Chapter 2

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

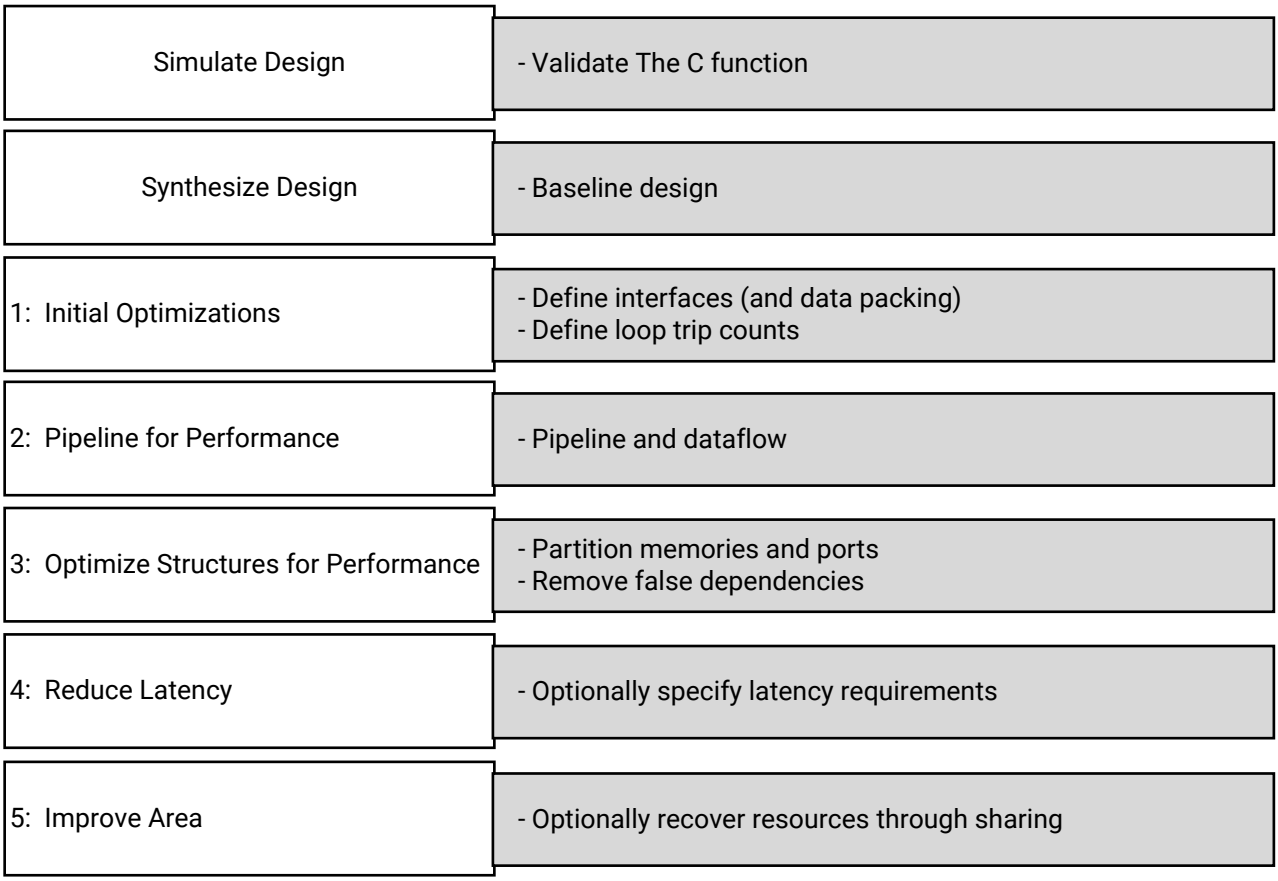
Hardware functions are synthesized into hardware in the Programmable Logic (PL) by the Vivado HLS compiler. This compiler automatically translates C/C++ code into an FPGA hardware implementation, and as with all compilers, does so using compiler defaults. In addition to the compiler defaults, Vivado 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.

There are two flows for optimizing the hardware functions.

- Top-down flow: In this flow, program decomposition into hardware functions proceeds top-down within the SDSoC environment, letting the system 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 the *SDAccel Environment Profiling and Optimization Guide* (UG1207) .

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 4: **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 5: **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.
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 not pipelined, it takes a minimum of N clock cycles to complete. Only then can the function read the next x input value.

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

In this type of coding style, the loops are typically operating on arrays and performing a shift register or line buffer functions. It is not uncommon to partition these arrays into individual elements as discussed in Chapter 3: 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. This example code shows a loop being used inside a function.

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

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

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

The issue in this example is that arrays are implemented using the efficient block RAM resources in the PL 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. Because the partitioning ensures there are now two dual-port block RAMs (with a total of four data ports), this allows elements 0, 1, 2, and 3 to be read in a single clock cycle.

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

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

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

Table 6: **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.

Table 6: **Optimization Strategy Step 3: Optimize Structures for Performance** *(cont'd)*

Directives and Configurations	Description
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 7: **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 8: **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 4

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:

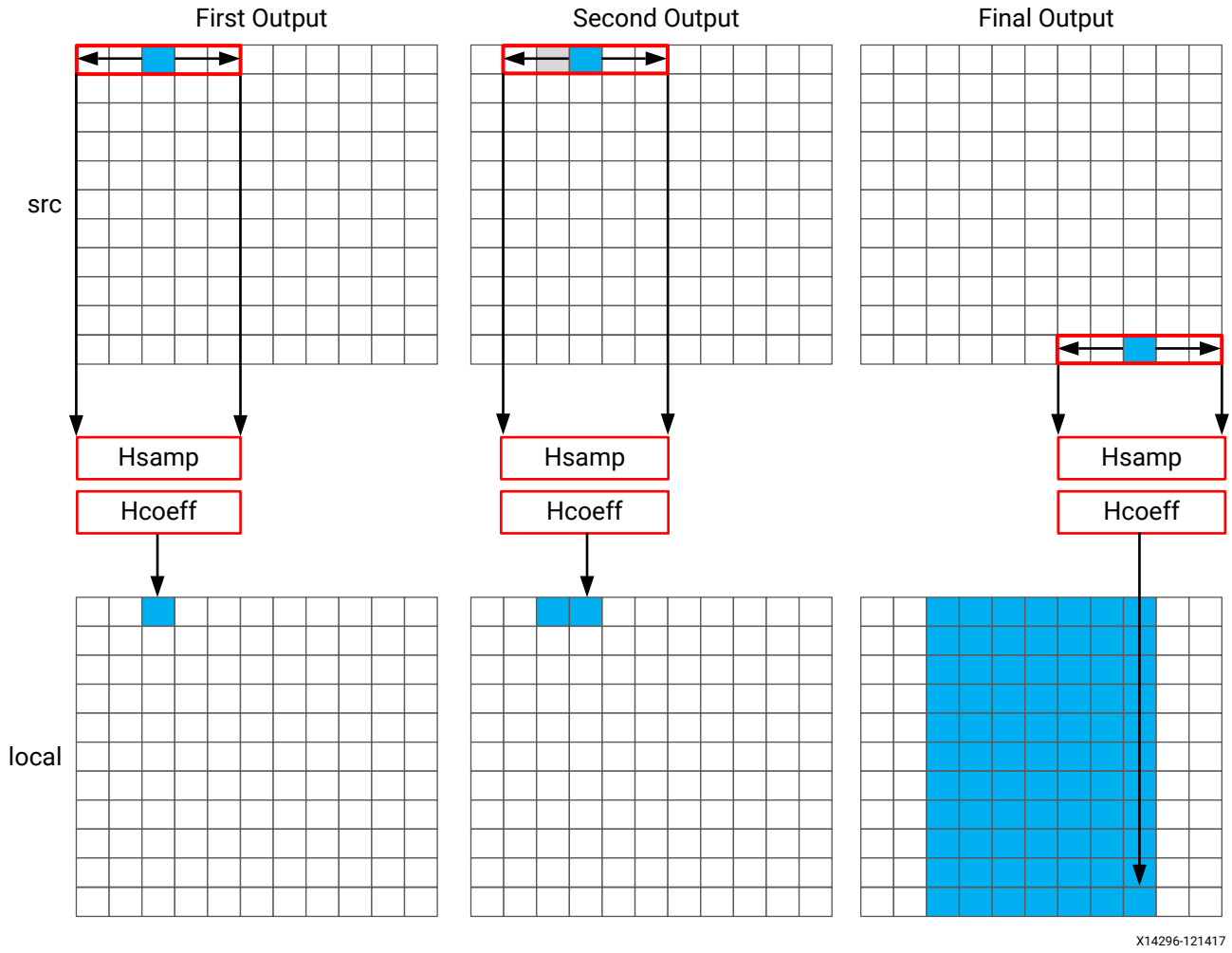
- A horizontal convolution.
- Followed by a vertical convolution.

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

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

// 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.

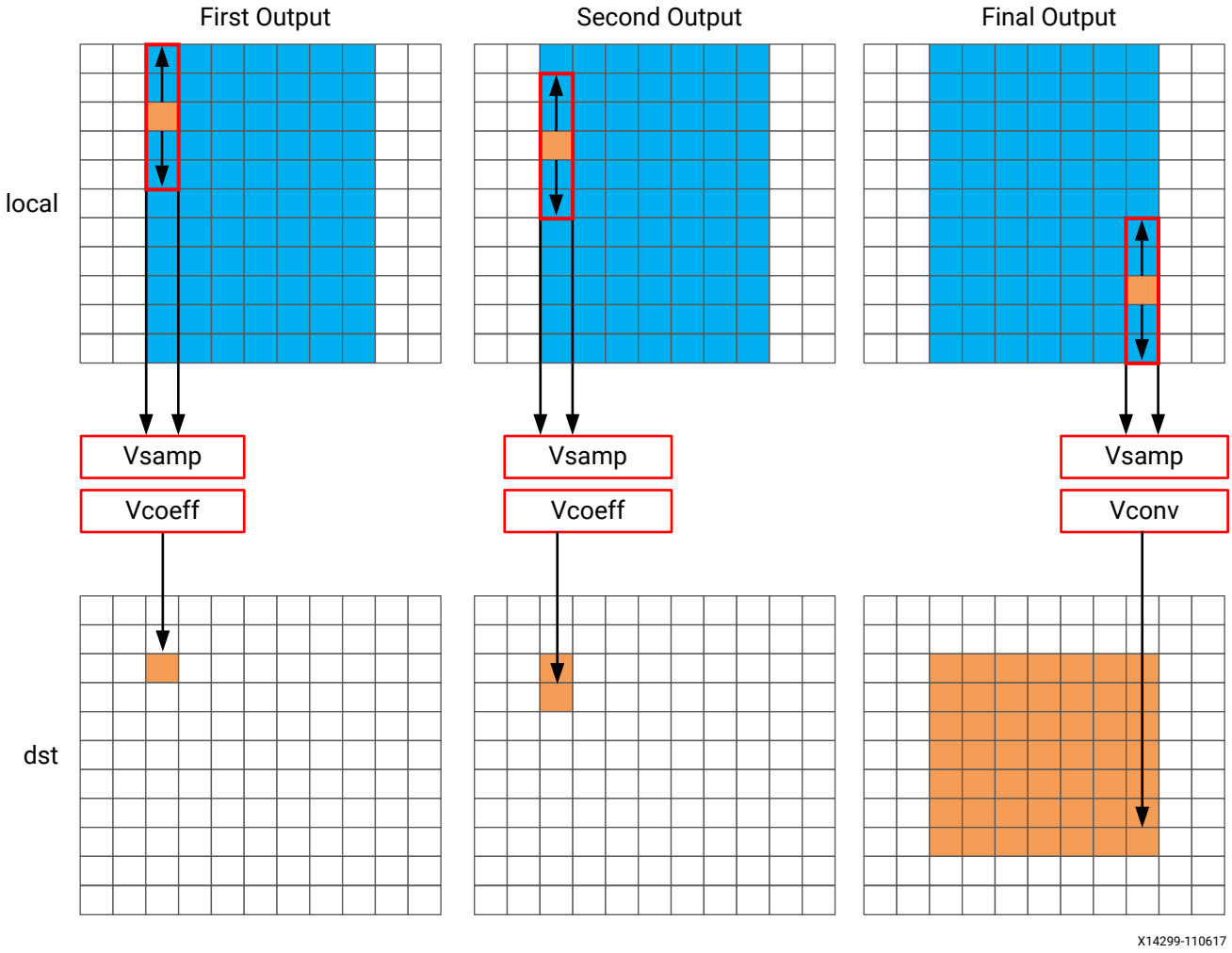
One of the keys to a high-performance FPGA is to minimize the access to and from the PS. Each access for data, which has previously been fetched, negatively impacts the performance of the system. An FPGA is capable of performing many concurrent calculations at once and reaching very high performance,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

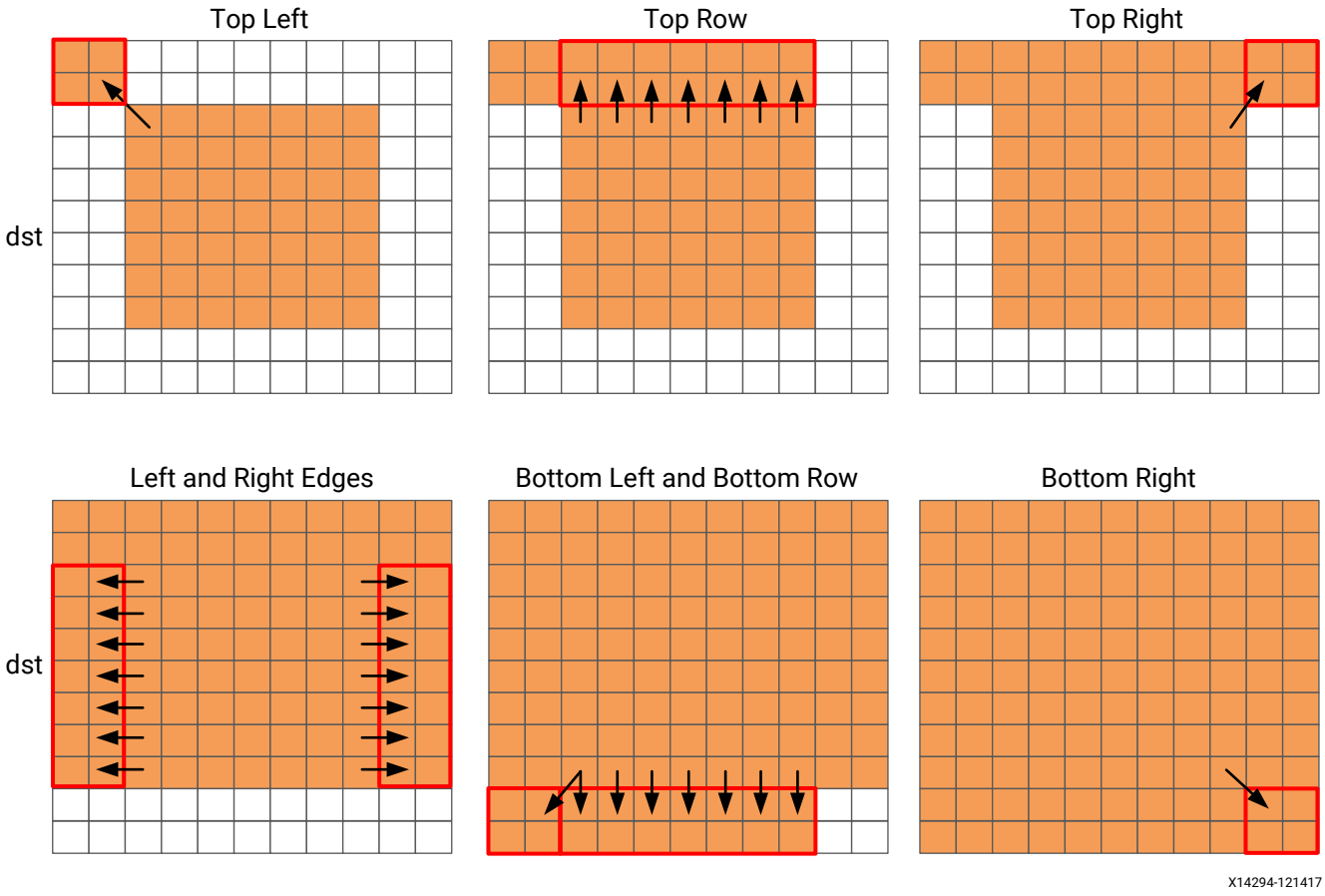
- Many clock cycles are spent to set the values in the output image `dst` to zero. In this case, approximately another two million cycles for a 1920*1080 image size.
- There are multiple accesses per pixel to re-read data stored in array `local`.
- There are multiple writes per pixel to the output array/port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

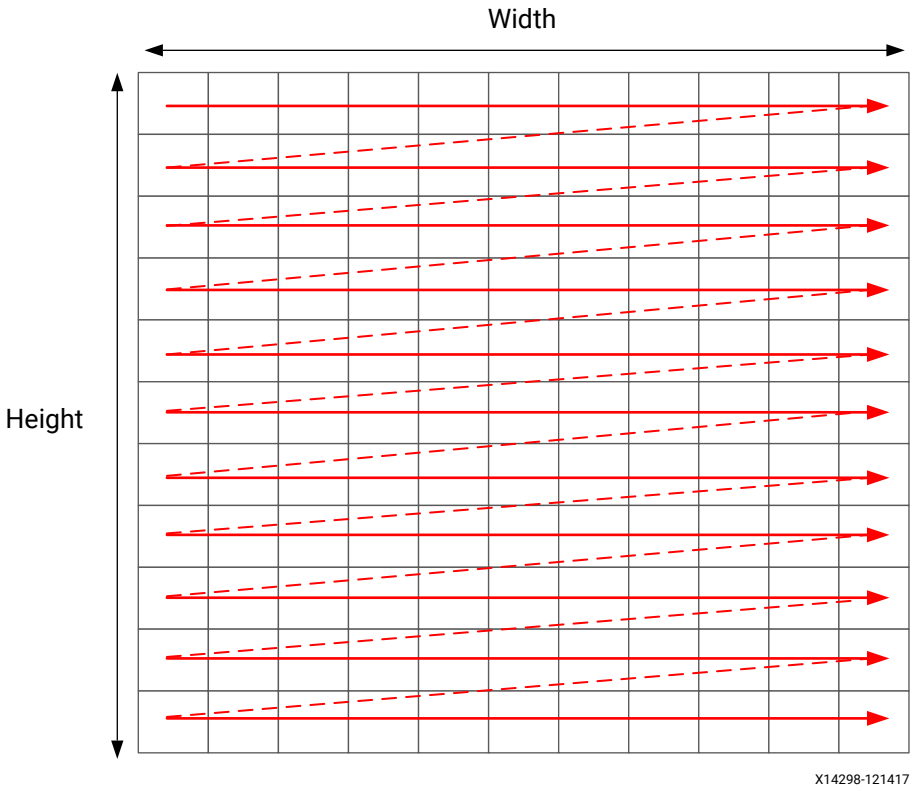
- Multiple accesses to read and then re-read data. Use local storage where possible.
- Accessing data in an arbitrary or random access manner. This requires the data to be stored locally in arrays and costs resources.
- Setting default values in arrays costs clock cycles and performance.

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.