# SDSoC Environment Optimization Guide

**UG1235 (v2016.4) March 9, 2017**

XILINX
ALL PROGRAMMABLE™

# Revision History

The following table shows the revision history for this document.

| Date | Version | Revision |
|---|---|---|
| 03/07/2017 | 2016.4 | Updated for SDx™ IDE 2016.4. |
| 11/30/2016 | 2016.3 | Initial documentation release for SDx IDE 2016.3, which includes both the SDSoC™ Environment and the SDAccel™ Environment. |

# Table of Contents

XILINX
ALL PROGRAMMABLE

*Chapter 1*

# Introduction

The Zynq®-7000 All Programmable SoC and Zynq UltraScale+™ MPSoC 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™ (software-defined system-on-chip) environment is a tool suite that includes an Eclipse-based integrated development environment (IDE) for implementing heterogeneous embedded systems using the Zynq-7000 SoC and Zynq UltraScale+ MPSoC platforms.

The SDSoC environment includes system compilers that transform C/C++ programs into complete hardware/software systems with select functions compiled into programmable logic to enable hardware acceleration of the selected functions. This guide provides software programmers with an appreciation 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.

Although a detailed knowledge of hardware is not required for using SDSoC, an appreciation of the hardware resources available on the device and how a hardware function achieves very high performance through increased parallelism will help you select the appropriate compiler optimization directives to meet your performance needs.

The methodology can be summarized as follows:

- Profile your software and identify functions that would benefit from hardware acceleration.
- Optimize the data transfer between the CPU, often referred to as the Programmable System (PS), and the Programmable Logic (PL) fabric used to implement the hardware.
- Optimize the parallelism of the hardware.

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

# Guide Organization

This guide employs the integrated profiling in SDSoC to analyze and understand which functions would benefit from hardware acceleration and the implications of C/C++ constructs on system performance. The chapters in this guide are organized as follows:

- Chapter 2: What is an SoC?

  This chapter introduces the computational elements available on an FPGA and how they compare to a processor. It covers topics such as FPGA memory hierarchy, logic elements, and how these elements interrelate.

- Chapter 3: Memory Access Optimizations

  This chapter details how the transfer of data between the PS and PL is implemented using datamovers, and how user-specified compiler directives can be used to select specific datamovers optimized for the particular memory architectures, addressing memory allocation, sharing, and cache coherency. The optimizations described in this chapter primarily improve memory accesses, and their effects on the overall design performance are described in detail.

- Chapter 4: Optimizing the Hardware Function

  This chapter describes the compiler directives used to optimized the hardware function and provides a methodology to apply them. These optimizations are primarily targeted to improve data path performance, and their effects on the overall design performance are described in detail.

- Chapter 5: Putting It All Together

  This chapter shows how the steps learned in this guide can be applied to a real-world design example and how existing hardware functions, developed by hardware engineers using Vivado HLS, can be easily incorporated into an SDSoC project. The optimizations used to improve this design target both data path and memory accesses. These optimizations are gradually applied to improve the kernel, and their effects are described in detail.

Chapter 2

# What is an SoC?

An FPGA is an integrated circuit (IC) that can be programmed for different algorithms after fabrication. Modern FPGA devices consist of up to two million logic cells that can be configured to implement a variety of software algorithms. Although the traditional FPGA design flow is more similar to a regular IC than a processor, an FPGA provides significant cost advantages in comparison to an IC development effort and offers the same level of performance in most cases. Another advantage of the FPGA when compared to the IC is its ability to be dynamically reconfigured. This process, which is the same as loading a program in a processor, can affect part or all of the resources available in the FPGA fabric.

When using SDSoC, it is important to have a basic understanding of the available resources in the FPGA fabric and how they interact to execute a target application. This chapter presents fundamental information about FPGAs, which is required to guide SDSoC to the best computational architecture for any algorithm.

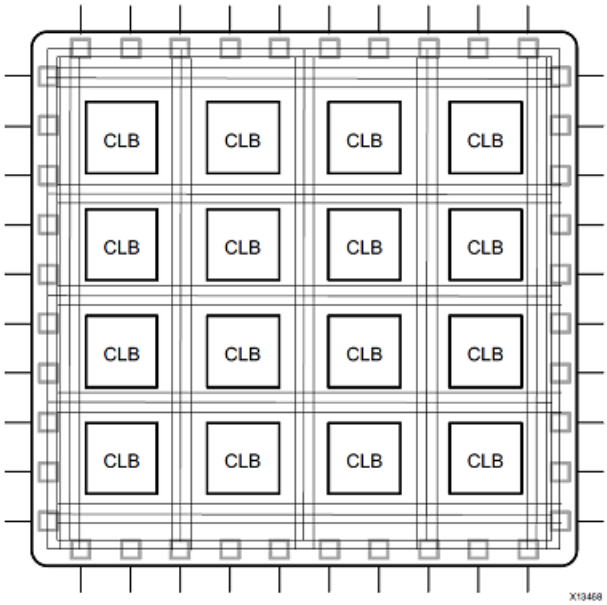
## SoC Architecture

Xilinx FPGAs are heterogeneous compute platforms that include Block RAMs, DSP Slices, PCI Express support, and programmable fabric. They enable parallelism and pipelining of applications across the entire platform as all of these compute resources can be used simultaneously. SDSoC is the tool provided by Xilinx to target and enable these compute resources for C/C++ hardware functions.

The basic structure of an FPGA is composed of the following elements:

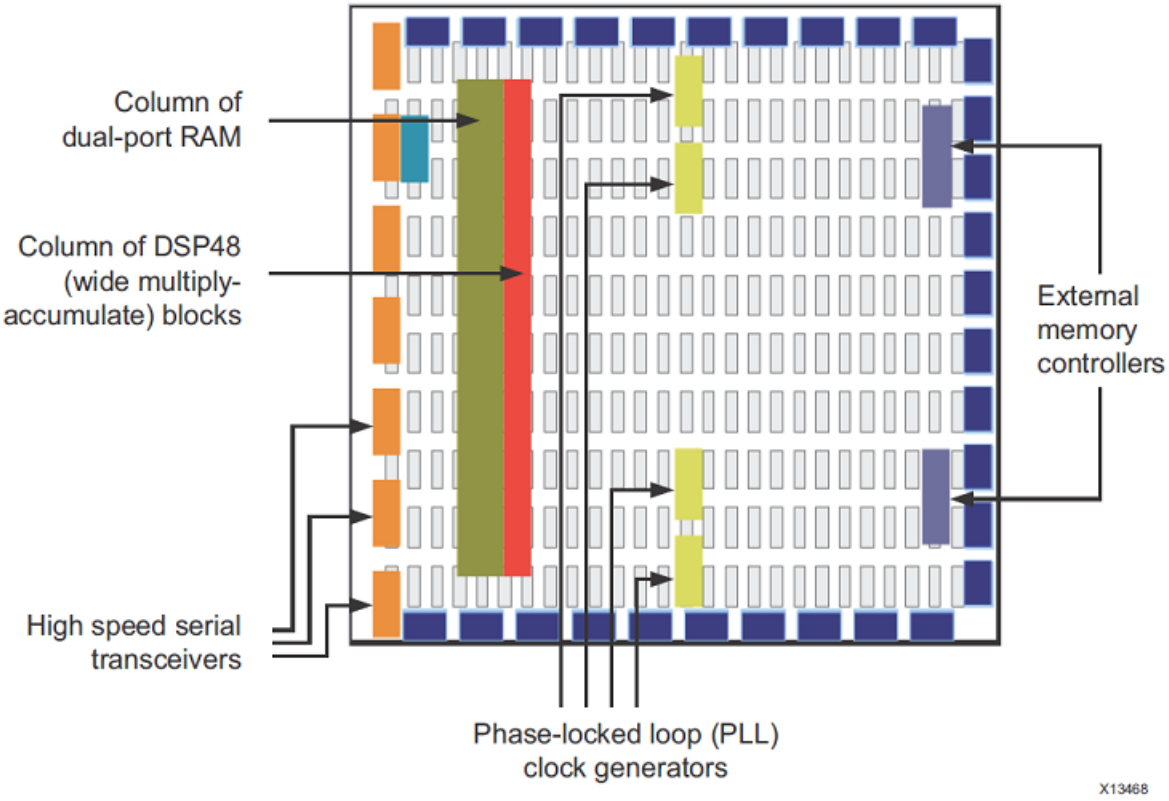
- Look-up table (LUT) - This element performs logic operations.
- Flip-Flop (FF) - This register element stores the result of the LUT.
- Wires - These elements connect elements to one another.
- Input/Output (I/O) pads - These physical ports get data in and out of the FPGA.

The combination of these elements results in the basic FPGA architecture shown in the following figure. Although this structure is sufficient for the implementation of any algorithm, the efficiency of the resulting implementation is limited in terms of computational throughput, required resources, and achievable clock frequency.




Contemporary FPGA architectures incorporate the basic elements along with additional computational and data storage blocks that increase the computational density and efficiency of the device. These additional elements, which are discussed in the following sections, include:

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

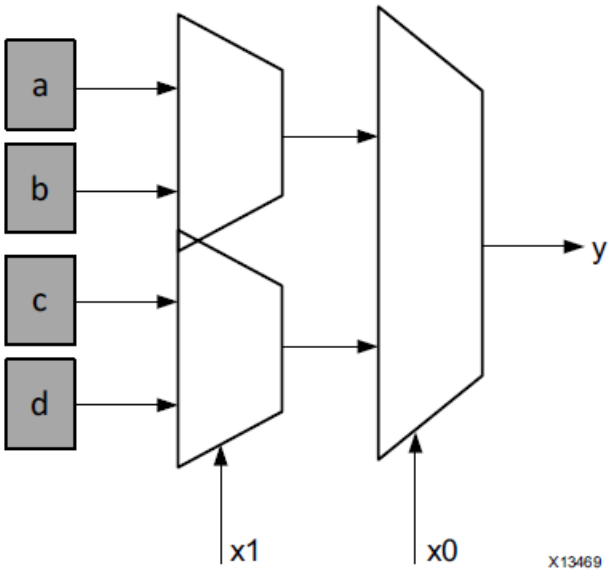



The above figure shows the combination of these elements on a contemporary FPGA architecture. This provides the FPGA with the flexibility to implement any software algorithm running on a processor. Note that all of these elements across the entire FPGA device can be used concurrently, creating a unique compute platform for C/C++ applications.

## LUT

The LUT is the basic building block of an FPGA and is capable of implementing any logic function of N Boolean variables. Essentially, this element is a truth table in which different combinations of the inputs implement different functions to yield output values. The limit on the size of the truth table is N, where N represents the number of inputs to the LUT. For the general N-input LUT, the number of memory locations accessed by the table is $2^N$. This allows the table to implement $2^{N^N}$ functions. Note that a typical value for N in Xilinx FPGAs is 6.

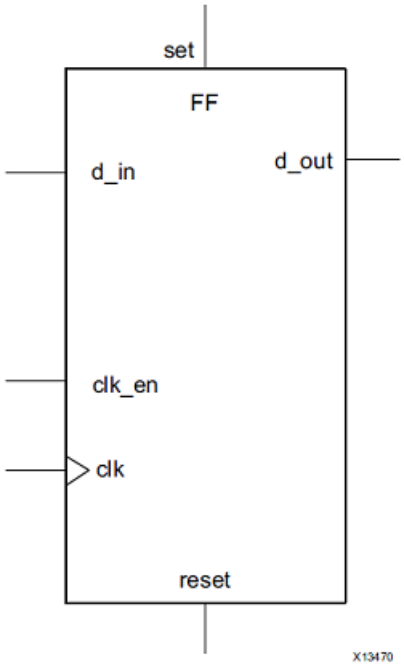
The hardware implementation of a LUT can be thought of as a collection of memory cells connected to a set of multiplexers, as shown in the following figure.




The inputs to the LUT act as selector bits on the multiplexer to select the result at a given point in time. It is important to keep this representation in mind, because a LUT can be used as both a function compute engine and a data storage element.

## Flip-Flop

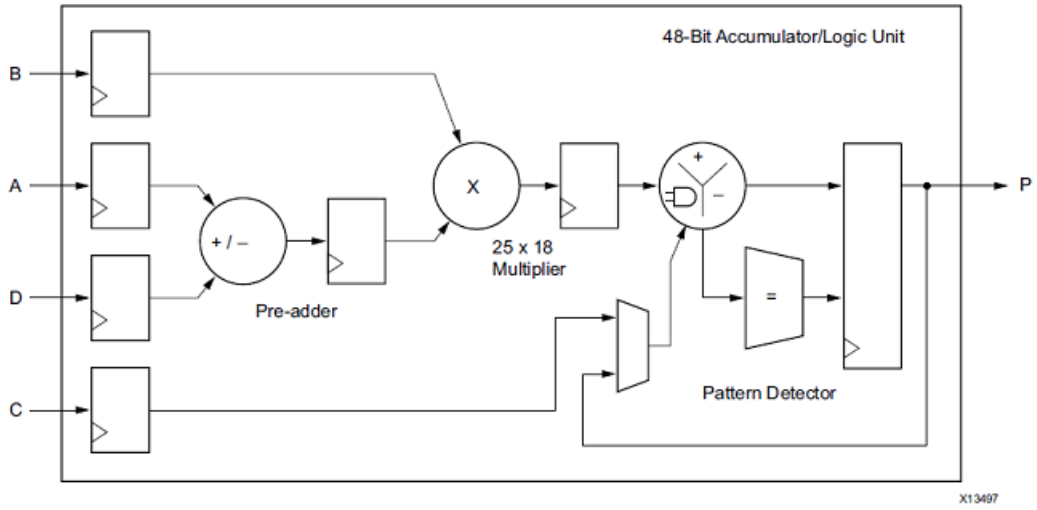
The basic structure of a flip-flop includes a data input, clock input, clock enable, reset, and data output as shown in the following figure.



During normal operation, any value at the data input port is latched and passed to the output on every pulse of the clock. The purpose of the clock enable pin is to allow the flip-flop to hold a specific value for more than one clock pulse. New data inputs are only latched and passed to the data output port when both clock and clock enable are equal to one.

## DSP48 Block

The most complex computational block available in a Xilinx FPGA and the Zynq®-7000 All Programmable SoC and Zynq UltraScale+™ MPSoC PL is the DSP48 block shown below.




The DSP48 block is an arithmetic logic unit (ALU) embedded into the fabric of the FPGA and is composed of a chain of three different blocks. The computational chain in the DSP48 contains an add/subtract unit connected to a multiplier connected to a final add/subtract/accumulate engine. This chain allows a single DSP48 unit to implement functions of the form:

```
P=Bx(A+D)+C
```

Or

```
P +=Bx(A+D)
```

The DSP48 block can be utilized by SDSoC to perform a lot of the computational load within hardware functions. The synthesis flow inside the SDSoC tool targets this block automatically.

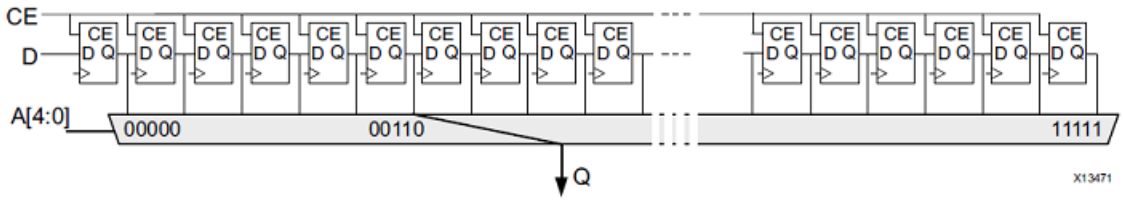
## Block RAM and Other Memories

The FPGA fabric includes embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These elements are block RAM, LUTs, and shift registers.

The block RAM is a dual-port RAM module instantiated into the FPGA fabric to provide on-chip storage for a relatively large set of data. The two types of block RAM memories available in a device can hold either 18k or 36k bits, and the available amount of these memories is device specific. The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations.

In C/C++ code, block RAM can implement either a RAM or a ROM, covering on-chip global, local, and private memory types. In a RAM configuration, the data can be read and written at any time during the runtime of the circuit. In contrast, in a ROM configuration, data can only be read during the runtime of the circuit. The data of the ROM is written as part of the FPGA configuration and cannot be modified in any way.

As previously discussed, the LUT is a small memory in which the contents of a truth table are written during device configuration. Due to the flexibility of the LUT structure in Xilinx FPGAs, these blocks can be used as 64-bit memories and are commonly referred to as distributed memories. This is the fastest kind of memory available on the FPGA, because it can be instantiated in any part of the fabric that improves the performance of the implemented circuit. The following figure shows the structure of an addressable shift register.




The shift register is a chain of registers connected to each other. The purpose of this structure is to provide data reuse along a computational path, such as with a filter. For example, a basic filter is composed of a chain of multipliers that multiply a data sample against a set of coefficients. By using a shift register to store the input data, a built-in data transport structure moves the data sample to the next multiplier in the chain on every clock cycle.

# FPGA Parallelism Versus Processor Architectures

When compared with processor architectures, the structures that comprise the Programmable Logic (PL) fabric in a Zynq®-7000 All Programmable SoC or a Zynq UltraScale+™ MPSoC enable a high degree of fine-grained parallelism in application execution. The custom processing microarchitecture generated by SDSoC for a hardware function presents a different execution paradigm from CPU execution. This must be taken into account when porting an application from a processor to a Zynq-7000 All Programmable SoC or to a Zynq UltraScale+ MPSoC. To examine the benefits of a PL-based execution paradigm, this section provides a brief review of processor program execution.

## Program Execution on a Processor

A processor, regardless of its type, executes a program as a sequence of instructions that translate into useful computations for the software application. This sequence of instructions is generated by processor compiler tools, such as the GNU Compiler Collection (GCC), which transform an algorithm expressed in C/C++ into assembly language constructs that are native to the processor. The job of a processor compiler is to take a C function of the form:

```
z=a+b;
```

and transform it into assembly code as follows:

```
ADD $R1,$R2,$R3
```

The assembly code above defines the addition operation to compute the value of z in terms of the internal registers of a processor. The input values for the computation are stored in registers R1 and R2, and the result of the computation is stored in register R3. The assembly code above is simplified as it does not express all the instructions needed to compute the value of z. This code only handles the computation after the data has arrived at the processor. Therefore, the compiler must create additional assembly language instructions to load the registers of the processor with data from a central memory and to write back the result to memory. The complete assembly program to compute the value of z is as follows:

```
LD      a,$R1
LD      b,$R2
ADD     R1,$R2,$R3
ST      $R3,c
```

This code shows that even a simple operation, such as the addition of two values, results in multiple assembly instructions. The computational latency of each instruction is not equal across instruction types. For example, depending on the location of a and b, the LD operations take a different number of clock cycles to complete. If the values are in the processor cache, these load operations complete within a few tens of clock cycles. If the values are in the main, double data rate (DDR) memory, these operations take hundreds of clock cycles to complete. If the values are on a hard drive, the load operations take even longer to complete. This is why software engineers with cache hit traces spend so much time restructuring their algorithms to increase the spatial locality of data in memory to increase the cache hit rate and decrease the processor time spent per instruction [2].

## Program Execution on an FPGA

The FPGA is an inherently parallel processing fabric capable of implementing any logical and arithmetic function that can run on a processor. The main difference is that the Vivado® High-Level Synthesis (HLS) compiler [3], which is used by SDSoC to transform C/C++ software descriptions into RTL, is not hindered by the restrictions of a cache and a unified memory space.

The computation of z is compiled by HLS into several LUTs required to achieve the size of the output operand. For example, assume that in the original software program the variables a, b, and z are defined with the short data type. This type, which defines a 16-bit data container, gets implemented as 16 LUTs by HLS. As a general rule, 1 LUT is equivalent to 1 bit of computation.

The LUTs used for the computation of z are exclusive to this operation only. Unlike a processor where all computations share the same ALU, an FPGA implementation instantiates independent sets of LUTs for each computation in the software algorithm.

In addition to assigning unique LUT resources per computation, the FPGA differs from a processor in both memory architecture and the cost of memory accesses. In an FPGA implementation, the HLS compiler arranges memories into multiple storage banks as close as possible to the point of use in the operation. This results in an instantaneous memory bandwidth, which far exceeds the capabilities of a processor. For example, the Xilinx Zynq®-7100 device has a total of 1,510 18k-bit BRAMs available. In terms of memory bandwidth, the memory layout of this device provides the software engineer with the capacity of 0.5M-bits per second at the register level and 23T-bits per second at the BRAM level.

With regard to computational throughput and memory bandwidth, the HLS compiler exercises the capabilities of the FPGA fabric through the processes of scheduling, pipelining, and dataflow. Although transparent to the user, these processes are integral stages of the software compilation process that extract the best possible circuit-level implementation of the software application.

## Scheduling

Scheduling is the process of identifying the data and control dependencies between different operations to determine when each will execute. In traditional FPGA design, this is a manual process also referred to as parallelizing the software algorithm for a hardware implementation.

HLS analyzes dependencies between adjacent operations as well as across time. This allows the compiler to group operations to execute in the same clock cycle and to set up the hardware to allow the overlap of function calls. The overlap of function call executions removes the processor restriction that requires the current function call to fully complete before the next function call to the same set of operations can begin. This process is called pipelining and is covered in detail in the following section and remaining chapters.

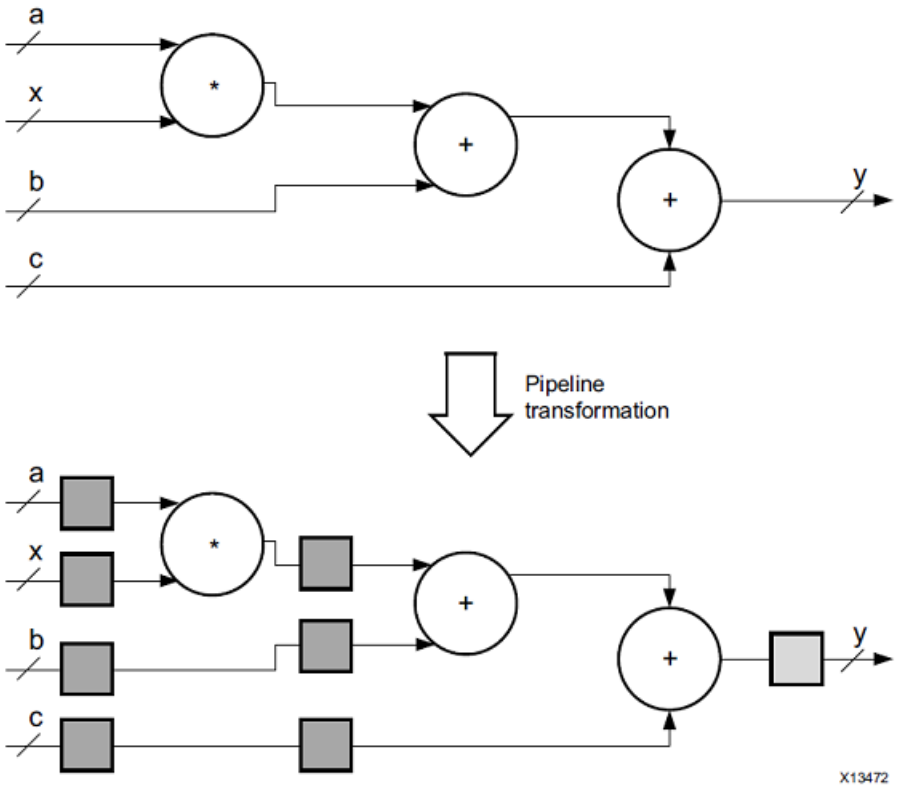
## Pipelining

Pipelining is a digital design technique that allows the designer to avoid data dependencies and increase the level of parallelism in an algorithm hardware implementation. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit is divided into a chain of independent stages. All stages in the chain run in parallel on the same clock cycle. The only difference is the source of data for each stage. Each stage in the computation receives its data values from the result computed by the preceding stage during the previous clock cycle. For example, consider the following function:

```
y=(a*x)+b+c
```

The HLS compiler instantiates one multiplier and two adder blocks to implement this function.

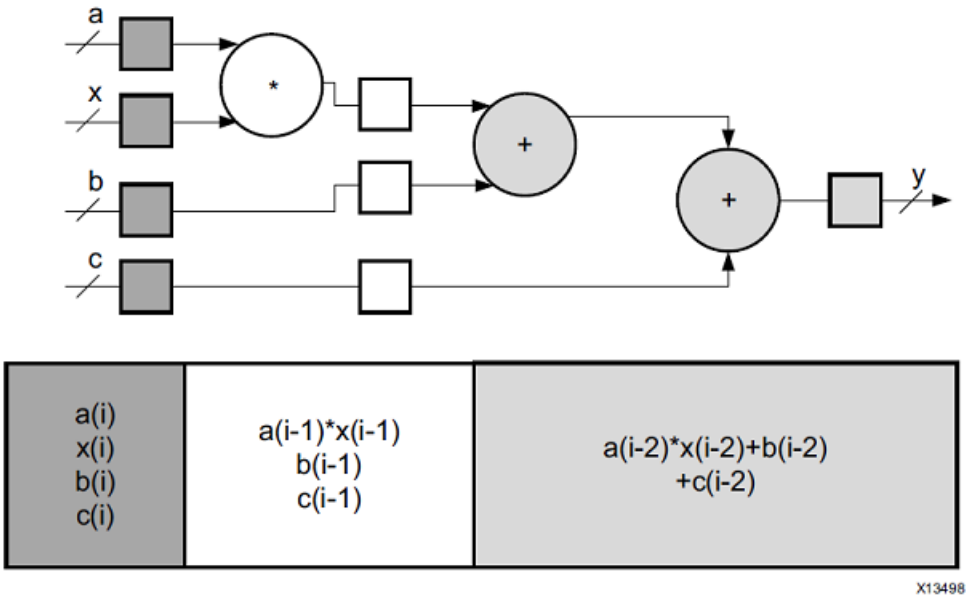
The following figure shows this compute structure and the effects of pipelining. It shows two implementations of the example function. The top implementation is the data path required to compute the result y without pipelining. This implementation behaves similarly to the corresponding software function in that all input values must be known at the start of the computation, and only one result y can be computed at a time. The bottom implementation shows the pipelined version of the same circuit.




The boxes in the data path in the above figure represent registers that are implemented by flip-flop blocks in the FPGA fabric. Each box can be counted as a single clock cycle. Therefore, in the pipelined version, the computation of each result y takes three clock cycles. By adding the register, each block is isolated into separate compute sections in time.

This means that the section with the multiplier and the section with the two adders can run in parallel and reduce the overall computational latency of the function. By running both sections of the data path in parallel, the block is essentially computing the values y and y' in parallel, where y' is the result of the next execution of the equation for y above. The initial computation of y, which is also referred to as the pipeline fill time, takes three clock cycles. However, after this initial computation, a new value of y is available at the output on every clock cycle. The computation pipeline contains overlapped data sets for the current and subsequent y computations.

The following figure shows a pipelined architecture in which raw data (dark gray), semi-computed data (white), and final data (light gray) exist simultaneously, and each stage result is captured in its own set of registers. Thus, although the latency for such computation is in multiple cycles, a new result can be produced on every cycle.




## Dataflow

Dataflow is another digital design technique, which is similar in concept to pipelining. The goal of dataflow is to express parallelism at a coarse-grain level. In terms of software execution, this transformation applies to parallel execution of functions within a single program.

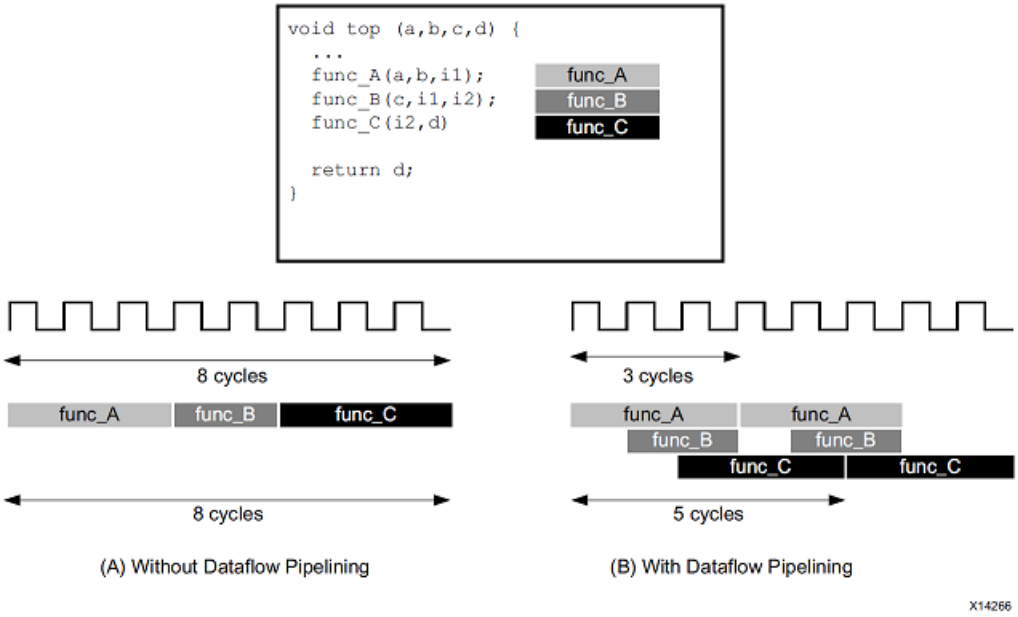
SDSoC extracts this level of parallelism by evaluating the interactions between different functions of a program based on their inputs and outputs. The simplest case of parallelism is when functions work on different data sets and do not communicate with each other. In this case, SDSoC allocates FPGA logic resources for each function and then runs the blocks independently. The more complex case, which is typical in software programs, is when one function provides results for another function. This case is referred to as the consumer-producer scenario.

Dataflow is automatically performed between functions which are marked for hardware acceleration. Given the following code example, where both function load_input and function store_output are marked for implementation in the hardware fabric and a consumer-producer relationship is present through variable buffer, SDSoC automatically performs dataflow, ensuring the consumer hardware function store_output starts operation as soon as data is available from producer function load_input.

```
main() {
int input_r[N], buffer[N], output_w[N];
...
load_input(input_r, buffer);
store_output(buffer, output_w);
...
}
```

When the functions marked for hardware acceleration contain sub-functions, a dataflow optimization directive is required to ensure the sub-functions execute in a dataflow manner and further enhance parallelism.

The preceding figure shows a conceptual view of dataflow pipelining. After synthesis, the default behavior is to execute and complete `func_A`, then `func_B`, and finally `func_C`. However, you can use the Vivado HLS `DATAFLOW` directive to schedule each function to execute as soon as data is available. In this example, the original function has a latency and interval of eight clock cycles. When you use dataflow optimization, the interval is reduced to only three clock cycles. The tasks shown in this example are functions, but you can perform dataflow optimization between functions, between functions and loops, and between loops.




SDSoC supports two use models for the consumer-producer scenario in sub-functions. In the first use model, the producer creates a complete data set before the consumer can start its operation. Parallelism is achieved by instantiating a pair of BRAM memories arranged as memory banks ping and pong. Each function can access only one memory bank, ping or pong, for the duration of a function call. When a new function call begins, the HLS-generated circuit switches the memory connections for both the producer and the consumer. This approach guarantees functional correctness but limits the level of achievable parallelism to across function calls.

In the second use model, the consumer can start working with partial results from the producer, and the achievable level of parallelism is extended to include execution within a function call. The HLS-generated modules for both functions are connected through the use of a first in, first out (FIFO) memory circuit. This memory circuit, which acts as a queue in software programming, provides data-level synchronization between the modules. At any point during a function call, both hardware modules are executing their programming. The only exception is that the consumer module waits for some data to be available from the producer before beginning computation. In HLS terminology, the wait time of the consumer module is referred to as the interval or initiation interval (II).

Chapter 3

# Memory Access Optimizations

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

Data is often stored in an external DDR. However, because accesses to an off-chip DDR can take time and reduce the system performance, it is well understood that in high-performance systems data can also be stored in a local cache to reduce the memory access times and improve run time. In addition to these memories, an FPGA provides local memory where small- to 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.

A well-designed hardware function minimizes the latency impact of accessing and storing the data through the extensive use of local memories.

A few suggested techniques include the following:

- Data Motion Optimization - This includes the transfer of data between the PS and the PL fabric. The SDSoC environment implements a default data motion architecture based on the arguments of the functions selected for the PL. However, optimization directives might be used, for example, to ensure data is stored in contiguous memory to improve the efficiency of the data transfer or a scatter-gather transfer is used to more efficiently transfer very large sets of data.
- Data Access Patterns - An FPGA excels at processing data quickly and in a highly concurrent manner. Poor data access patterns interrupt the flow of data and leave the computational power of an FPGA waiting for data. 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 utilize the block RAM on the FPGA and are physically located near the computation. They allow one-cycle reads and writes, thus drastically improving memory access performance. Copying the data efficiently using optimal data access patterns from the DDR to these local memories can be done very quickly using burst transactions and can considerably improve performance.

# Data Motion Optimization

The data motion network is the hardware used to connect the programming 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 or Class**

A single `struct`/`class` 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`/`class`, the data motion network is the same as an `array` discussed previously.

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

## Memory Allocation

The `sdscc/sds++` compilers analyze your program and select 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.

```
sds_alloc(size_t size); // guarantees physically contiguous memory
sds_mmap(void *paddr, size_t size, void *vaddr); // paddr must point to
contiguous memory
sds_register_dmabuf(void *vaddr, int fd); // assumes physically contiguous
memory
```

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

```
WARNING: [SDSoC 0-0] 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
```

## 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 synthesizeable 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 `memcpy` 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, storing 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 `sdscc` 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.

```
ERROR: [SDSoC 0:0] The bound callers of accelerator foo have different/
indeterminate data size for port p.
```

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
```

## Data Cache Coherency

The `sdscc/sds++` compilers automatically generate 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

An FPGA is selected to implement the C code due to the superior performance of the FPGA - the massively parallel architecture of an FPGA allows it to perform operations much faster than the inherently sequential operations of a processor, and users typically wish 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:

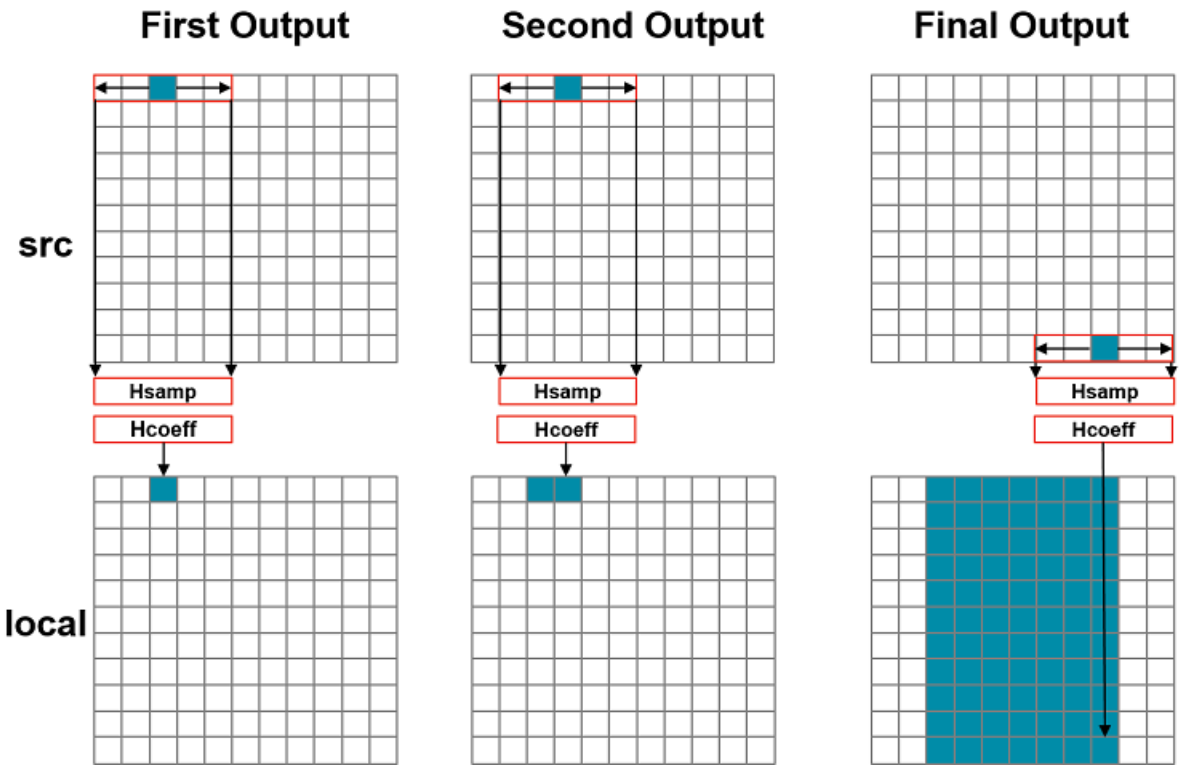
- A horizontal convolution.
- Followed by a vertical convolution.
- Followed by a manipulation of the border pixels.

```
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 col = 0; col < height; col++){
    HconvWfor(int row = border_width; row < width - border_width; row++){
      Hconv:for(int i = - border_width; i <= border_width; i++){
      }
```

```
    }
  }

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

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

### *Standard Horizontal Convolution*

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



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 and intuitive. There are, however, some issues with this C code that will negatively impact the quality of the hardware results.

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 cross-compilers targeting Zynq®-7000 All Programmable SoC or Zynq UltraScale+™ MPSoC, as well as many host systems, this amount of local storage can lead to stack overflows at run time (for example, running on the target device, or running co-sim flows within Vivado HLS). 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 sufficent stack space. (Note that 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 loop Clear_Local 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.

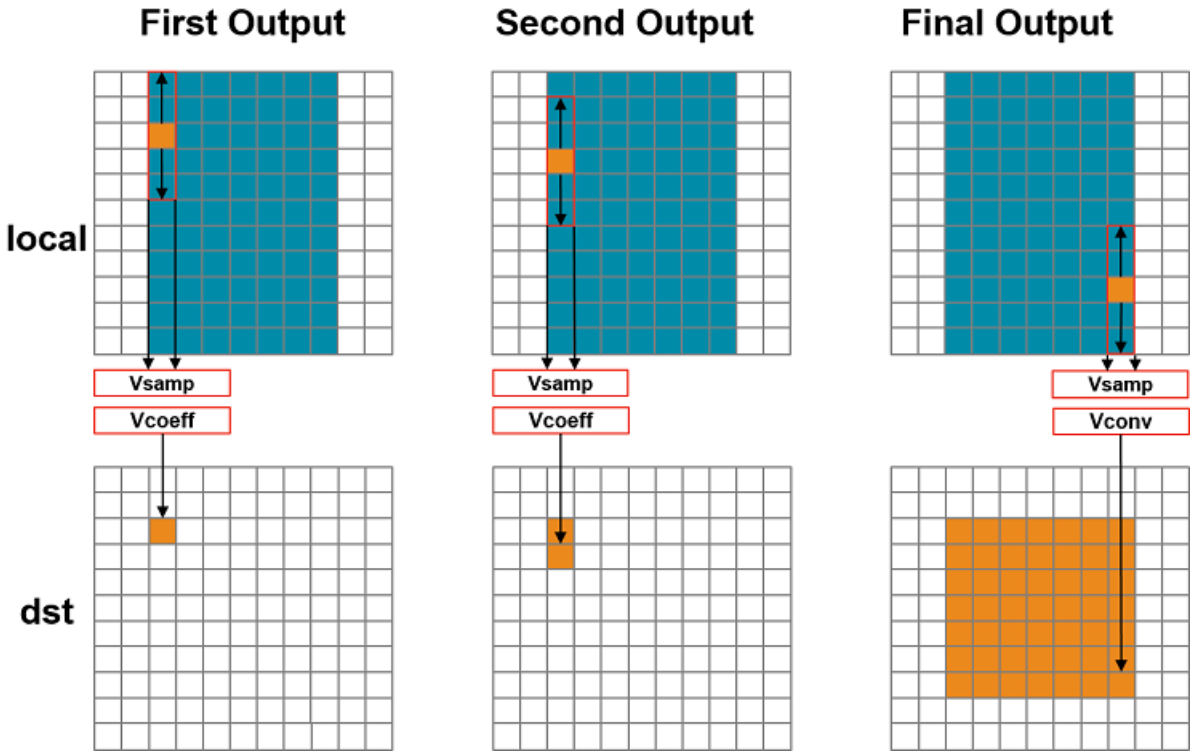
One of the keys to a high-performance FPGA is to minimize the access to and from the PS. Each access for data, which has previously been fetched, negatively impacts the performance of the system. An FPGA is capable of performing many concurrent calculations at once and reaching very high performance, 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 DMA 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.




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 code for performing these operations is shown below.

```
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.

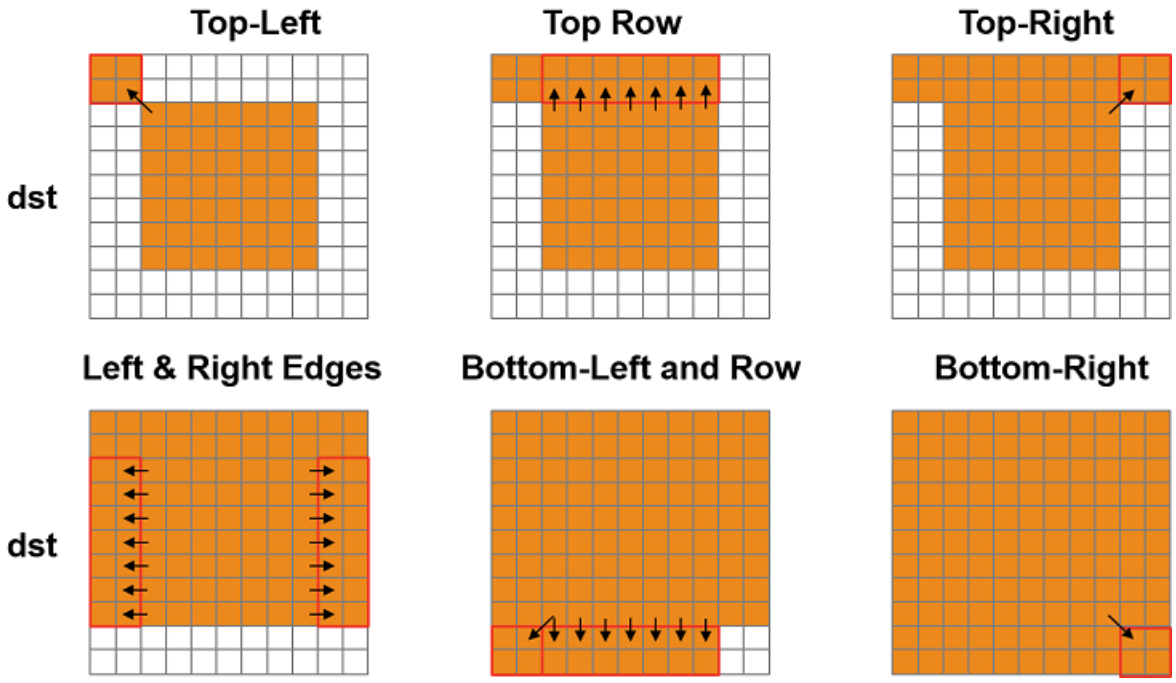
- Many clock cycles are spent to set the values in the output image `dst` to zero. In this case, approximately another two million cycles for a 1920*1080 image size.
- There are multiple accesses per pixel to re-read data stored in array `local`.
- There are multiple writes per pixel to the output array/port `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 via 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.




The border region is populated with the nearest valid value. The following code performs the operations shown in the 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];
  }
  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:

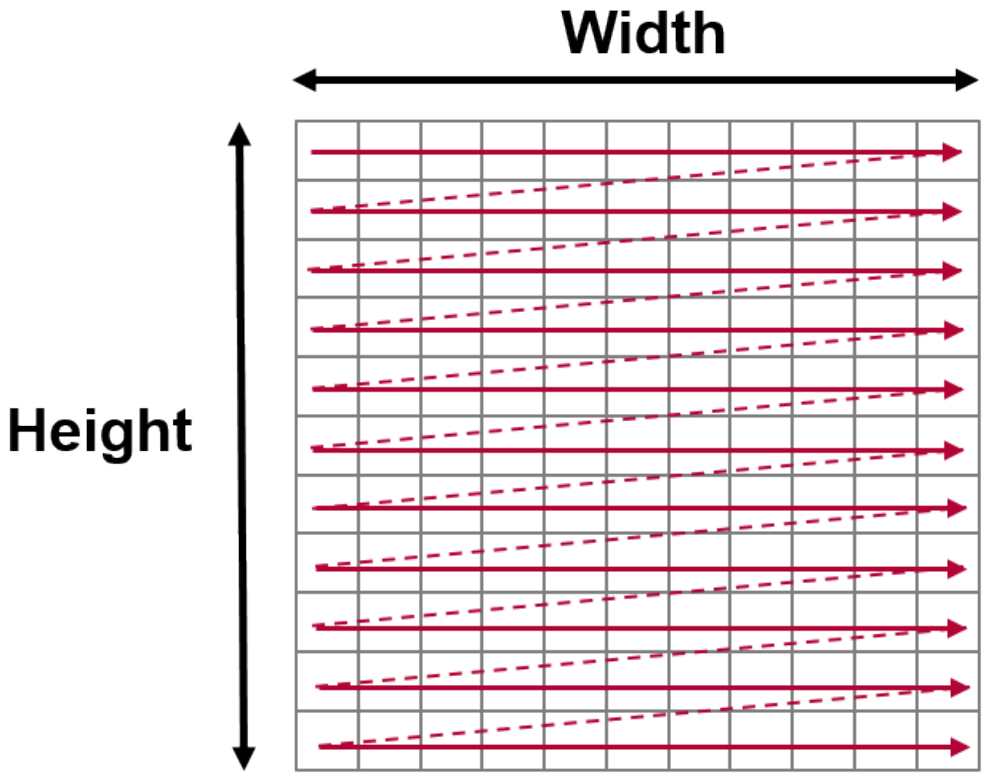
- Multiple accesses to read and then re-read data. Use local storage where possible.
- Accessing data in an arbitrary or random access manner. This requires the data to be stored locally in arrays and costs resources.
- Setting default values in arrays costs clock cycles and performance.

## 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 to:

- 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.
- Embrace 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.




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.

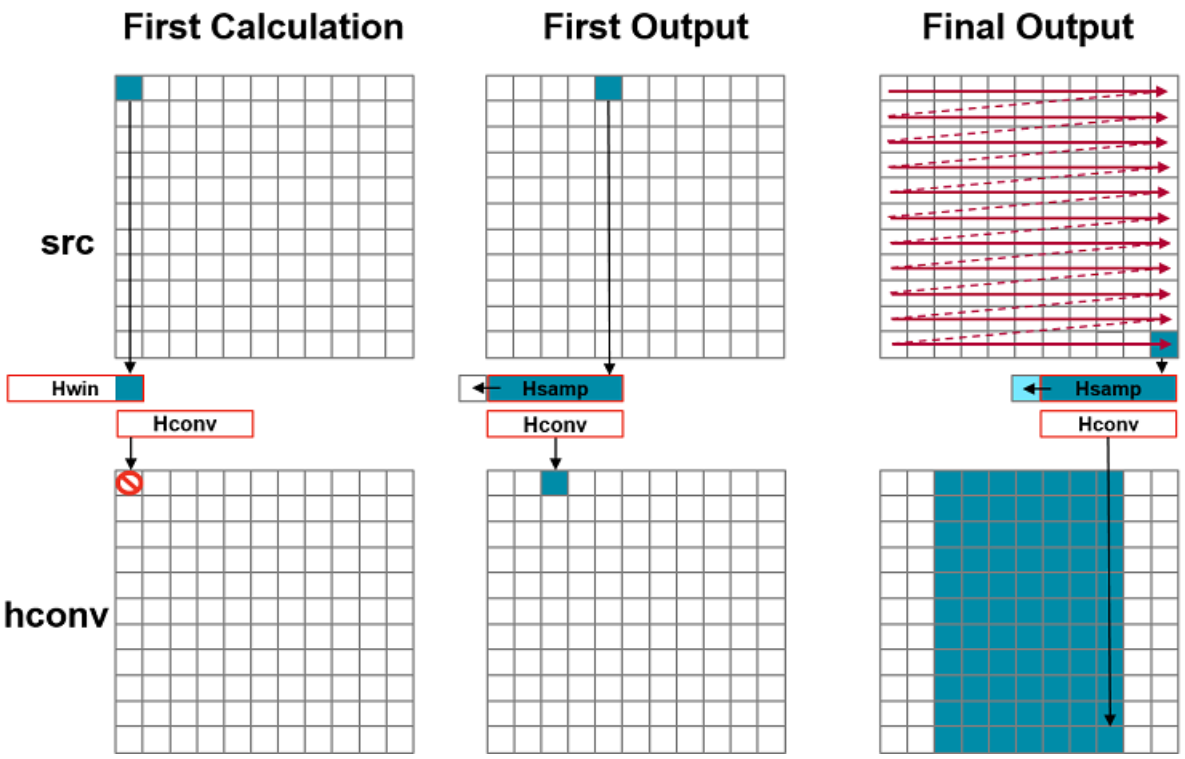
```
template<typename T, int K>
static void convolution_strm(
int width,
int height,
T src[TEST_IMG_ROWS][TEST_IMG_COLS],
T dst[TEST_IMG_ROWS][TEST_IMG_COLS],
const T *hcoeff,
const T *vcoeff)
{

T hconv_buffer[MAX_IMG_COLS*MAX_IMG_ROWS];
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*

To perform the calculation in a more efficient manner for FPGA implementation, the horizontal convolution is computed as shown in the following figure.




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.

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.

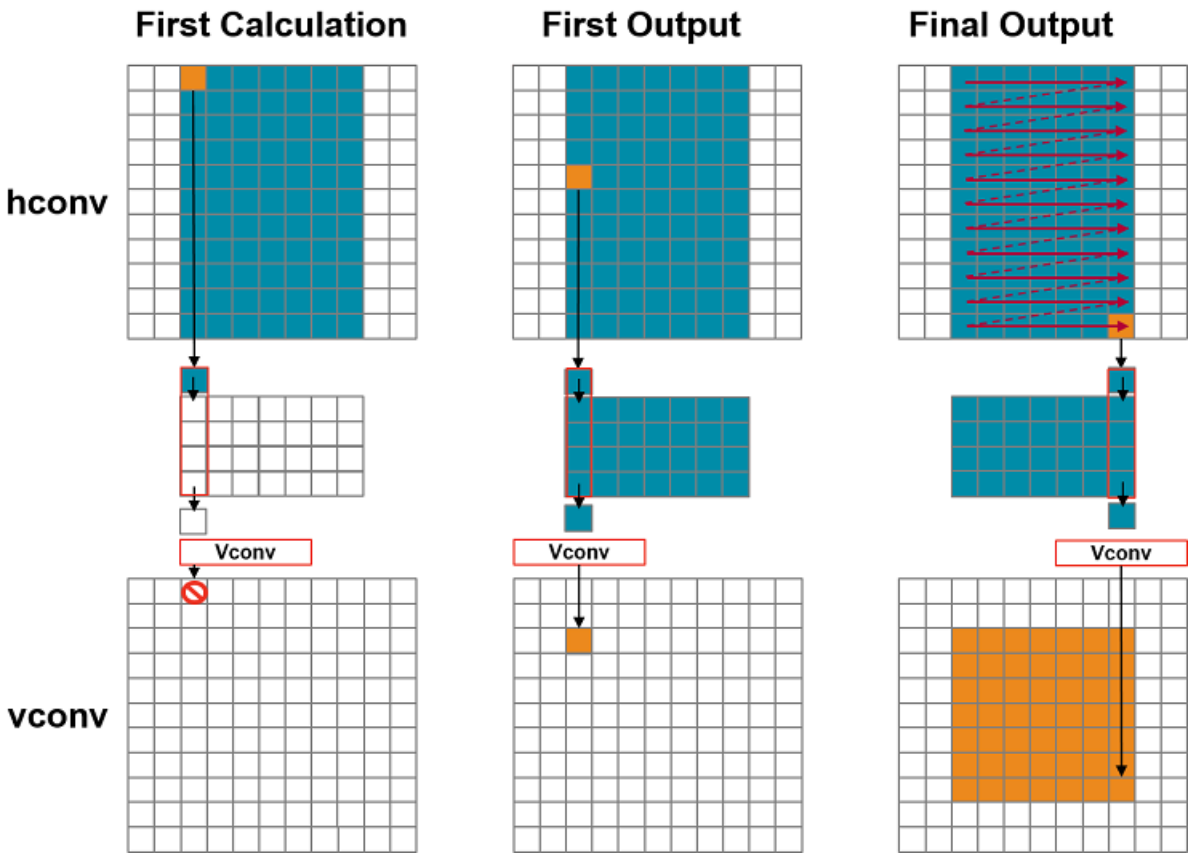
```
// Horizontal convolution
phconv=hconv_buffer; // set / reset pointer to start of buffer

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




Once 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.

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];
```

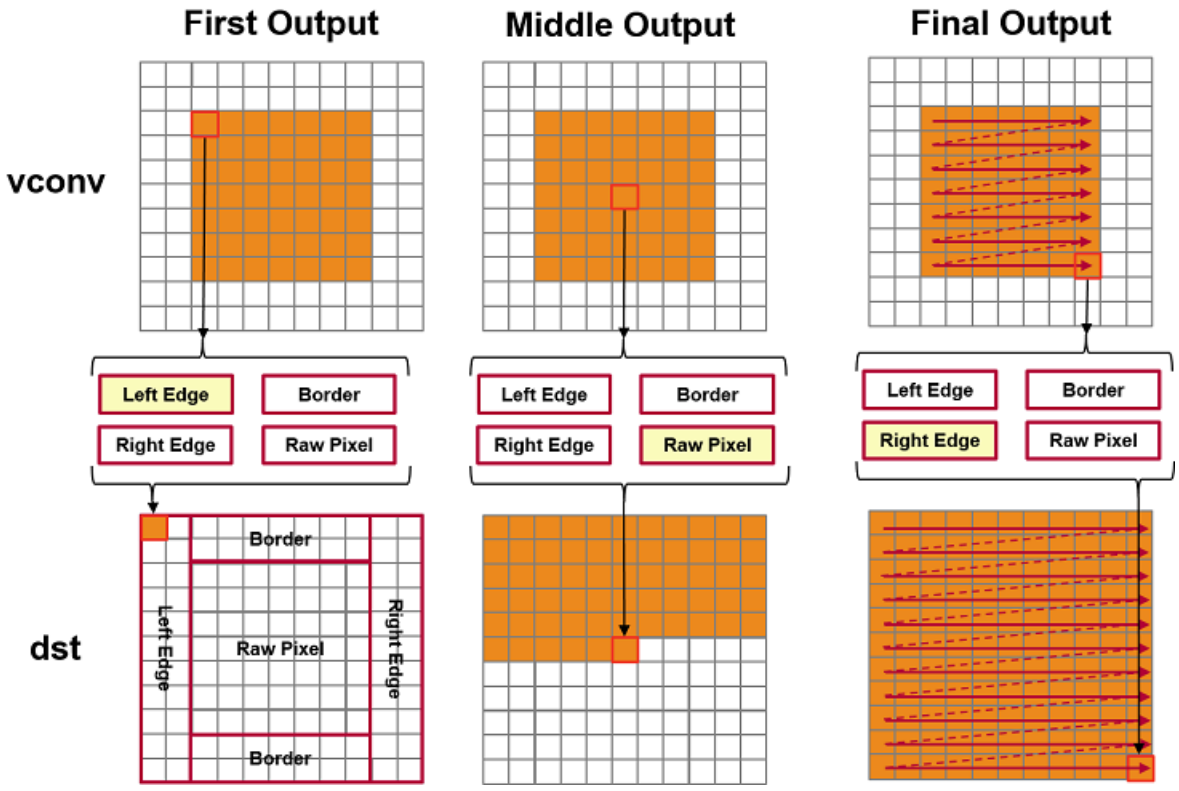
```
      if (i > 0)
        linebuf[i - 1][row] = vwin_val;
    }
    if (col >= K - 1) {
      *pvconv++ = out_val;
  }
}
```

The code above once again 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.




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 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.

XILINX
ALL PROGRAMMABLE

*Chapter 4*

# Optimizing the Hardware Function

The SDSoC environment employs heterogeneous cross-compilation, with ARM CPU-specific cross compilers for the Zynq-7000 SoC and Zynq UltraScale+ MPSoC CPUs, and Vivado HLS as a PL cross-compiler for hardware functions. This section explains the default behavior and optimization directives associated with the Vivado HLS cross-compiler.

The default behavior of Vivado HLS 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 Vivado HLS. 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 Vivado Design Suite User Guide: High-Level Synthesis (UG902).

It is highly recommended to review the methodology and obtain a global perspective of hardware function optimization before reviewing the details of specific optimization.

## Hardware Function Optimization Methodology

Hardware functions are synthesized into hardware in the PL by the Vivado HLS 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, Vivado HLS 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.

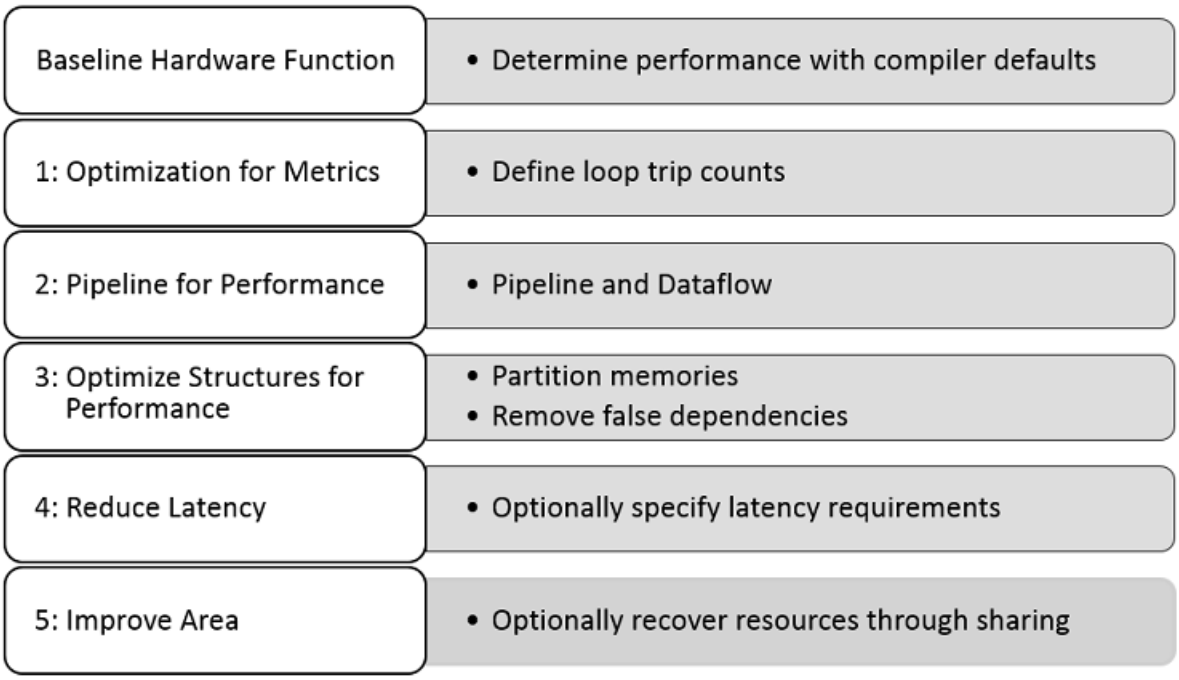
The are two flows for optimizing the hardware functions.

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

- Bottom-up flow: In this flow, the hardware functions are optimized in isolation from the system using the Vivado HLS 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. Examples of both flows are provided in Putting It All Together.

The optimization methodology for hardware functions is shown in the figure below.




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

**IMPORTANT:** *Designs will reach the optimum performance after step 3.*

Step 4 is used to minimize, or specifically control, the latency through the design and is only required for applications where this is of concern. Step 5 explains how to reduce the resources required for hardware implementation and is typically only applied 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.

# Baseline The Hardware Functions

Before seeking to 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. This is achieved by selecting the functions to implement hardware and building the project.

After the project has been built, a report is available in the reports section of the IDE (and provided at `<project name>/<build_config>/_sds/vhls/<hw_function>/solution/syn/report/<hw_function>.rpt`). This report details the performance estimates and utilization estimates.

The key factors in the performance estimates are the timing, interval, and latency in that order.

- The timing summary shows the target and estimated clock frequency. If the estimated clock frequency is greater than the target, *the hardware will not function at this clock frequency*. The clock frequency should be reduced by using the Data Motion Network Clock Frequency option in the Project Settings. 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 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 (e.g., 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 PL with typically little benefit. The hardware function will often be ideal as 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. It is, however, 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.

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 on the device. 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.

# Optimization for Metrics

The following table shows the first directive you should think about adding to your design.

**Table 1: 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.

# Pipeline 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 the highest level of performance. The following table shows the directives you can use for pipelining.

**Table 2: Optimization Strategy Step 1: 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. |

| Directives and Configurations | Description |
|---|---|
| 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 wih 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, refer to 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 Config commands are used to change the optimization default settings and are only available from within Vivado HLS when using a bottom-up flow. Refer to Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more details.*

## 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.

Fundamentally, 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.

An example outline of frame-based C code is shown below.

```
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.

Reviewing the advantages and disadvantages of placing the PIPELINE directive at each of these locations helps explain the best location to place the pipeline directive for 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. It is unlikely this is 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
- 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 large highly parallel hardware design unable to process all the data in parallel and would lead to 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. However, the only way to make use of the additional hardware would be to 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 result in 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 will cause Loop3 to be completely unrolled but to 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 complier 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 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 Optimize 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.

# Optimize 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 logic. 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.

```
#include "bottleneck.h"
dout_t bottleneck(din_t mem[N]) {
  dout_t sum=0;
  int i;
  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];
  }
  return sum;
}
```

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 fabric. 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, etc., from the original array and the second partition contains elements 1, 3, 5, etc. With a dual-port block RAM, this allows elements 0, 1, 2, and 3 to be read in a single clock cycle.

```
#include "bottleneck.h"
dout_t bottleneck(din_t mem[N]) {
#pragma HLS ARRAY_PARTITION variable=mem cyclic factor=2 dim=1
  dout_t sum=0;
  int i;
  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];
  }
  return sum;
}
```

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 3: 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.

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 creating 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 Config commands are used to change the optimization default settings and are only available from within Vivado HLS when using a bottom-up flow. Refer to Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more details.

If optimization directives cannot be used to improve the initiation interval, it might require changes to the code. Examples of this are discussed in Vivado Design Suite User Guide: High-Level Synthesis (UG902).

# 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 4: Optimization Strategy Step 4: Reduce Latency**

| Directive | Description |
|---|---|
| LATENCY | Allows a minimum and maximum latency constraint to be specified. |
| 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 of importance 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 II), the next step might be to reduce the area while maintaining the same performance. This step can be optimal because there is nothing to be gained by 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 in fact 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.

For most applications, the depth can be specified as 1, resulting in the memory channel being implemented as a simple register. If, however, 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 symptom will be 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 to consider when attempting to minimize the resources used to implement the design.

**Table 5: 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 Config commands are used to change the optimization default settings and are only available from within Vivado HLS when using a bottom-up flow. Refer to Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more details.*

# 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 (e.g., Opt_ver1 and UnOpt_ver) might be created in the Project Settings window using the **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 RESOUCES 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, simply 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).

Chapter 5

# Putting It All Together

This chapter describes a some real-world examples and shows how they 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.
- For the bottom-up flow a stereo vision block matching algorithm is used.

For both flows, the final optimized design is available from within the SDSoC environment. To access the examples, create a new SDx project using the IDE, select either the zc706 or zcu102 hardware platforms, and from the New Project Templates window select **File IO Dense Optical Flow** for the top-down example or **File IO Stereo Block Matching**.

This chapter explains what optimization directives were applied to the fully optimized examples and why.

## 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 code to implement this algorithm is shown below, 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);
```

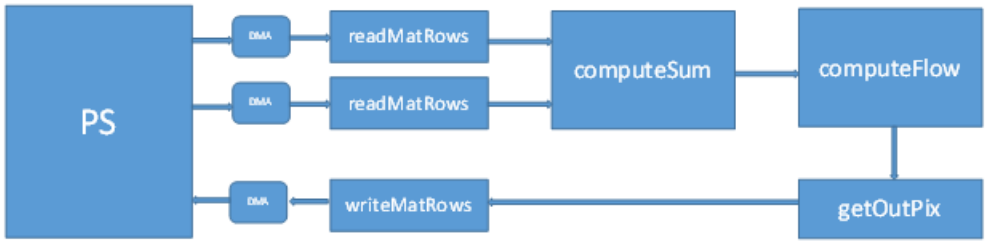
```
        read_yuv_frame(inY1, WIDTH, WIDTH, HEIGHT, f);
        printf("read 1st %dx%d frame\n", WIDTH, HEIGHT);
        read_yuv_frame(inY2, WIDTH, WIDTH, HEIGHT, f);
        printf("read 2nd %dx%d frame\n", WIDTH, HEIGHT);

        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 is typical for a top-down design flow using standard C/C++ data types. Function `fpa_optflow` is shown below and contains the sub-function `readMatRows`, `computeSum`, `computeFlow`, `getOutPix`, and `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);
  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 thus would 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.




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 very 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. A few optimization directives can be used to ensure the system meets the target performance.

## Optical Flow Memory Access Optimization

As noted earlier in the methodology, the first task is to optimize the transfer of data. In this case, because the system is to process steaming video where each sample is processed in consecutive order, the memory transfer optimization is to ensure the SDSoC environment understands that all accesses are sequential in nature.

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 to be 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:

- 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 simply specified as sequential, allowing the most efficient hardware to be used to connect the functions in the 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 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 function getOutPix. 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 happens automatically for small functions, but 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 function getOutPix is the loop that processes data at the level of each pixel and is optimized with the PIPELINE directive to ensure it processes 1 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 will by default be implemented in a block RAM, which has a maximum of two ports. As shown in the following code summary:

- img1Win is used in a for-loop that is pipelined to process 1 sample per clock cycle.
- img1Win is read from, 8 + (KMEDP1-1) + (KMEDP1-1) times within the for-loop.
- img1Win is 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 array csIxix has a DEPENDENCE directive applied to it. The array is read from and then written to via different indices, as shown below, 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
   ...
}
```

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 there exists any condition 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 shows an example of the 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
```

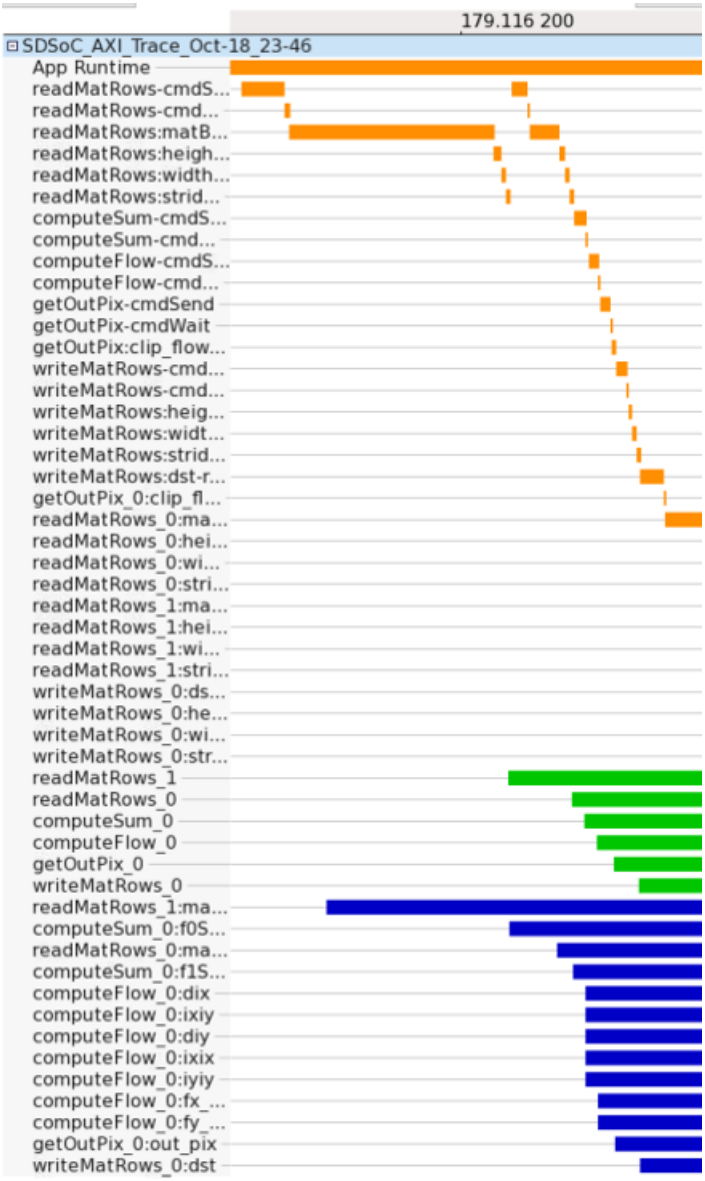
In sequential C code where each iteration completes before the next starts, it is clear what order the reads and writes occur. However, in a concurrent hardware pipeline the accesses can overlap and occur in different orders. As can be seen clearly above, it is possible for the read from index 8, as 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 and the DEPENDENCE directive is used with the setting false to tell the compiler, "I, the user state that it is okay to ignore this," thus removing the write-after-read anti-dependence and allowing the compiler to create pipelined hardware which performs with II = 1.

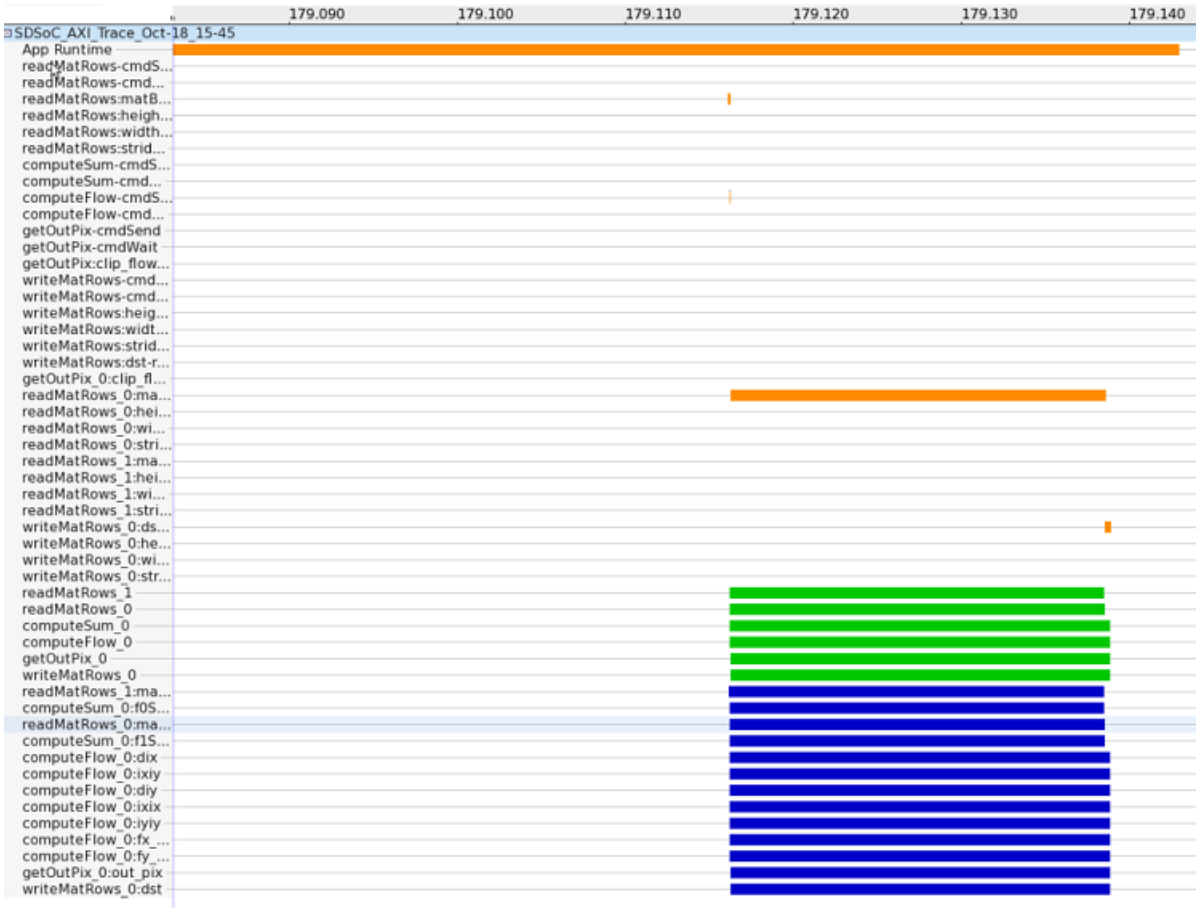
The DEPENDENCE directive is typically used to inform the compiler of algorithm behaviors and conditions external to the function of which is it unaware from static analysis of the code. If a DEPENDENCE directive is set incorrectly, the issue will be discovered in hardware emulation if the results from 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 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 before they start. Each hardware function begins to process data as soon as data becomes available.




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



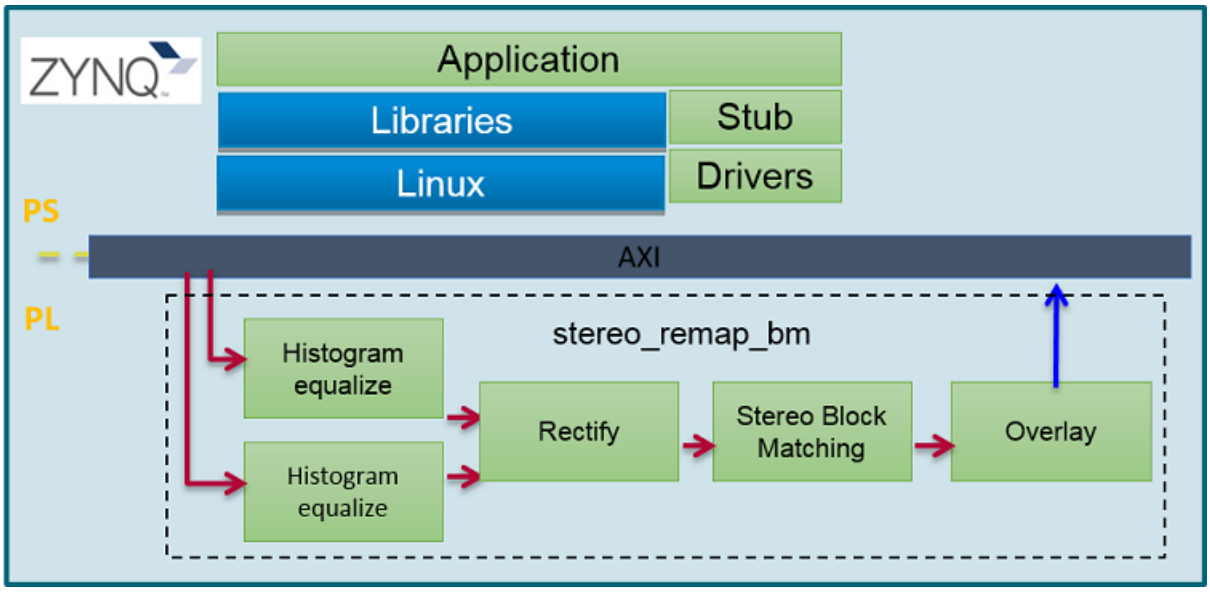

To get the duration time, hover on top of one of the lanes to obtain a pop-up that shows the duration of the accelerator run time. The execution time is just under 15.5 ms, and this meets the targeted 16.8 ms necessary to achieve 60 frames per second.

# Bottom-Up: Stereo Vision Algorithm

The stereo vision algorithm uses images from two cameras that are displaced horizontally from each other. This provides two different views of the scene from different vantage points, similar to human vision. The relative depth information from the scene can be obtained by comparing 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 Vivado HLS and then integrate the pre-optimized hardware function with software in the SDSoC environment. This flow allows hardware designers already knowledgeable with HLS to build and optimize the entire hardware function using advanced HLS features first 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 Vivado HLS 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 hardware function stereo_remap_bm to be incorporated into the SDSoC environment.




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 code for the existing hardware function `stereo_remap_bm` is shown below with the optimization pragmas highlighted. Before reviewing the optimization directives, there are a few things to note about the function.

- The hardware function contains sub-functions readLRinput, writeDispOut and writeDispOut which have also been optimized.
- The hardware function also uses pre-optimized functions, prefixed with the namespace hls, from the Vivado HLS video library `hls_video.h`. Tthese sub-functions use their own data type 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 Hardware Function Optimization Methodology, the primary optimization directives used are the PIPELINE and DATAFLOW directives. In addition, the LOOP_TRIPCOUNT directive is used.

In keeping with 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, the exact value of which is unknown at compile time. The estimated tripcount, or loop iteration count, allows the reports generated by Vivado HLS 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 function `stereo_remap_bm` is composed of the optimized sub-functions and a number of functions from the Vivado HLS video library (`hls_video.h`). Details of the library functions provided by Vivado HLS are provided in Vivado Design Suite User Guide: High-Level Synthesis (UG902). The functions provided in the Vivado HLS 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 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 a Vivado HLS 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 which is based on the Vivado HLS 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 function `stereo_remap_bm` 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 Vivado HLS hardware data types into the SDSoC environment, ensure the functions have been fully verified through C compilation and hardware simulation within the Vivado HLS environment.*

**IMPORTANT:** *The `hls::stream` data type is designed for use within Vivado HLS, 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 Vivado HLS 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.

## Stereo Vision Memory Access Optimization

After importing the pre-optimized hardware function into a project in the SDSoC environment, the first task is to remove any interface optimizations. The interface between the PS and the hardware function is automatically managed and optimized based on the data types of the hardware function and the data access. Refer to 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 Vivado HLS 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);
```

Because the data is transferred in a sequential manner, the first thing to ensure is that the access pattern is defined as SEQUENTIAL for both arguments. The next optimization is to ensure the data transfer is not interrupted by a scatter gather DMA operation specifying the memory_attribute to be PHYSICAL_CONTIGUOUS|NON_CACHEABLE. This also requires that the memory be 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

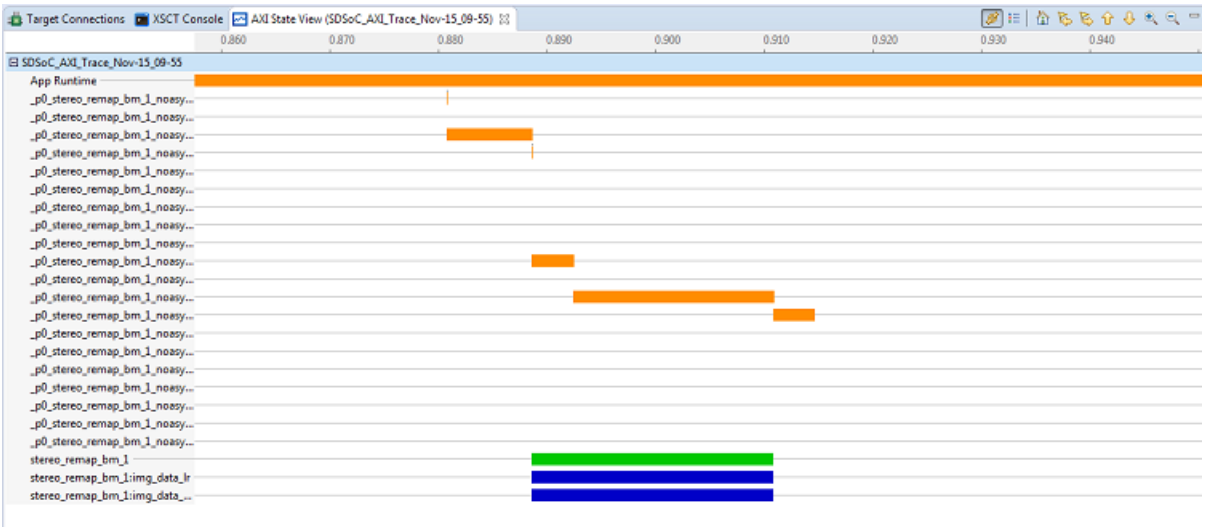
```
#pragma SDS data access_pattern(img_data_lr:SEQUENTIAL)
#pragma SDS data
mem_attribute(img_data_lr:PHYSICAL_CONTIGUOUS|NON_CACHEABLE)
#pragma SDS data copy(img_data_lr[0:stride*height])
#pragma SDS data access_pattern(img_data_disp:SEQUENTIAL)
#pragma SDS data
mem_attribute(img_data_disp:PHYSICAL_CONTIGUOUS|NON_CACHEABLE)
#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 Vivado HLS is wrapped, as in this example, to ensure Vivado HLS 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 figure below shows the complete view of the event traces, and all hardware functions and data transfers executing in parallel for the highest performing system.




To get the duration time, hover on top of one of the lanes to obtain a pop-up that shows the duration of the accelerator run time. The execution time is <to be done> ms and this meets the targeted 16.8 ms necessary to achieve 60 frames per second for live video.

XILINX
ALL PROGRAMMABLE

*Appendix A*

# Additional Resources and Legal Notices

## Xilinx Resources

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

## Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips

## References

These documents provide supplemental material useful with this guide:

1. *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDSoC Environment User Guide* (UG1027)
3. *SDSoC Environment Optimization Guide* (UG1235)
4. *SDSoC Environment Tutorial: Introduction* (UG1028)
5. *SDSoC Environment Platform Development Guide* (UG1146)
6. SDSoC Development Environment web page
7. *UltraFast Embedded Design Methodology Guide* (UG1046)
8. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* (UG850)
9. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
10. *PetaLinux Tools Documentation: Workflow Tutorial* (UG1156)
11. Vivado® Design Suite Documentation
12. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

# 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 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 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.