCE directives).

- Specify a specific latency or reduce the size of the dataflow memory channels and use the ALLOCATION and RESOURCES directives to further reduce area.

Note: It might sometimes be necessary to make adjustments to the code to meet performance.

In summary, the goal is to always meet performance first, before reducing area. If the strategy is to create a design with the fewest resources, omit the steps to improving performance, although the baseline results might be very close to the smallest possible design.

Throughout the optimization process, it is highly recommended to review the console output (or log file) after compilation. When the compiler cannot reach the specified performance goals of an optimization, it automatically relaxes the goals (except the clock frequency) and creates a design with the goals that can be satisfied. It is important to review the output from the compilation log files and reports to understand what optimizations have been performed.

For specific details on applying optimizations, refer to *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Optimization Guidelines

This section documents several fundamental optimization techniques to enhance hardware function performance using the Vivado HLS tool. These techniques include: function inlining, loop and function pipelining, loop unrolling, increasing local memory bandwidth, and streaming data flow between loops and functions.

Related Information
Function Inlining
Loop Pipelining and Loop Unrolling
Increasing Local Memory Bandwidth
Data Flow Pipelining

Function Inlining

Similar to function inlining of software functions, it can be beneficial to inline hardware functions.

Function inlining replaces a function call by substituting a copy of the function body after resolving the actual and formal arguments. After that, the inlined function is dissolved and no longer appears as a separate level of hierarchy. Function inlining allows operations within the inlined function be optimized more effectively with surrounding operations, thus improving the overall latency or the initiation interval for a loop.

To inline a function, put `#pragma HLS inline` at the beginning of the body of the desired function. The following code snippet directs the Vivado HLS tool to inline the `mmult_kernel` function:

```
void mmult_kernel(float in_A[A_NROWS][A_NCOLS],
                  float in_B[A_NCOLS][B_NCOLS],
                  float out_C[A_NROWS][B_NCOLS])
{
#pragma HLS INLINE
    int index_a, index_b, index_d;
    // rest of code body omitted
}
```

Loop Pipelining and Loop Unrolling

Both loop pipelining and loop unrolling improve the performance of the hardware functions by exploiting the parallelism between loop iterations. The basic concepts of loop pipelining and loop unrolling and example codes to apply these techniques are shown and the limiting factors to achieve optimal performance using these techniques are discussed.

Loop Pipelining

In sequential languages such as C/C++, the operations in a loop are executed sequentially, and the next iteration of the loop can only begin when the last operation in the current loop iteration is complete. Loop pipelining allows the operations in a loop to be implemented in a concurrent manner as shown in the following figure.

Figure 9: **Loop Pipelining**

Send Feedback
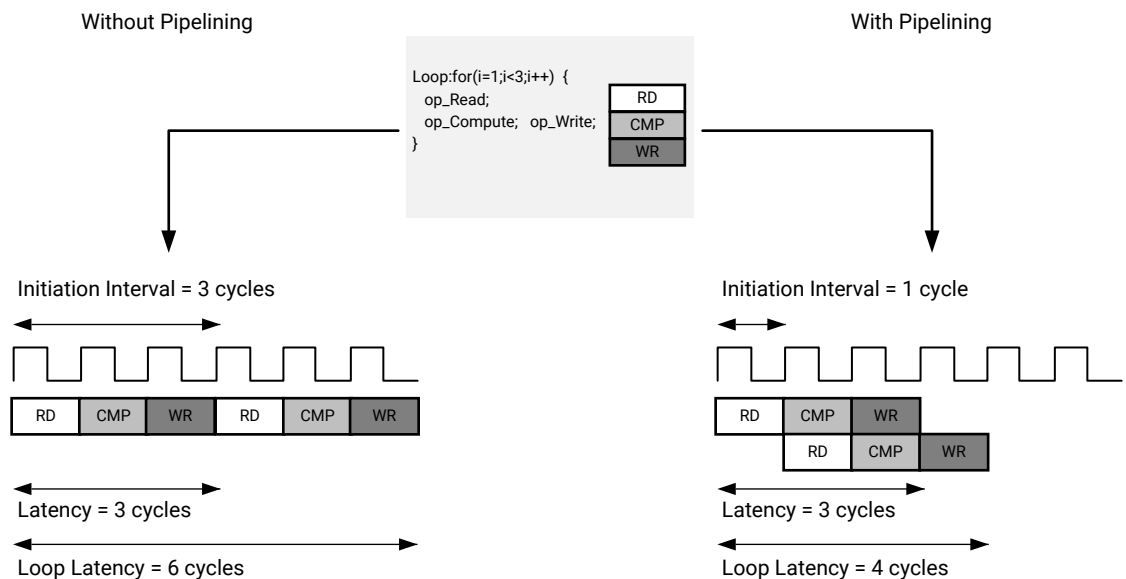
As shown in the previous figure, without pipelining, there are three clock cycles between the two RD operations, and it requires six clock cycles for the entire loop to finish. However, with pipelining, there is only one clock cycle between the two RD operations, and it requires four clock cycles for the entire loop to finish, that is, the next iteration of the loop can start before the current iteration is finished.

An important term for loop pipelining is called initiation interval (II), which is the number of clock cycles between the start times of consecutive loop iterations. In the above figure, the II is one because there is only one clock cycle between the start times of consecutive loop iterations.

To pipeline a loop, put `#pragma HLS pipeline` at the beginning of the loop body, as illustrated in the following code snippet. The Vivado HLS tool tries to pipeline the loop with minimum II.

```
for (index_a = 0; index_a < A_NROWS; index_a++) {
    for (index_b = 0; index_b < B_NCOLS; index_b++) {
#pragma HLS PIPELINE II=1
        float result = 0;
        for (index_d = 0; index_d < A_NCOLS; index_d++) {
            float product_term = in_A[index_a][index_d] * in_B[index_d]
[index_b];
            result += product_term;
        }
        out_C[index_a * B_NCOLS + index_b] = result;
    }
}
```

Loop Unrolling

Loop unrolling is another technique to exploit parallelism between loop iterations. It creates multiple copies of the loop body and adjust the loop iteration counter accordingly. The following code snippet shows a normal rolled loop:

```
int sum = 0;
for(int i = 0; i < 10; i++) {
    sum += a[i];
}
```

After the loop is unrolled by a factor of two, the loop becomes:

```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

Unrolling a loop by a factor of <N> creates <N> copies of the loop body, the loop variable referenced by each copy is updated accordingly (such as the a[i+1] in the above code snippet), and the loop iteration counter is also updated accordingly (such as the i+=2 in the above code snippet).

Send Feedback

Loop unrolling creates more operations in each loop iteration, so that the Vivado HLS tool can exploit more parallelism among these operations. More parallelism means more throughput and higher system performance.

- When the factor <N> is less than the total number of loop iterations (10 in the example above), it is called a *partial unroll*.

- When the factor <N> is the same as the number of loop iterations, it is called a *full unroll*. While *full unroll* requires that the loop bounds are known at compile time, it exposes the most parallelism.

To unroll a loop, put `#pragma HLS unroll [factor=N]` at the beginning of the loop. Without the optional `factor=N`, the loop will be fully unrolled.

```
int sum = 0;
for(int i = 0; i < 10; i++) {
#pragma HLS unroll factor=2
    sum += a[i];
}
```

Factors Limiting the Parallelism Achieved by Loop Pipelining and Loop Unrolling

Both loop pipelining and loop unrolling exploit the parallelism between loop iterations. However, parallelism between loop iterations is limited by two main factors:

- The data dependencies between loop iterations.

- The number of available hardware resources.

A data dependence from an operation in one iteration to another operation in a subsequent iteration is called a loop-carried dependence. It implies that the operation in the subsequent iteration cannot start until the operation in the current iteration has finished computing the data input for the operation in subsequent iteration. Loop-carried dependencies fundamentally limit the initiation interval that can be achieved using loop pipelining and the parallelism that can be exploited using loop unrolling.

The following example demonstrates loop-carried dependencies among operations producing and consuming variables `a` and `b`.

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

Send Feedback

Operations in the next iteration of this loop can not start until the current iteration has calculated and updated the values of `a` and `b`. Array accesses are a common source of loop-carried dependencies, as shown in the following example:

```
for (i = 1; i < N; i++)
    mem[i] = mem[i-1] + i;
```

In this case, the next iteration of the loop must wait until the current iteration updates the content of the array. In case of loop pipelining, the minimum initiation interval (II) is the total number of clock cycles required for the memory read, the add operation, and the memory write

Another performance limiting factor for loop pipelining and loop unrolling is the number of available hardware resources. The following figure shows an example the issues created by resource limitations, which in this case prevents the loop to be pipelined with an initiation interval of 1.

Figure 10: **Resource Contention**
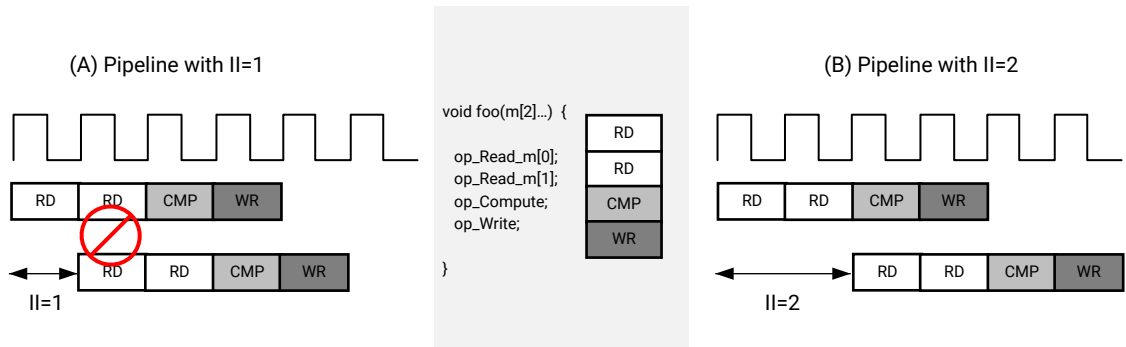
In this example, if the loop is pipelined with an initiation interval of one, there are two read operations. If the memory has only a single port, then the two read operations cannot be executed simultaneously and must be executed in two cycles. So the minimal initiation interval can only be two, as shown in part (B) of the figure. The same can happen with other hardware resources. For example, if the `op_compute` is implemented with a DSP core which cannot accept new inputs every cycle, and there is only one such DSP core. Then `op_compute` cannot be issued to the DSP core each cycle, and an initiation interval of one is not possible.

Increasing Local Memory Bandwidth

This section shows several ways provided by the Vivado HLS tool to increase local memory bandwidth, which can be used together with loop pipelining and loop unrolling to improve system performance.

Send Feedback

Arrays are intuitive and useful constructs in C/C++ programs. They allow for the algorithm to be easily captured and understood. In the HLS tool, each array is implemented by default with a single port memory resource; however, such memory implementation might not be the most ideal memory architecture for performance- oriented programs. Refer to Loop Pipelining and Loop Unrolling for an example of resource contention caused by limited memory ports.
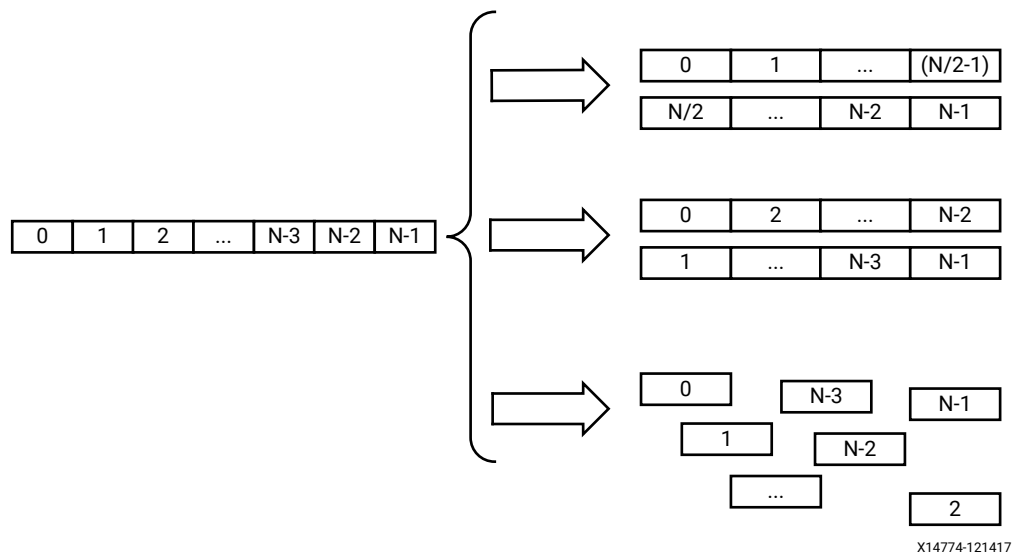
Array Partitioning

Arrays can be partitioned into smaller arrays. Physical implementation of memories have only a limited number of read ports and write ports, which can limit the throughput of a load/store intensive algorithm. The memory bandwidth can sometimes be improved by splitting up the original array (implemented as a single memory resource) into multiple smaller arrays (implemented as multiple memories), effectively increasing the number of load/store ports.

The Vivado HLS tool provides three types of array partitioning, as shown in the following figure.

- **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:** The default operation is to split the array into its individual elements. This corresponds to implementing an array as a collection of registers rather than as a memory.

Figure 11: **Multi-dimension Array Partitioning**

X14774-121417

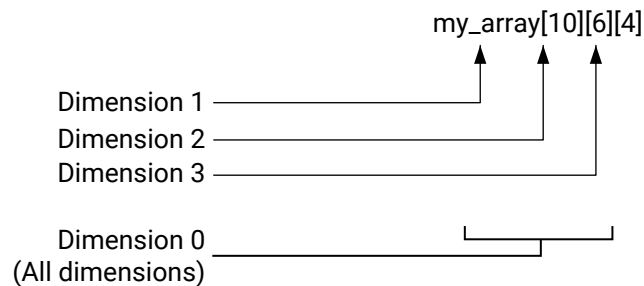To partition an array in the HLS tool, insert this in the hardware function source code:

```
#pragma HLS array_partition variable=<variable> <block, cyclic, complete>
factor=<int> dim=<int>
```

For *block* and *cyclic* partitioning, the `factor` option can be used to specify the number of arrays which are created. In the figure above, a factor of two is used, dividing the array into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the last array will have fewer than average elements.

When partitioning multi-dimensional arrays, the `dim` option can be used to specify which dimension is partitioned. The following figure shows an example of partitioning different dimensions of a multi-dimensional array.

Figure 12: **Multi-dimension Array Partitioning**

X14769-121417

Array Reshaping

Arrays can also be reshaped to increase the memory bandwidth. Reshaping takes different elements from a dimension in the original array, and combines them into a single wider element. Array reshaping is similar to array partitioning, but instead of partitioning into multiple arrays, it widens array elements. The following figure illustrates the concept of array reshaping.

Send Feedback

Figure 13: **Array Reshaping**
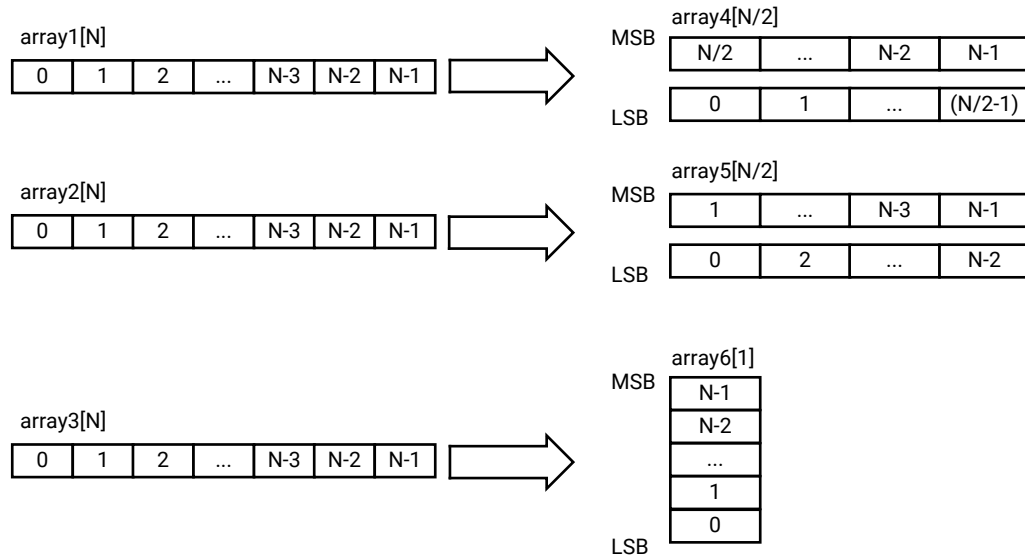
To use array reshaping in the Vivado HLS tool, insert this in the hardware function source code:

```
#pragma HLS array_reshape variable=<variable> <block, cyclic, complete>
factor=<int> dim=<int>
```

The options have the same meaning as the array partition pragma.

Data Flow Pipelining

The previously discussed optimization techniques are all "fine grain" parallelizing optimizations at the level of operators, such as multiplier, adder, and memory load/store operations. These techniques optimize the parallelism between these operators. Data flow pipelining on the other hand, exploits the "coarse grain" parallelism at the level of functions and loops. Data flow pipelining can increase the concurrency between functions and loops.

Function Data Flow Pipelining

The default behavior for a series of function calls in the Vivado HLS tool is to complete a function before starting the next function. In the following figure, part (A) shows the latency without function data flow pipelining. Assuming it takes eight cycles for the three functions to complete, the code requires eight cycles before a new input can be processed by `func_A` and also eight cycles before an output is written by `func_C` (assume the output is written at the end of `func_C`).

Send Feedback

Figure 14: **Function Data Flow Pipelining**
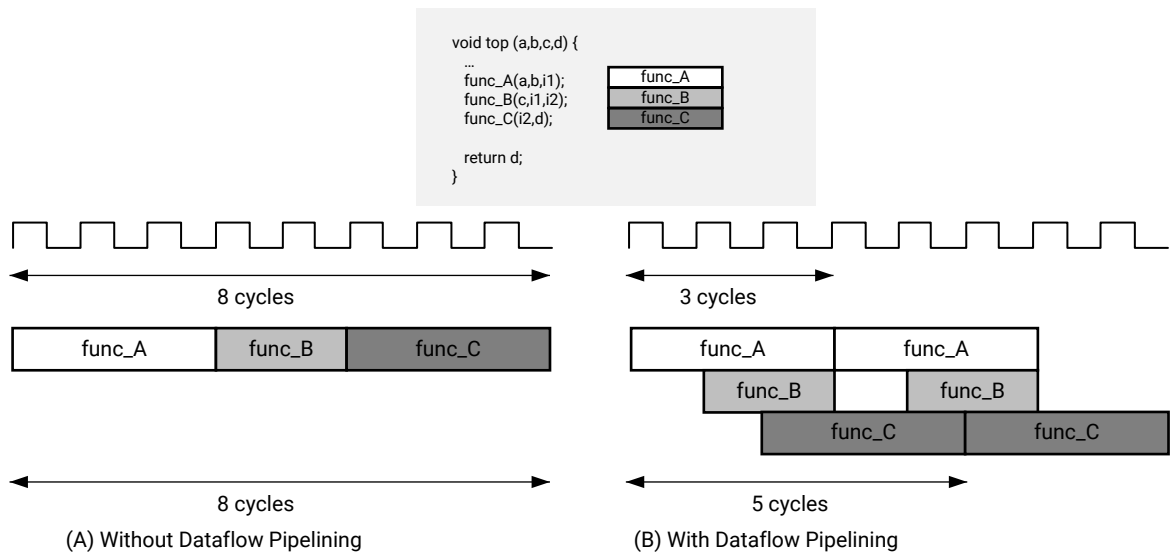
An example execution with data flow pipelining is shown in the part (B) of the figure above. Assuming the execution of `func_A` takes three cycles, `func_A` can begin processing a new input every three clock cycles rather than waiting for all the three functions to complete, resulting in increased throughput, The complete execution to produce an output then requires only five clock cycles, resulting in shorter overall latency.

The HLS tool implements function data flow pipelining by inserting "channels" between the functions. These channels are implemented as either ping-pong buffers or FIFOs, depending on the access patterns of the producer and the consumer of the data.

- If a function parameter (producer or consumer) is an array, the corresponding channel is implemented as a multi-buffer using standard memory accesses (with associated address and control signals).

- For scalar, pointer and reference parameters, as well as the function return, the channel is implemented as a FIFO, which uses less hardware resources (no address generation), but requires that the data is accessed sequentially.

To use function data flow pipelining, put `#pragma HLS dataflow` where the data flow optimization is desired. The following code snippet shows an example:

```
void top(a, b, c, d) {
#pragma HLS dataflow
    func_A(a, b, i1);
    func_B(c, i1, i2);
    func_C(i2, d);
}
```

Loop Dataflow Pipelining

Data flow pipelining can also be applied to loops in similar manner as it can be applied to functions. It enables a sequence of loops, normally executed sequentially, to execute concurrently. Data flow pipelining should be applied to a function, loop or region which contains either all function or all loops: do not apply on a scope which contains a mixture of loops and functions.

The following figure shows the advantages data flow pipelining can produce when applied to loops. Without data flow pipelining, loop `N` must execute and complete all iterations before loop `M` can begin. The same applies to the relationship between loops `M` and `P`. In this example, it is eight cycles before loop `N` can start processing the next value and eight cycles before an output is written (assuming the output is written when loop `P` finishes).

Figure 15: **Loop Data Flow Pipelining**

(A) Without Dataflow Pipelining   (B) With Dataflow Pipelining
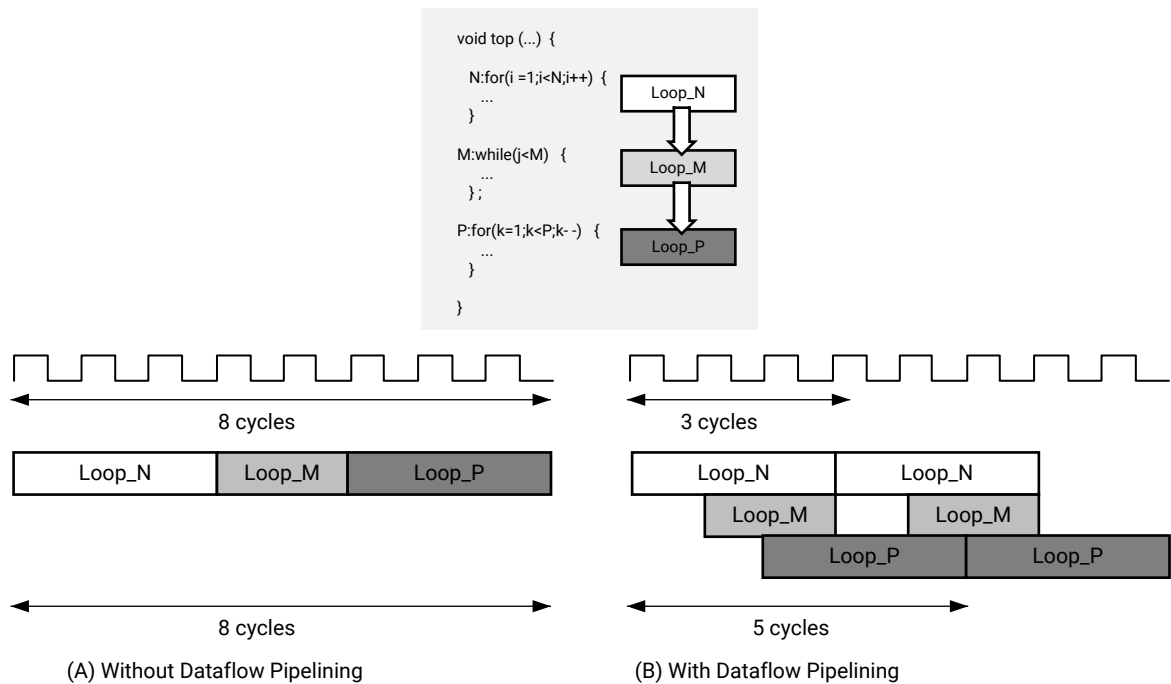
X14771-051419

With data flow pipelining, these loops can operate concurrently. An example execution with data flow pipelining is shown in part (B) of the figure above. Assuming the loop `M` takes three cycles to execute, the code can accept new inputs every three cycles. Similarly, it can produce an output value every five cycles, using the same hardware resources. The Vivado HLS tool automatically inserts channels between the loops to ensure data can flow asynchronously from one loop to the next. As with data flow pipelining, the channels between the loops are implemented either as multi-buffers or FIFOs.

To use loop data flow pipelining, put `#pragma HLS dataflow` where you want the data flow optimization.

Hardware Function Interfacing

After defining what function is needed for acceleration, there are a few key items to ensure compilation is valid. The Vivado HLS tool data types (`ap_int`, `ap_uint`, `ap_fixed`, etc.) cannot be part of the function parameter list that the software part of the application calls. These data types are unique to the HLS tool and have no bearing outside of the intended tool and associated compiler.

For example, if the following function was written in the HLS tool, the parameter list needs to be adjusted, and the function body has to handle moving the data from the HLS tool to a more generic data type, as shown below:

```
void foo(ap_int *a, ap_int *b, ap_int *c) { /* Function body */ }
```

This needs to be modified if using local variables:

```
void foo(int *a, int *b, int *c) {
    ap_int *local_a = a;
    ap_int *local_b = b;
    ap_int *local_c = c;
    // Remaining function body
}
```

IMPORTANT! *Initializing local variables with input data can consume too much memory in the accelerator. Therefore, casting the input data types to the appropriate HLS data types will work instead.*

Send Feedback

Memory Access Optimizations

Efficient memory accesses are critical to the performance of hardware functions running on an FPGA. The first category of optimizations that can be made to a hardware function is improving memory accesses and the data transfer rate between the PS and PL.

Data is often stored in an external memory. Accesses to an off-chip can take time and reduce the system performance. To reduce the memory access times and improve the runtime in high-performance systems, data can be stored in a local cache.

In addition to these memories, an FPGA provides local memory, where small and medium-sized data blocks can be stored and efficiently accessed. The memory architecture of a system, which uses FPGAs to accelerate performance, is similar to that of a CPU+GPU or CPU+DSP system, where consideration is given to making the most common memory accesses through the most local memory.

The following techniques can help improve the design of a hardware function by minimizing the latency impact of accessing and storing the data through the extensive use of local memories.

- **Data Motion Optimization:** This includes the transfer of data between the PS and the PL components. The SDSoC™ environment implements a default data motion architecture based on the arguments of the functions selected for the PL. However, it could use optimization directives to ensure data is stored in contiguous memory. This would improve the efficiency of the data transfer. A scatter-gather transfer could also be used to transfer very large sets of data more efficiently.

- **Data Access Patterns:** An FPGA excels at processing data quickly, in a highly concurrent manner. Poor data access patterns interrupt the flow of data and leave the computational power of an FPGA underused. Good data access patterns minimize the use of re-reading data and increase the use of conditional branching to process one sample in multiple ways.

- **On-Chip Memories:** On-chip memories (that is, programmable logic RAM, or PLRAM) use the block RAM on the FPGA, and are physically located near the computation. They allow one-cycle reads and writes, which drastically improve memory access performance. Copying data using optimal and efficient data access patterns from the DDR to these local memories can be done very quickly using burst transactions and can considerably improve performance. Performance could also be improved by using optimal data access patterns, such as burst transactions, to efficiently copy the data from the DDR to these local memories.

Send Feedback

Data Motion Optimization

The data motion network is the hardware used to connect the program running on the PS to the hardware function implemented in the PL fabric. The SDSoC environment automatically creates a data motion network based on the data types; however, directives can be used to optimize the implementation for both speed and hardware resources. The default data motion networks created for each data type are:

- **Scalar:** Scalar data types are always transferred by the `AXI_LITE` data mover. This is a memory-mapped interface that requires very few hardware resources to implement, consisting of single word read/write transfers with handshaking. This makes it the ideal choice for function arguments that are only transferred once per invocation. These types rarely need to be addressed when optimizing the system.

- **Array:** Arrays contain multiple data values and are more efficiently transferred using fast DMA transfers. The SDSoC environment provides a number of options to control both the efficiency of the hardware and the resources used to implement the hardware, even allowing for the selection of specific data movers.

- **Pointers:** By default, in the absence of any pragmas, a pointer argument is taken to be a scalar, even though in C/C++ it might denote a one-dimensional array type. If a pointer is written to or read from multiple times, pragmas should be used to ensure the efficient transfer of data.

- **Struct:** A single `struct` is flattened, and each data member uses its own data mover depending on whether it is a scalar or array. For an array of `struct`, the data motion network is the same as an `array` discussed previously.

With an understanding of the SDSoC environment default behavior, select the most optimum data motion network based on your own particular needs.

Send Feedback

Memory Allocation

The `sds++/sdscc` (referred to as `sds++`) compiler analyzes your program and selects data movers to match the requirements for each hardware function call between software and hardware, based on payload size, hardware interface on the accelerator, and properties of the function arguments. When the compiler can guarantee an array argument is located in physically contiguous memory, it can use the most efficient data movers. Allocating or memory-mapping arrays with the following `sds_lib` library functions can inform the compiler that memory is physically contiguous.

```
// guarantees physically contiguous memory
sds_alloc(size_t size);

// paddr must point to contiguous memory
sds_mmap(void *paddr, size_t size, void *vaddr);

// assumes physically contiguous memory
sds_register_dmabuf(void *vaddr, int fd);
```

It is possible that due to the program structure, the `sds++` compiler cannot definitively deduce the memory contiguity, and when this occurs, it issues a warning message, as shown:

```
WARNING: [DMAnalysis 83-4492] Unable to determine the memory attributes
passed to foo_arg_A of function foo at foo.cpp:102
```

You can inform the compiler that the data is allocated in a physically contiguous memory by inserting the following pragma immediately before the function declaration.

Note: The pragma does not guarantee physically contiguous allocation of memory; your code must use `sds_alloc` to allocate such memory).

```
#pragma SDS data mem_attribute (A:PHYSICAL_CONTIGUOUS) // default is
NON_PHYSICAL_CONTIGUOUS
```

It is important to know how the data will be used, because it helps determine what data movers to use. This is because some require contiguous allocation, and others do not. For example, if the data to be used is random, it will use a scatter-gather DMA, which causes these types of overhead:

- walking page tables

- pinning pages

- creating descriptors

- cleaning up when the accelerator is completed

Send Feedback

Copy and Shared Memory Semantics

By default, hardware function calls involve copy-in, copy-out semantics for function arguments. It is possible to impose a shared memory model for hardware function arguments, but you must keep in mind that while throughput on burst transfers is quite good, the latency to external DDR from the programmable logic is significantly higher than it is for the CPU. The following pragma, inserted immediately preceding the function declaration, is used to declare that a variable transfer employs shared memory semantics.

```
#pragma SDS data zero_copy(A[0:<array_size>]) // array_size = number of
elements
```

Within a synthesizable hardware function, it is usually inefficient to read/write single words from the shared memory (specified using the ZERO_COPY pragma). A more efficient approach is to employ a pipelined for-loop to read/write data from memory in bursts and store it in a local memory.

For copy and zero-copy memory semantics, another efficient alternative is to stream data between programmable logic and external DDR to maximize memory efficiency. This stores data in local memory within a hardware function whenever you need to make non-sequential and repeated accesses to variables. For example, video applications typically have data coming in as pixel streams and implement line buffers in FPGA memory to support multiple accesses to the pixel stream data.

To declare to the `sds++/sdscc` (referred to as `sds++`) compiler command that a hardware function can admit streaming access for an array data transfer (that is, each element is accessed precisely once in index order), insert the following pragma immediately preceding the function prototype.

```
#pragma SDS data access_pattern(A:SEQUENTIAL) // access pattern =
SEQUENTIAL | RANDOM
```

For arrays passed as pointer typed arguments to hardware functions, sometimes the compilers can infer transfer size, but if they cannot, they issue the following message:

```
WARNING: [DMAnalysis 83-4439] Cannot determine data size for argument p of
function foo
```

Use the following to specify the size of the data to be transferred.

```
#pragma SDS data copy(p[0:<array_size>]) // for example, int *p
```

You can vary the data transfer size on a per function call basis to avoid transferring data that is not required by a hardware function by setting `<array_size>` in the pragma definition to be an expression defined in the scope of the function call (that is, all variables in the size expression must be scalar arguments to the function), for example:

```
#pragma SDS data copy(A[0:L+2*T/3]) // scalar arguments L, T to same
function
```

The `sds++` compiler implements function call interfaces by creating stub functions that have a specific synchronization code (such as `cf_wait()`) to preserve program behavior and ensure consistency.

Related Information
Execution Model of an SDSoC Application

Data Cache Coherency

The `sds++/sdscc` (referred to as `sds++`) compiler automatically generates software configuration code for each data mover required by the system, including interfacing to underlying device drivers, as needed. The default assumption is that the system compiler maintains cache coherency for the memory allocated to arrays passed between the CPU and hardware functions; consequently, the compiler might generate code to perform a cache flush before transferring data to a hardware function and to perform a cache-invalidate before transferring data from a hardware function to the memory. Both actions are necessary for correctness, but have performance implications. When using Zynq® device HP ports, for example, you can override the default when you know that the CPU will not access the memory indicating that the correctness of the application does not depend on cache coherency. To avoid the overhead of unnecessary cache flushes, use the following API to allocate memory.

```
void *sds_alloc_non_cacheable(size_t size)
```

A typical use case of non-cacheable memory is a video application where some frame buffers are accessed by programmable logic but not the CPU.

Data Access Patterns

Designers select an FPGA to implement C code due to the superior performance of the FPGA— the parallel architecture of an FPGA allows it to perform operations much faster than the inherently sequential operations of a processor, and users typically want to take advantage of that performance.

The focus here is on understanding the impact that the access patterns inherent in the C code might have on the results. Although the access patterns of most concern are those into and out of the hardware function, it is worth considering the access patterns within functions as any bottlenecks within the hardware function will negatively impact the transfer rate into and out of the function.

To highlight how some data access patterns can negatively impact performance and demonstrate how other patterns can be used to fully embrace the parallelism and high performance capabilities of an FPGA, this section reviews an image convolution algorithm.

- The first part reviews the algorithm and highlights the data access aspects that limit the performance in an FPGA.

- The second part shows how the algorithm might be written to achieve the highest performance possible.

Algorithm with Poor Data Access Patterns

A standard convolution function applied to an image is used here to demonstrate how the C code can negatively impact the performance that is possible from an FPGA. In this example, a horizontal and then vertical convolution is performed on the data. Because the data at the edge of the image lies outside the convolution windows, the final step is to address the data around the border.

The algorithm structure can be summarized as follows:

- A horizontal convolution.

- Followed by a vertical convolution.

- Followed by a manipulation of the border pixels.

```
static void convolution_orig(
  int width,
  int height,
  const T *src,
  T *dst,
  const T *hcoeff,
  const T *vcoeff) {
  T local[MAX_IMG_ROWS*MAX_IMG_COLS];

// Horizontal convolution
  HconvH:for(int row = 0; row < height; row++){
    HconvWfor(int col = border_width; col < width - border_width; col++){
      Hconv:for(int i = - border_width; i <= border_width; i++){
        ...
        }
      }
    }

// Vertical convolution
  VconvH:for(int row = border_width; row < height - border_width; row++){
    VconvW:for(int col = 0; col < width; col++){
      Vconv:for(int i = - border_width; i <= border_width; i++){
```

Send Feedback

```
        }
      }
    }
  }

// Border pixels
  Top_Border:for(int col = 0; col < border_width; col++){
  }
  Side_Border:for(int row = border_width; row < height - border_width; row+
+){
  }
  Bottom_Border:for(int = height - border_width; row < height; row++){
  }
}
```

Standard Horizontal Convolution

The first step is to perform the convolution in the horizontal direction, as shown in the following figure.

Figure 16: **Convolution in Horizontal Direction**
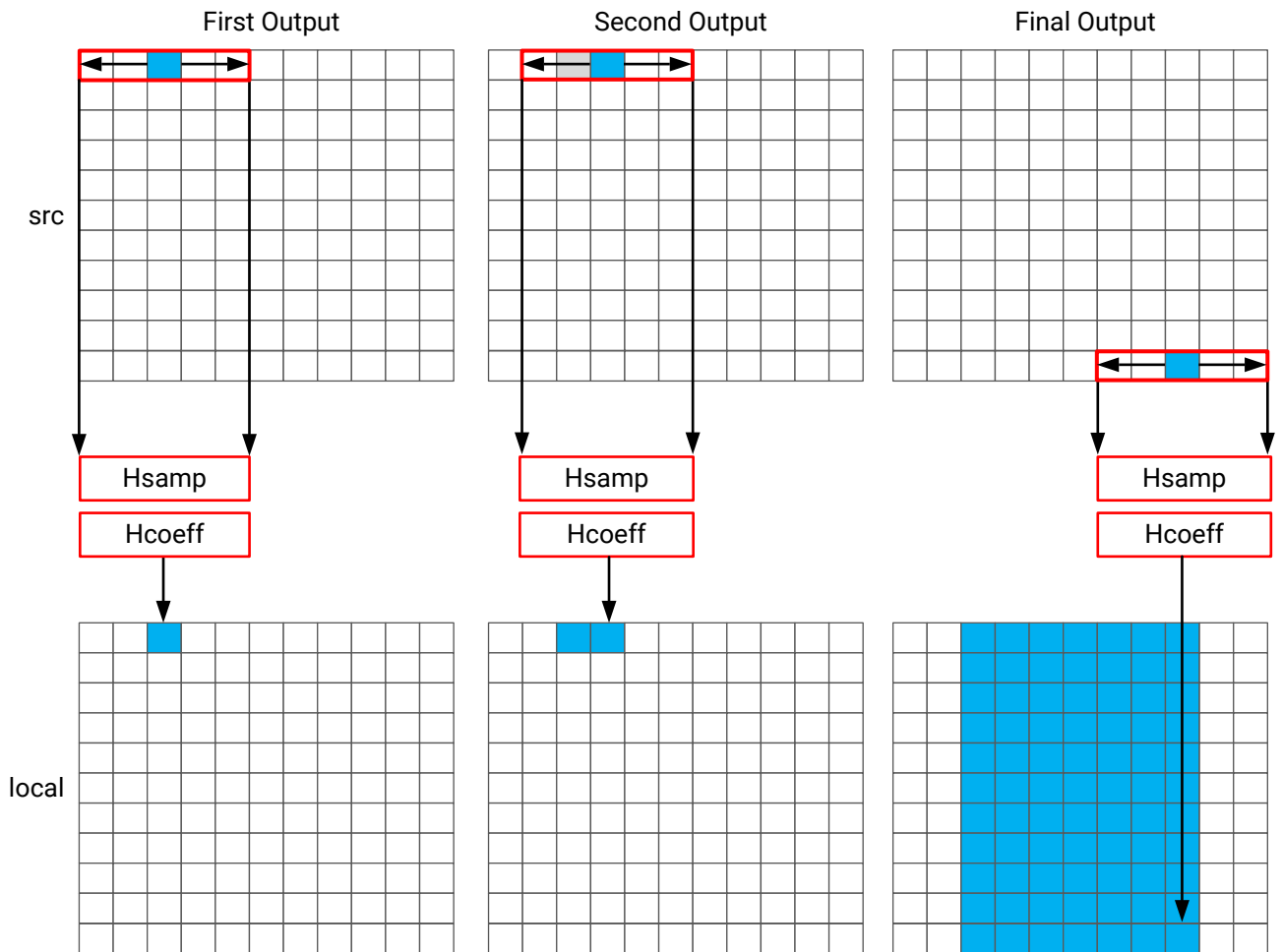
X14296-121417

The convolution is performed using K samples of data and K convolution coefficients. In the figure above, K is shown as 5, however, the value of K is defined in the code. To perform the convolution, a minimum of K data samples are required. The convolution window cannot start at the first pixel because the window would need to include pixels that are outside the image.

By performing a symmetric convolution, the first K data samples from input `src` can be convolved with the horizontal coefficients, and the first output calculated. To calculate the second output, the next set of K data samples is used. This calculation proceeds along each row until the final output is written.

The C code for performing this operation is shown below.

```
const int conv_size = K;

const int border_width = int(conv_size / 2);


#ifndef __SYNTHESIS__

T * const local = new T[MAX_IMG_ROWS*MAX_IMG_COLS];

#else // Static storage allocation for HLS, dynamic otherwise

T local[MAX_IMG_ROWS*MAX_IMG_COLS];

#endif


Clear_Local:for(int i = 0; i < height * width; i++){

  local[i]=0;

}


// Horizontal convolution

HconvH:for(int col = 0; col < height; col++){

  HconvWfor(int row = border_width; row < width - border_width; row++){

    int pixel = col * width + row;

    Hconv:for(int i = - border_width; i <= border_width; i++){

      local[pixel] += src[pixel + i] * hcoeff[i + border_width];

    }

  }

}
```

The code is straightforward; however, there are some issues with this C code that will negatively impact the quality of the hardware results.

Send Feedback

The first issue is the large storage requirements during C compilation. The intermediate results in the algorithm are stored in an internal local array. This requires an array of `HEIGHT*WIDTH`, which for a standard video image of 1920*1080 will hold 2,073,600 values.

- For the compilers targeting Zynq®-7000 SoC or Zynq® UltraScale+™ MPSoC devices, as well as many host systems, this amount of local storage can lead to stack overflows at runtime (for example, running on the target device, or running co-sim flows within the Vivado® High-Level Synthesis (HLS) tool). The data for a local array is placed on the stack and not the heap, which is managed by the OS. When cross-compiling with `arm-linux-gnueabihf-g++` use the `-Wl,"-z stacksize=4194304"` linker option to allocate sufficient stack space.

  Note: The syntax for this option varies for different linkers

- When a function will only be run in hardware, a useful way to avoid such issues is to use the `__SYNTHESIS__` macro. This macro is automatically defined by the system compiler when the hardware function is synthesized into hardware. The code shown above uses dynamic memory allocation during C simulation to avoid any compilation issues and only uses static storage during synthesis. A downside of using this macro is the code verified by C simulation is not the same code that is synthesized. In this case, however, the code is not complex, and the behavior will be the same.

- The main issue with this local array is the quality of the FPGA implementation. Because this is an array it will be implemented using internal FPGA block RAM. This is a very large memory to implement inside the FPGA. It might require a larger and more costly FPGA device. The use of block RAM can be minimized by using the `DATAFLOW` optimization and streaming the data through small efficient FIFOs, but this will require the data to be used in a streaming sequential manner. There is currently no such requirement.

The next issue relates to the performance: the initialization for the local array. The `Clear_Local` loop is used to set the values in array local to zero. Even if this loop is pipelined in the hardware to execute in a high-performance manner, this operation still requires approximately two million clock cycles (`HEIGHT*WIDTH`) to implement. While this memory is being initialized, the system cannot perform any image processing. This same initialization of the data could be performed using a temporary variable inside loop HConv to initialize the accumulation before the write.

Finally, the throughput of the data, and thus the system performance, is fundamentally limited by the data access pattern.

- To create the first convolved output, the first K values are read from the input.

- To calculate the second output, a new value is read, and then the same K-1 values are re-read.

One of the keys to a high-performance FPGA is to minimize the access to and from the PS. Each access for data, which has previously been fetched, negatively impacts the performance of the system. An FPGA is capable of performing many concurrent calculations at once and reaching very high performance, but not while the flow of data is constantly interrupted by re-reading values.

Note: To maximize performance, data should only be accessed once from the PS and small units of local storage (small to medium sized arrays) should be used for data which must be reused.

With the code shown above, the data cannot be continuously streamed directly from the processor using a operation because the data is required to be re-read time and again.

Standard Vertical Convolution

The next step is to perform the vertical convolution shown in the following figure.

Figure 17: **Vertical Convolution**
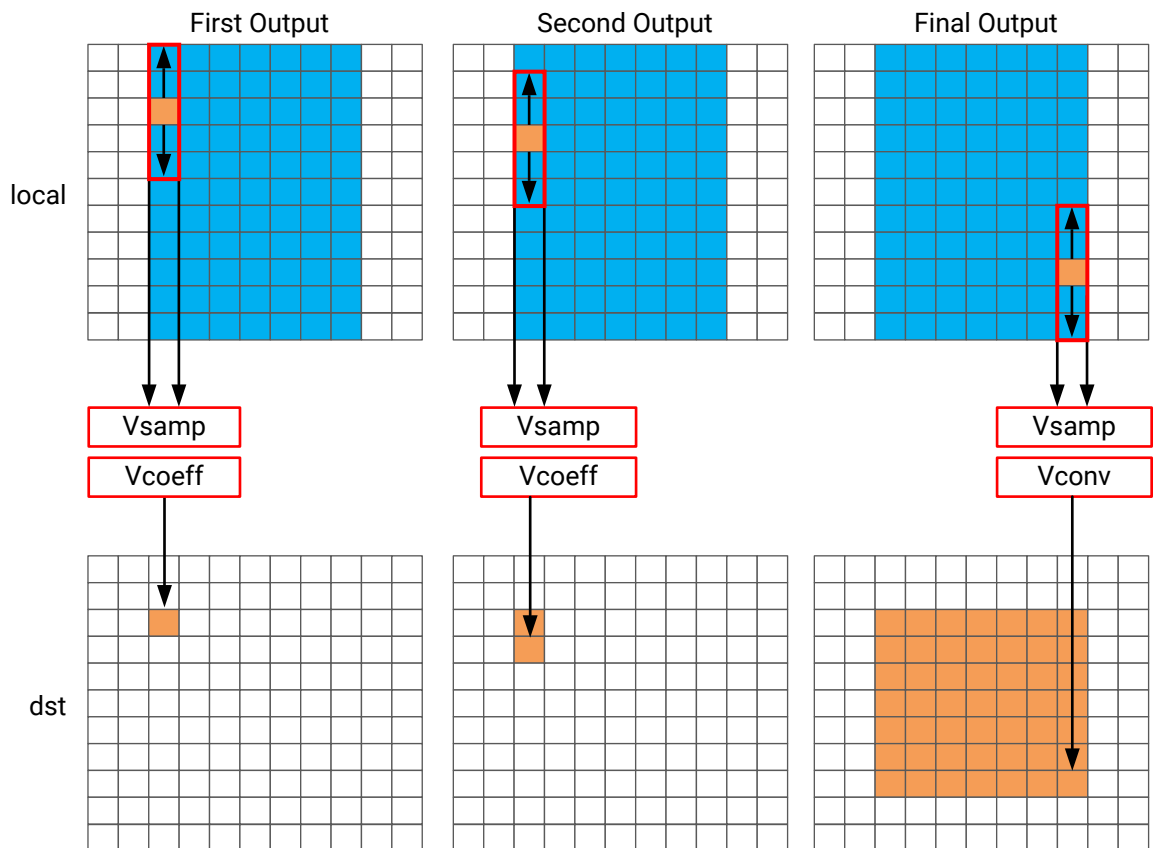
X14299-011519

The process for the vertical convolution is similar to the horizontal convolution. A set of K data samples is required to convolve with the convolution coefficients, Vcoeff in this case. After the first output is created using the first K samples in the vertical direction, the next set of K values is used to create the second output. The process continues down through each column until the final output is created.

After the vertical convolution, the image is now smaller than the source image `src`, due to both the horizontal and vertical border effect.

The following is the code for performing these operations:

```
Clear_Dst:for(int i = 0; i < height * width; i++){
  dst[i]=0;
}
// Vertical convolution
VconvH:for(int col = border_width; col < height - border_width; col++){
  VconvW:for(int row = 0; row < width; row++){
    int pixel = col * width + row;
    Vconv:for(int i = - border_width; i <= border_width; i++){
      int offset = i * width;
      dst[pixel] += local[pixel + offset] * vcoeff[i + border_width];
    }
  }
}
```

This code highlights similar issues to those already discussed with the horizontal convolution code:

- Many clock cycles are spent to set the values in the output image `dst` to zero. In this case, approximately another two million cycles for a 1920*1080 image size.

- There are multiple accesses per pixel to re-read data stored in array `local`.

- There are multiple writes per pixel to the output array/port `dst`.

The access patterns in the code above in fact creates the requirement to have such a large local array. The algorithm requires the data on row K to be available to perform the first calculation. Processing data down the rows before proceeding to the next column requires the entire image to be stored locally. This requires that all values be stored and results in large local storage on the FPGA.

In addition, when you reach the stage where you wish to use compiler directives to optimize the performance of the hardware function, the flow of data between the horizontal and vertical loop cannot be managed using a FIFO (a high-performance and low-resource unit), because the data is not streamed out of array `local`, a FIFO can only be used with sequential access patterns. Instead, this code which requires arbitrary/random accesses requires a ping-pong block RAM to improve performance. This doubles the memory requirements for the implementation of the local array to approximately four million data samples, which is too large for an FPGA.

Standard Border Pixel Convolution

The final step in performing the convolution is to create the data around the border. These pixels can be created by simply reusing the nearest pixel in the convolved output. The following figures shows how this is achieved.

Send Feedback

Figure 18: **Standard Border Pixel Convolution**
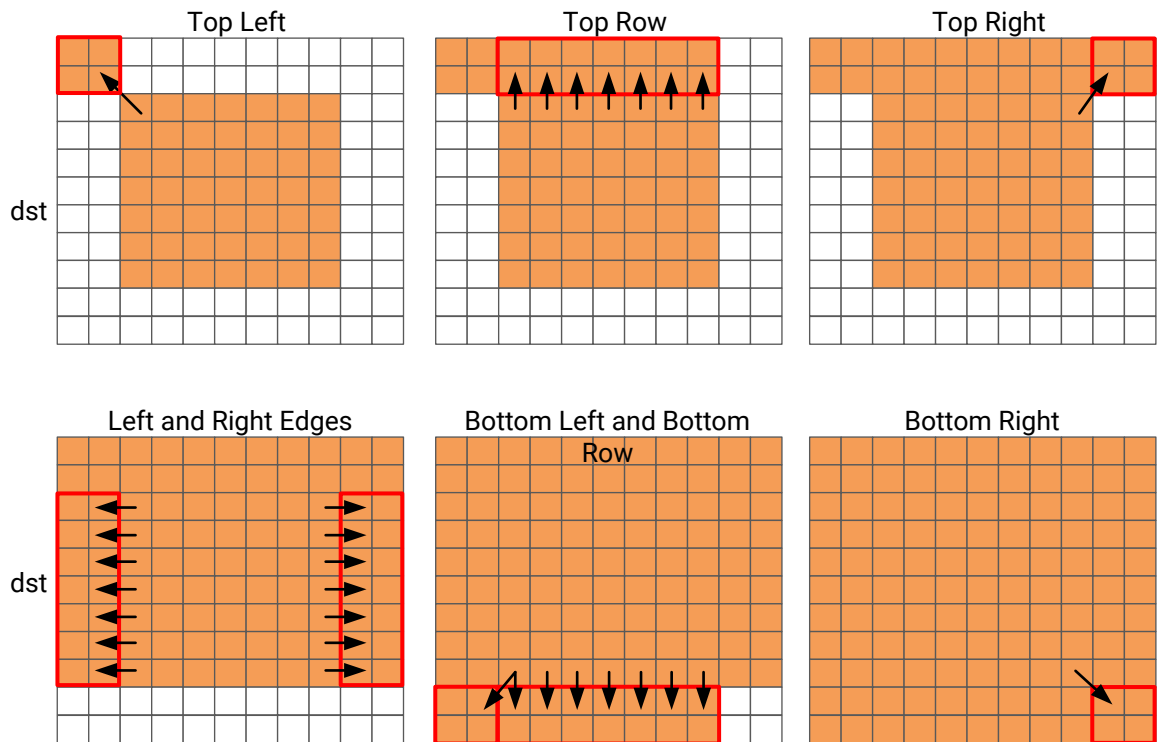
The border region is populated with the nearest valid value. The following code performs the operations shown in the previous figure.

```
int border_width_offset = border_width * width;
int border_height_offset = (height - border_width - 1) * width;

// Border pixels

Top_Border:for(int col = 0; col < border_width; col++){
  int offset = col * width;
  for(int row = 0; row < border_width; row++){
    int pixel = offset + row;
    dst[pixel] = dst[border_width_offset + border_width];
  }
  for(int row = border_width; row < width - border_width; row++){
    int pixel = offset + row;
    dst[pixel] = dst[border_width_offset + row];
  }
  for(int row = width - border_width; row < width; row++){
    int pixel = offset + row;
    dst[pixel] = dst[border_width_offset + width - border_width - 1];
  }
}

Side_Border:for(int col = border_width; col < height - border_width; col++){
  int offset = col * width;
  for(int row = 0; row < border_width; row++){
    int pixel = offset + row;
    dst[pixel] = dst[offset + border_width];
```

Send Feedback

```
    }
  for(int row = width - border_width; row < width; row++){
    int pixel = offset + row;
    dst[pixel] = dst[offset + width - border_width - 1];
  }
}

Bottom_Border:for(int col = height - border_width; col < height; col++){
  int offset = col * width;
  for(int row = 0; row < border_width; row++){
    int pixel = offset + row;
    dst[pixel] = dst[border_height_offset + border_width];
  }
  for(int row = border_width; row < width - border_width; row++){
    int pixel = offset + row;
    dst[pixel] = dst[border_height_offset + row];
  }
  for(int row = width - border_width; row < width; row++){
    int pixel = offset + row;
    dst[pixel] = dst[border_height_offset + width - border_width - 1];
  }
}
```

The code suffers from the same repeated access for data. The data stored outside the FPGA in the array `dst` must now be available to be read as input data re-read multiple times. Even in the first loop, `dst[border_width_offset + border_width]` is read multiple times but the values of `border_width_offset` and `border_width` do not change.

This code is very intuitive to both read and write. When implemented with the SDSoC environment it is approximately 120M clock cycles, which meets or slightly exceeds the performance of a CPU. However, as shown in the next section, optimal data access patterns ensure this same algorithm can be implemented on the FPGA at a rate of one pixel per clock cycle, or approximately 2M clock cycles.

The summary from this review is that the following poor data access patterns negatively impact the performance and size of the FPGA implementation:

- Multiple accesses to read and then re-read data. Use local storage where possible.

- Accessing data in an arbitrary or random access manner. This requires the data to be stored locally in arrays and costs resources.

- Setting default values in arrays costs clock cycles and performance.

## Algorithm with Optimal Data Access Patterns

The key to implementing the convolution example reviewed in the previous section as a high-performance design with minimal resources is doing the following:

- Maximize the flow of data through the system. Refrain from using any coding techniques or algorithm behavior that inhibits the continuous flow of data.

- Maximize the reuse of data. Use local caches to ensure there are no requirements to re-read data and the incoming data can keep flowing.

- Use conditional branching. This is expensive on a CPU, GPU, or DSP but optimal in an FPGA.

The first step is to understand how data flows through the system into and out of the FPGA. The convolution algorithm is performed on an image. When data from an image is produced and consumed, it is transferred in a standard raster-scan manner as shown in the following figure.

Figure 19: **Raster-Scan**
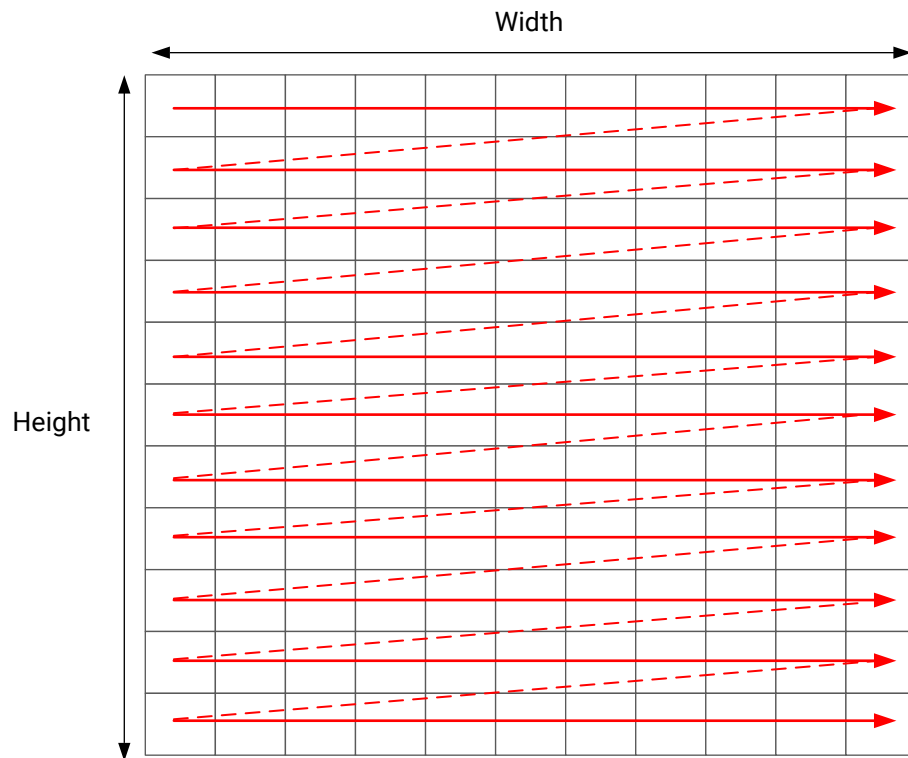
X14298-121417

If the data is transferred to the FPGA in a streaming manner, the FPGA should process it in a streaming manner and transfer it back from the FPGA in this manner.

The convolution algorithm shown below embraces this style of coding. At this level of abstraction a concise view of the code is shown. However, there are now intermediate buffers, `hconv` and `vconv`, between each loop. Because these are accessed in a streaming manner, they are optimized into single registers in the final implementation.

```
template<typename T, int K>
static void convolution_strm(
int width,
int height,
T src[TEST_IMG_ROWS][TEST_IMG_COLS],
T dst[TEST_IMG_ROWS][TEST_IMG_COLS],
const T *hcoeff,
const T *vcoeff)
{

T hconv_buffer[MAX_IMG_COLS*MAX_IMG_ROWS];
```

Send Feedback

```
T vconv_buffer[MAX_IMG_COLS*MAX_IMG_ROWS];
T *phconv, *pvconv;

// These assertions let HLS know the upper bounds of loops
assert(height < MAX_IMG_ROWS);
assert(width < MAX_IMG_COLS);
assert(vconv_xlim < MAX_IMG_COLS - (K - 1));
// Horizontal convolution
HConvH:for(int col = 0; col < height; col++) {
  HConvW:for(int row = 0; row < width; row++) {
    HConv:for(int i = 0; i < K; i++) {
    }
  }
}
// Vertical convolution
VConvH:for(int col = 0; col < height; col++) {
  VConvW:for(int row = 0; row < vconv_xlim; row++) {
    VConv:for(int i = 0; i < K; i++) {
    }
  }
}
Border:for (int i = 0; i < height; i++) {
  for (int j = 0; j < width; j++) {
  }
}
```

All three processing loops now embrace conditional branching to ensure the continuous processing of data.

Optimal Horizontal Convolution

The algorithm must use the K previous samples to compute the convolution result. It therefore copies the sample into a temporary cache `hwin`. This use of local storage means there is no need to re-read values from the PS and interrupt the flow of data. For the first calculation, there are not enough values in hwin to compute a result, so conditionally, no output values are written. To perform the calculation in a more efficient manner for FPGA implementation, the horizontal convolution is computed.

The algorithm keeps reading input samples and caching them into `hwin`. Each time it reads a new sample, it pushes an unneeded sample out of `hwin`. The first time an output value can be written is after the Kth input has been read. An output value can now be written. The algorithm proceeds in this manner along the rows until the final sample has been read. At that point, only the last K samples are stored in `hwin`; all that is required to compute the convolution.

As shown below, the code to perform these operations uses both local storage to prevent re-reads from the PL (the reads from local storage can be performed in parallel in the final implementation), and the extensive use of conditional branching to ensure each new data sample can be processed in a different manner.

```
// Horizontal convolution
phconv=hconv_buffer; // set / reset pointer to start of buffer

// These assertions let HLS know the upper bounds of loops
assert(height < MAX_IMG_ROWS);
```

Send Feedback

```
assert(width < MAX_IMG_COLS);
assert(vconv_xlim < MAX_IMG_COLS - (K - 1));
HConvH:for(int col = 0; col < height; col++) {
  HConvW:for(int row = 0; row < width; row++) {
#pragma HLS PIPELINE
    T in_val = *src++;
    // Reset pixel value on-the-fly - eliminates an O(height*width) loop
    T out_val = 0;
      HConv:for(int i = 0; i < K; i++) {
         hwin[i] = i < K - 1 ? hwin[i + 1] : in_val;
         out_val += hwin[i] * hcoeff[i];
      }
      if (row >= K - 1) {
         *phconv++=out_val;
      }
  }
}
```

An interesting point to note in the code above is the use of the temporary variable `out_val` to perform the convolution calculation. This variable is set to zero before the calculation is performed, negating the need to spend two million clock cycles to reset the values, as in the previous example.

Throughout the entire process, the samples in the src input are processed in a raster-streaming manner. Every sample is read in turn. The outputs from the task are either discarded or used, but the task keeps constantly computing. This represents a difference from code written to perform on a CPU.

Optimal Vertical Convolution

The vertical convolution represents a challenge to the streaming data model preferred by an FPGA. The data must be accessed by column but you do not wish to store the entire image. The solution is to use line buffers, as shown in the following figure.

Figure 20: **Line Buffers**
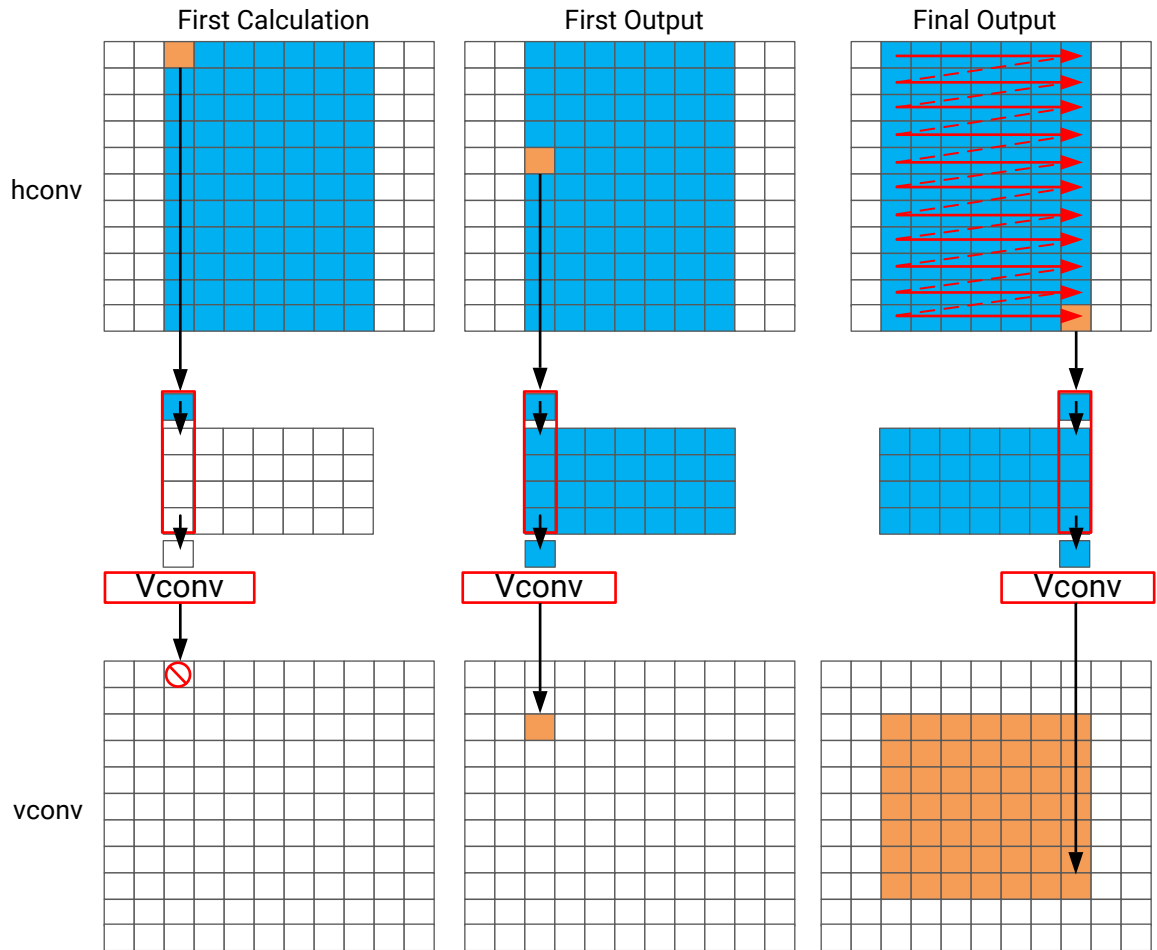
X14300-011519

Again, the samples are read in a streaming manner, this time from the local buffer `hconv`. The algorithm requires at least K-1 lines of data before it can process the first sample. All the calculations performed before this are discarded through the use of conditionals.

A line buffer allows K-1 lines of data to be stored. Each time a new sample is read, another sample is pushed out the line buffer. An interesting point to note here is that the newest sample is used in the calculation, and then the sample is stored into the line buffer and the old sample ejected out. This ensures that only K-1 lines are required to be cached rather than an unknown number of lines, and minimizes the use of local storage. Although a line buffer does require multiple lines to be stored locally, the convolution kernel size K is always much less than the 1080 lines in a full video image.

Send Feedback

The first calculation can be performed when the first sample on the Kth line is read. The algorithm then proceeds to output values until the final pixel is read.

```
// Vertical convolution
phconv=hconv_buffer; // set/reset pointer to start of buffer
pvconv=vconv_buffer; // set/reset pointer to start of buffer
VConvH:for(int col = 0; col < height; col++) {
  VConvW:for(int row = 0; row < vconv_xlim; row++) {
#pragma HLS DEPENDENCE variable=linebuf inter false
#pragma HLS PIPELINE
    T in_val = *phconv++;
    // Reset pixel value on-the-fly - eliminates an O(height*width) loop
    T out_val = 0;
    VConv:for(int i = 0; i < K; i++) {
      T vwin_val = i < K - 1 ? linebuf[i][row] : in_val;
      out_val += vwin_val * vcoeff[i];
      if (i > 0)
        linebuf[i - 1][row] = vwin_val;
    }
    if (col >= K - 1) {
      *pvconv++ = out_val;
    }
  }
}
```

The code processes all the samples in the design in a streaming manner. The task is constantly running. Following a coding style where you minimize the number of re-reads (or re-writes) forces you to cache the data locally. This is an ideal strategy when targeting an FPGA.

Optimal Border Pixel Convolution

The final step in the algorithm is to replicate the edge pixels into the border region. To ensure the constant flow of data and data reuse, the algorithm makes use of local caching. The following figure shows how the border samples are aligned into the image.

- Each sample is read from the `vconv` output from the vertical convolution.

- The sample is then cached as one of four possible pixel types.

- The sample is then written to the output stream.

Figure 21: **Border Sample Alignment**
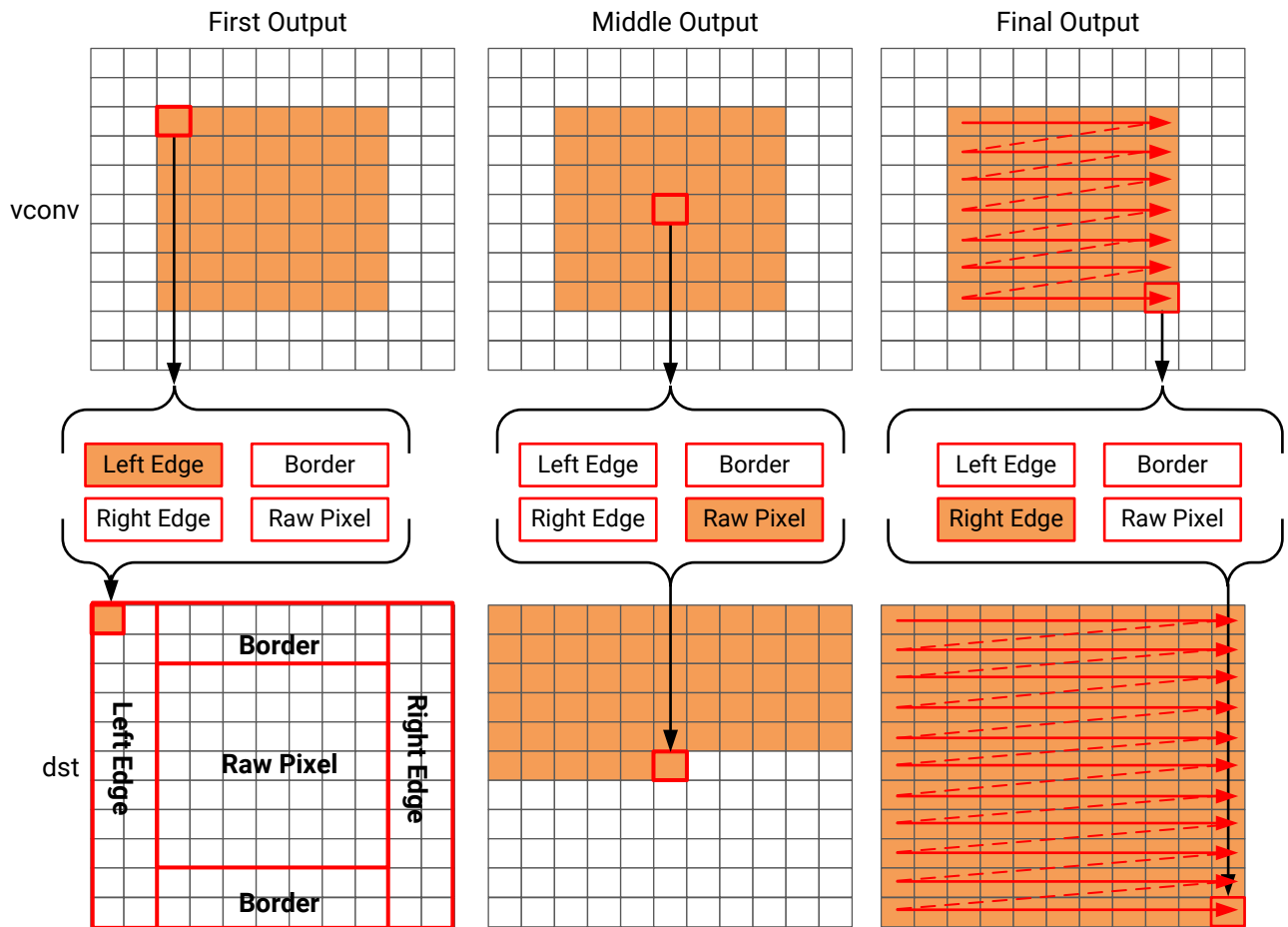
The code for determining the location of the border pixels is shown here.

```
// Border pixels
pvconv=vconv_buffer; // set/reset pointer to start of buffer
Border:for (int i = 0; i < height; i++) {
  for (int j = 0; j < width; j++) {
    T pix_in, l_edge_pix, r_edge_pix, pix_out;
#pragma HLS PIPELINE
    if (i == 0 || (i > border_width && i < height - border_width)) {
      // read a pixel out of the video stream and cache it for
      // immediate use and later replication purposes
      if (j < width - (K - 1)) {
        pix_in = *pvconv++;
        borderbuf[j] = pix_in;
      }
      if (j == 0) {
        l_edge_pix = pix_in;
      }
      if (j == width - K) {
        r_edge_pix = pix_in;
      }
    }
```

```
      // Select output value from the appropriate cache resource
      if (j <= border_width) {
        pix_out = l_edge_pix;
      } else if (j >= width - border_width - 1) {
        pix_out = r_edge_pix;
      } else {
         pix_out = borderbuf[j - border_width];
      }
      *dst++=pix_out;
    }
}
```

A notable difference with this new code is the extensive use of conditionals inside the tasks. This allows the task, after it is pipelined, to continuously process data. The result of the conditionals does not impact the execution of the pipeline. The result will impact the output values, but the pipeline with keep processing as long as input samples are available.

Optimal Data Access Patterns

The following list summarizes how to ensure your data access patterns result in the most optimal performance on an FPGA.

- Minimize data input reads. After data has been read into the block, it can easily feed many parallel paths but the inputs to the hardware function can be bottlenecks to performance. Read data once and use a local cache if the data must be reused.

- Minimize accesses to arrays, especially large arrays. Arrays are implemented in block RAM which like I/O ports only have a limited number of ports and can be bottlenecks to performance. Arrays can be partitioned into smaller arrays and even individual registers but partitioning large arrays will result in many registers being used. Use small localized caches to hold results such as accumulations and then write the final result to the array.

- Seek to perform conditional branching inside pipelined tasks rather than conditionally execute tasks, even pipelined tasks. Conditionals are implemented as separate paths in the pipeline. Allowing the data from one task to flow into the next task with the conditional performed inside the next task will result in a higher performing system.

- Minimize output writes for the same reason as input reads, namely, that ports are bottlenecks. Replicating additional accesses only pushes the issue further back into the system.

For C code which processes data in a streaming manner, consider employing a coding style that promotes read-once/write-once to function arguments because this ensures the function can be efficiently implemented in an FPGA. It is much more productive to design an algorithm in C that results in a high-performance FPGA implementation than debug why the FPGA is not operating at the performance required.

# Performance Measurement Using the AXI Performance Monitor

The AXI Performance Monitor (APM) module is used to monitor basic information about data transfers between the PS Arm® cores and the hardware in the PL. It captures statistics such as number of read/write transactions, throughput, and latency for the AXI transactions on the busses in the system.

This section demonstrates how to insert an APM core into the system, monitor the instrumented system, and view the performance data produced.

Create a Standalone Project and Implementing APM

1. Open the SDSoC™ environment, and create a new SDSoC project/workspace using any platform or operating system selection. Select the **Matrix Multiplication and Addition** template.

2. In Application Project Settings, select **Insert AXI Performance Monitor**.

   Enabling this option and building the project adds the APM IP core to your hardware system. The APM IP uses a small amount of resources in the programmable logic. The SDSoC environment connects the APM to the hardware/software interface ports, which are the general purpose (GP) and high performance (HP) ports.

Send Feedback
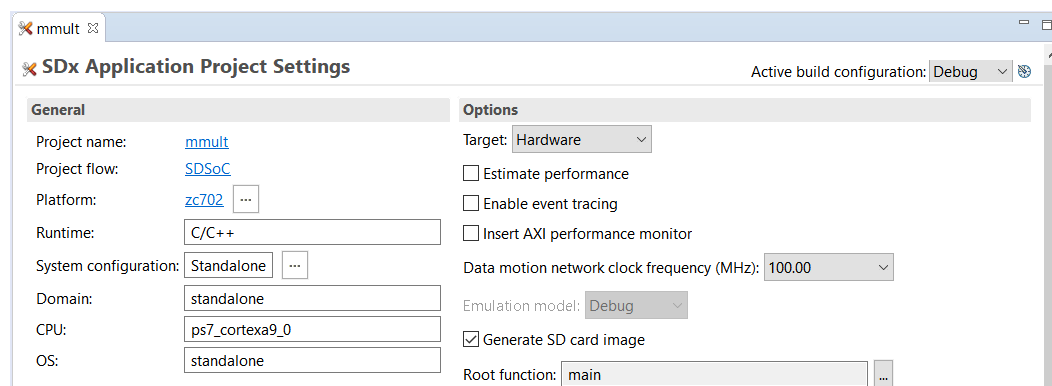
3. Select the `mmult` and `madd` functions to be implemented in hardware. Clean and build the project using the Debug configuration (selected by default).

Monitor the Standalone Instrumented System

To monitor the system, do the following:

1. After the build completes, connect the board to your computer, and power up the board.

2. Click the **Run → Debug Configurations** button to open the window.

3. Select **Xilinx SDx Application Debugger** in the Debug configurations tree.

4. Click **New Launch Configuration** to create a new SDx Application Debugger configuration.

5. For Debug Type, select **Standalone Application Debug**.

6. For Connection, select **Local**.

7. Select the **Performance Analysis** check box in the Main tab.

   After you select **Performance Analysis**, the performance analysis options are populated automatically in the Main tab.
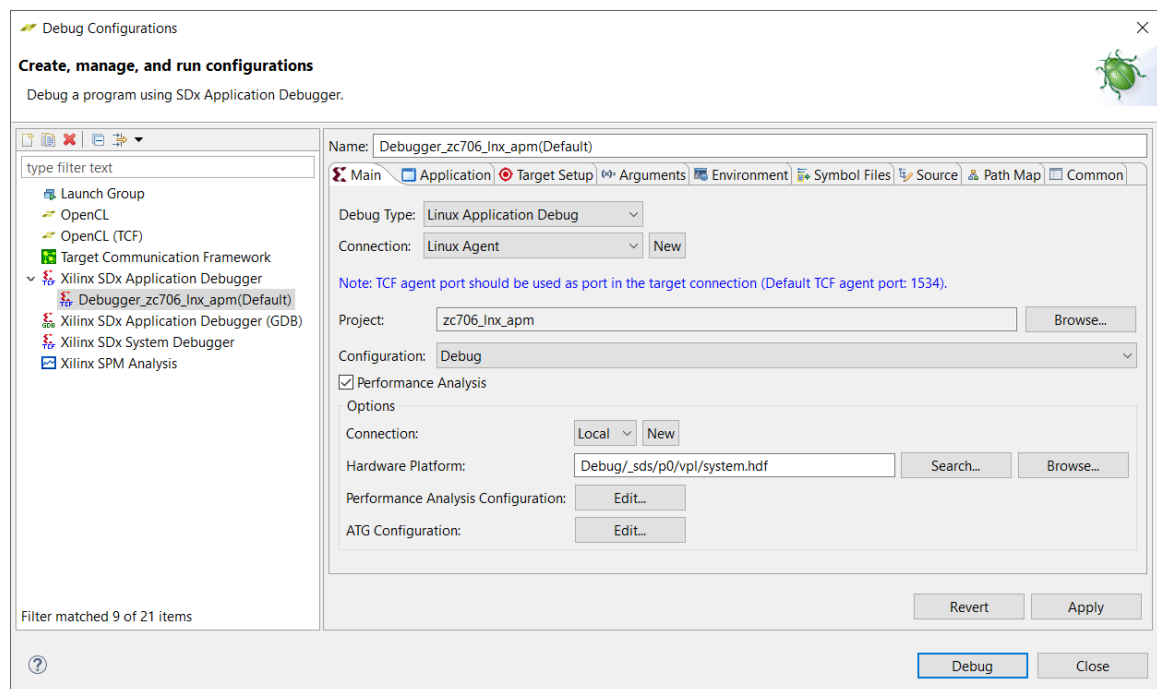

8. Click **Apply**, and then click **Debug**.

   If prompted to switch perspectives, click **Yes**.

Send Feedback

9. After the Debug Perspective opens, click **Window → Perspective**, and select **Performance Analysis** in the Open Perspective dialog, and click **OK**.

10. To resume the application, select the Debug tab, and click **Resume**.
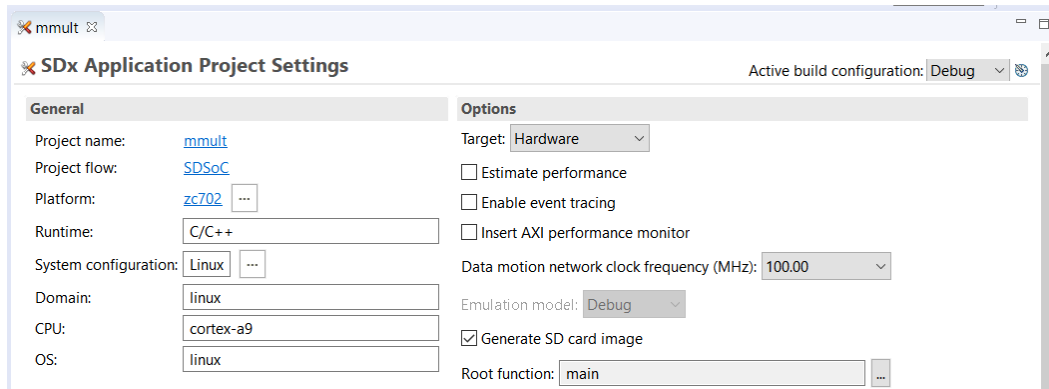
If prompted to switch perspectives, click **No** to stay on the Performance Analysis perspective.
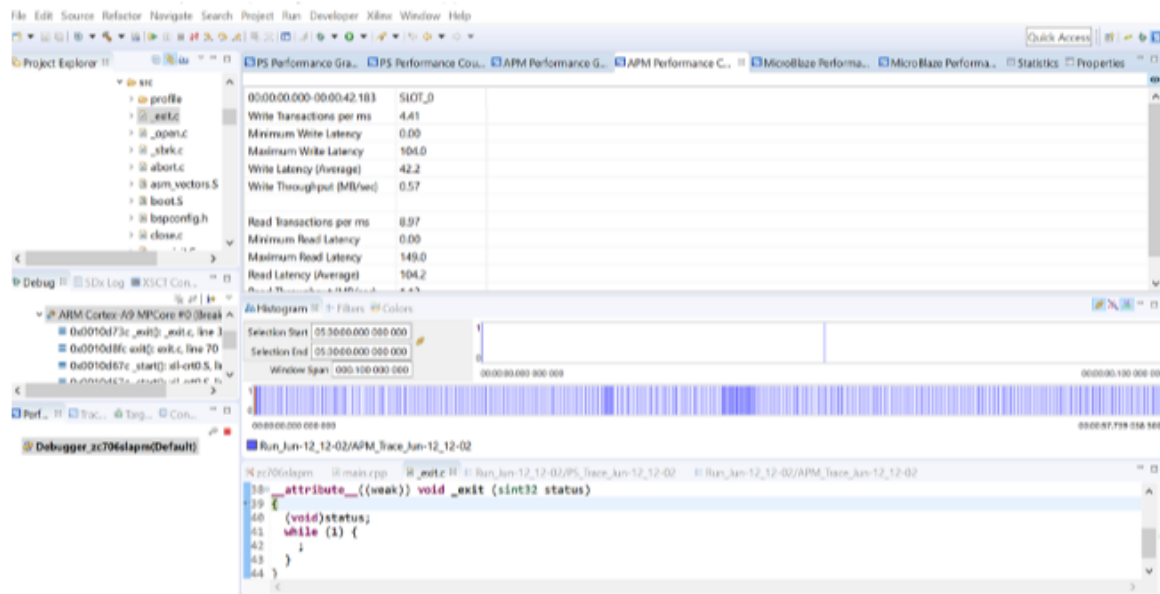
Create a Linux Project and Implement APM

1. Open the SDSoC environment, and create a new project/workspace using any platform or operating system selection. Select the **Matrix Multiplication and Addition Template**.

2. In SDx Application Project Settings, select **Insert AXI Performance Monitor**.

Enabling this option, and building the project adds the APM IP core to your hardware system. The APM IP uses a small amount of resources in the programmable logic. The SDSoC environment connects the APM to the hardware/software interface ports, which are the GP and HP.

3. Select the `mmult` and `madd` functions to be implemented in hardware.

4. Clean and build the project using the Debug configuration, which is selected by default.

The following figure shows the APM Counter:


Monitor the Linux Instrumented System

1. With the completed build, copy the contents of the `sd_card` directory onto a SD card, and boot Linux on the board.

2. Connect the board to your computer using both UART and JTAG cables.

3. Set up the Linux TCF agent target connection with the IP address of the board. See the SDK Help topic on TCF for more information.

4. Click the **Run → Debug Configuration** menu to open Debug Configurations.

5. Select **Xilinx Sdx Application Debugger** in the Debug configuration tree.

6. Click **New Launch Configuration** to create a new Application Debugger configuration.

7. For Debug Type, select **Linux Application Debug**.

Send Feedback

8. For Connection, select **Local**.

9. Select the **Performance Analysis** check box.

   After the Performance Analysis check box is selected, all the performance analysis options is populated automatically in the **Main** tab.
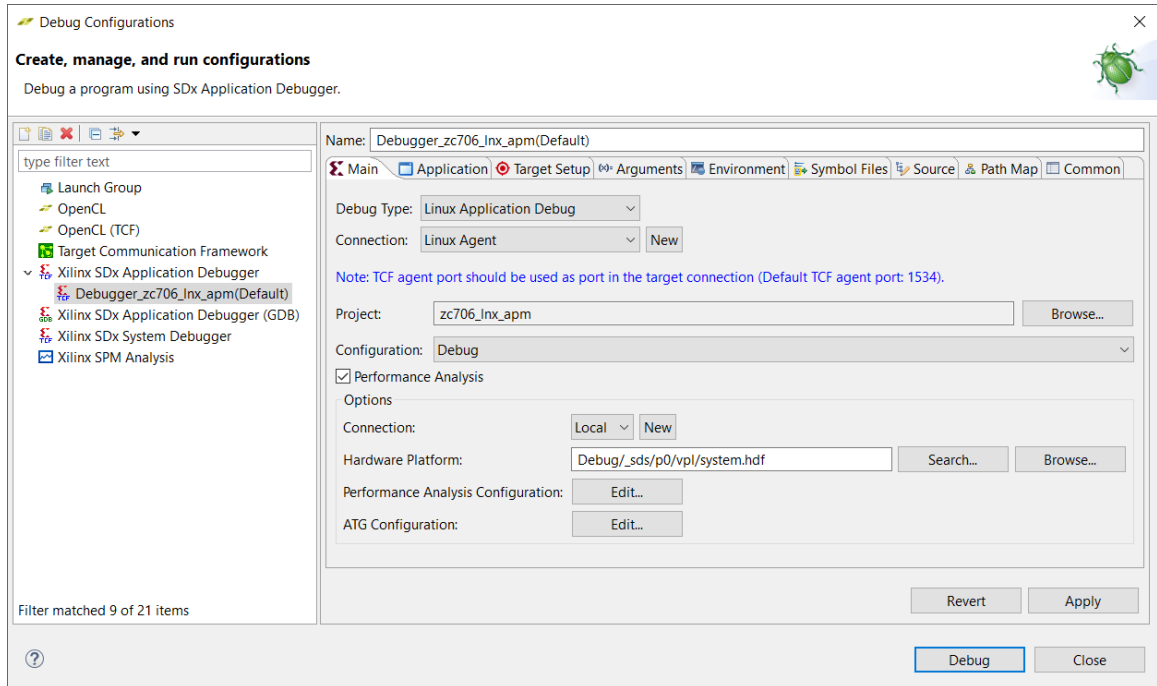
10. Click **Apply**.

11. Click **Debug**.

   If prompted to switch perspective to Debug, click **Yes**.

12. After Debug Perspective is displayed, click **Window → Perspective**, and then select **Performance Analysis** in the Open Perspective dialog.

13. Click **OK**.

14. Select the **Debug** tab, and click on **Resume** to resume the application.

   If prompted to switch perspectives, click **No** to stay on the Performance Analysis perspective.

15. After your program completes execution, click the **Stop Analysis** button.
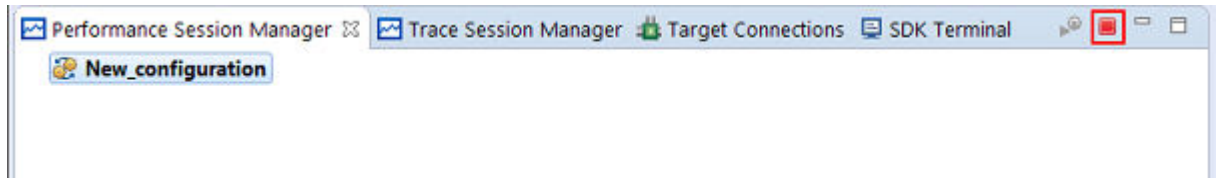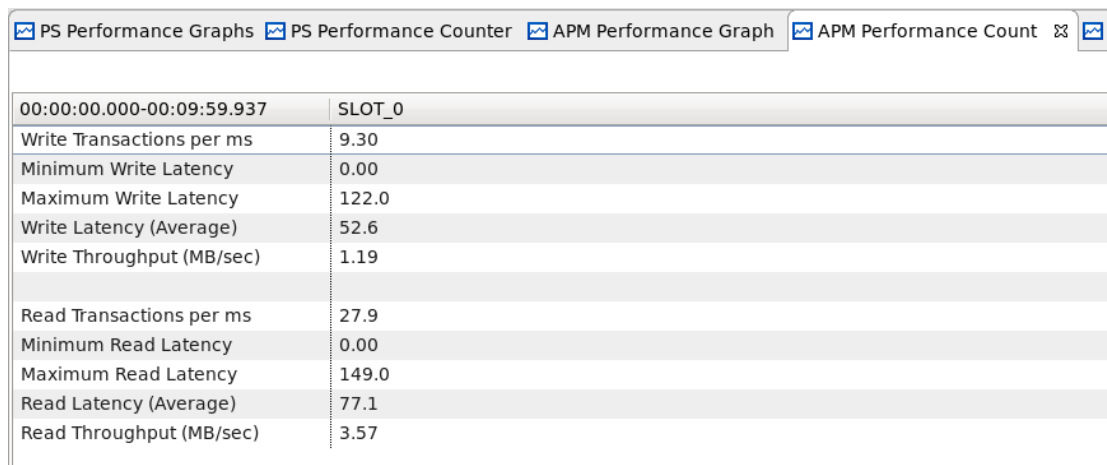

   If prompted by the Confirm Perspective Switch dialog box to stay in the Performance Analysis perspective, click **No**.

16. Scroll through the analysis plots in the lower portion of the perspective to view different performance statistics.

17. Click in any plot area to show a bigger version in the middle of the perspective.

The orange box allows you to focus on a particular time slice of data.

Analyzing the Performance

In this system, the APM is connected to the one port in use between the PS, PL, and the GP port.

Figure 22: **APM Performance Count**

00:00:00.000-00:09:59.937	SLOT_0
Write Transactions per ms	9.30
Minimum Write Latency	0.00
Maximum Write Latency	122.0
Write Latency (Average)	52.6
Write Throughput (MB/sec)	1.19
Read Transactions per ms	27.9
Minimum Read Latency	0.00
Maximum Read Latency	149.0
Read Latency (Average)	77.1
Read Throughput (MB/sec)	3.57

The multiplier and adder accelerator cores are both connected to the accelerator coherency port (ACP) for data input and output.

The GP port is used to issue control commands and get the status of the accelerator cores only, not for data transfer. The blue Slot 0 is connected to the GP port, and the green Slot 1 is connect to the ACP.

Note: The ACP port is not supported on Zynq UltraScale+ MPSoC device for SDSoC environment flows.

The APM is configured in Profile mode with two monitoring slots for each port (ACP and GP). Profile mode provides event counting functionality for each slot. The type of statistics computed by the APM for both reading and writing include:

* **Transaction Count:** Total number of requests that occur on the bus.

* **Byte Counter:** Total number of bytes sent (used for write throughput calculation).

* **Latency:** Time between the start of the address issuance and the last element sent.

The latency and byte counter statistics are used by the APM to automatically compute the throughput (in MB/s). The latency and throughput values shown are for the 50 ms time interval.

Minimum, maximum, and averages also display for latency and throughput statistics.

Send Feedback

Real-World Examples

This chapter describes some real-world examples and shows the following:

- How these examples are optimized using both the top-down flow and bottom-up flow

  ◦ The top-down flow is demonstrated using a Lucas-Kanade (LK) Optical Flow algorithm.

  ◦ The bottom-up flow is demonstrated using a stereo vision block matching algorithm.

- What optimization directives were applied

- Why those directives were chosen

Top-Down: Optical Flow Algorithm

The Lucas-Kanade (LK) method is a widely used, differential method for optical flow estimation or the estimation of movement of pixels between two related images. In this example system, the related images are the current and previous images of a video stream. The LK method is a compute intensive algorithm and works over a window of neighboring pixels using the least square difference to find matching pixels.

The following code example shows how to implement this algorithm, where two input files are read in, processed through function `fpga_optflow`, and the results written to an output file.

```
int main()
{
    FILE *f;
    pix_t *inY1 = (pix_t *)sds_alloc(HEIGHT*WIDTH);
    yuv_t *inCY1 = (yuv_t *)sds_alloc(HEIGHT*WIDTH*2);
    pix_t *inY2 = (pix_t *)sds_alloc(HEIGHT*WIDTH);
    yuv_t *inCY2 = (yuv_t *)sds_alloc(HEIGHT*WIDTH*2);
    yuv_t *outCY = (yuv_t *)sds_alloc(HEIGHT*WIDTH*2);
    printf("allocated buffers\n");

    f = fopen(FILEINAME,"rb");
    if (f == NULL) {
        printf("failed to open file %s\n", FILEINAME);
        return -1;
    }
    printf("opened file %s\n", FILEINAME);

    read_yuv_frame(inY1, WIDTH, WIDTH, HEIGHT, f);
    printf("read 1st %dx%d frame\n", WIDTH, HEIGHT);
    read_yuv_frame(inY2, WIDTH, WIDTH, HEIGHT, f);
```

Send Feedback

```
        printf("read 2nd %dx%d frame\n", WIDTH, HEIGHT);

        fclose(f);
        printf("closed file %s\n", FILEINAME);

        convert_Y8toCY16(inY1, inCY1, HEIGHT*WIDTH);
        printf("converted 1st frame to 16bit\n");
        convert_Y8toCY16(inY2, inCY2, HEIGHT*WIDTH);
        printf("converted 2nd frame to 16bit\n");

        fpga_optflow(inCY1, inCY2, outCY, HEIGHT, WIDTH, WIDTH, 10.0);
        printf("computed optical flow\n");

        // write optical flow data image to disk
        write_yuv_file(outCY, WIDTH, WIDTH, HEIGHT, ONAME);

        sds_free(inY1);
        sds_free(inCY1);
        sds_free(inY2);
        sds_free(inCY2);
        sds_free(outCY);
        printf("freed buffers\n");

return 0;
}
```

This method is typical for a top-down design flow using standard C/C++ data types.

Function `fpa_optflow` is shown in the following code example and contains the following sub-functions:

- `readMatRows`

- `computeSum`

- `computeFlow`

- `getOutPix`

- `writeMatRows`

```
int fpga_optflow (yuv_t *frame0, yuv_t *frame1, yuv_t *framef, int height,
int width, int stride, float clip_flowmag)
{
#ifdef COMPILEFORSW
      int img_pix_count = height*width;
#else
      int img_pix_count = 10;
#endif

  if (f0Stream == NULL) f0Stream = (pix_t *) malloc(sizeof(pix_t) *
img_pix_count);
  if (f1Stream == NULL) f1Stream = (pix_t *) malloc(sizeof(pix_t) *
img_pix_count);
  if (ffStream == NULL) ffStream = (yuv_t *) malloc(sizeof(yuv_t) *
img_pix_count);

  if (ixix == NULL) ixix = (int *) malloc(sizeof(int) * img_pix_count);
  if (ixiy == NULL) ixiy = (int *) malloc(sizeof(int) * img_pix_count);
  if (iyiy == NULL) iyiy = (int *) malloc(sizeof(int) * img_pix_count);
  if (dix == NULL) dix = (int *) malloc(sizeof(int) * img_pix_count);
```

Send Feedback

```
  if (diy == NULL) diy = (int *) malloc(sizeof(int) * img_pix_count);

  if (fx == NULL) fx = (float *) malloc(sizeof(float) * img_pix_count);
  if (fy == NULL) fy = (float *) malloc(sizeof(float) * img_pix_count);

  readMatRows (frame0, f0Stream, height, width, stride);
  readMatRows (frame1, f1Stream, height, width, stride);

  computeSum (f0Stream, f1Stream, ixix, ixiy, iyiy, dix, diy, height,
width);
  computeFlow (ixix, ixiy, iyiy, dix, diy, fx, fy, height, width);
  getOutPix (fx, fy, ffStream, height, width, clip_flowmag);

  writeMatRows (ffStream, framef, height, width, stride);

  return 0;
}
```
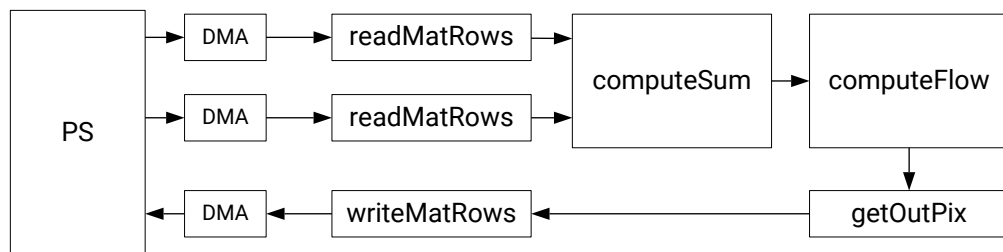
In this example, all of the functions in `fpga_optflow` are processing live video data, and can benefit from hardware acceleration with DMAs used to transfer the data to and from the PS. If all five functions are annotated to be hardware functions, the topology of the system is shown in the following figure:

Figure 23: **System Topology**

X22628-051019

The system can be compiled into hardware and event tracing used to analyze the performance in detail.

The issue here is that it takes a long time to complete— approximately 15 seconds for a single frame. To process HD video, the system should process 60 frames per second or one frame every 16.7 ms. You can use optimization directives, as described below, to ensure the system meets the target performance.

Optical Flow Memory Access Optimization

The first task is to optimize the transfer of data. In this case, because the system will process steaming video, where each sample is processed in consecutive order, the memory transfer optimization is used to ensure the SDSoC™ environment interprets all accesses as sequential in nature.

Send Feedback

This is performed by adding SDS pragmas before the function signatures for all functions involved.

```
#pragma SDS data access_pattern(matB:SEQUENTIAL, pixStream:SEQUENTIAL)
#pragma SDS data mem_attribute(matB:PHYSICAL_CONTIGUOUS)
#pragma SDS data copy(matB[0:stride*height])
void readMatRows (yuv_t *matB, pix_t* pixStream,
        int height, int width, int stride);

#pragma SDS data access_pattern(pixStream:SEQUENTIAL, dst:SEQUENTIAL)
#pragma SDS data mem_attribute(dst:PHYSICAL_CONTIGUOUS)
#pragma SDS data copy(dst[0:stride*height])
void writeMatRows (yuv_t* pixStream, yuv_t *dst,
        int height, int width, int stride);

#pragma SDS data access_pattern(f0Stream:SEQUENTIAL, f1Stream:SEQUENTIAL)
#pragma SDS data access_pattern(ixix_out:SEQUENTIAL, ixiy_out:SEQUENTIAL,
iyiy_out:SEQUENTIAL)
#pragma SDS data access_pattern(dix_out:SEQUENTIAL, diy_out:SEQUENTIAL)
void computeSum(pix_t* f0Stream, pix_t* f1Stream,
        int* ixix_out, int* ixiy_out, int* iyiy_out,
        int*  dix_out, int* diy_out,
        int height, int width);

#pragma SDS data access_pattern(ixix:SEQUENTIAL, ixiy:SEQUENTIAL,
iyiy:SEQUENTIAL)
#pragma SDS data access_pattern(dix:SEQUENTIAL, diy:SEQUENTIAL)
#pragma SDS data access_pattern(fx_out:SEQUENTIAL, fy_out:SEQUENTIAL)
void computeFlow(int* ixix, int* ixiy, int* iyiy,
        int* dix, int* diy,
        float* fx_out, float* fy_out,
        int height, int width);

#pragma SDS data access_pattern(fx:SEQUENTIAL, fy:SEQUENTIAL,
out_pix:SEQUENTIAL)
void getOutPix (float* fx, float* fy, yuv_t* out_pix,
        int height, int width, float clip_flowmag);
```

For the `readMatRows` and `writeMatRows` function arguments, which interface to the processor, the memory transfers are specified as sequential accesses from physically contiguous memory, and the data should be copied to and from the hardware function, and not simply accessed from the accelerator. This ensures the data is copied efficiently. The following options are available:

- **Sequential:** The data is transferred in the same sequential manner as it is processed. This type of transfer requires the least amount of hardware overhead for high data processing rates and means an area efficient datamover is used.

- **Contiguous:** The data is accessed from contiguous memory. This ensures there is no scatter-gather overhead in the data transfer rate and an efficient fast hardware datamover is used. This directive is supported by the associated `scs_alloc` library call in the `main()` function, which ensures data for these arguments is stored in contiguous memory.

- **Copy:** The data is copied to and from the accelerator, negating the need for data accesses back to the CPU or DDR memory. Because pointers are used, the size of the data to be copied is specified.

For the remaining hardware functions, the data transfers are specified as sequential, allowing the most efficient hardware to be used to connect the functions in the programmable logic (PL) fabric.

Optical Flow Hardware Function Optimization

The hardware functions also require optimization directives to execute at the highest level of performance. These are already present in the design example. Reviewing these highlights the lessons learned from Understanding the Hardware Function Optimization Methodology. Most of the hardware functions in this design example are optimized using primarily the PIPELINE directive, in a manner similar to the `getOutPix` function.

Review of the `getOutPix` function shows:

- The sub-functions have an INLINE optimization applied to ensure the logic from these functions is merged with the function above. This automatically occurs for small functions, but the use of this directive ensures the sub-functions are always inlined, and there is no need to pipeline the sub-functions.

- The inner loop of the `getOutPix` function is the loop that processes data at the level of each pixel and is optimized with the PIPELINE directive to ensure it processes one pixel per clock.

```
pix_t getLuma (float fx, float fy, float clip_flowmag)
{
#pragma HLS inline
  float rad = sqrtf (fx*fx + fy*fy);

  if (rad > clip_flowmag) rad = clip_flowmag; // clamp to MAX
  rad /= clip_flowmag;                        // convert 0..MAX to 0.0..1.0
  pix_t pix = (pix_t) (255.0f * rad);

  return pix;
}

pix_t getChroma (float f, float clip_flowmag)
{
#pragma HLS inline
  if (f >   clip_flowmag ) f =  clip_flowmag; // clamp big positive f to
MAX
  if (f < (-clip_flowmag)) f = -clip_flowmag; // clamp big negative f to -
MAX
  f /= clip_flowmag;                  // convert -MAX..MAX to -1.0..1.0
  pix_t pix = (pix_t) (127.0f * f + 128.0f);  // convert -1.0..1.0 to
-127..127 to 1..255

  return pix;
}

void getOutPix (float* fx,
                float* fy,
                yuv_t* out_pix,
                int height, int width, float clip_flowmag)
{
  int pix_index = 0;
  for (int r = 0; r < height; r++) {
    for (int c = 0; c < width; c++) {
```

```
      #pragma HLS PIPELINE
      float fx_ = fx[pix_index];
      float fy_ = fy[pix_index];

      pix_t outLuma = getLuma (fx_, fy_, clip_flowmag);
      pix_t outChroma = (c&1)? getChroma (fy_, clip_flowmag) : getChroma
(fx_, clip_flowmag);
      yuv_t yuvpix;

      yuvpix = ((yuv_t)outChroma << 8) | outLuma;

      out_pix[pix_index++] = yuvpix;
    }
  }
}
```

If you examine the `computeSum` function, you will find examples of the ARRAY_PARTITION and DEPENDENCE directives. In this function, the ARRAY_PARTITION directive is used on array `img1Win`. Because `img1Win` is an array, it is implemented by default in a block RAM, which has a maximum of two ports, as shown in the following code summary:

- `img1Win`: Used in a for-loop that is pipelined to process 1 sample per clock cycle.

- `img1Win`: Read from 8 + (KMEDP1-1) + (KMEDP1-1) times within the for-loop.

- `img1Win`: Written to (KMEDP1-1) + (KMEDP1-1) times within the for-loop.

```
void computeSum(pix_t* f0Stream,
                pix_t* f1Stream,
                int*   ixix_out,
                int*   ixiy_out,
                int*   iyiy_out,
                int*   dix_out,
                int*   diy_out)
{

  static pix_t img1Win [2 * KMEDP1], img2Win [1 * KMEDP1];
  #pragma HLS ARRAY_PARTITION variable=img1Win complete dim=0
   ...
  for (int r = 0; r < MAX_HEIGHT; r++) {
    for (int c = 0; c < MAX_WIDTH; c++) {
      #pragma HLS PIPELINE
      ...
      int cIxTopR = (img1Col_ [wrt] - img1Win [wrt*2 + 2-2]) /2 ;
      int cIyTopR = (img1Win [ (wrt+1)*2 + 2-1] - img1Win [ (wrt-1)*2 +
2-1])  /2;
      int delTopR = img1Win [wrt*2 + 2-1] - img2Win [wrt*1 + 1-1];
      ...
      int cIxBotR = (img1Col_ [wrb] - img1Win [wrb*2 + 2-2]) /2 ;
      int cIyBotR = (img1Win [ (wrb+1)*2 + 2-1] - img1Win [ (wrb-1)*2 +
2-1]) /2;
      int delBotR = img1Win [wrb*2 + 2-1] - img2Win [wrb*1 + 1-1];
      ...
      // shift windows
      for (int i = 0; i < KMEDP1; i++) {
        img1Win [i * 2] = img1Win [i * 2 + 1];
      }
      for (int i=0; i < KMEDP1; ++i) {
        img1Win  [i*2 + 1] = img1Col_ [i];
        ...
```

```
          }
          ...
      } // for c
    }  // for r
    ...
}
```

Because a block RAM only supports a maximum of two accesses per clock cycle, all of these accesses cannot be made in one clock cycle. As noted previously in the methodology, the ARRAY_PARTITION directive is used to partition the array into smaller blocks, in this case into individual elements, by using the complete option. This enables parallel access to all elements of the array at the same time and ensures that the for-loop processes data every clock cycle.

The final optimization directive worth reviewing is the DEPENDENCE directive. The `csIxix` array has a DEPENDENCE directive applied to it. The array is read from and then written to using different indices, as shown in the following code example, and performs these reads and writes within a pipelined loop.

```
 void computeSum(pix_t* f0Stream,
                 pix_t* f1Stream,
                 int*   ixix_out,
                 int*   ixiy_out,
                 int*   iyiy_out,
                 int*   dix_out,
                 int*   diy_out)
{
  ...
   static int csIxix [MAX_WIDTH], csIxiy [MAX_WIDTH], csIyiy [MAX_WIDTH],
csDix [MAX_WIDTH], csDiy [MAX_WIDTH];
   ...
   #pragma HLS DEPENDENCE variable=csIxix inter WAR false
   ...
   int zIdx= - (KMED-2);
   int nIdx = zIdx + KMED-2;

   for (int r = 0; r < MAX_HEIGHT; r++) {
     for (int c = 0; c < MAX_WIDTH; c++) {
        #pragma HLS PIPELINE
        ...
        if (zIdx >= 0) {
          csIxixL = csIxix [zIdx];
          ...
        }
        ...
         csIxix [nIdx] = csIxixR;
         ...
         zIdx++;
         if (zIdx == MAX_WIDTH) zIdx = 0;
         nIdx++;
         if (nIdx == MAX_WIDTH) nIdx = 0;
         ...
      } // for c
    }  // for r
    ...
}
```
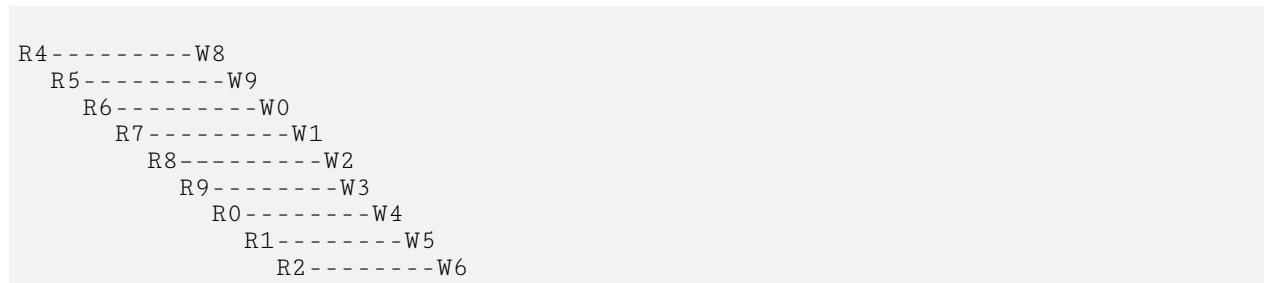
Send Feedback

When a loop is pipelined in hardware, the accesses to the array overlap in time. The compiler analyzes all accesses to an array and issues a warning if any condition exists where the write in iteration `N` overwrites the data for iteration `N + K`, thus changing the value. The warning prevents implementing a pipeline with `II = 1`.

The following example shows read and write operations for a loop over multiple iterations for an array with indices 0 through 9. As in the code above, it is possible for the address counters to differ between the read and write operations and to return to zero, before all loop iterations are complete. The operations are shown overlapped in time, just like a pipelined implementation.

```
R4---------W8
   R5---------W9
      R6---------W0
         R7--------W1
            R8--------W2
               R9-------W3
                  R0-------W4
                     R1-------W5
                        R2-------W6
```

In sequential C code, where each iteration completes before the next starts, it is clear what order the reads and writes occur. However, in a concurrent hardware pipeline, the accesses can overlap and occur in different orders. As can be seen clearly above, it is possible for the read from index 8, as noted by R8, to occur in time before the write to index 8 (W8) which is meant to occur some iterations before R8.

The compiler warns of this condition, and the DEPENDENCE directive is used with the setting `false` to tell the compiler that there is no dependence on read-after-write, allowing the compiler to create the pipelined hardware which performs with `II=1`.

The DEPENDENCE directive is typically used to inform the compiler of algorithm behaviors and conditions external to the function of which is it unaware from static analysis of the code. If a DEPENDENCE directive is set incorrectly, the issue will be discovered in hardware emulation, if the results from the hardware are different from those achieved with the software.

Optical Flow Results

With both the data transfers and hardware functions optimized, the hardware functions are recompiled, and the performance is analyzed using event traces. The figure below shows the start of the event traces, and clearly shows the pipelined hardware functions do not execute until the previous function has completed. Each hardware function begins to process data as soon as data becomes available.

Figure 24: **Trace Result**
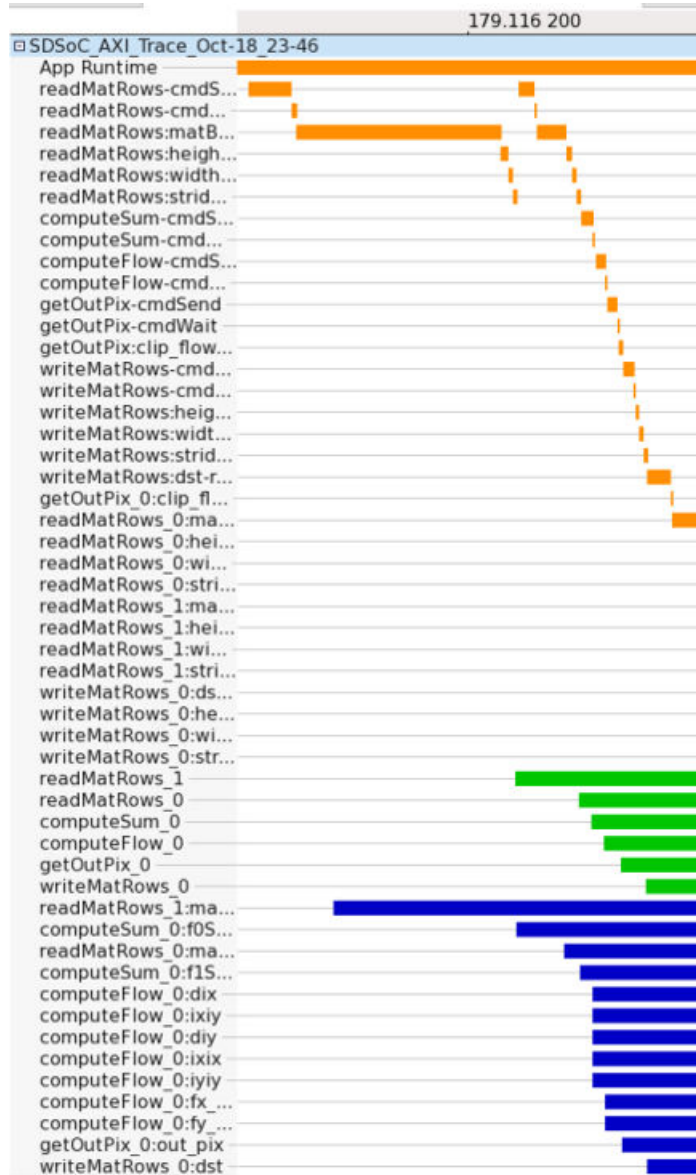
The complete view of the event traces shows all hardware functions and data transfers executing in parallel for the highest performing system, as shown in the following figure.

Figure 25: **Event Traces**
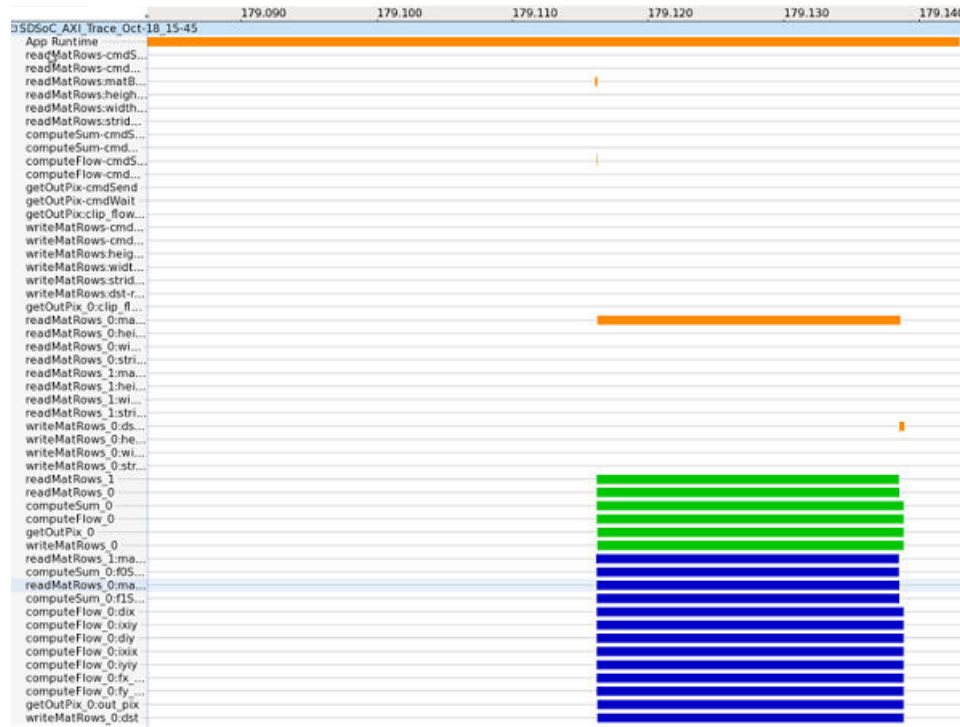
To get the duration time, hover on top of one of the lanes to obtain a popup window that shows the duration of the accelerator runtime. The execution time is just under 15.5 ms; this meets the targeted 16.8 ms necessary to achieve 60 frames per second. The following figure shows the AXI State View for trace legend:

Send Feedback

Figure 26: **AXI State View Trace Legend**
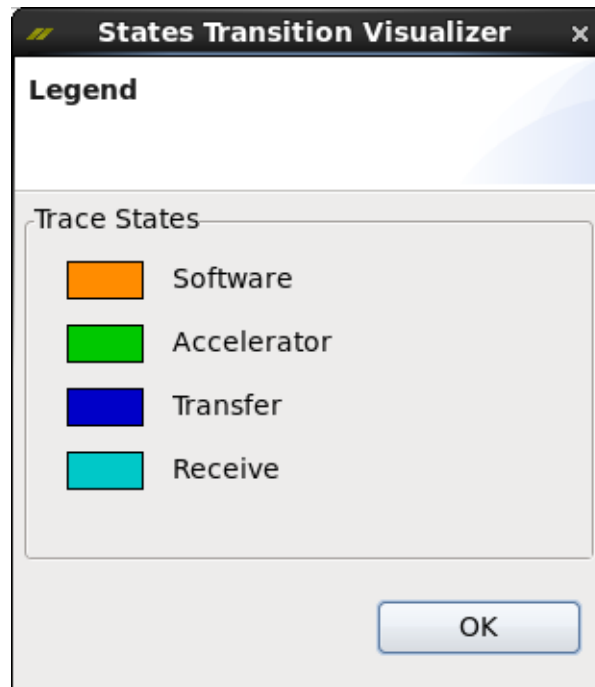
- **Software:** Execution done on the Arm® processor core.

- **Accelerator:** Execution done in the accelerator(s).

- **Transfer:** Data being transferred from Arm core.

- **Receive:** Data being received by the Arm processor core.

# Bottom-Up: Stereo Vision Algorithm

The stereo vision algorithm uses images from two cameras horizontally displaced from each other. This provides two different views of the scene from different vantage points, similar to human vision. To obtain the relative depth information from the scene, compare the two images to build a disparity map. The disparity map encodes the relative positions of objects in the horizontal coordinates such that the values are inversely proportional to the scene depth at the corresponding pixel location.

The bottom-up methodology starts with a fully optimized hardware design that is already synthesized using the Vivado® High-Level Synthesis (HLS) tool and then integrate the pre-optimized hardware function with software in the SDSoC environment.

This flow allows hardware designers who are already knowledgeable with the HLS tool to build and optimize the entire hardware function first, using advanced HLS tool features and then for software programmers to leverage this existing work.

The following section uses the stereo vision design example to take you through the steps of starting with an optimized hardware function in the HLS tool and build an application that integrates the full system with hardware and software running on the board using the SDSoC environment. The following figure shows the final system to be realized, and highlights the existing `stereo_remap_bm` hardware function to be incorporated into the SDSoC environment.

Figure 27: **Block Diagram of System**
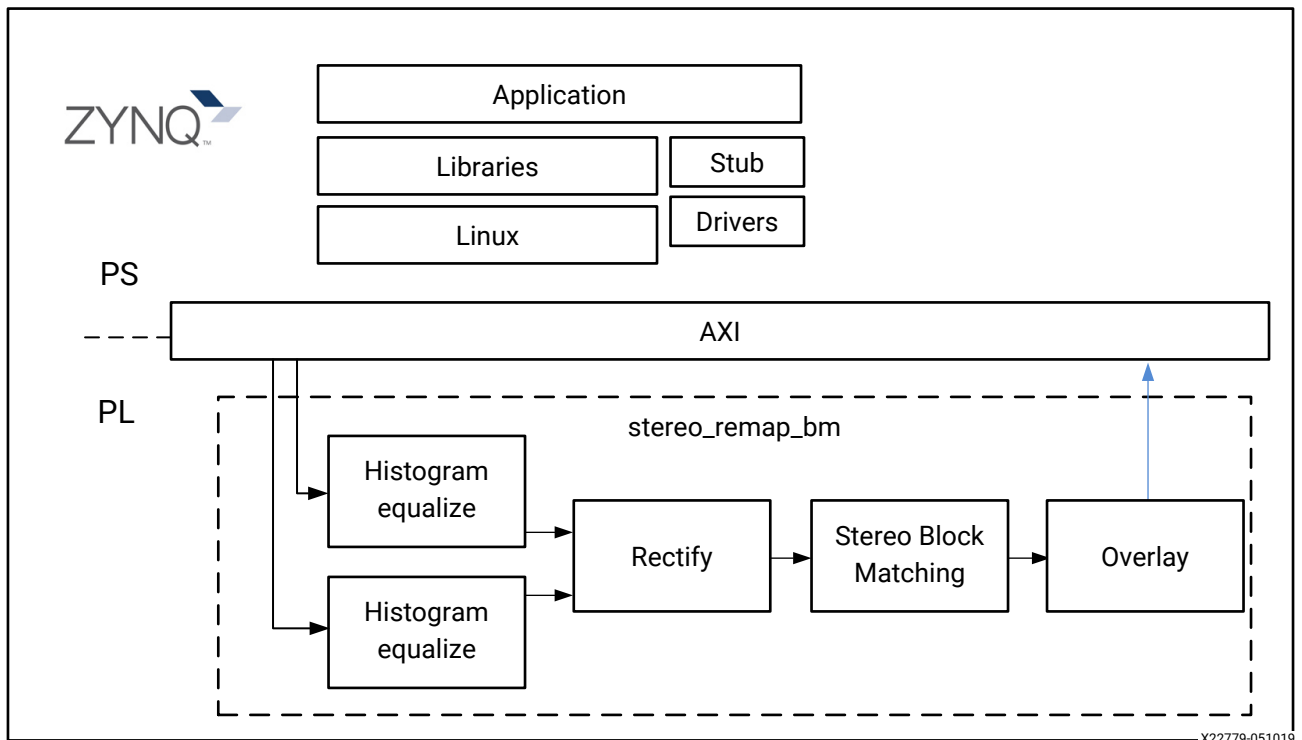
In the bottom-up flow, the general optimization methodology for the SDSoC environment, as detailed in this guide, is reversed. By definition, you would start with an optimized hardware function, and then seek to incorporate it into the SDSoC environment and optimize the data transfers.

Stereo Vision Hardware Function Optimization

The following code example shows the existing `stereo_remap_bm` hardware function with the optimization pragmas highlighted. Before reviewing the optimization directives, note the following details about the function:

- The hardware function contains sub-functions `readLRinput`, `writeDispOut`, and `writeDispOut` that have also been optimized.

Send Feedback

- The hardware function also uses pre-optimized functions, prefixed with the namespace `hls`, from the Vivado HLS tool video library, `hls_video.h`. These sub-functions use their own data type of `MAT`.

```
#include "hls_video.h"
#include "top.h"
#include "transform.h"

void readLRinput (yuv_t *inLR,
        hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1>& img_l,
        hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1>& img_r,
        int height, int dual_width, int width, int stride)
{

  for (int i=0; i < height; ++i) {
#pragma HLS loop_tripcount min=1080 max=1080 avg=1080
    for (int j=0; j < stride; ++j) {
#pragma HLS loop_tripcount min=1920 max=1920 avg=1920
    #pragma HLS PIPELINE
      yuv_t tmpData = inLR [i*stride + j];       // from yuv_t array:
consume height*stride
      if (j < width)
          img_l.write (tmpData & 0x00FF);    // to HLS_8UC1 stream
      else if (j < dual_width)
          img_r.write (tmpData & 0x00FF);    // to HLS_8UC1 stream
    }
  }
}

void writeDispOut(hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1>& img_d,
                  yuv_t *dst,
                  int height, int width, int stride)
{
  pix_t tmpOut;
  yuv_t outData;

  for (int i=0; i < height; ++i) {
#pragma HLS loop_tripcount min=1080 max=1080 avg=1080
    for (int j=0; j < stride; ++j) {
#pragma HLS loop_tripcount min=960 max=960 avg=960
#pragma HLS PIPELINE
      if (j < width) {
        tmpOut = img_d.read().val[0];
        outData = ((yuv_t) 0x8000) | ((yuv_t)tmpOut);
        dst [i*stride +j] = outData;
      }
      else {
        outData = (yuv_t) 0x8000;
        dst [i*stride +j] = outData;
      }
    }
  }
}

namespace hls {
void SaveAsGray(
        Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16SC1>& src,
        Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1>& dst)
{
    int height = src.rows;
    int width = src.cols;
    for (int i = 0; i < height; i++) {
```

```
#pragma HLS loop_tripcount min=1080 max=1080 avg=1080
        for (int j = 0; j < width; j++) {
#pragma HLS loop_tripcount min=960 max=960 avg=960
#pragma HLS pipeline II=1
            Scalar<1, short> s;
            Scalar<1, unsigned char> d;
            src >> s;

            short uval = (short) (abs ((int)s.val[0]));

            // Scale to avoid overflow.  The right scaling here for a
            // good picture depends on the NDISP parameter during
            // block matching.
            d.val[0] = (unsigned char)(uval >> 1);
            //d.val[0] = (unsigned char)(s.val[0] >> 1);
            dst << d;
        }
    }
}
} // namespace hls

int stereo_remap_bm_new(
        yuv_t *img_data_lr,
        yuv_t *img_data_disp,
        hls::Window<3, 3, param_T > &lcameraMA_l,
        hls::Window<3, 3, param_T > &lcameraMA_r,
        hls::Window<3, 3, param_T > &lirA_l,
        hls::Window<3, 3, param_T > &lirA_r,
        param_T (&ldistC_l)[5],
        param_T (&ldistC_r)[5],
        int height,          // 1080
        int dual_width,       // 1920 (two 960x1080 images side by side)
        int stride_in,        // 1920 (two 960x1080 images side by side)
        int stride_out)       // 960
{
    int width = dual_width/2; // 960

#pragma HLS DATAFLOW

    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_l(height, width);
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_r(height, width);

    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_l_remap(height,
width);     // remapped left image
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_r_remap(height,
width);     // remapped left image
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_8UC1> img_d(height, width);

    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16SC2> map1_l(height, width);
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16SC2> map1_r(height, width);
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16UC2> map2_l(height, width);
    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16UC2> map2_r(height, width);

    hls::Mat<IMG_HEIGHT, IMG_WIDTH, HLS_16SC1> img_disp(height, width);
    hls::StereoBMState<15, 32, 32> state;

// ddr -> kernel streams: extract luma from left and right yuv images
// store it in single channel HLS_8UC1 left and right Mat's
    readLRinput (img_data_lr, img_l, img_r, height, dual_width, width,
stride_in);

///////////////////////// remap left and right images, all types are
HLS_8UC1 //////////
```

```
    hls::InitUndistortRectifyMapInverse(lcameraMA_l, ldistC_l, lirA_l,
map1_l, map2_l);
    hls::Remap<8>(img_l, img_l_remap, map1_l, map2_l, HLS_INTER_LINEAR);
    hls::InitUndistortRectifyMapInverse(lcameraMA_r, ldistC_r, lirA_r,
map1_r, map2_r);
    hls::Remap<8>(img_r, img_r_remap, map1_r, map2_r, HLS_INTER_LINEAR);

////////// find disparity of remapped images //////////
    hls::FindStereoCorrespondenceBM(img_l_remap, img_r_remap, img_disp,
state);
    hls::SaveAsGray(img_disp, img_d);

  // kernel stream -> ddr : output single wide
  writeDispOut (img_d, img_data_disp, height, width, stride_out);

  return 0;
}

int stereo_remap_bm(
        yuv_t *img_data_lr,
        yuv_t *img_data_disp,
        int height,         // 1080
        int dual_width,     // 1920 (two 960x1080 images side by side)
        int stride_in,      // 1920 (two 960x1080 images side by side)
        int stride_out)     // 960
{
//1920*1080
//#pragma HLS interface m_axi port=img_data_lr depth=2073600
//#pragma HLS interface m_axi port=img_data_disp depth=2073600

    hls::Window<3, 3, param_T > lcameraMA_l;
    hls::Window<3, 3, param_T > lcameraMA_r;
    hls::Window<3, 3, param_T > lirA_l;
    hls::Window<3, 3, param_T > lirA_r;
    param_T ldistC_l[5];
    param_T ldistC_r[5];

    for (int i=0; i<3; i++) {
        for (int j=0; j<3; j++) {
            lcameraMA_l.val[i][j]=cameraMA_l[i*3+j];
            lcameraMA_r.val[i][j]=cameraMA_r[i*3+j];
            lirA_l.val[i][j]=irA_l[i*3+j];
            lirA_r.val[i][j]=irA_r[i*3+j];
        }
    }
    for (int i=0; i<5; i++) {
        ldistC_l[i] = distC_l[i];
        ldistC_r[i] = distC_r[i];
    }

    int ret = stereo_remap_bm_new(img_data_lr,
                                  img_data_disp,
                                  lcameraMA_l,
                                  lcameraMA_r,
                                  lirA_l,
                                  lirA_r,
                                  ldistC_l,
                                  ldistC_r,
                                  height,
```

```
                                            dual_width,
                                            stride_in,
                                            stride_out);
    return ret;
}
```

As noted in Understanding the Hardware Function Optimization Methodology, the primary optimization directives used are the PIPELINE and DATAFLOW directives. Additionally, the LOOP_TRIPCOUNT directive is used.

Based on the recommendations for optimizing hardware functions, which process frames of data, the PIPELINE directives are all applied to for-loops that process data at the sample level, or in this case, the pixel level. This ensures hardware pipelining is used to achieve the highest performing design.

The LOOP_TRIPCOUNT directives are used on for-loops, for which the upper bound of the loop index is defined by a variable, and the exact value, which is unknown at compile time. The estimated tripcount, or loop iteration count, allows the reports generated by the HLS tool to include expected values for latency and initiation interval (II), instead of unknowns. This directive has no impact on the hardware created—it only impacts reporting.

The top-level `stereo_remap_bm` function is composed of the optimized sub-functions and a number of functions from the HLS tool video library (`hls_video.h`). For details about the library functions provided by the HLS tool video library, refer to *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

The functions provided in the HLS tool video library are already pre-optimized and contain all the optimization directives to ensure they are implemented with the highest possible performance. The top-level function is therefore composed of sub-functions that are all optimized, and it only requires the DATAFLOW directive to ensure each sub-function starts to execute in hardware as soon as data becomes available.

```
int stereo_remap_bm(..) {

#pragma HLS DATAFLOW
  readLRinput (img_data_lr, img_l, img_r, height, dual_width, width, stride
  hls::InitUndistortRectifyMapInverse(lcameraMA_l, ldistC_l, lirA_l,
map1_l, map2_l);
  hls::Remap<8>(img_l, img_l_remap, map1_l, map2_l, HLS_INTER_LINEAR);
  hls::InitUndistortRectifyMapInverse(lcameraMA_r, ldistC_r, lirA_r,
map1_r, map2_r);
  hls::Remap<8>(img_r, img_r_remap, map1_r, map2_r, HLS_INTER_LINEAR);
  hls::Duplicate(img_l_remap, img_l_remap_bm, img_l_remap_pt);
  hls::FindStereoCorrespondenceBM(img_l_remap_bm, img_r_remap, img_disp,
state);
  hls::SaveAsGray(img_disp, img_d);
  writeDispOut (img_l_remap_pt, img_d, img_data_disp, height, dual_width,
width, stride);

}
```

In general, the DATAFLOW optimization is not required because the SDSoC™ environment automatically ensures that data is passed from one hardware function to the next, as soon as it becomes available; however, in this example, the functions within `stereo_remap_bm` are using the HLS tool data type `hls::stream`, which cannot be compiled on the Arm® processor and cannot be used in the hardware function interface in the SDSoC environment. For this reason, the top-level hardware function must be `stereo_remap_bm` and thus, the DATAFLOW directive is used to achieve high-performance transfers between the sub-functions. If this were not the case, the DATAFLOW directive could be removed and each sub-function within `stereo_remap_bm` could be specified as a hardware function.

The hardware functions in this design example use the data type `Mat`, that is based on the HLS tool data type `hls::stream`. The `hls::stream` data type can only be accessed in a sequential manner. Data is pushed on and popped off.

- In software simulation, the `hls::stream` data type has infinite size.

- In hardware, the `hls::stream` data type is implemented as a single register and can only store one data value at a time, because it is expected that the streaming data is consumed before the previous value is overwritten.

By specifying the top-level `stereo_remap_bm` function as the hardware function, the effects of these hardware types can be ignored in the software environment; however, when these functions are incorporated into the SDSoC environment, they cannot be compiled on the Arm processor, and the system can only be verified through hardware emulation, executing on the target platform, or both.

IMPORTANT! *When incorporating hardware functions that contain the HLS tool hardware data types into the SDSoC environment, ensure the functions have been fully verified through C compilation and hardware simulation within the HLS tool environment.*

IMPORTANT! *The `hls::stream` data type is designed for use within the HLS tool, but is unsuitable for running software on embedded CPUs. Therefore, this type should not be part of the top-level hardware function interface.*

If any of the arguments of the hardware function use any HLS tool specific data types, the function must be enclosed by a top-level C/C++ wrapper function that exposes only native C/C++ types in the function argument list.

Optimizing the Data Motion Network

After importing the pre-optimized hardware function into a project in the SDSoC environment, the first task is to remove any interface optimizations. Based on the data types of the hardware function and data access, the interface between the PS and the hardware function is managed and automatically optimized. See Data Motion Optimization.

- Remove any INTERFACE directives present in the hardware function.

- Remove any DATA_PACK directives that reference variables present in the hardware function argument list.

- Remove any of the Vivado HLS tool hardware data types by enclosing the top-level function in wrappers that only use native C/C++ types for the function arguments.

In this example, the functions to be accelerated are captured inside a single top-level hardware function, `stereo_remap_bm`.

```
int main() {

  unsigned char *inY = (unsigned char *)sds_alloc(HEIGHT*DUALWIDTH);
  unsigned short *inCY = (unsigned short *)sds_alloc(HEIGHT*DUALWIDTH*2);
  unsigned short *outCY = (unsigned short *)sds_alloc(HEIGHT*DUALWIDTH*2);
  unsigned char *outY = (unsigned char *)sds_alloc(HEIGHT*DUALWIDTH);

  // read double wide image from disk
  if (read_yuv_file(inY, DUALWIDTH, DUALWIDTH, HEIGHT, FILEINAME) != 0)
    return -1;

  convert_Y8toCY16(inY, inCY, HEIGHT*DUALWIDTH);

  stereo_remap_bm(inCY, outCY, HEIGHT, DUALWIDTH, DUALWIDTH);

  // write single wide image to disk
  convert_CY16toY8(outCY, outY, HEIGHT*DUALWIDTH);
  write_yuv_file(outY, DUALWIDTH, DUALWIDTH, HEIGHT, ONAME);

  // write single wide image to disk
  sds_free(inY);
  sds_free(inCY);
  sds_free(outCY);
  sds_free(outY);
  return 0;
}
```

The key to optimizing the memory accesses to the hardware is to review the data types passed into the hardware function. Reviewing the function signature shows the key variables names to optimize: the input and output data streams `img_data_lr` and `img_data_disp`.

```
int stereo_remap_bm(
        yuv_t *img_data_lr,
        yuv_t *img_data_disp,
        int height,
        int dual_width,
        int stride);
```

Send Feedback

Because the data is transferred in a sequential manner, first ensure that the access pattern is defined as `SEQUENTIAL` for both arguments. For the next optimization, ensure the data transfer is not interrupted by a scatter gather DMA operation by specifying the `memory_attribute` as `PHYSICAL_CONTIGUOUS|NON_CACHEABLE`. This also requires that the memory is allocated with `sds_alloc` from `sds_lib`.

```
#include "sds_lib.h"
int main() {

  unsigned char *inY = (unsigned char *)sds_alloc(HEIGHT*DUALWIDTH);
  unsigned short *inCY = (unsigned short *)sds_alloc(HEIGHT*DUALWIDTH*2);
  unsigned short *outCY = (unsigned short *)sds_alloc(HEIGHT*DUALWIDTH*2);
  unsigned char *outY = (unsigned char *)sds_alloc(HEIGHT*DUALWIDTH);

}
```

Finally, the copy directive is used to ensure the data is explicitly copied to the accelerator, and that the data is not accessed from shared memory.

```
#pragma SDS data access_pattern(img_data_lr:SEQUENTIAL)
#pragma SDS data mem_attribute(img_data_lr:PHYSICAL_CONTIGUOUS|
NON_CACHEABLE)
#pragma SDS data copy(img_data_lr[0:stride*height])
#pragma SDS data access_pattern(img_data_disp:SEQUENTIAL)
#pragma SDS data mem_attribute(img_data_disp:PHYSICAL_CONTIGUOUS|
NON_CACHEABLE)
#pragma SDS data copy(img_data_disp[0:stride*height])
int stereo_remap_bm(
        yuv_t *img_data_lr,
        yuv_t *img_data_disp,
        int height,
        int dual_width,
        int stride);
```

With these optimization directives, the memory access between the PS and PL is optimized for the most efficient transfers.

Stereo Vision Results

After the hardware function optimized with the Vivado HLS tool is wrapped, as in this example, to ensure the HLS tool hardware data types are not exposed at the interface of the hardware function, any interface directives are removed and the data transfers optimized, the hardware functions are recompiled, and the performance is analyzed using event traces.

The following figure shows the complete view of the event traces, and all hardware functions and data transfers executing in parallel for the highest performing system.

Figure 28: **Event Traces**
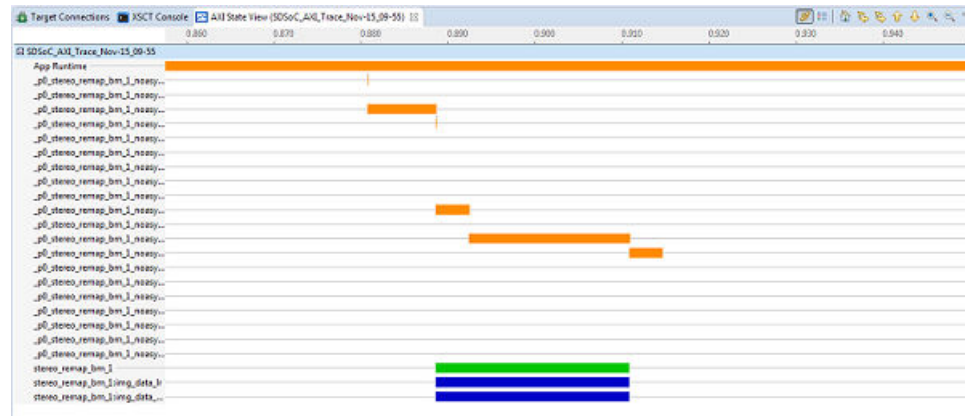
To get the duration time, hover over one of the lanes to display a popup window that shows the duration of the accelerator runtime. The execution time is 15.86 ms; this meets the targeted 16.8 ms necessary to achieve 60 frames per second for live video.

Send Feedback

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. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- 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

These documents provide supplemental material useful with this guide:

Send Feedback

1. *SDSoC Environments Release Notes, Installation, and Licensing Guide* (UG1294)

2. *SDSoC Environment User Guide* (UG1027)

3. *SDSoC Environment Getting Started Tutorial* (UG1028)

4. *SDSoC Environment Tutorial: Platform Creation* (UG1236)

5. *SDSoC Environment Platform Development Guide* (UG1146)

6. *SDSoC Environment Profiling and Optimization Guide* (UG1235)

7. *SDx Command and Utility Reference Guide* (UG1279)

8. *SDSoC Environment Programmers Guide* (UG1278)

9. *SDSoC Environment Debugging Guide* (UG1282)

10. *SDx Pragma Reference Guide* (UG1253)

11. *UltraFast Embedded Design Methodology Guide* (UG1046)

12. *Zynq-7000 SoC Software Developers Guide* (UG821)

13. *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137)

14. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 SoC User Guide* (UG850)

15. *ZCU102 Evaluation Board User Guide* (UG1182)

16. *PetaLinux Tools Documentation: Reference Guide* (UG1144)

17. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

18. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

19. SDSoC Development Environment web page

20. Vivado® Design Suite Documentation

Training Resources

1. SDSoC Development Environment and Methodology

2. Advanced SDSoC Development Environment and Methodology

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright