Vivado HLS Optimization Methodology Guide

UG1270 (v2018.1) April 4, 2018

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
04/04/2018 Version 2018.1	
Entire document	Minor editorial updates.

Table of Contents

Chapter 1

Introduction

This guide provides details on how to perform optimizations using Vivado HLS. The optimization process consists of directives which specify which optimizations are performed and a methodology which shows how optimizations may be applied in a deterministic and efficient manner.

HLS Pragmas

Optimizations in Vivado HLS

In both SDAccel and SDSoC projects, the hardware kernel must be synthesized from the OpenCL, C, or C++ language, into RTL that can be implemented into the programmable logic of a Xilinx device. Vivado HLS synthesizes the RTL from the OpenCL, C, and C++ language descriptions.

Vivado HLS is intended to work with your SDAccel or SDSoC Development Environment project without interaction. However, Vivado HLS also provides pragmas that can be used to optimize the design: reduce latency, improve throughput performance, and reduce area and device resource utilization of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

IMPORTANT!:
Although the SDSoC environment supports the use of HLS pragmas, it does not support pragmas applied to any argument of the function interface (interface, array partition, or data_pack pragmas). Refer to "Optimizing the Hardware Function" in the SDSoC Environment Optimization Guide (UG1235) for more information.

The Vivado HLS pragmas include the optimization types specified below:

Table 1: **Vivado HLS Pragmas by Type**

Type	Attributes
Kernel Optimization	• pragma HLS allocation • pragma HLS clock • pragma HLS expression_balance • pragma HLS latency • pragma HLS reset • pragma HLS resource • pragma HLS top
Function Inlining	• pragma HLS inline • pragma HLS function_instantiate
Interface Synthesis	• pragma HLS interface • pragma HLS protocol
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurrence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_map • pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS data_pack

OpenCL Attributes

Optimizations in OpenCL

This section describes OpenCL attributes that can be added to source code to assist system optimization by the SDAccel compiler, `xocc`, the SDSoC system compilers, `sdscc` and `sds++`, and Vivado HLS synthesis.

SDx provides OpenCL attributes to optimize your code for data movement and kernel performance. The goal of data movement optimization is to maximize the system level data throughput by maximizing interface bandwidth utilization and DDR bandwidth utilization. The goal of kernel computation optimization is to create processing logic that can consume all the data as soon as they arrive at kernel interfaces. This is generally achieved by expanding the processing code to match the data path with techniques such as function inlining and pipelining, loop unrolling, array partitioning, dataflowing, etc.

The OpenCL attributes include the types specified below:

Table 2: **OpenCL __attributes__ by Type**

Type	Attributes
Kernel Size	• reqd_work_group_size • vec_type_hint • work_group_size_hint • xcl_max_work_group_size • xcl_zero_global_work_offset
Function Inlining	• always_inline
Task-level Pipeline	• xcl_dataflow • xcl_reqd_pipe_depth
Pipeline	• xcl_pipeline_loop • xcl_pipeline_workitems
Loop Unrolling	• opencl_unroll_hint

Table 2: **OpenCL __attributes__ by Type** *(cont'd)*

Type	Attributes
Array Optimization	• xcl_array_partition • xcl_array_reshape **Note:** Array variables only accept a single array optimization attribute.

TIP: *The SDAccel and SDSoC compilers also support many of the standard attributes supported by* `gcc`, *such as* `always_inline`, `noinline`, `unroll`, *and* `nounroll`.

Directives

To view details on the attributes in the following table see the Command Reference section in *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Table 3: **Vivado HLS Pragmas by Type**

Type	Attributes
Kernel Optimization	• set_directive_allocation • set_directive_clock • set_directive_expression_balance • set_directive_latency • set_directive_reset • set_directive_resource • set_directive_top
Function Inlining	• set_directive_inline • set_directive_function_instantiate
Interface Synthesis	• set_directive_interface • set_directive_protocol
Task-level Pipeline	• set_directive_dataflow • set_directive_stream

Table 3: **Vivado HLS Pragmas by Type** *(cont'd)*

Type	Attributes
Pipeline	• set_directive_pipeline • set_directive_occurrence
Loop Unrolling	• set_directive_unroll • set_directive_dependence
Loop Optimization	• set_directive_loop_flatten • set_directive_loop_merge • set_directive_loop_tripcount
Array Optimization	• set_directive_array_map • set_directive_array_partition • set_directive_array_reshape
Structure Packing	• set_directive_data_pack

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

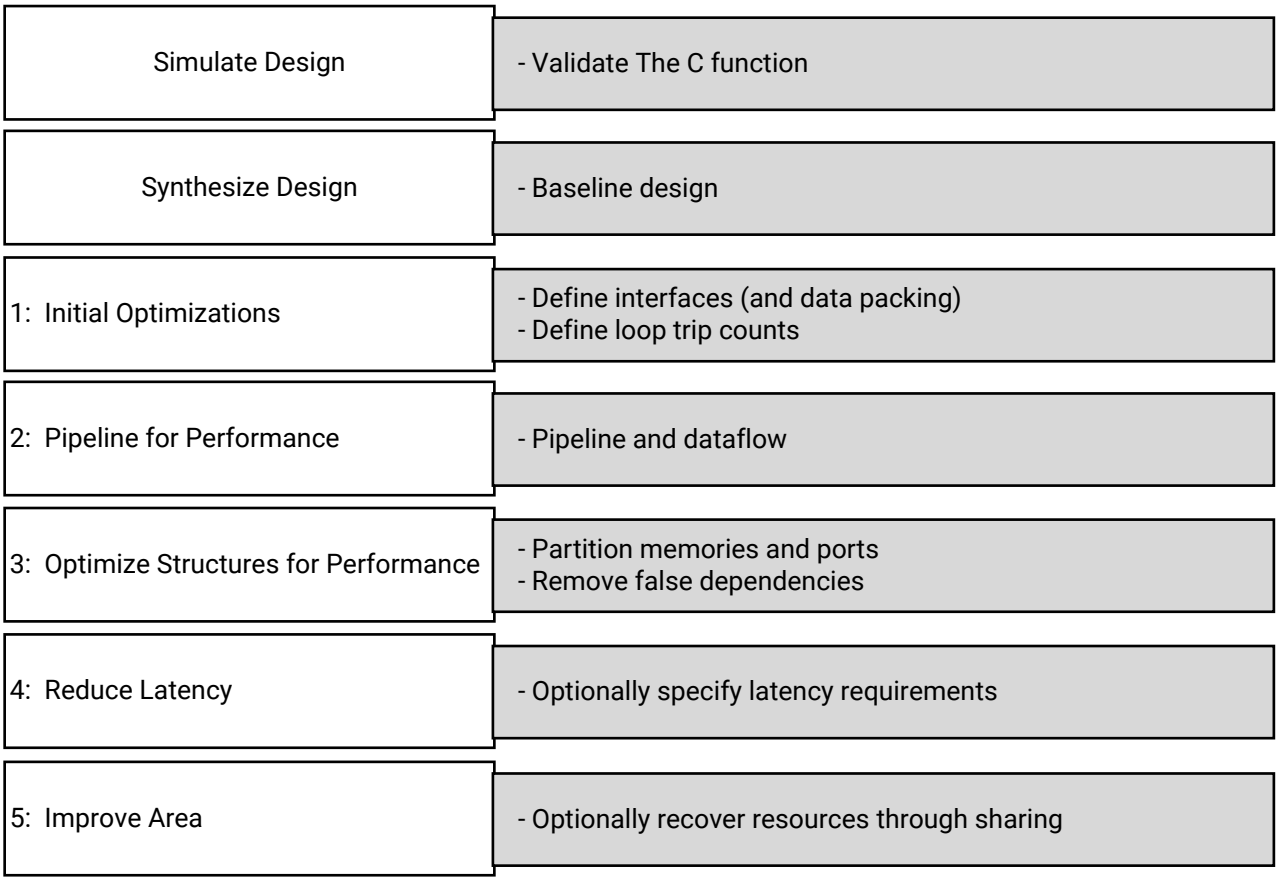
Hardware functions are synthesized into hardware in the Programmable Logic (PL) by the Vivado HLS compiler. This compiler automatically translates C/C++ code into an FPGA hardware implementation, and as with all compilers, does so using compiler defaults. In addition to the compiler defaults, Vivado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

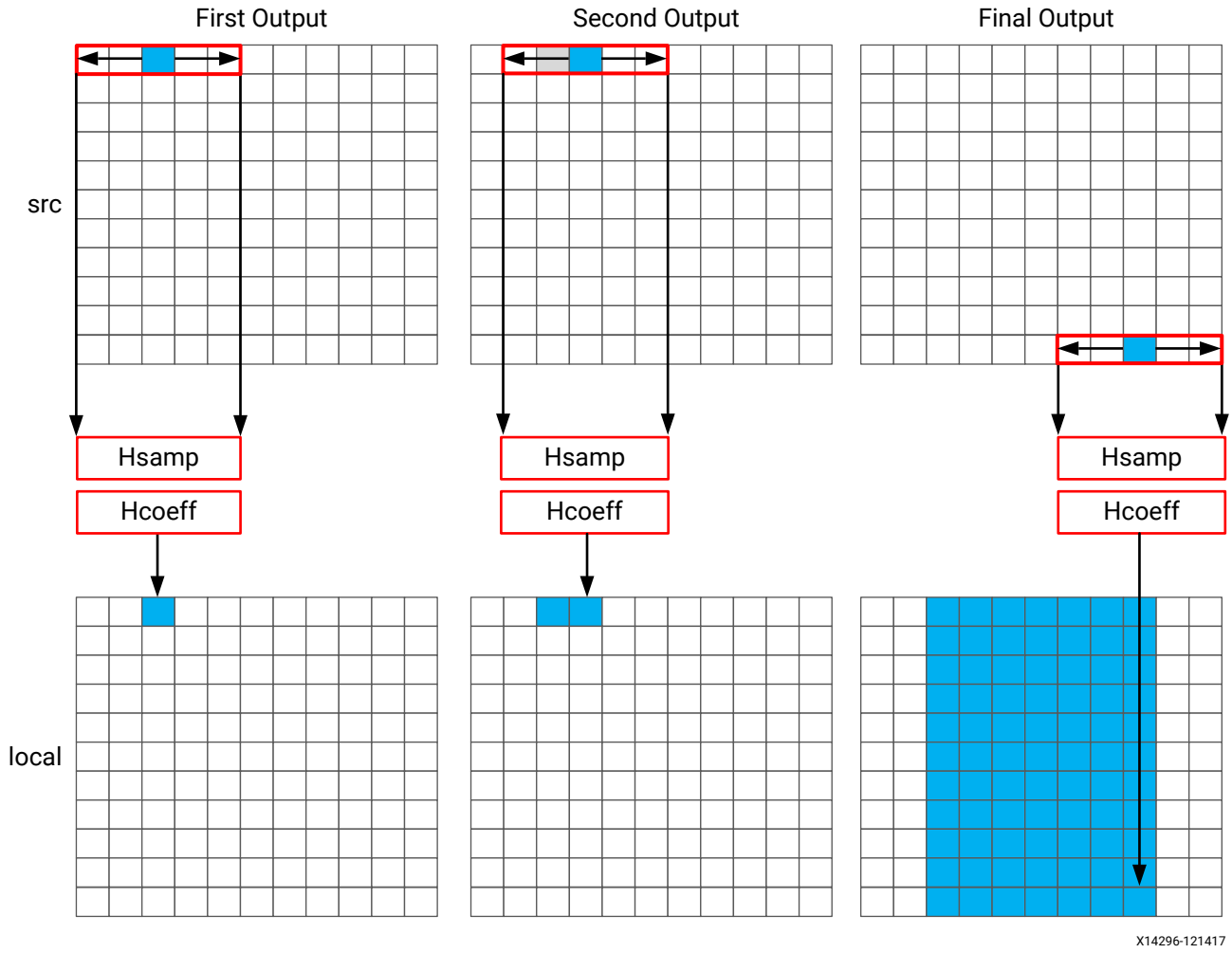
- A horizontal convolution.
- Followed by a vertical convolution.

- 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.

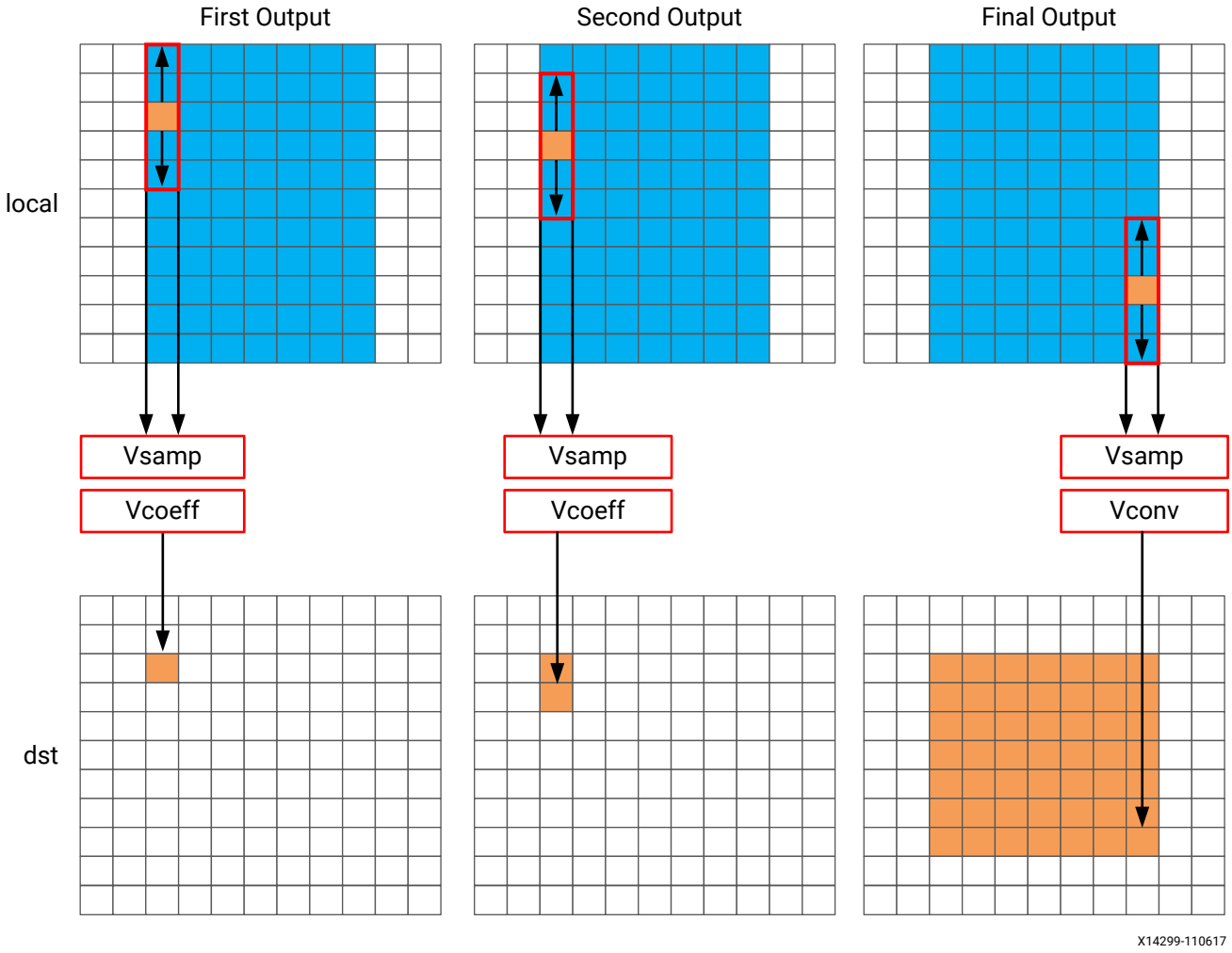
One of the keys to a high-performance FPGA is to minimize the access to and from the PS. Each access for data, which has previously been fetched, negatively impacts the performance of the system. An FPGA is capable of performing many concurrent calculations at once and reaching very high performance,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

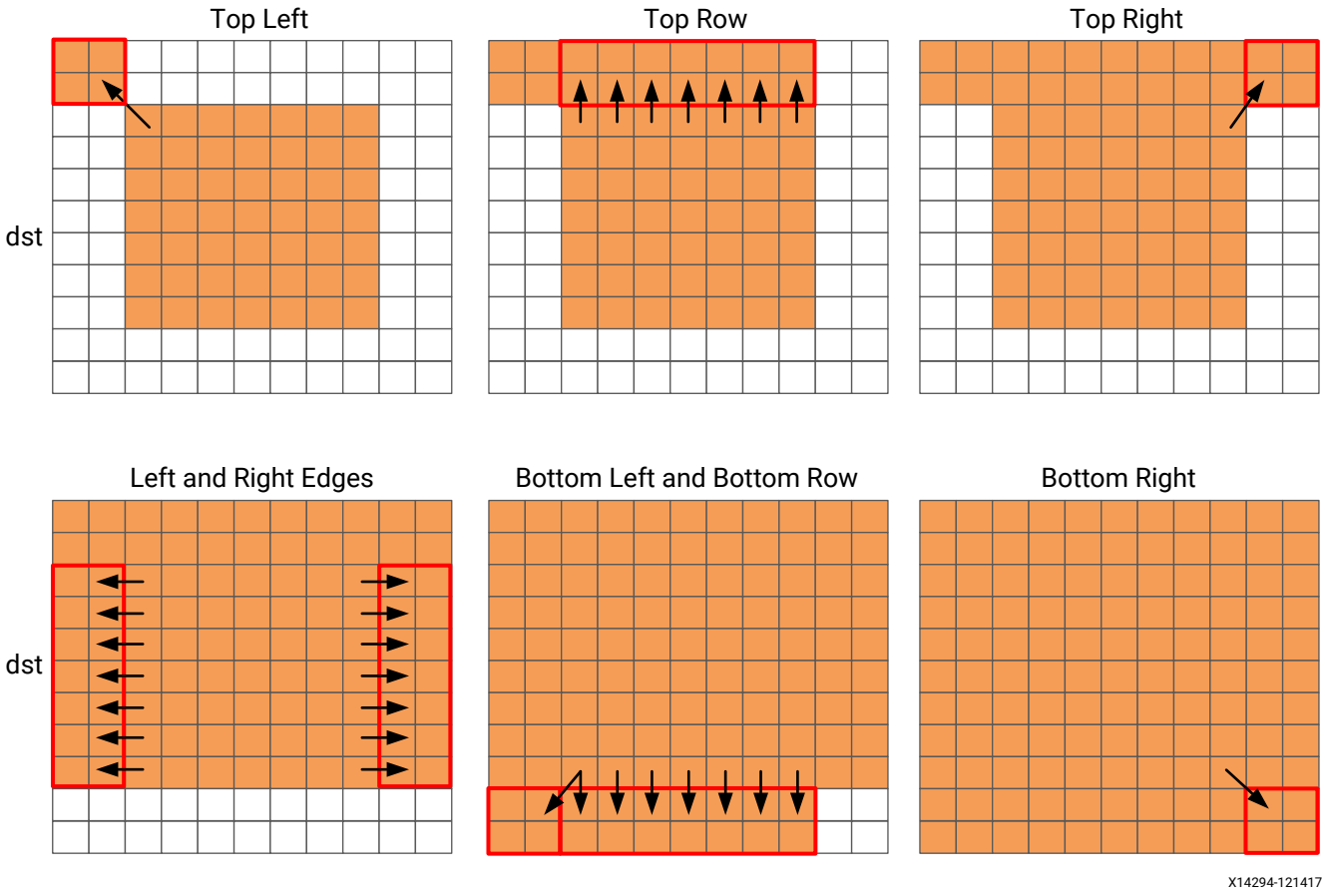
- Many clock cycles are spent to set the values in the output image `dst` to zero. In this case, approximately another two million cycles for a 1920*1080 image size.
- There are multiple accesses per pixel to re-read data stored in array `local`.
- There are multiple writes per pixel to the output array/port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

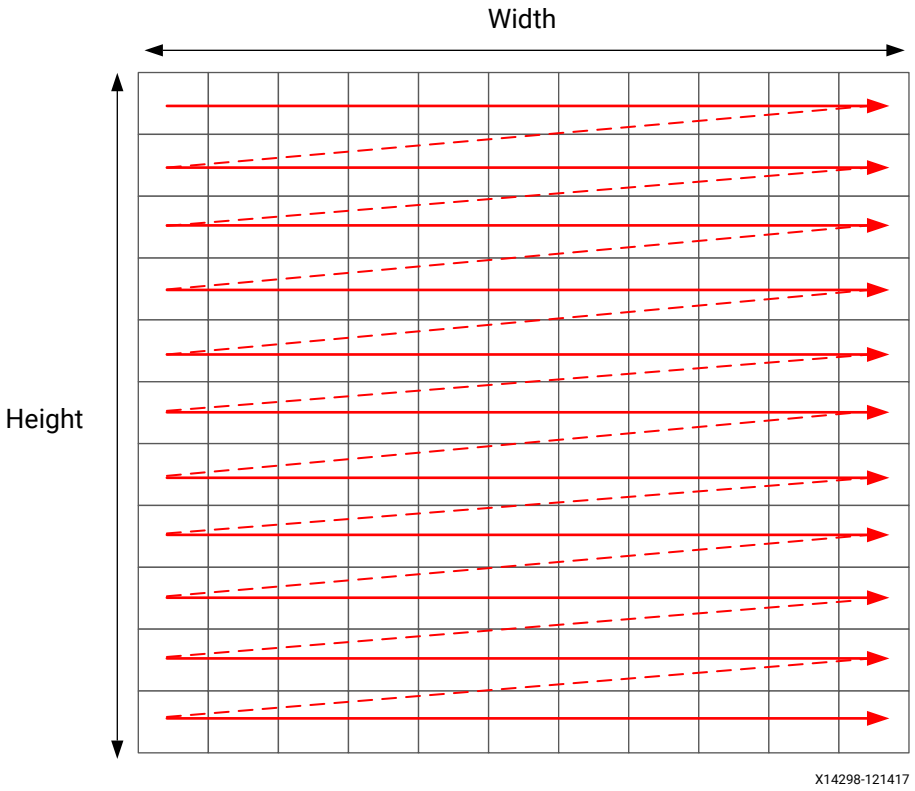
- Multiple accesses to read and then re-read data. Use local storage where possible.
- Accessing data in an arbitrary or random access manner. This requires the data to be stored locally in arrays and costs resources.
- Setting default values in arrays costs clock cycles and performance.

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.

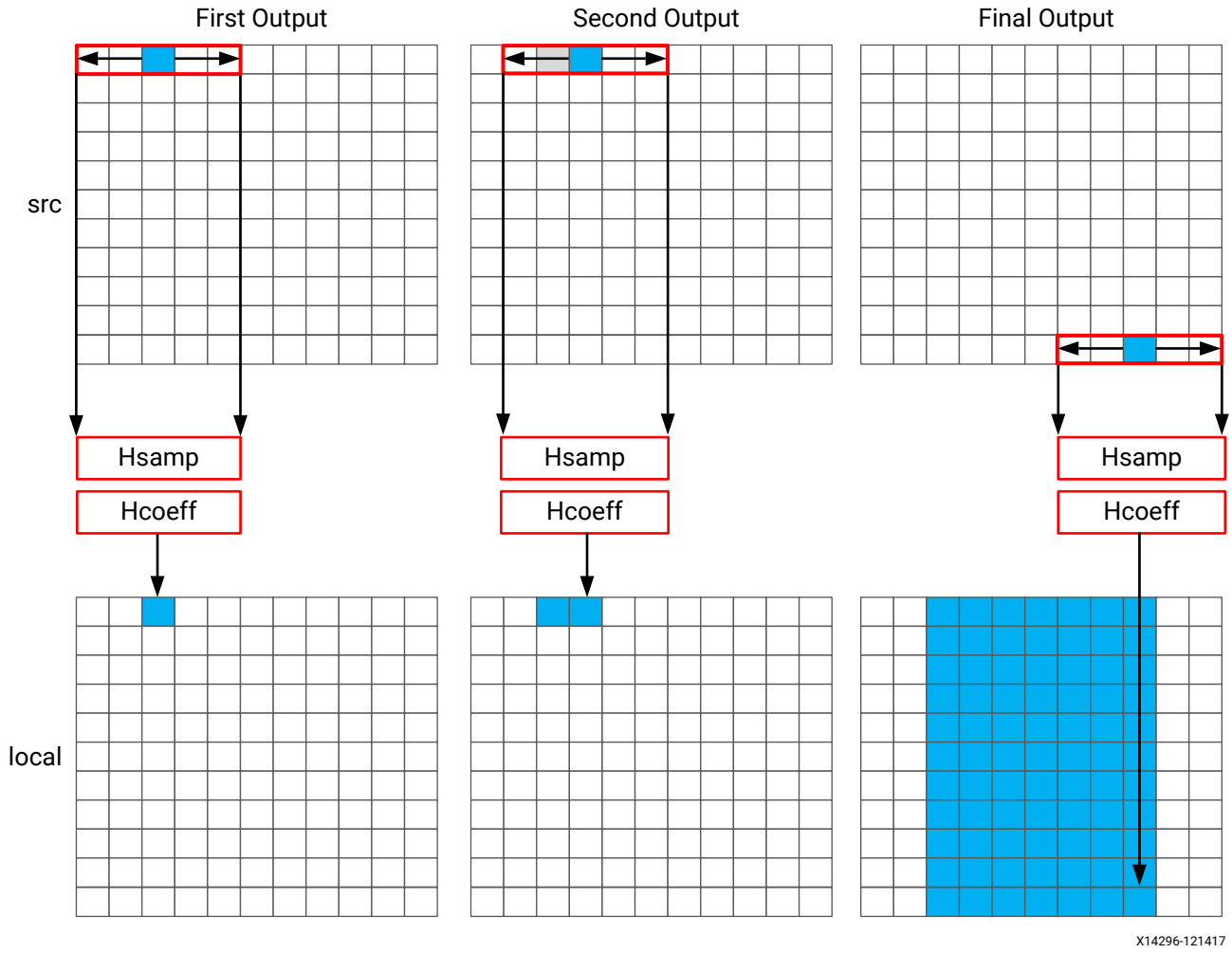
```
template<typename T, int K>
static void convolution_strm(
int width,
int height,
T src[TEST_IMG_ROWS][TEST_IMG_COLS],
T dst[TEST_IMG_ROWS][TEST_IMG_COLS],
const T *hcoeff,
const T *vcoeff)
{

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.

Chapter 5

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.

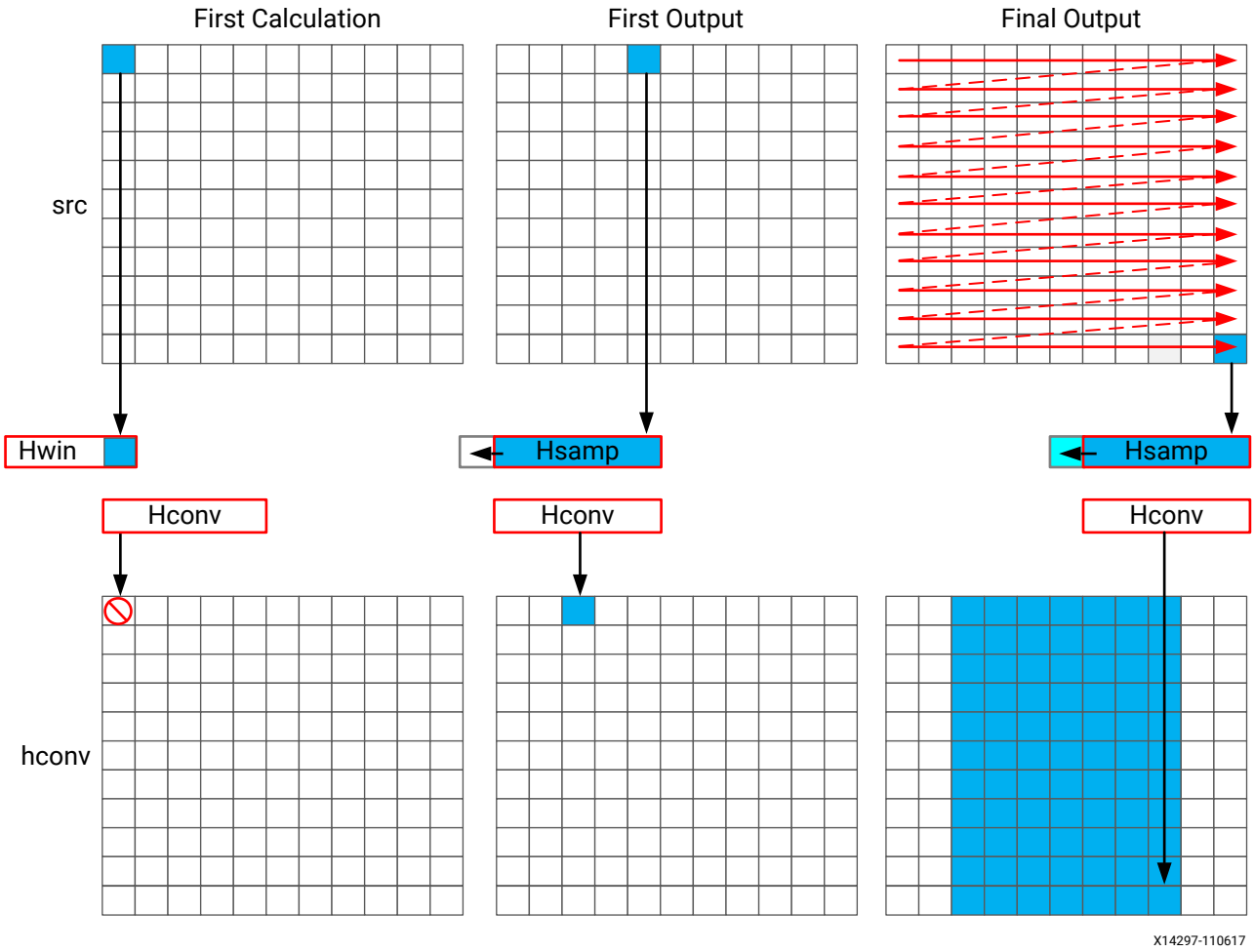
One of the keys to a high-performance FPGA is to minimize the access to and from the PS. Each access for data, which has previously been fetched, negatively impacts the performance of the system. An FPGA is capable of performing many concurrent calculations at once and reaching very high performance,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

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.

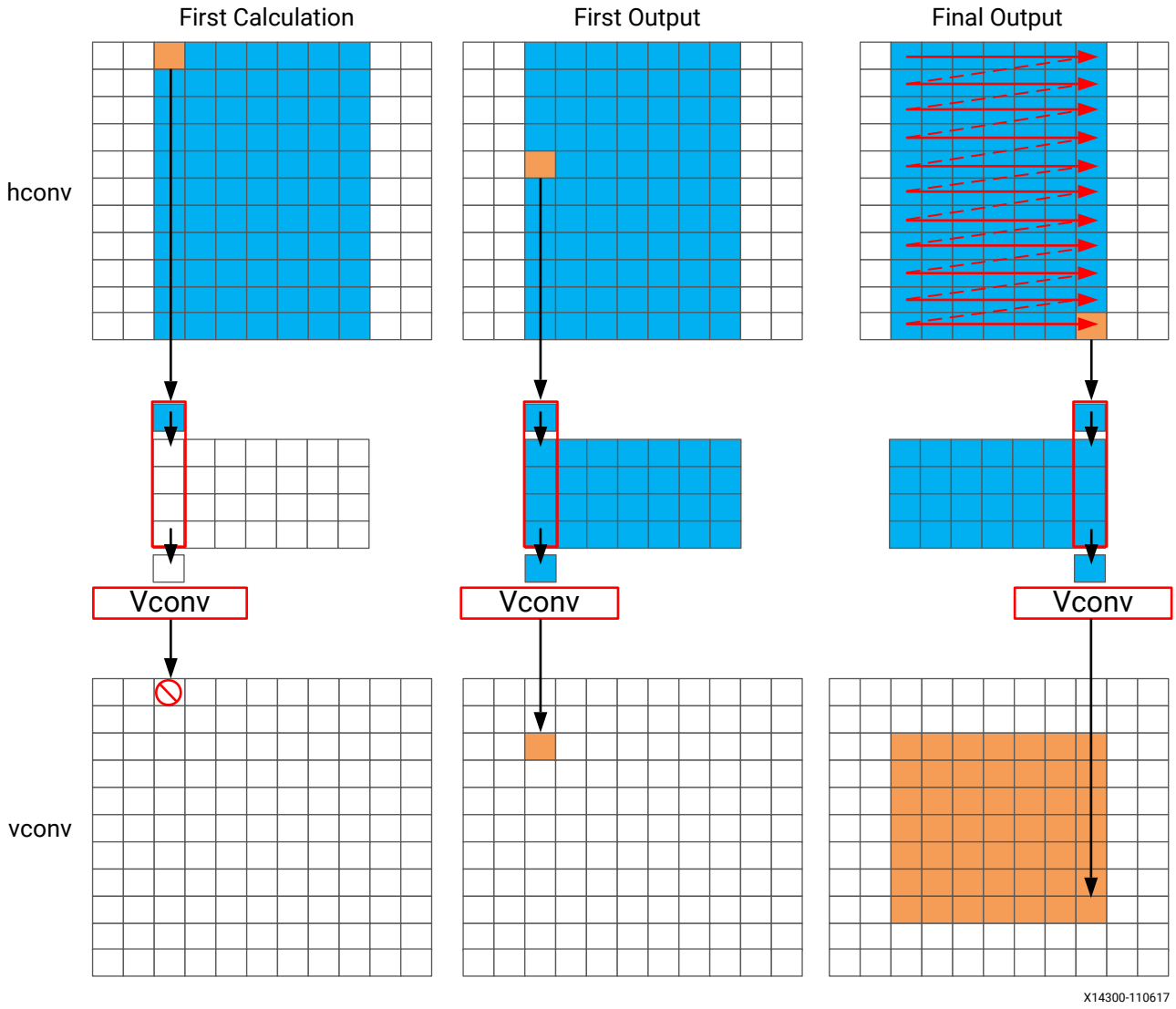
```
// Horizontal convolution
phconv=hconv_buffer; // set / reset pointer to start of buffer

// 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.

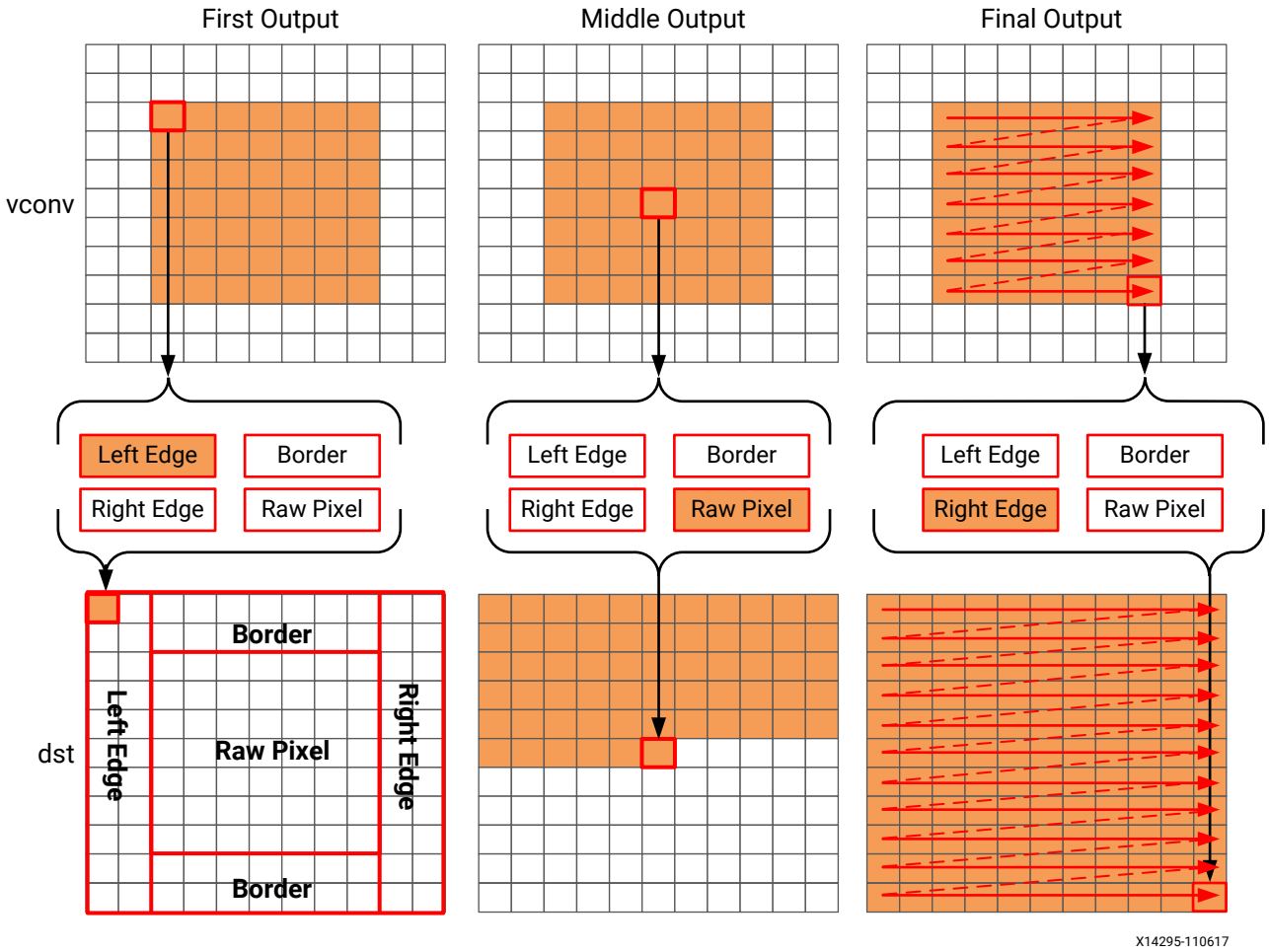
```
// Vertical convolution
phconv=hconv_buffer; // set/reset pointer to start of buffer
pvconv=vconv_buffer; // set/reset pointer to start of buffer
VConvH:for(int col = 0; col < height; col++) {
  VConvW:for(int row = 0; row < vconv_xlim; row++) {
#pragma HLS DEPENDENCE variable=linebuf inter false
#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.

Appendix A

OpenCL Attributes

Optimizations in OpenCL

This section describes OpenCL attributes that can be added to source code to assist system optimization by the SDAccel compiler, `xocc`, the SDSoC system compilers, `sdscc` and `sds++`, and Vivado HLS synthesis.

SDx provides OpenCL attributes to optimize your code for data movement and kernel performance. The goal of data movement optimization is to maximize the system level data throughput by maximizing interface bandwidth utilization and DDR bandwidth utilization. The goal of kernel computation optimization is to create processing logic that can consume all the data as soon as they arrive at kernel interfaces. This is generally achieved by expanding the processing code to match the data path with techniques such as function inlining and pipelining, loop unrolling, array partitioning, dataflowing, etc.

The OpenCL attributes include the types specified below:

Table 9: **OpenCL __attributes__ by Type**

Type	Attributes
Kernel Size	• reqd_work_group_size • vec_type_hint • work_group_size_hint • xcl_max_work_group_size • xcl_zero_global_work_offset
Function Inlining	• always_inline
Task-level Pipeline	• xcl_dataflow • xcl_reqd_pipe_depth
Pipeline	• xcl_pipeline_loop • xcl_pipeline_workitems
Loop Unrolling	• opencl_unroll_hint

Table 9: **OpenCL __attributes__ by Type** *(cont'd)*

Type	Attributes
Array Optimization	• xcl_array_partition • xcl_array_reshape **Note:** Array variables only accept a single array optimization attribute.

TIP: *The SDAccel and SDSoC compilers also support many of the standard attributes supported by* `gcc`*, such as* `always_inline`*,* `noinline`*,* `unroll`*, and* `nounroll`*.*

always_inline

Description

The `always_inline` attribute indicates that a function must be inlined. This attribute is a standard feature of GCC, and a standard feature of the SDx compilers.

TIP: *The* `noinline` *attribute is also a standard feature of GCC, and is also supported by SDx compilers.*

This attribute enables a compiler optimization to have a function inlined into the calling function. The inlined function is dissolved and no longer appears as a separate level of hierarchy in the RTL.

In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with surrounding operations in the calling function. However, an inlined function can no longer be shared with other functions, so the logic may be duplicated between the inlined function and a separate instance of the function which can be more broadly shared. While this can improve performance, this will also increase the area required for implementing the RTL.

For OpenCL kernels, the SDx compiler uses its own rules to inline or not inline a function. To directly control inlining functions, you should use the `always_inline` or `noinline` attributes.

By default, inlining is only performed on the next level of function hierarchy, not sub-functions.

IMPORTANT!: *When used with the* `xcl_dataflow` *attribute, the compiler will ignore the* `always_inline` *attribute and not inline the function.*

Syntax

Place the attribute in the OpenCL source before the function definition to always have it inlined whenever the function is called.

```
__attribute__((always_inline))
```

Examples

This example adds the `always_inline` attribute to function `foo`:

```
__attribute__((always_inline))
  void foo ( a, b, c, d ) {
  ...
}
```

This example prevents the inlining of the function `foo`:

```
__attribute__((noinline))
  void foo ( a, b, c, d ) {
  ...
}
```

See Also

- https://gcc.gnu.org
- *SDAccel Environment Profiling and Optimization Guide* (UG1207)

opencl_unroll_hint

Description

IMPORTANT!: *This is a compiler hint which the compiler may ignore.*

Loop unrolling is the first optimization technique available in SDAccel. The purpose of the loop unroll optimization is to expose concurrency to the compiler. This newly exposed concurrency reduces latency and improves performance, but also consumes more FPGA fabric resources.

The `opencl_unroll_hint` attribute is part of the OpenCL Language Specification, and specifies that loops (`for`, `while`, `do`) can be unrolled by the OpenCL compiler. See "Unrolling Loops" in *SDAccel Environment Optimization Guide* (UG1207) for more information.

The `opencl_unroll_hint` attribute qualifier must appear immediately before the loop to be affected. You can use this attribute to specify full unrolling of the loop, partial unrolling by a specified amount, or to disable unrolling of the loop.

Syntax

Place the attribute in the OpenCL source before the loop definition:

```
__attribute__((opencl_unroll_hint(n)))
```

Where:

- *n* is an optional loop unrolling factor and must be a positive integer, or compile time constant expression. An unroll factor of 1 disables unrolling.

TIP: *If `n` is not specified, the compiler automatically determines the unrolling factor for the loop.*

Example 1

The following example unrolls the `for` loop by a factor of 2. This results in two parallel loop iterations instead of four sequential iterations for the compute unit to complete the operation.

```
__attribute__((opencl_unroll_hint(2)))
for(int i = 0; i < LENGTH; i++) {
bufc[i] = bufa[i] * bufb[i];
}
```

Conceptually the compiler transforms the loop above to the code below:

```
for(int i = 0; i < LENGTH; i+=2) {
bufc[i] = bufa[i] * bufb[i];
bufc[i+1] = bufa[i+1] * bufb[i+1];
}
```

See Also

- *SDAccel Environment Optimization Guide* (UG1207)
- https://www.khronos.org/
- *The OpenCL C Specification*

reqd_work_group_size

Description

When OpenCL™ kernels are submitted for execution on an OpenCL device, they execute within an index space, called an ND range, which can have 1, 2, or 3 dimensions. This is called the global size in the OpenCL API. The work-group size defines the amount of the ND range that can be processed by a single invocation of a kernel compute unit. The work-group size is also called the local size in the OpenCL API. The OpenCL compiler can determine the work-group size based on the properties of the kernel and selected device. Once the work-group size (local size) has been determined, the ND range (global size) is divided automatically into work-groups, and the work-groups are scheduled for execution on the device.

Although the OpenCL compiler can define the work-group size, the specification of the `reqd_work_group_size` attribute on the kernel to define the work-group size is highly recommended for FPGA implementations of the kernel. The attribute is recommended for performance optimization during the generation of the custom logic for a kernel. See "OpenCL Execution Model" in the *SDAccel Environment Profiling and Optimization Guide* (UG1207) for more information.

TIP: *In the case of an FPGA implementation, the specification of the `reqd_work_group_size` attribute is highly recommended as it can be used for performance optimization during the generation of the custom logic for a kernel.*

OpenCL kernel functions are executed exactly one time for each point in the ND range index space. This unit of work for each point in the ND range is called a work-item. Work-items are organized into work-groups, which are the unit of work scheduled onto compute units. The optional `reqd_work_group_size` defines the work-group size of a compute unit that must be used as the `local_work_size` argument to `clEnqueueNDRangeKernel`. This allows the compiler to optimize the generated code appropriately for this kernel.

Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel:

```
__attribute__((reqd_work_group_size(X, Y, Z)))
```

Where:

- *X*, *Y*, *Z*: Specifies the ND range of the kernel. This represents each dimension of a three dimensional matrix specifying the size of the work-group for the kernel.

Examples

The following OpenCL API C kernel code shows a vector addition design where two arrays of data are summed into a third array. The required size of the work-group is 16x1x1. This kernel will execute 16 times to produce a valid result.

```
#include <clc.h>
// For VHLS OpenCL C kernels, the full work group is synthesized
__attribute__ ((reqd_work_group_size(16, 1, 1)))
__kernel void
vadd(__global int* a,
__global int* b,
__global int* c)
{
int idx = get_global_id(0);
c[idx] = a[idx] + b[idx];
}
```

See Also

- *SDAccel Environment Profiling and Optimization Guide* (UG1207)
- https://www.khronos.org/
- *The OpenCL C Specification*

vec_type_hint

Description

IMPORTANT!: *This is a compiler hint which the compiler may ignore.*

The optional `__attribute__((vec_type_hint(<type>)))` is part of the OpenCL Language Specification, and is a hint to the OpenCL compiler representing the computational width of the kernel, providing a basis for calculating processor bandwidth utilization when the compiler is looking to autovectorize the code.

By default, the kernel is assumed to have the `__attribute__((vec_type_hint(int)))` qualifier. This lets you specify a different vectorization type.

Implicit in autovectorization is the assumption that any libraries called from the kernel must be re-compilable at run time to handle cases where the compiler decides to merge or separate workitems. This probably means that such libraries can never be hard coded binaries or that hard coded binaries must be accompanied either by source or some re-targetable intermediate representation. This may be a code security question for some.

Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel:

```
__attribute__((vec_type_hint(<type>)))
```

Where:

- *<type>*: is one of the built-in vector types listed in the following table, or the constituent scalar element types.

***Note*:** When not specified, the kernel is assumed to have an INT type.

Table 10: **Vector Types**

Type	Description
char*n*	A vector of *n* 8-bit signed two's complement integer values.
uchar*n*	A vector of *n* 8-bit unsigned integer values.
short*n*	A vector of *n* 16-bit signed two's complement integer values.
ushort*n*	A vector of *n* 16-bit unsigned integer values.
int*n*	A vector of *n* 32-bit signed two's complement integer values.
uint*n*	A vector of *n* 32-bit unsigned integer values.
long*n*	A vector of *n* 64-bit signed two's complement integer values.
ulong*n*	A vector of *n* 64-bit unsigned integer values.
float*n*	A vector of *n* 32-bit floating-point values.
double*n*	A vector of *n* 64-bit floating-point values.

***Note*:** *n* is assumed to be 1 when not specified. The vector data type names defined above where n is any value other than 2, 3, 4, 8 and 16, are also reserved. That is to say, n can only be specified as 2,3,4,8, and 16.

Examples

The following example autovectorizes assuming double-wide integer as the basic computation width:

```
#include <clc.h>
// For VHLS OpenCL C kernels, the full work group is synthesized
__attribute__((vec_type_hint(double)))
__attribute__ ((reqd_work_group_size(16, 1, 1)))
__kernel void
...
```

See Also

- *SDAccel Environment Optimization Guide* (UG1207)
- https://www.khronos.org/
- *The OpenCL C Specification*

work_group_size_hint

Description

IMPORTANT!: *This is a compiler hint which the compiler may ignore.*

The work-group size in the OpenCL standard defines the size of the ND range space that can be handled by a single invocation of a kernel compute unit. When OpenCL kernels are submitted for execution on an OpenCL device, they execute within an index space, called an ND range, which can have 1, 2, or 3 dimensions. See "OpenCL Execution Model" in *SDAccel Environment Optimization Guide* (UG1207) for more information.

OpenCL kernel functions are executed exactly one time for each point in the ND range index space. This unit of work for each point in the ND range is called a work-item. Unlike `for` loops in C, where loop iterations are executed sequentially and in-order, an OpenCL runtime and device is free to execute work-items in parallel and in any order.

Work-items are organized into work-groups, which are the unit of work scheduled onto compute units. The optional `work_group_size_hint` attribute is part of the OpenCL Language Specification, and is a hint to the compiler that indicates the work-group size value most likely to be specified by the `local_work_size` argument to `clEnqueueNDRangeKernel`. This allows the compiler to optimize the generated code according to the expected value.

TIP: *In the case of an FPGA implementation, the specification of the* `reqd_work_group_size` *attribute instead of the* `work_group_size_hint` *is highly recommended as it can be used for performance optimization during the generation of the custom logic for a kernel.*

Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel:

```
__attribute__((work_group_size_hint(X, Y, Z)))
```

Where:

- *X*, *Y*, *Z*: Specifies the ND range of the kernel. This represents each dimension of a three dimensional matrix specifying the size of the work-group for the kernel.

Examples

The following example is a hint to the compiler that the kernel will most likely be executed with a work-group size of 1:

```
__attribute__((work_group_size_hint(1, 1, 1)))
__kernel void
...
```

See Also

- *SDAccel Environment Optimization Guide* (UG1207)
- https://www.khronos.org/
- *The OpenCL C Specification*

xcl_array_partition

Description

IMPORTANT!: *Array variables only accept one attribute. While* `xcl_array_partition` *does support multi-dimensional arrays, you can only reshape one dimension of the array with a single attribute.*

One of the advantages of the FPGA over other compute devices for OpenCL programs is the ability for the application programmer to customize the memory architecture all throughout the system and into the compute unit. By default, The SDAccel compiler generates a memory architecture within the compute unit that maximizes local and private memory bandwidth based on static code analysis of the kernel code. Further optimization of these memories is possible based on attributes in the kernel source code, which can be used to specify physical layouts and implementations of local and private memories. The attribute in the SDAccel compiler to control the physical layout of memories in a compute unit is `array_partition`.

For one dimensional arrays, the `array_partition` attribute implements an array declared within kernel code as multiple physical memories instead of a single physical memory. The selection of which partitioning scheme to use depends on the specific application and its performance goals. The array partitioning schemes available in the SDAccel compiler are `cyclic`, `block`, and `complete`.

Syntax

Place the attribute with the definition of the array variable:

```
__attribute__((xcl_array_partition(<type>, <factor>,
<dimension>)))
```

Where:

- *<type>*: Specifies one of the following partition types:
  - `cyclic`: Cyclic partitioning is the implementation of an array as a set of smaller physical memories that can be accessed simultaneously by the logic in the compute unit. The array is partitioned cyclically by putting one element into each memory before coming back to the first memory to repeat the cycle until the array is fully partitioned.
  - `block`: Block partitioning is the physical implementation of an array as a set of smaller memories that can be accessed simultaneously by the logic inside of the compute unit. In this case, each memory block is filled with elements from the array before moving on to the next memory.
  - `complete`: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers.
  - The default *type* is `complete`.
- *<factor>*: For cyclic type partitioning, the *factor* specifies how many physical memories to partition the original array into in the kernel code. For Block type partitioning, the *factor* specifies the number of elements from the original array to store in each physical memory.

IMPORTANT!: *For `complete` type partitioning, the `factor` is not specified.*

- *<dimension>*: Specifies which array dimension to partition. Specified as an integer from 1 to *N*. SDAccel supports arrays of N dimensions and can partition the array on any single dimension.

Example 1

For example, consider the following array declaration:

```
int buffer[16];
```

The integer array, named buffer, stores 16 values that are 32-bits wide each. Cyclic partitioning can be applied to this array with the following declaration:

```
int buffer[16] __attribute__((xcl_array_partition(cyclic,4,1)));
```

In this example, the cyclic *partition_type* attribute tells SDAccel to distribute the contents of the array among four physical memories. This attribute increases the immediate memory bandwidth for operations accessing the array buffer by a factor of four.

All arrays inside of a compute unit in the context of SDAccel are capable of sustaining a maximum of two concurrent accesses. By dividing the original array in the code into four physical memories, the resulting compute unit can sustain a maximum of eight concurrent accesses to the array buffer.

Example 2

Using the same integer array as found in Example 1, block partitioning can be applied to the array with the following declaration:

```
int buffer[16] __attribute__((xcl_array_partition(block,4,1)));
```

Since the size of the block is four, SDAccel will generate four physical memories, sequentially filling each memory with data from the array.

Example 3

Using the same integer array as found in Example 1, complete partitioning can be applied to the array with the following declaration:

```
int buffer[16] __attribute__((xcl_array_partition(complete, 1)));
```

In this example the array is completely partitioned into distributed RAM, or 16 independent registers in the programmable logic of the kernel. Because complete is the default, the same effect can also be accomplished with the following declaration:

```
int buffer[16] __attribute__((xcl_array_partition));
```

While this creates an implementation with the highest possible memory bandwidth, it is not suited to all applications. The way in which data is accessed by the kernel code through either constant or data dependent indexes affects the amount of supporting logic that SDx has to build around each register to ensure functional equivalence with the usage in the original code. As a general best practice guideline for SDx, the complete partitioning attribute is best suited for arrays in which at least one dimension of the array is accessed through the use of constant indexes.

See Also

- xcl_array_reshape
- pragma HLS array_partition
- *SDAccel Environment Optimization Guide* (UG1207)
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

xcl_array_reshape

Description

IMPORTANT!: *Array variables only accept one attribute. While* `xcl_array_reshape` *does support multi-dimensional arrays, you can only reshape one dimension of the array with a single attribute.*

Combines array partitioning with vertical array mapping.

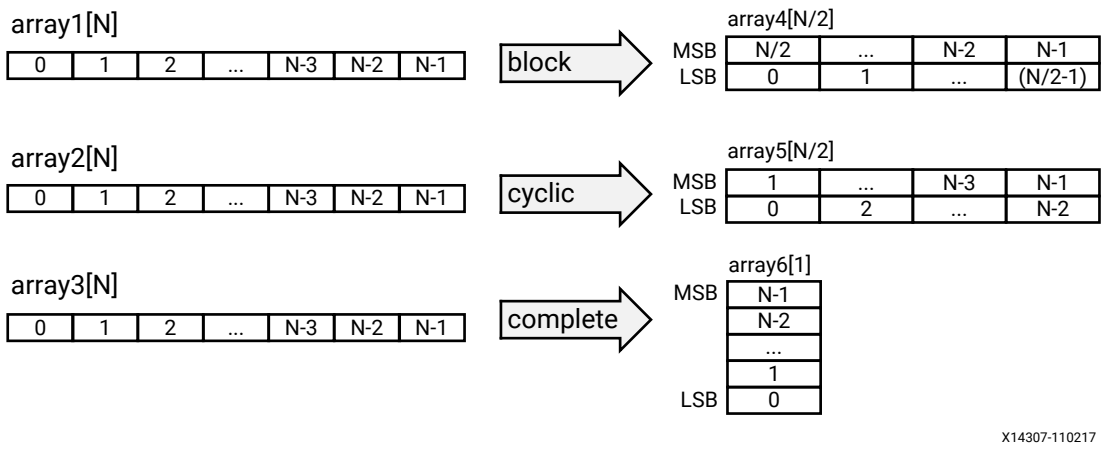
The `ARRAY_RESHAPE` attribute combines the effect of `ARRAY_PARTITION`, breaking an array into smaller arrays, and concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing parallel access to the data. This attribute creates a new array with fewer elements but with greater bit-width, allowing more data to be accessed in a single clock cycle.

Given the following code:

```
void foo (...) {
int array1[N] __attribute__((xcl_array_reshape(block, 2, 1)));
int array2[N] __attribute__((xcl_array_reshape(cycle, 2, 1)));
int array3[N] __attribute__((xcl_array_reshape(complete, 1)));
...
}
```

The ARRAY_RESHAPE attribute transforms the arrays into the form shown in the following figure:

Figure 1: **ARRAY_RESHAPE**



Syntax

Place the attribute with the definition of the array variable:

```
__attribute__((xcl_array_reshape(<type>,<factor>,
<dimension>)))
```

Where:

- *<type>*: Specifies one of the following partition types:
  - `cyclic`: Cyclic partitioning is the implementation of an array as a set of smaller physical memories that can be accessed simultaneously by the logic in the compute unit. The array is partitioned cyclically by putting one element into each memory before coming back to the first memory to repeat the cycle until the array is fully partitioned.
  - `block`: Block partitioning is the physical implementation of an array as a set of smaller memories that can be accessed simultaneously by the logic inside of the compute unit. In this case, each memory block is filled with elements from the array before moving on to the next memory.

    - `complete`: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers.
    - The default *`type`* is `complete`.

- *`<factor>`*: For cyclic type partitioning, the *`factor`* specifies how many physical memories to partition the original array into in the kernel code. For Block type partitioning, the *`factor`* specifies the number of elements from the original array to store in each physical memory.

IMPORTANT!: *For `complete` type partitioning, the `factor` should not be specified.*

- *`<dimension>`*: Specifies which array dimension to partition. Specified as an integer from 1 to *N*. SDAccel supports arrays of N dimensions and can partition the array on any single dimension.

Example 1

Reshapes (partition and maps) an 8-bit array with 17 elements, AB[17], into a new 32-bit array with five elements using block mapping.

```
int AB[17] __attribute__((xcl_array_reshape(block,4,1)));
```

TIP: *A `factor` of 4 indicates that the array should be divided into four. So 17 elements is reshaped into an array of 5 elements, with four times the bit-width. In this case, the last element, AB[17], is mapped to the lower eight bits of the fifth element, and the rest of the fifth element is empty.*

Example 2

Reshapes the two-dimensional array AB[6][4] into a new array of dimension [6][2], in which dimension 2 has twice the bit-width:

```
int AB[6][4] __attribute__((xcl_array_reshape(block,2,2)));
```

Example 3

Reshapes the three-dimensional 8-bit array, AB[4][2][2] in function `foo`, into a new single element array (a register), 128 bits wide (4*2*2*8):

```
int AB[4][2][2] __attribute__((xcl_array_reshape(complete,0)));
```

TIP: *A `dimension` of 0 means to reshape all dimensions of the array.*

See Also

- xcl_array_partition
- pragma HLS array_reshape

- *SDAccel Environment Optimization Guide* (UG1207)
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

xcl_data_pack

Description

Packs the data fields of a struct into a single scalar with a wider word width.

The `xcl_data_pack` attribute is used for packing all the elements of a `struct` into a single wide vector to reduce the memory required for the variable. This allows all members of the `struct` to be read and written to simultaneously. The bit alignment of the resulting new wide-word can be inferred from the declaration order of the `struct` fields. The first field takes the LSB of the vector, and the final element of the `struct` is aligned with the MSB of the vector.

TIP: *Any arrays declared inside the `struct` are completely partitioned and reshaped into a wide scalar and packed with other scalar fields.*

If a `struct` contains arrays, those arrays can be optimized using the `xcl_array_partition` attribute to partition the array. The `xcl_data_pack` attribute performs a similar operation as the complete partitioning of the `xcl_array_partition` attribute, reshaping the elements in the `struct` to a single wide vector.

A `struct` cannot be optimized with `xcl_data_pack` and also partitioned. The `xcl_data_pack` and `xcl_array_partition` attributes are mutually exclusive.

You should exercise some caution when using the `xcl_data_pack` optimization on structs with large arrays. If an array has 4096 elements of type int, this will result in a vector (and port) of width 4096*32=131072 bits. SDx can create this RTL design, however it is very unlikely logic synthesis will be able to route this during the FPGA implementation.

Syntax

Place within the region where the `struct` variable is defined:

```
__attribute__((xcl_data_pack(<variable>, <name>)))
```

Where:

- *<variable>*: is the variable to be packed.
- *<name>*: Specifies the name of resultant variable after packing. If no *<name>* is specified, the input *<variable>* is used.

Example 1

Packs struct array AB[17] with three 8-bit field fields (typedef struct {unsigned char R, G, B;} pixel) in function `foo`, into a new 17 element array of 24 bits.

```
typedef struct{
unsigned char R, G, B;
} pixel;

pixel AB[17] __attribute__((xcl_data_pack(AB)));
```

See Also

- pragma HLS data_pack
- *SDAccel Environment Profiling and Optimization Guide* (UG1207)

xcl_dataflow

Description

The `xcl_dataflow` attribute enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design.

All operations are performed sequentially in a C description. In the absence of any directives that limit resources (such as `pragma HLS allocation`), Vivado HLS seeks to minimize latency and improve concurrency. However, data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation. The dataflow optimization enables the operations in a function or loop to start operation before the previous function or loop completes all its operations.

When dataflow optimization is specified, Vivado HLS analyzes the dataflow between sequential functions or loops and create channels (based on pingpong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, Vivado HLS attempts to minimize the initiation interval and start operation as soon as data is available.

TIP: *Vivado HLS provides dataflow configuration settings. The* `config_dataflow` *command specifies the default memory channel and FIFO depth used in dataflow optimization. Refer to the Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more information.*

For the DATAFLOW optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent Vivado HLS from performing the DATAFLOW optimization, refer to *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information:

- Single-producer-consumer violations
- Bypassing tasks
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions

IMPORTANT!: *If any of these coding styles are present, Vivado HLS issues a message and does not perform DATAFLOW optimization.*

Finally, the DATAFLOW optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the DATAFLOW optimization, you must apply the optimization to the loop, the sub-function, or inline the sub-function.

Syntax

Assign the `dataflow` attribute before the function definition or the loop definition:

```
__attribute__((xcl_dataflow))
```

Examples

Specifies dataflow optimization within function `foo`.

```
#pragma HLS dataflow
```

See Also

- pragma HLS dataflow
- *SDAccel Environment Optimization Guide* (UG1207)
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

xcl_dependence

Description

The `xcl_dependence` attribute is used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).

Vivado HLS automatically detects dependencies:

- Within loops (loop-independent dependence), or
- Between different iterations of a loop (loop-carry dependence).

These dependencies impact when operations can be scheduled, especially during function and loop pipelining.

- Loop-independent dependence: The same element is accessed in the same loop iteration.

```
for (i=0;i<N;i++) {
 A[i]=x;
 y=A[i];
}
```

- Loop-carry dependence: The same element is accessed in a different loop iteration.

```
for (i=0;i<N;i++) {
 A[i]=A[i-1]*2;
}
```

Under certain complex scenarios automatic dependence analysis can be too conservative and fail to filter out false dependencies. Under certain circumstances, such as variable dependent array indexing, or when an external requirement needs to be enforced (for example, two inputs are never the same index), the dependence analysis might be too conservative. The `xcl_dependence` attribute allows you to explicitly specify the dependence and resolve a false dependence.

IMPORTANT!: *Specifying a false dependency, when in fact the dependency is not false, can result in incorrect hardware. Be sure dependencies are correct (true or false) before specifying them.*

Syntax

This attribute must be assigned at the declaration of the variable:

```
__attribute__((xcl_dependence(<class> <type> <direction> distance=<int>
<dependent>)))
```

Where:

- *<class>*: Specifies a class of variables in which the dependence needs clarification. Valid values include `array` or `pointer`.

TIP: *<class> is mutually exclusive with variable= as you can either specify a variable or a class of variables.*

- *<type>*: Valid values include `intra` or `inter`. Specifies whether the dependence is:
  - `intra`: dependence within the same loop iteration. When dependence *<type>* is specified as `intra`, and *<dependent>* is false, Vivado HLS may move operations freely within a loop, increasing their mobility and potentially improving performance or area. When *<dependent>* is specified as true, the operations must be performed in the order specified.
  - `inter`: dependence between different loop iterations. This is the default *<type>*. If dependence *<type>* is specified as `inter`, and *<dependent>* is false, it allows Vivado HLS to perform operations in parallel if the function or loop is pipelined, or the loop is unrolled, or partially unrolled, and prevents such concurrent operation when *<dependent>* is specified as true.
- *<direction>*: Valid values include `RAW`, `WAR`, or `WAW`. This is relevant for loop-carry dependencies only, and specifies the direction for a dependence:
  - `RAW` (Read-After-Write - true dependence) The write instruction uses a value used by the read instruction.
  - `WAR` (Write-After-Read - anti dependence) The read instruction gets a value that is overwritten by the write instruction.
  - `WAW` (Write-After-Write - output dependence) Two write instructions write to the same location, in a certain order.
- `distance=`*<int>*: Specifies the inter-iteration distance for array access. Relevant only for loop-carry dependencies where dependence is set to `true`.
- *<dependent>*: Specifies whether a dependence needs to be enforced (`true`) or removed (`false`). The default is `true`.

Example 1

In the following example, Vivado HLS does not have any knowledge about the value of `cols` and conservatively assumes that there is always a dependence between the write to `buff_A[1][col]` and the read from `buff_A[1][col]`. In an algorithm such as this, it is unlikely `cols` will ever be zero, but Vivado HLS cannot make assumptions about data dependencies. To overcome this deficiency, you can use the `xcl_dependence` attribute to state that there is no dependence between loop iterations (in this case, for both `buff_A` and `buff_B`).

```
void foo(int rows, int cols, ...)
  for (row = 0; row < rows + 1; row++) {
    for (col = 0; col < cols + 1; col++)
__attribute__((xcl_pipeline_loop(II=1)))
    {
      if (col < cols) {
      buff_A[2][col] = buff_A[1][col] __attribute__((xcl_dependence(inter
false))); // read from buff_A
      buff_A[1][col] = buff_A[0][col]; // write to buff_A
      buff_B[1][col] = buff_B[0][col] __attribute__((xcl_dependence(inter
false)));
      temp = buff_A[0][col];
}
```

Example 2

Removes the dependence between `Var1` in the same iterations of `loop_1` in function `foo`.

```
__attribute__((xcl_dependence(intra false)));
```

Example 3

Defines the dependence on all arrays in `loop_2` of function `foo` to inform Vivado HLS that all reads must happen after writes (RAW) in the same loop iteration.

```
__attribute__((xcl_dependence(array intra RAW true)));
```

See Also

- pragma HLS dependence
- *SDAccel Environment Profiling and Optimization Guide* (UG1207)
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

xcl_max_work_group_size

Description

Use this attribute instead of `reqd_work_group_size` when you need to specify a larger kernel than the 4K size.

Extends the default maximum work group size supported in SDx by the `reqd_work_group_size` attribute. SDx supports work size larger than 4096 with the Xilinx attribute `xcl_max_work_group_size`.

***Note*:** The actual workgroup size limit is dependent on the Xilinx device selected for the platform.

Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel:

```
__attribute__((xcl_max_work_group_size(X, Y, Z)))
```

Where:

- *X*, *Y*, *Z*: Specifies the ND range of the kernel. This represents each dimension of a three dimensional matrix specifying the size of the work-group for the kernel.

Example 1

Below is the kernel source code for an un-optimized adder. No attributes were specified for this design other than the work size equal to the size of the matrices (i.e., 64x64). That is, iterating over an entire workgroup will fully add the input matrices a and b and output the result to output. All three are global integer pointers, which means each value in the matrices is four bytes and is stored in off-chip DDR global memory.

```
#define RANK 64
__kernel __attribute__ ((reqd_work_group_size(RANK, RANK, 1)))
void madd(__global int* a, __global int* b, __global int* output) {
int index = get_local_id(1)*get_local_size(0) + get_local_id(0);
output[index] = a[index] + b[index];
}
```

This local work size of (64, 64, 1) is the same as the global work size. It should be noted that this setting creates a total work size of 4096.

***Note*:** This is the largest work size that SDAccel supports with the standard OpenCL attribute reqd_work_group_size. SDAccel supports work size larger than 4096 with the Xilinx attribute xcl_max_work_group_size.

Any matrix larger than 64x64 would need to only use one dimension to define the work size. That is, a 128x128 matrix could be operated on by a kernel with a work size of (128, 1, 1), where each invocation operates on an entire row or column of data.

See Also

- *SDAccel Environment Optimization Guide* (UG1207)
- https://www.khronos.org/
- *The OpenCL C Specification*

xcl_pipeline_loop

Description

You can pipeline a loop to improve latency and maximize kernel throughput and performance.

Although unrolling loops increases concurrency, it does not address the issue of keeping all elements in a kernel data path busy at all times. Even in an unrolled case, loop control dependencies can lead to sequential behavior. The sequential behavior of operations results in idle hardware and a loss of performance.

Xilinx addresses this issue by introducing a vendor extension on top of the OpenCL 2.0 specification for loop pipelining: `xcl_pipeline_loop`.

By default, the XOCC compiler automatically pipelines loops with a trip count more than 64, or unrolls loops with a trip count less than 64. This should provide good results. However, you can choose to pipeline loops (instead of the automatic unrolling) by explicitly specifying the `nounroll` attribute and `xcl_pipeline_loop` attribute before the loop.

Syntax

Place the attribute in the OpenCL source before the loop definition:

```
__attribute__((xcl_pipeline_loop))
```

Examples

The following example pipelines `LOOP_1` of function `vaccum` to improve performance:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vaccum(__global const int* a, __global const int* b, __global int*
result)
{
int tmp = 0;
```

```
__attribute__((xcl_pipeline_loop))
LOOP_1: for (int i=0; i < 32; i++) {
tmp += a[i] * b[i];
}
result[0] = tmp;
}
```

See Also

- pragma HLS pipeline
- *SDAccel Environment Optimization Guide* (UG1207)
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

xcl_pipeline_workitems

Description

Pipeline a work item to improve latency and throughput. Work item pipelining is the extension of loop pipelining to the kernel work group. This is necessary for maximizing kernel throughput and performance.

Syntax

Place the attribute in the OpenCL source before the elements to pipeline:

```
__attribute__((xcl_pipeline_workitems))
```

Example 1

In order to handle the `reqd_work_group_size` attribute in the following example, SDAccel automatically inserts a loop nest to handle the three-dimensional characteristics of the ND range (3,1,1). As a result of the added loop nest, the execution profile of this kernel is like an unpipelined loop. Adding the `xcl_pipeline_workitems` attribute adds concurrency and improves the throughput of the code.

```
kernel
__attribute__ ((reqd_work_group_size(3,1,1)))
void foo(...)
{
...
__attribute__((xcl_pipeline_workitems)) {
int tid = get_global_id(0);
op_Read(tid);
```

```
op_Compute(tid);
op_Write(tid);
}
...
}
```

Example 2

The following example adds the work-item pipeline to the appropriate elements of the kernel:

```
__kernel __attribute__ ((reqd_work_group_size(8, 8, 1)))
void madd(__global int* a, __global int* b, __global int* output)
{
int rank = get_local_size(0);
__local unsigned int bufa[64];
__local unsigned int bufb[64];
__attribute__((xcl_pipeline_workitems)) {
int x = get_local_id(0);
int y = get_local_id(1);
bufa[x*rank + y] = a[x*rank + y];
bufb[x*rank + y] = b[x*rank + y];
}
barrier(CLK_LOCAL_MEM_FENCE);
__attribute__((xcl_pipeline_workitems)) {
int index = get_local_id(1)*rank + get_local_id(0);
output[index] = bufa[index] + bufb[index];
}
}
```

See Also

- pragma HLS pipeline
- *SDAccel Environment Optimization Guide* (UG1207)
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

xcl_reqd_pipe_depth

Description

IMPORTANT!: *Pipes must be declared in lower case alphanumerics. In addition, `printf()` is not supported with variables used in pipes.*

The OpenCL 2.0 specification introduces a new memory object called pipe. A pipe stores data organized as a FIFO. Pipes can be used to stream data from one kernel to another inside the FPGA device without having to use the external memory, which greatly improves the overall system latency.

In the SDAccel development environment, pipes must be statically defined outside of all kernel functions:. The depth of a pipe must be specified by using the `xcl_reqd_pipe_depth` attribute in the pipe declaration:

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)));
```

Pipes can only be accessed using standard OpenCL `read_pipe()` and `write_pipe()` built-in functions in non-blocking mode, or using Xilinx extended `read_pipe_block()` and `write_pipe_block()` functions in blocking mode.

IMPORTANT!: *A given pipe, can have one and only one producer and consumer in different kernels.*

Pipe objects are not accessible from the host CPU. The status of pipes can be queried using OpenCL `get_pipe_num_packets()` and `get_pipe_max_packets()` built-in functions. See *The OpenCL C Specification* from Khronos OpenCL Working Group for more details on these built-in functions.

Syntax

This attribute must be assigned at the declaration of the pipe object:

```
pipe int id __attribute__((xcl_reqd_pipe_depth(n)));
```

Where:

- *id*: Specifies an identifier for the pipe, which must consist of lower-case alphanumerics. For example `infifo1` not `inFifo1`.
- *n*: Specifies the depth of the pipe. Valid depth values are 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768.

Examples

The following is the `dataflow_pipes_ocl` example from Xilinx GitHub that use pipes to pass data from one processing stage to the next using blocking `read_pipe_block()` and `write_pipe_block()` functions:

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel : Read Data from Global Memory and write into Pipe P0
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
{
__attribute__((xcl_pipeline_loop))
mem_rd: for (int i = 0 ; i < size ; i++)
{
//blocking Write command to pipe P0
write_pipe_block(p0, &input[i]);
}
```

```
}
// Adder Stage Kernel: Read Input data from Pipe P0 and write the result
// into Pipe P1
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void adder_stage(int inc, int size)
{
__attribute__((xcl_pipeline_loop))
execute: for(int i = 0 ; i < size ; i++)
{
int input_data, output_data;
//blocking read command to Pipe P0
read_pipe_block(p0, &input_data);
output_data = input_data + inc;
//blocking write command to Pipe P1
write_pipe_block(p1, &output_data);
}
}
// Output Stage Kernel: Read result from Pipe P1 and write the result to
// Global Memory
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
__attribute__((xcl_pipeline_loop))
mem_wr: for (int i = 0 ; i < size ; i++)
{
//blocking read command to Pipe P1
read_pipe_block(p1, &output[i]);
}
}
```

See Also

- *SDAccel Environment Optimization Guide* (UG1207)
- https://www.khronos.org/
- *The OpenCL C Specification*

xcl_zero_global_work_offset

Description

If you use `clEnqueueNDRangeKernel` with the `global_work_offset` set to NULL or all zeros, you can use this attribute to tell the compiler that the `global_work_offset` is always zero.

This attribute can improve memory performance when you have memory accesses like:

```
A[get_global_id(x)] = ...;
```

Note: You can specify `reqd_work_group_size`, `vec_type_hint`, and `xcl_zero_global_work_offset` together to maximize performance.

Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel:

```
__kernel __attribute__((xcl_zero_global_work_offset))
void test (__global short *input, __global short *output, __constant short
*constants) { }
```

See Also

- reqd_work_group_size
- vec_type_hint
- clEnqueueNDRangeKernel
- *SDAccel Environment Optimization Guide* (UG1207)

HLS Pragmas

Optimizations in Vivado HLS

In both SDAccel and SDSoC projects, the hardware kernel must be synthesized from the OpenCL, C, or C++ language, into RTL that can be implemented into the programmable logic of a Xilinx device. Vivado HLS synthesizes the RTL from the OpenCL, C, and C++ language descriptions.

Vivado HLS is intended to work with your SDAccel or SDSoC Development Environment project without interaction. However, Vivado HLS also provides pragmas that can be used to optimize the design: reduce latency, improve throughput performance, and reduce area and device resource utilization of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

IMPORTANT!:
Although the SDSoC environment supports the use of HLS pragmas, it does not support pragmas applied to any argument of the function interface (interface, array partition, or data_pack pragmas). Refer to "Optimizing the Hardware Function" in the SDSoC Environment Optimization Guide (UG1235) for more information.

The Vivado HLS pragmas include the optimization types specified below:

Table 11: **Vivado HLS Pragmas by Type**

Type	Attributes
Kernel Optimization	• pragma HLS allocation • pragma HLS clock • pragma HLS expression_balance • pragma HLS latency • pragma HLS reset • pragma HLS resource • pragma HLS top
Function Inlining	• pragma HLS inline • pragma HLS function_instantiate

Table 11: **Vivado HLS Pragmas by Type** *(cont'd)*

Type	Attributes
Interface Synthesis	• pragma HLS interface • pragma HLS protocol
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurrence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_map • pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS data_pack

pragma HLS allocation

Description

Specifies instance restrictions to limit resource allocation in the implemented kernel. This defines, and can limit, the number of RTL instances and hardware resources used to implement specific functions, loops, operations or cores. The `ALLOCATION` pragma is specified inside the body of a function, a loop, or a region of code.

For example, if the C source has four instances of a function `foo_sub`, the `ALLOCATION` pragma can ensure that there is only one instance of `foo_sub` in the final RTL. All four instances of the C function are implemented using the same RTL block. This reduces resources utilized by the function, but negatively impacts performance.

The operations in the C code, such as additions, multiplications, array reads, and writes, can be limited by the `ALLOCATION` pragma. Cores, which operators are mapped to during synthesis, can be limited in the same manner as the operators. Instead of limiting the total number of multiplication operations, you can choose to limit the number of combinational multiplier cores, forcing any remaining multiplications to be performed using pipelined multipliers (or vice versa).

The ALLOCATION pragma applies to the scope it is specified within: a function, a loop, or a region of code. However, you can use the `-min_op` argument of the `config_bind` command to globally minimize operators throughout the design.

TIP: *For more information refer to "Controlling Hardware Resources" and `config_bind` in Vivado Design Suite User Guide: High-Level Synthesis (UG902).*

Syntax

Place the pragma inside the body of the function, loop, or region where it will apply.

```
#pragma HLS allocation instances=<list> \
limit=<value> <type>
```

Where:

- `instances=<list>`: Specifies the names of functions, operators, or cores.
- `limit=<value>`: Optionally specifies the limit of instances to be used in the kernel.
- `<type>`: Specifies that the allocation applies to a function, an operation, or a core (hardware component) used to create the design (such as adders, multipliers, pipelined multipliers, and block RAM). The type is specified as one of the following::
  - `function`: Specifies that the allocation applies to the functions listed in the `instances=` list. The function can be any function in the original C or C++ code that has NOT been:
    - Inlined by the `pragma HLS inline`, or the `set_directive_inline` command, or
    - Inlined automatically by Vivado HLS.
  - `operation`: Specifies that the allocation applies to the operations listed in the `instances=` list. Refer to *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for a complete list of the operations that can be limited using the `ALLOCATION` pragma.
  - `core`: Specifies that the `ALLOCATION` applies to the cores, which are the specific hardware components used to create the design (such as adders, multipliers, pipelined multipliers, and block RAM). The actual core to use is specified in the `instances=` option. In the case of cores, you can specify which the tool should use, or you can define a limit for the specified core.

Example 1

Given a design with multiple instances of function `foo`, this example limits the number of instances of `foo` in the RTL for the hardware kernel to 2.

```
#pragma HLS allocation instances=foo limit=2 function
```

Example 2

Limits the number of multiplier operations used in the implementation of the function `my_func` to 1. This limit does not apply to any multipliers outside of `my_func`, or multipliers that might reside in sub-functions of `my_func`.

TIP: *To limit the multipliers used in the implementation of any sub-functions, specify an allocation directive on the sub-functions or inline the sub-function into function `my_func`.*

```
void my_func(data_t angle) {
#pragma HLS allocation instances=mul limit=1 operation
...
}
```

See Also

- pragma HLS function_instantiate
- pragma HLS inline
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

pragma HLS array_map

Description

Combines multiple smaller arrays into a single large array to help reduce block RAM resources.

Designers typically use the `pragma HLS array_map` command (with the same `instance=` target) to combine multiple smaller arrays into a single larger array. This larger array can then be targeted to a single larger memory (RAM or FIFO) resource.

Each array is mapped into a block RAM or UltraRAM, when supported by the device. The basic block RAM unit provided in an FPGA is 18K. If many small arrays do not use the full 18K, a better use of the block RAM resources is to map many small arrays into a single larger array.

TIP: *If a block RAM is larger than 18K, they are automatically mapped into multiple 18K units.*

The ARRAY_MAP pragma supports two ways of mapping small arrays into a larger one:

- Horizontal mapping: this corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements.
- Vertical mapping: this corresponds to creating a new array by concatenating the original words in the array. Physically, this gets implemented as a single array with a larger bit-width.

The arrays are concatenated in the order that the pragmas are specified, starting at:

- Target element zero for horizontal mapping, or
- Bit zero for vertical mapping.

Syntax

Place the pragma in the C source within the boundaries of the function where the array variable is defined.

```
#pragma HLS array_map variable=<name> instance=<instance> \
<mode> offset=<int>
```

Where:

- `variable=<name>`: A required argument that specifies the array variable to be mapped into the new target array `<instance>`.
- `instance=<instance>`: Specifies the name of the new array to merge arrays into.
- `<mode>`: Optionally specifies the array map as being either `horizontal` or `vertical`.
  - Horizontal mapping is the default `<mode>`, and concatenates the arrays to form a new array with more elements.
  - Vertical mapping concatenates the array to form a new array with longer words.
- `offset=<int>`: Applies to horizontal type array mapping only. The offset specifies an integer value offset to apply before mapping the array into the new array `<instance>`. For example:
  - Element 0 of the array variable maps to element `<int>` of the new target.
  - Other elements map to `<int+1>`, `<int+2>`... of the new target.

IMPORTANT!: *If an offset is not specified, Vivado HLS calculates the required offset automatically to avoid overlapping array elements.*

Example 1

Arrays `array1` and `array2` in function `foo` are mapped into a single array, specified as `array3` in the following example:

```
void foo (...) {
int8 array1[M];
int12 array2[N];
#pragma HLS ARRAY_MAP variable=array1 instance=array3 horizontal
#pragma HLS ARRAY_MAP variable=array2 instance=array3 horizontal
...
loop_1: for(i=0;i<M;i++) {
array1[i] = ...;
array2[i] = ...;
...
}
...
}
```

Example 2

This example provides a horizontal mapping of array A[10] and array B[15] in function `foo` into a single new array AB[25].

- Element AB[0] will be the same as A[0].
- Element AB[10] will be the same as B[0] because no `offset=` option is specified.
- The bit-width of array AB[25] will be the maximum bit-width of either A[10] or B[15].

```
#pragma HLS array_map variable=A instance=AB horizontal
#pragma HLS array_map variable=B instance=AB horizontal
```

Example 3

The following example performs a vertical concatenation of arrays C and D into a new array CD, with the bit-width of C and D combined. The number of elements in CD is the maximum of the original arrays, C or D:

```
#pragma HLS array_map variable=C instance=CD vertical
#pragma HLS array_map variable=D instance=CD vertical
```

See Also

- pragma HLS array_partition
- pragma HLS array_reshape
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

pragma HLS array_partition

Description

Partitions an array into smaller arrays or individual elements.

This partitioning:

- Results in RTL with multiple small memories or multiple registers instead of one large memory.
- Effectively increases the amount of read and write ports for the storage.
- Potentially improves the throughput of the design.
- Requires more memory instances or registers.

Syntax

Place the pragma in the C source within the boundaries of the function where the array variable is defined.

```
#pragma HLS array_partition variable=<name> \
<type>  factor=<int>  dim=<int>
```

where

- `variable=<name>`: A required argument that specifies the array variable to be partitioned.
- `<type>`: Optionally specifies the partition type. The default type is `complete`. The following types are supported:
  - `cyclic`: Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. For example, if `factor=3` is used:
    - Element 0 is assigned to the first new array
    - Element 1 is assigned to the second new array.
    - Element 2 is assigned to the third new array.
    - Element 3 is assigned to the first new array again.
  - `block`: Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the `factor=` argument.
  - `complete`: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. This is the default `<type>`.

- `factor=<int>`: Specifies the number of smaller arrays that are to be created.

IMPORTANT!: *For complete type partitioning, the factor is not specified. For block and cyclic partitioning the `factor=` is required.*

- `dim=<int>`: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to `N`, for an array with `N` dimensions:
  - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
  - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.

Example 1

This example partitions the 13 element array, AB[13], into four arrays using block partitioning:

```
#pragma HLS array_partition variable=AB block factor=4
```

TIP:
Because four is not an integer factor of 13:

- *Three of the new arrays have three elements each,*
- *One array has four elements (AB[9:12]).*

Example 2

This example partitions dimension two of the two-dimensional array, AB[6][4] into two new arrays of dimension [6][2]:

```
#pragma HLS array_partition variable=AB block factor=2 dim=2
```

Example 3

This example partitions the second dimension of the two-dimensional `in_local` array into individual elements.

```
int in_local[MAX_SIZE][MAX_DIM];
        #pragma HLS ARRAY_PARTITION variable=in_local complete dim=2
```

See Also

- pragma HLS array_map
- pragma HLS array_reshape
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

- xcl_array_partition
- *SDAccel Environment Optimization Guide* (UG1207)

pragma HLS array_reshape

Description

Combines array partitioning with vertical array mapping.

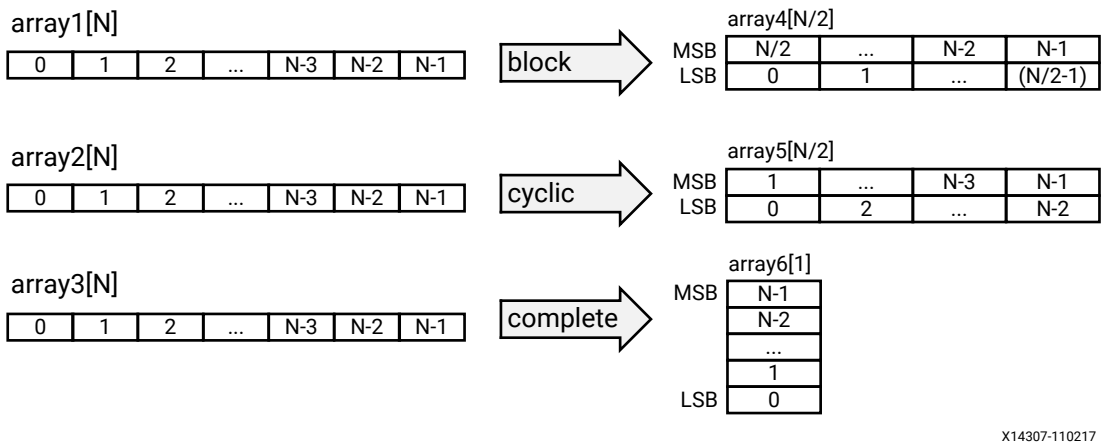
The `ARRAY_RESHAPE` pragma combines the effect of `ARRAY_PARTITION`, breaking an array into smaller arrays, with the effect of the vertical type of `ARRAY_MAP`, concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing the primary benefit of partitioning: parallel access to the data. This pragma creates a new array with fewer elements but with greater bit-width, allowing more data to be accessed in a single clock cycle.

Given the following code:

```
void foo (...) {
int array1[N];
int array2[N];
int array3[N];
#pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
...
}
```

The ARRAY_RESHAPE pragma transforms the arrays into the form shown in the following figure:

Figure 2: **ARRAY_RESHAPE Pragma**



Syntax

Place the pragma in the C source within the region of a function where the array variable is defines.

```
#pragma HLS array_reshape variable=<name> \
<type>  factor=<int>  dim=<int>
```

Where:

- *<name>*: A required argument that specifies the array variable to be reshaped.
- *<type>*: Optionally specifies the partition type. The default type is `complete`. The following types are supported:
  - `cyclic`: Cyclic reshaping creates smaller arrays by interleaving elements from the original array. For example, if `factor=3` is used, element 0 is assigned to the first new array, element 1 to the second new array, element 2 is assigned to the third new array, and then element 3 is assigned to the first new array again. The final array is a vertical concatenation (word concatenation, to create longer words) of the new arrays into a single array.
  - `block`: Block reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks where N is the integer defined by `factor=`, and then combines the N blocks into a single array with `word-width*N`.
  - `complete`: Complete reshaping decomposes the array into temporary individual elements and then recombines them into an array with a wider word. For a one-dimension array this is equivalent to creating a very-wide register (if the original array was N elements of M bits, the result is a register with `N*M` bits). This is the default type of array reshaping.
- `factor=`*<int>*: Specifies the amount to divide the current array by (or the number of temporary arrays to create). A factor of 2 splits the array in half, while doubling the bit-width. A factor of 3 divides the array into three, with triple the bit-width.

IMPORTANT!: *For complete type partitioning, the factor is not specified. For block and cyclic reshaping the `factor=` is required.*

- `dim=`*<int>*: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to *N*, for an array with *N* dimensions:
  - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
  - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.
- `object`: A keyword relevant for container arrays only. When the keyword is specified the `ARRAY_RESHAPE` pragma applies to the objects in the container, reshaping all dimensions of the objects within the container, but all dimensions of the container itself are preserved. When the keyword is not specified the pragma applies to the container array and not the objects.

Example 1

Reshapes (partition and maps) an 8-bit array with 17 elements, AB[17], into a new 32-bit array with five elements using block mapping.

```
#pragma HLS array_reshape variable=AB block factor=4
```

TIP: *factor=4 indicates that the array should be divided into four. So 17 elements is reshaped into an array of 5 elements, with four times the bit-width. In this case, the last element, AB[17], is mapped to the lower eight bits of the fifth element, and the rest of the fifth element is empty.*

Example 2

Reshapes the two-dimensional array AB[6][4] into a new array of dimension [6][2], in which dimension 2 has twice the bit-width:

```
#pragma HLS array_reshape variable=AB block factor=2 dim=2
```

Example 3

Reshapes the three-dimensional 8-bit array, AB[4][2][2] in function `foo`, into a new single element array (a register), 128 bits wide (4*2*2*8):

```
#pragma HLS array_reshape variable=AB complete dim=0
```

TIP: *dim=0 means to reshape all dimensions of the array.*

See Also

- pragma HLS array_map
- pragma HLS array_partition
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
- *SDAccel Environment Profiling and Optimization Guide* (UG1207)

pragma HLS clock

Description

Applies the named clock to the specified function.

C and C++ designs support only a single clock. The clock period specified by `create_clock` is applied to all functions in the design.

SystemC designs support multiple clocks. Multiple named clocks can be specified using the `create_clock` command and applied to individual SC_MODULEs using `pragma HLS clock`. Each SC_MODULE is synthesized using a single clock.

Syntax

Place the pragma in the C source within the body of the function.

```
#pragma HLS clock domain=<clock>
```

Where:

- domain=`<clock>`: Specifies the clock name.

IMPORTANT!: *The specified clock must already exist by the `create_clock` command. There is no pragma equivalent of the create_clock command. See the Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more information.*

Example 1

Assume a SystemC design in which the top-level, `foo_top`, has clocks ports `fast_clock` and `slow_clock`. However, `foo_top` uses only `fast_clock` within its function. A sub-block, `foo_sub`, uses only `slow_clock`.

In this example, the following `create_clock` commands are specified in the `script.tcl` file which is specified when the Vivado HLS tool is launched:

```
create_clock -period 15 fast_clk
create_clock -period 60 slow_clk
```

Then the following pragmas are specified in the C source file to assign the clock to the specified functions, `foo_sub` and `foo_top`:

```
foo_sub (p, q) {
  #pragma HLS clock domain=slow_clock
  ...
}
void foo_top { a, b, c, d} {
  #pragma HLS clock domain=fast_clock
  ...
```

See Also

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

pragma HLS data_pack

Description

Packs the data fields of a `struct` into a single scalar with a wider word width.

The `DATA_PACK` pragma is used for packing all the elements of a `struct` into a single wide vector to reduce the memory required for the variable, while allowing all members of the `struct` to be read and written to simultaneously. The bit alignment of the resulting new wide-word can be inferred from the declaration order of the `struct` fields. The first field takes the LSB of the vector, and the final element of the `struct` is aligned with the MSB of the vector.

If the `struct` contains arrays, the `DATA_PACK` pragma performs a similar operation as the `ARRAY_RESHAPE` pragma and combines the reshaped array with the other elements in the `struct`. Any arrays declared inside the `struct` are completely partitioned and reshaped into a wide scalar and packed with other scalar fields. However, a `struct` cannot be optimized with `DATA_PACK` and `ARRAY_PARTITION` or `ARRAY_RESHAPE`, as those pragmas are mutually exclusive.

IMPORTANT!: *You should exercise some caution when using the `DATA_PACK` optimization on `struct` objects with large arrays. If an array has 4096 elements of type int, this will result in a vector (and port) of width 4096*32=131072 bits. Vivado HLS can create this RTL design, however it is very unlikely logic synthesis will be able to route this during the FPGA implementation.*

In general, Xilinx recommends that you use arbitrary precision (or bit-accurate) data types. Standard C types are based on 8-bit boundaries (8-bit, 16-bit, 32-bit, 64-bit), however, using arbitrary precision data types in a design lets you specify the exact bit-sizes in the C code prior to synthesis. The bit-accurate widths result in hardware operators that are smaller and faster. This allows more logic to be placed in the FPGA and for the logic to execute at higher clock frequencies. However, the `DATA_PACK` pragma also lets you align data in the packed `struct` along 8-bit boundaries if needed.

If a `struct` port is to be implemented with an AXI4 interface you should consider using the `DATA_PACK <byte_pad>` option to automatically align member elements of the `struct` to 8-bit boundaries. The AXI4-Stream protocol requires that `TDATA` ports of the IP have a width in multiples of 8. It is a specification violation to define an AXI4-Stream IP with a `TDATA` port width that is not a multiple of 8, therefore, it is a requirement to round up `TDATA` widths to byte multiples. Refer to "Interface Synthesis and Structs" in *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information.

Syntax

Place the pragma near the definition of the `struct` variable to pack:

```
#pragma HLS data_pack variable=<variable> \
instance=<name> <byte_pad>
```

Where:

- `variable=<variable>`: is the variable to be packed.
- `instance=<name>`: Specifies the name of resultant variable after packing. If no `<name>` is specified, the input `<variable>` is used.
- `<byte_pad>`: Optionally specifies whether to pack data on an 8-bit boundary (8-bit, 16-bit, 24-bit...). The two supported values for this option are:
  - `struct_level`: Pack the whole `struct` first, then pad it upward to the next 8-bit boundary.
  - `field_level`: First pad each individual element (field) of the `struct` on an 8-bit boundary, then pack the `struct`.

TIP: *Deciding whether multiple fields of data should be concatenated together before (`field_level`) or after (`struct_level`) alignment to byte boundaries is generally determined by considering how atomic the data is. Atomic information is data that can be interpreted on its own, whereas non-atomic information is incomplete for the purpose of interpreting the data. For example, atomic data can consist of all the bits of information in a floating point number. However, the exponent bits in the floating point number alone would not be atomic. When packing information into `TDATA`, generally non-atomic bits of data are concatenated together (regardless of bit width) until they form atomic units. The atomic units are then aligned to byte boundaries using pad bits where necessary.*

Example 1

Packs `struct` array AB[17] with three 8-bit field fields (R, G, B) into a new 17 element array of 24 bits.

```
typedef struct{
unsigned char R, G, B;
} pixel;

pixel AB[17];
#pragma HLS data_pack variable=AB
```

Example 2

Packs struct pointer AB with three 8-bit fields (typedef struct {unsigned char R, G, B;} pixel) in function `foo`, into a new 24-bit pointer.

```
typedef struct{
unsigned char R, G, B;
} pixel;

pixel AB;
#pragma HLS data_pack variable=AB
```

Example 3

In this example the `DATA_PACK` pragma is specified for `in` and `out` arguments to `rgb_to_hsv` function to instruct the compiler to do pack the structure on an 8-bit boundary to improve the memory access:

```
void rgb_to_hsv(RGBcolor* in,  // Access global memory as RGBcolor struct-
wise
                HSVcolor* out, // Access Global Memory as HSVcolor struct-
wise
                int size) {
#pragma HLS data_pack variable=in struct_level
#pragma HLS data_pack variable=out struct_level
...
}
```

See Also

- pragma HLS array_partition
- pragma HLS array_reshape
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

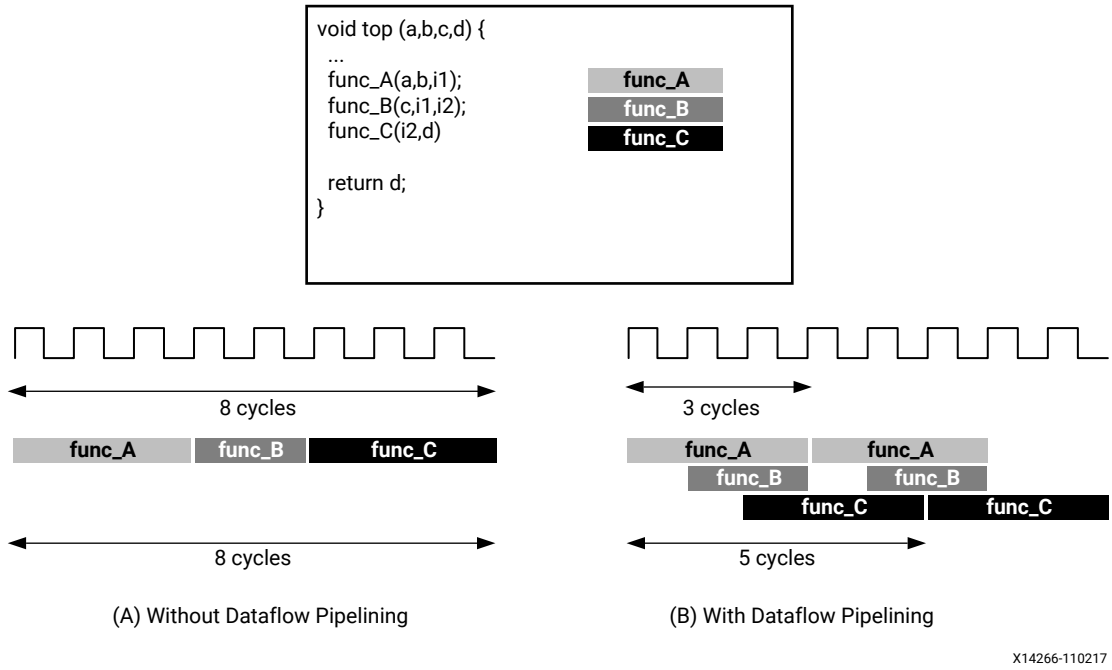
pragma HLS dataflow

Description

The `DATAFLOW` pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design.

All operations are performed sequentially in a C description. In the absence of any directives that limit resources (such as `pragma HLS allocation`), Vivado HLS seeks to minimize latency and improve concurrency. However, data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation. The `DATAFLOW` optimization enables the operations in a function or loop to start operation before the previous function or loop completes all its operations.

Figure 3: **DATAFLOW Pragma**



When the `DATAFLOW` pragma is specified, Vivado HLS analyzes the dataflow between sequential functions or loops and create channels (based on pingpong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, Vivado HLS attempts to minimize the initiation interval and start operation as soon as data is available.

TIP: *The `config_dataflow` command specifies the default memory channel and FIFO depth used in dataflow optimization. Refer to the `config_dataflow` command in the Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more information.*

For the `DATAFLOW` optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent Vivado HLS from performing the `DATAFLOW` optimization:

- Single-producer-consumer violations
- Bypassing tasks
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions

IMPORTANT!: *If any of these coding styles are present, Vivado HLS issues a message and does not perform DATAFLOW optimization.*

Finally, the `DATAFLOW` optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the `DATAFLOW` optimization, you must apply the optimization to the loop, the sub-function, or inline the sub-function.

Syntax

Place the pragma in the C source within the boundaries of the region, function, or loop.

```
#pragma HLS dataflow
```

Example 1

Specifies `DATAFLOW` optimization within the loop `wr_loop_j`.

```
        wr_loop_j: for (int j = 0; j < TILE_PER_ROW; ++j) {
#pragma HLS DATAFLOW
            wr_buf_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
                wr_buf_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
#pragma HLS PIPELINE
                    // should burst TILE_WIDTH in WORD beat
                    outFifo >> tile[m][n];
                }
            }
            wr_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
                wr_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
#pragma HLS PIPELINE
                    outx[TILE_HEIGHT*TILE_PER_ROW*TILE_WIDTH*i
+TILE_PER_ROW*TILE_WIDTH*m+TILE_WIDTH*j+n] = tile[m][n];
                }
            }
```

See Also

- pragma HLS allocation
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
- xcl_dataflow
- *SDAccel Environment Optimization Guide* (UG1207)

pragma HLS dependence

Description

The `DEPENDENCE` pragma is used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).

Vivado HLS automatically detects dependencies:

- Within loops (loop-independent dependence), or
- Between different iterations of a loop (loop-carry dependence).

These dependencies impact when operations can be scheduled, especially during function and loop pipelining.

- Loop-independent dependence: The same element is accessed in the same loop iteration.

```
for (i=0;i<N;i++) {
 A[i]=x;
 y=A[i];
}
```

- Loop-carry dependence: The same element is accessed in a different loop iteration.

```
for (i=0;i<N;i++) {
 A[i]=A[i-1]*2;
}
```

Under certain complex scenarios automatic dependence analysis can be too conservative and fail to filter out false dependencies. Under certain circumstances, such as variable dependent array indexing, or when an external requirement needs to be enforced (for example, two inputs are never the same index), the dependence analysis might be too conservative. The `DEPENDENCE` pragma allows you to explicitly specify the dependence and resolve a false dependence.

IMPORTANT!: *Specifying a false dependency, when in fact the dependency is not false, can result in incorrect hardware. Be sure dependencies are correct (true or false) before specifying them.*

Syntax

Place the pragma within the boundaries of the function where the dependence is defined.

```
#pragma HLS dependence variable=<variable> <class> \
<type> <direction> distance=<int> <dependent>
```

Where:

- `variable=<variable>`: Optionally specifies the variable to consider for the dependence.
- `<class>`: Optionally specifies a class of variables in which the dependence needs clarification. Valid values include `array` or `pointer`.

TIP: *`<class>` and `variable=` do not need to be specified together as you can either specify a variable or a class of variables within a function.*

- `<type>`: Valid values include `intra` or `inter`. Specifies whether the dependence is:
  - `intra`: dependence within the same loop iteration. When dependence `<type>` is specified as `intra`, and `<dependent>` is false, Vivado HLS may move operations freely within a loop, increasing their mobility and potentially improving performance or area. When `<dependent>` is specified as true, the operations must be performed in the order specified.
  - `inter`: dependence between different loop iterations. This is the default `<type>`. If dependence `<type>` is specified as `inter`, and `<dependent>` is false, it allows Vivado HLS to perform operations in parallel if the function or loop is pipelined, or the loop is unrolled, or partially unrolled, and prevents such concurrent operation when `<dependent>` is specified as true.
- `<direction>`: Valid values include `RAW`, `WAR`, or `WAW`. This is relevant for loop-carry dependencies only, and specifies the direction for a dependence:
  - `RAW` (Read-After-Write - true dependence) The write instruction uses a value used by the read instruction.
  - `WAR` (Write-After-Read - anti dependence) The read instruction gets a value that is overwritten by the write instruction.
  - `WAW` (Write-After-Write - output dependence) Two write instructions write to the same location, in a certain order.
- `distance=<int>`: Specifies the inter-iteration distance for array access. Relevant only for loop-carry dependencies where dependence is set to `true`.
- `<dependent>`: Specifies whether a dependence needs to be enforced (`true`) or removed (`false`). The default is `true`.

Example 1

In the following example, Vivado HLS does not have any knowledge about the value of `cols` and conservatively assumes that there is always a dependence between the write to `buff_A[1][col]` and the read from `buff_A[1][col]`. In an algorithm such as this, it is unlikely `cols` will ever be zero, but Vivado HLS cannot make assumptions about data dependencies. To overcome this deficiency, you can use the `DEPENDENCE` pragma to state that there is no dependence between loop iterations (in this case, for both `buff_A` and `buff_B`).

```
void foo(int rows, int cols, ...)
  for (row = 0; row < rows + 1; row++) {
    for (col = 0; col < cols + 1; col++) {
      #pragma HLS PIPELINE II=1
      #pragma HLS dependence variable=buff_A inter false
      #pragma HLS dependence variable=buff_B inter false
      if (col < cols) {
      buff_A[2][col] = buff_A[1][col]; // read from buff_A[1][col]
      buff_A[1][col] = buff_A[0][col]; // write to buff_A[1][col]
      buff_B[1][col] = buff_B[0][col];
      temp = buff_A[0][col];
}
```

Example 2

Removes the dependence between `Var1` in the same iterations of `loop_1` in function `foo`.

```
#pragma HLS dependence variable=Var1 intra false
```

Example 3

Defines the dependence on all arrays in `loop_2` of function `foo` to inform Vivado HLS that all reads must happen after writes (RAW) in the same loop iteration.

```
#pragma HLS dependence array intra RAW true
```

See Also

- pragma HLS pipeline
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
- xcl_pipeline_loop
- *SDAccel Environment Optimization Guide* (UG1207)

pragma HLS expression_balance

Description

Sometimes a C-based specification is written with a sequence of operations resulting in a long chain of operations in RTL. With a small clock period, this can increase the latency in the design. By default, Vivado HLS rearranges the operations using associative and commutative properties. This rearrangement creates a balanced tree that can shorten the chain, potentially reducing latency in the design at the cost of extra hardware.

The `EXPRESSION_BALANCE` pragma allows this expression balancing to be disabled, or to be expressly enabled, within a specified scope.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS expression_balance off
```

Where:

- `off`: Turns off expression balancing at this location.

TIP: *Leaving this option out of the pragma enables expression balancing, which is the default mode.*

Example 1

This example explicitly enables expression balancing in function `my_Func`:

```
void my_func(char inval, char incr) {
  #pragma HLS expression_balance
```

Example 2

Disables expression balancing within function `my_Func`:

```
void my_func(char inval, char incr) {
  #pragma HLS expression_balance off
```

See Also

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

pragma HLS function_instantiate

Description

The FUNCTION_INSTANTIATE pragma is an optimization technique that has the area benefits of maintaining the function hierarchy but provides an additional powerful option: performing targeted local optimizations on specific instances of a function. This can simplify the control logic around the function call and potentially improve latency and throughput.

By default:

- Functions remain as separate hierarchy blocks in the RTL.
- All instances of a function, at the same level of hierarchy, make use of a single RTL implementation (block).

The FUNCTION_INSTANTIATE pragma is used to create a unique RTL implementation for each instance of a function, allowing each instance to be locally optimized according to the function call. This pragma exploits the fact that some inputs to a function may be a constant value when the function is called, and uses this to both simplify the surrounding control structures and produce smaller more optimized function blocks.

Without the FUNCTION_INSTANTIATE pragma, the following code results in a single RTL implementation of function foo_sub for all three instances of the function in foo. Each instance of function foo_sub is implemented in an identical manner. This is fine for function reuse and reducing the area required for each instance call of a function, but means that the control logic inside the function must be more complex to account for the variation in each call of foo_sub.

```
char foo_sub(char inval, char incr) {
#pragma HLS function_instantiate variable=incr
 return inval + incr;
}
void foo(char inval1, char inval2, char inval3,
 char *outval1, char *outval2, char * outval3)
{
 *outval1 = foo_sub(inval1, 1);
 *outval2 = foo_sub(inval2, 2);
 *outval3 = foo_sub(inval3, 3);
}
```

In the code sample above, the FUNCTION_INSTANTIATE pragma results in three different implementations of function foo_sub, each independently optimized for the incr argument, reducing the area and improving the performance of the function. After FUNCTION_INSTANTIATE optimization, foo_sub is effectively be transformed into three separate functions, each optimized for the specified values of incr.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS function_instantiate variable=<variable>
```

Where:

- `variable=<variable>`: A required argument that defines the function argument to use as a constant.

Example 1

In the following example, the FUNCTION_INSTANTIATE pragma placed in function `swInt`) allows each instance of function `swInt` to be independently optimized with respect to the `maxv` function argument:

```
void swInt(unsigned int *readRefPacked, short *maxr, short *maxc, short
*maxv){
#pragma HLS function_instantiate variable=maxv
    uint2_t d2bit[MAXCOL];
    uint2_t q2bit[MAXROW];
#pragma HLS array partition variable=d2bit,q2bit cyclic factor=FACTOR

    intTo2bit<MAXCOL/16>((readRefPacked + MAXROW/16), d2bit);
    intTo2bit<MAXROW/16>(readRefPacked, q2bit);
    sw(d2bit, q2bit, maxr, maxc, maxv);
}
```

See Also

- pragma HLS allocation
- pragma HLS inline
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

pragma HLS inline

Description

Removes a function as a separate entity in the hierarchy. After inlining, the function is dissolved into the calling function and no longer appears as a separate level of hierarchy in the RTL. In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with surrounding operations. An inlined function cannot be shared. This can increase area required for implementing the RTL.

The INLINE pragma applies differently to the scope it is defined in depending on how it is specified:

- INLINE: Without arguments, the pragma means that the function it is specified in should be inlined upward into any calling functions or regions.
- INLINE OFF: Specifies that the function it is specified in should NOT be inlined upward into any calling functions or regions. This disables the inline of a specific function that may be automatically inlined, or inlined as part of a region or recursion.
- INLINE REGION: This applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function, but not inlining recursively through the hierarchy.
- INLINE RECURSIVE: This applies the pragma to the region or the body of the function it is assigned in. It applies downward, recursively inlining the contents of the region or function.

By default, inlining is only performed on the next level of function hierarchy, not sub-functions. However, the recursive option lets you specify inlining through levels of the hierarchy.

Syntax

Place the pragma in the C source within the body of the function or region of code.

```
#pragma HLS inline <region | recursive | off>
```

Where:

- region: Optionally specifies that all functions in the specified region (or contained within the body of the function) are to be inlined, applies to the scope of the region.
- recursive: By default, only one level of function inlining is performed, and functions within the specified function are not inlined. The recursive option inlines all functions recursively within the specified function or region.
- off: Disables function inlining to prevent specified functions from being inlined. For example, if recursive is specified in a function, this option can prevent a particular called function from being inlined when all others are.

TIP: *Vivado HLS automatically inlines small functions and using the INLINE pragma with the off option may be used to prevent this automatic inlining.*

Example 1

This example inlines all functions within the region it is specified in, in this case the body of foo_top, but does not inline any lower level functions within those functions.

```
void foo_top { a, b, c, d} {
  #pragma HLS inline region
  ...
```

Example 2

The following example, inlines all functions within the body of `foo_top`, inlining recursively down through the function hierarchy, except function `foo_sub` is not inlined. The recursive pragma is placed in function `foo_top`. The pragma to disable inlining is placed in the function `foo_sub`:

```
foo_sub (p, q) {
#pragma HLS inline off
int q1 = q + 10;
foo(p1,q);// foo_3
...
}
void foo_top { a, b, c, d} {
  #pragma HLS inline region recursive
  ...
  foo(a,b);//foo_1
  foo(a,c);//foo_2
  foo_sub(a,d);
  ...
}
```

***Note*:** Notice in this example, that INLINE applies downward to the contents of function `foo_top`, but applies upward to the code calling `foo_sub`.

Example 3

This example inlines the `copy_output` function into any functions or regions calling `copy_output`.

```
void copy_output(int *out, int out_lcl[OSize * OSize], int output) {
#pragma HLS INLINE
    // Calculate each work_item's result update location
    int stride = output * OSize * OSize;

    // Work_item updates output filter/image in DDR
    writeOut: for(int itr = 0; itr < OSize * OSize; itr++) {
    #pragma HLS PIPELINE
        out[stride + itr] = out_lcl[itr];
    }
```

See Also

- pragma HLS allocation
- pragma HLS function_instantiate
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

pragma HLS interface

Description

In C based design, all input and output operations are performed, in zero time, through formal function arguments. In an RTL design these same input and output operations must be performed through a port in the design interface and typically operate using a specific I/O (input-output) protocol. For more information, refer to "Managing Interfaces" in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

The `INTERFACE` pragma specifies how RTL ports are created from the function definition during interface synthesis.

The ports in the RTL implementation are derived from:

- Any function-level protocol that is specified.
- Function arguments.
- Global variables accessed by the top-level function and defined outside its scope.

Function-level protocols, also called block-level I/O protocols, provide signals to control when the function starts operation, and indicate when function operation ends, is idle, and is ready for new inputs. The implementation of a function-level protocol:

- Is specified by the *<mode>* values `ap_ctrl_none`, `ap_ctrl_hs` or `ap_ctrl_chain`. The `ap_ctrl_hs` block-level I/O protocol is the default.
- Are associated with the function name.

Each function argument can be specified to have its own port-level (I/O) interface protocol, such as valid handshake (`ap_vld`) or acknowledge handshake (`ap_ack`). Port Level interface protocols are created for each argument in the top-level function and the function return, if the function returns a value. The default I/O protocol created depends on the type of C argument. After the block-level protocol has been used to start the operation of the block, the port-level IO protocols are used to sequence data into and out of the block.

If a global variable is accessed, but all read and write operations are local to the design, the resource is created in the design. There is no need for an I/O port in the RTL. If the global variable is expected to be an external source or destination, specify its interface in a similar manner as standard function arguments. See the examples below.

When the `INTERFACE` pragma is used on sub-functions, only the `register` option can be used. The *<mode>* option is not supported on sub-functions.

TIP: *Vivado HLS automatically determines the I/O protocol used by any sub-functions. You cannot control these ports except to specify whether the port is registered.*

Syntax

Place the pragma within the boundaries of the function.

```
#pragma HLS interface <mode> port=<name> bundle=<string> \
register register_mode=<mode> depth=<int> offset=<string> \
clock=<string> name=<string> \
num_read_outstanding=<int> num_write_outstanding=<int> \
max_read_burst_length=<int> max_write_burst_length=<int>
```

Where:

- *<mode>*: Specifies the interface protocol mode for function arguments, global variables used by the function, or the block-level control protocols. For detailed descriptions of these different modes see "Interface Synthesis Reference" in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902). The mode can be specified as one of the following:
  - `ap_none`: No protocol. The interface is a data port.
  - `ap_stable`: No protocol. The interface is a data port. Vivado HLS assumes the data port is always stable after reset, which allows internal optimizations to remove unnecessary registers.
  - `ap_vld`: Implements the data port with an associated `valid` port to indicate when the data is valid for reading or writing.
  - `ap_ack`: Implements the data port with an associated `acknowledge` port to acknowledge that the data was read or written.
  - `ap_hs`: Implements the data port with associated `valid` and `acknowledge` ports to provide a two-way handshake to indicate when the data is valid for reading and writing and to acknowledge that the data was read or written.
  - `ap_ovld`: Implements the output data port with an associated `valid` port to indicate when the data is valid for reading or writing.

IMPORTANT!: *Vivado HLS implements the input argument or the input half of any read/write arguments with mode `ap_none`.*

  - `ap_fifo`: Implements the port with a standard FIFO interface using data input and output ports with associated active-Low FIFO `empty` and `full` ports.

Note: You can only use this interface on read arguments or write arguments. The `ap_fifo` mode does not support bidirectional read/write arguments.

  - `ap_bus`: Implements pointer and pass-by-reference ports as a bus interface.
  - `ap_memory`: Implements array arguments as a standard RAM interface. If you use the RTL design in Vivado IP integrator, the memory interface appears as discrete ports.

  - `bram`: Implements array arguments as a standard RAM interface. If you use the RTL design in Vivado IP integrator, the memory interface appears as a single port.
- `axis`: Implements all ports as an AXI4-Stream interface.
  - `s_axilite`: Implements all ports as an AXI4-Lite interface. Vivado HLS produces an associated set of C driver files during the Export RTL process.
  - `m_axi`: Implements all ports as an AXI4 interface. You can use the `config_interface` command to specify either 32-bit (default) or 64-bit address ports and to control any address offset.
  - `ap_ctrl_none`: No block-level I/O protocol.

***Note*:** Using the `ap_ctrl_none` mode might prevent the design from being verified using the C/RTL co-simulation feature.

  - `ap_ctrl_hs`: Implements a set of block-level control ports to `start` the design operation and to indicate when the design is `idle`, `done`, and `ready` for new input data.

***Note*:** The `ap_ctrl_hs` mode is the default block-level I/O protocol.

  - `ap_ctrl_chain`: Implements a set of block-level control ports to `start` the design operation, `continue` operation, and indicate when the design is `idle`, `done`, and `ready` for new input data.

***Note*:** The `ap_ctrl_chain` interface mode is similar to `ap_ctrl_hs` but provides an additional input signal `ap_continue` to apply back pressure. Xilinx recommends using the `ap_ctrl_chain` block-level I/O protocol when chaining Vivado HLS blocks together.

- `port=<name>`: Specifies the name of the function argument, function return, or global variable which the `INTERFACE` pragma applies to.

TIP: *Block-level I/O protocols (`ap_ctrl_none`, `ap_ctrl_hs`, or `ap_ctrl_chain`) can be assigned to a port for the function `return` value.*

- `bundle=<string>`: Groups function arguments into AXI interface ports. By default, Vivado HLS groups all function arguments specified as an AXI4-Lite (`s_axilite`) interface into a single AXI4-Lite port. Similarly, all function arguments specified as an AXI4 (`m_axi`) interface are grouped into a single AXI4 port. This option explicitly groups all interface ports with the same `bundle=<string>` into the same AXI interface port and names the RTL port the value specified by `<string>`.
- `register`: An optional keyword to register the signal and any relevant protocol signals, and causes the signals to persist until at least the last cycle of the function execution. This option applies to the following interface modes:
  - `ap_none`
  - `ap_ack`
  - `ap_vld`
  - `ap_ovld`

    - ap_hs
    - ap_stable
    - axis
    - s_axilite

TIP: *The* `-register_io` *option of the* `config_interface` *command globally controls registering all inputs/outputs on the top function. Refer to Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more information.*

- `register_mode= <forward|reverse|both|off>`: Used with the `register` keyword, this option specifies if registers are placed on the `forward` path (TDATA and TVALID), the `reverse` path (TREADY), on `both` paths (TDATA, TVALID, and TREADY), or if none of the port signals are to be registered (`off`). The default `register_mode` is `both`. AXI-Stream (`axis`) side-channel signals are considered to be data signals and are registered whenever the TDATA is registered.
- `depth=<int>`: Specifies the maximum number of samples for the test bench to process. This setting indicates the maximum size of the FIFO needed in the verification adapter that Vivado HLS creates for RTL co-simulation.

TIP: *While* `depth` *is usually an option, it is required for* `m_axi` *interfaces.*

- `offset=<string>`: Controls the address offset in AXI4-Lite (`s_axilite`) and AXI4 (`m_axi`) interfaces.
    - For the `s_axilite` interface, *<string>* specifies the address in the register map.
    - For the `m_axi` interface, *<string>* specifies on of the following values:
        - `direct`: Generate a scalar input offset port.
        - `slave`: Generate an offset port and automatically map it to an AXI4-Lite slave interface.
        - `off`: Do not generate an offset port.

TIP: *The* `-m_axi_offset` *option of the* `config_interface` *command globally controls the offset ports of all M_AXI interfaces in the design.*

- `clock=<name>`: Optionally specified only for interface mode `s_axilite`. This defines the clock signal to use for the interface. By default, the AXI-Lite interface clock is the same clock as the system clock. This option is used to specify a separate clock for the AXI-Lite (`s_axilite`) interface.

TIP: *If the* `bundle` *option is used to group multiple top-level function arguments into a single AXI-Lite interface, the clock option need only be specified on one of the bundle members.*

- `num_read_outstanding=<int>`: For AXI4 (`m_axi`) interfaces, this option specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size: `num_read_outstanding*max_read_burst_length*word_size`.

- `num_write_outstanding=<int>`: For AXI4 (`m_axi`) interfaces, this option specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size: `num_write_outstanding*max_write_burst_length*word_size`
- `max_read_burst_length=<int>`: For AXI4 (`m_axi`) interfaces, this option specifies the maximum number of data values read during a burst transfer.
- `max_write_burst_length=<int>`: For AXI4 (`m_axi`) interfaces, this option specifies the maximum number of data values written during a burst transfer.
- `name=<string>`: This option is used to rename the port based on your own specification. The generated RTL port will use this name.

Example 1

In this example, both function arguments are implemented using an AXI4-Stream interface:

```
void example(int A[50], int B[50]) {
  //Set the HLS native interface types
  #pragma HLS INTERFACE axis port=A
  #pragma HLS INTERFACE axis port=B
  int i;
  for(i = 0; i < 50; i++){
    B[i] = A[i] + 5;
  }
}
```

Example 2

The following turns off block-level I/O protocols, and is assigned to the function return value:

```
#pragma HLS interface ap_ctrl_none port=return
```

The function argument `InData` is specified to use the `ap_vld` interface, and also indicates the input should be registered:

```
#pragma HLS interface ap_vld register port=InData
```

This exposes the global variable `lookup_table` as a port on the RTL design, with an `ap_memory` interface:

```
pragma HLS interface ap_memory port=lookup_table
```

Example 3

This example defines the INTERFACE standards for the ports of the top-level `transpose` function. Notice the use of the `bundle=` option to group signals.

```
// TOP LEVEL - TRANSPOSE
void transpose(int* input, int* output) {
    #pragma HLS INTERFACE m_axi port=input offset=slave bundle=gmem0
    #pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem1

    #pragma HLS INTERFACE s_axilite port=input bundle=control
    #pragma HLS INTERFACE s_axilite port=output bundle=control
    #pragma HLS INTERFACE s_axilite port=return bundle=control

    #pragma HLS dataflow
```

See Also

- pragma HLS protocol
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

pragma HLS latency

Description

Specifies a minimum or maximum latency value, or both, for the completion of functions, loops, and regions. Latency is defined as the number of clock cycles required to produce an output. Function latency is the number of clock cycles required for the function to compute all output values, and return. Loop latency is the number of cycles to execute all iterations of the loop. See "Performance Metrics Example" of *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Vivado HLS always tries to the minimize latency in the design. When the LATENCY pragma is specified, the tool behavior is as follows:

- Latency is greater than the minimum, or less than the maximum: The constraint is satisfied. No further optimizations are performed.
- Latency is less than the minimum: If Vivado HLS can achieve less than the minimum specified latency, it extends the latency to the specified value, potentially increasing sharing.
- Latency is greater than the maximum: If Vivado HLS cannot schedule within the maximum limit, it increases effort to achieve the specified constraint. If it still fails to meet the maximum latency, it issues a warning, and produces a design with the smallest achievable latency in excess of the maximum.

TIP: *You can also use the LATENCY pragma to limit the efforts of the tool to find an optimum solution. Specifying latency constraints for scopes within the code: loops, functions, or regions, reduces the possible solutions within that scope and improves tool runtime. Refer to "Improving Run Time and Capacity" of Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more information.*

Syntax

Place the pragma within the boundary of a function, loop, or region of code where the latency must be managed.

```
#pragma HLS latency min=<int> max=<int>
```

Where:

- `min=<int>`: Optionally specifies the minimum latency for the function, loop, or region of code.
- `max=<int>`: Optionally specifies the maximum latency for the function, loop, or region of code.

***Note*:** Although both min and max are described as optional, one must be specified.

Example 1

Function `foo` is specified to have a minimum latency of 4 and a maximum latency of 8:

```
int foo(char x, char a, char b, char c) {
  #pragma HLS latency min=4 max=8
  char y;
  y = x*a+b+c;
  return y
}
```

Example 2

In the following example `loop_1` is specified to have a maximum latency of 12. Place the pragma in the loop body as shown:

```
void foo (num_samples, ...) {
  int i;
  ...
  loop_1: for(i=0;i< num_samples;i++) {
  #pragma HLS latency max=12
    ...
    result = a + b;
  }
}
```

Example 3

The following example creates a code region and groups signals that need to change in the same clock cycle by specifying zero latency:

```
// create a region { } with a latency = 0
{
  #pragma HLS LATENCY max=0 min=0
  *data = 0xFF;
  *data_vld = 1;
}
```

See Also

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

pragma HLS loop_flatten

Description

Allows nested loops to be flattened into a single loop hierarchy with improved latency.

In the RTL implementation, it requires one clock cycle to move from an outer loop to an inner loop, and from an inner loop to an outer loop. Flattening nested loops allows them to be optimized as a single loop. This saves clock cycles, potentially allowing for greater optimization of the loop body logic.

Apply the `LOOP_FLATTEN` pragma to the loop body of the inner-most loop in the loop hierarchy. Only perfect and semi-perfect loops can be flattened in this manner:

- Perfect loop nests:
  - Only the innermost loop has loop body content.
  - There is no logic specified between the loop statements.
  - All loop bounds are constant.
- Semi-perfect loop nests:
  - Only the innermost loop has loop body content.
  - There is no logic specified between the loop statements.
  - The outermost loop bound can be a variable.

- Imperfect loop nests: When the inner loop has variable bounds (or the loop body is not exclusively inside the inner loop), try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

Syntax

Place the pragma in the C source within the boundaries of the nested loop.

```
#pragma HLS loop_flatten off
```

Where:

- `off`: Is an optional keyword that prevents flattening from taking place. Can prevent some loops from being flattened while all others in the specified location are flattened.

***Note*:** The presence of the LOOP_FLATTEN pragma enables the optimization.

Example 1

Flattens `loop_1` in function `foo` and all (perfect or semi-perfect) loops above it in the loop hierarchy, into a single loop. Place the pragma in the body of `loop_1`.

```
void foo (num_samples, ...) {
  int i;
  ...
  loop_1: for(i=0;i< num_samples;i++) {
   #pragma HLS loop_flatten
   ...
    result = a + b;
  }
}
```

Example 2

Prevents loop flattening in `loop_1`:

```
loop_1: for(i=0;i< num_samples;i++) {
   #pragma HLS loop_flatten off
   ...
```

See Also

- pragma HLS loop_merge
- pragma HLS loop_tripcount
- pragma HLS unroll
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

pragma HLS loop_merge

Description

Merge consecutive loops into a single loop to reduce overall latency, increase sharing, and improve logic optimization. Merging loops:

- Reduces the number of clock cycles required in the RTL to transition between the loop-body implementations.
- Allows the loops be implemented in parallel (if possible).

The `LOOP_MERGE` pragma will seek to merge all loops within the scope it is placed. For example, if you apply a `LOOP_MERGE` pragma in the body of a loop, Vivado HLS applies the pragma to any sub-loops within the loop but not to the loop itself.

The rules for merging loops are:

- If the loop bounds are variables, they must have the same value (number of iterations).
- If the loop bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bounds and constant bounds cannot be merged.
- The code between loops to be merged cannot have side effects. Multiple execution of this code should generate the same results (a=b is allowed, a=a+1 is not).
- Loops cannot be merged when they contain FIFO reads. Merging changes the order of the reads. Reads from a FIFO or FIFO interface must always be in sequence.

Syntax

Place the pragma in the C source within the required scope or region of code:

```
#pragma HLS loop_merge force
```

where

- `force`: An optional keyword to force loops to be merged even when Vivado HLS issues a warning.

IMPORTANT!: *In this case, you must manually insure that the merged loop will function correctly.*

Examples

Merges all consecutive loops in function `foo` into a single loop.

```
void foo (num_samples, ...) {
  #pragma HLS loop_merge
  int i;
  ...
  loop_1: for(i=0;i< num_samples;i++) {
    ...
```

All loops inside `loop_2` (but not `loop_2` itself) are merged by using the `force` option. Place the pragma in the body of `loop_2`.

```
loop_2: for(i=0;i< num_samples;i++) {
#pragma HLS loop_merge force
...
```

See Also

- pragma HLS loop_flatten
- pragma HLS loop_tripcount
- pragma HLS unroll
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

pragma HLS loop_tripcount

Description

The `TRIPCOUNT` pragma can be applied to a loop to manually specify the total number of iterations performed by a loop.

IMPORTANT!: *The `TRIPCOUNT` pragma is for analysis only, and does not impact the results of synthesis.*

Vivado HLS reports the total latency of each loop, which is the number of clock cycles to execute all iterations of the loop. The loop latency is therefore a function of the number of loop iterations, or tripcount.

The tripcount can be a constant value. It may depend on the value of variables used in the loop expression (for example, `x<y`), or depend on control statements used inside the loop. In some cases Vivado HLS cannot determine the tripcount, and the latency is unknown. This includes cases in which the variables used to determine the tripcount are:

- Input arguments, or
- Variables calculated by dynamic operation.

In cases where the loop latency is unknown or cannot be calculate, the `TRIPCOUNT` pragma lets you specify minimum and maximum iterations for a loop. This lets the tool analyze how the loop latency contributes to the total design latency in the reports, and helps you determine appropriate optimizations for the design.

Syntax

Place the pragma in the C source within the body of the loop:

```
#pragma HLS loop_tripcount min=<int> max=<int> avg=<int>
```

Where:

- `max=<int>`: Specifies the maximum number of loop iterations.
- `min=<int>`: Specifies the minimum number of loop iterations.
- `avg=<int>`: Specifies the average number of loop iterations.

Examples

In this example `loop_1` in function `foo` is specified to have a minimum tripcount of 12 and a maximum tripcount of 16:

```
void foo (num_samples, ...) {
  int i;
  ...
  loop_1: for(i=0;i< num_samples;i++) {
   #pragma HLS loop_tripcount min=12 max=16
   ...
    result = a + b;
  }
}
```

See Also

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

pragma HLS occurrence

Description

When pipelining functions or loops, the `OCCURRENCE` pragma specifies that the code in a region is executed less frequently than the code in the enclosing function or loop. This allows the code that is executed less often to be pipelined at a slower rate, and potentially shared within the top-level pipeline. To determine the OCCURRENCE:

- A loop iterates N times.
- However, part of the loop body is enabled by a conditional statement, and as a result only executes M times, where N is an integer multiple of M.
- The conditional code has an occurrence that is N/M times slower than the rest of the loop body.

For example, in a loop that executes 10 times, a conditional statement within the loop only executes 2 times has an occurrence of 5 (or 10/2).

Identifying a region with the `OCCURRENCE` pragma allows the functions and loops in that region to be pipelined with a higher initiation interval that is slower than the enclosing function or loop.

Syntax

Place the pragma in the C source within a region of code.

```
#pragma HLS occurrence cycle=<int>
```

Where:

- `cycle=<int>`: Specifies the occurrence N/M, where:
  - N is the number of times the enclosing function or loop is executed .
  - M is the number of times the conditional region is executed.

IMPORTANT!: *N must be an integer multiple of M.*

Examples

In this example, the region `Cond_Region` has an occurrence of 4 (it executes at a rate four times less often than the surrounding code that contains it):

```
Cond_Region: {
#pragma HLS occurrence cycle=4
...
}
```

See Also

- pragma HLS pipeline
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

pragma HLS pipeline

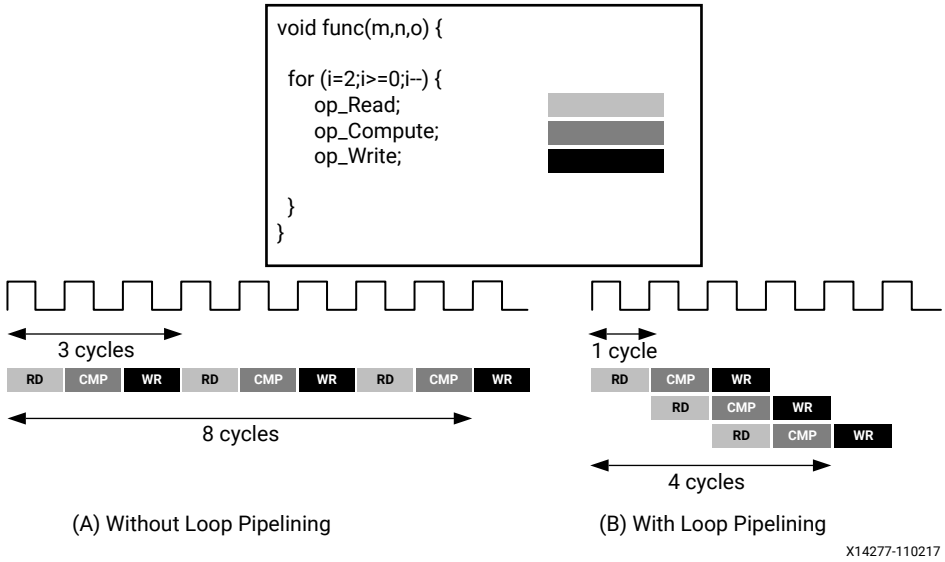
Description

The `PIPELINE` pragma reduces the initiation interval for a function or loop by allowing the concurrent execution of operations.

A pipelined function or loop can process new inputs every N clock cycles, where N is the initiation interval (II) of the loop or function. The default initiation interval for the `PIPELINE` pragma is 1, which processes a new input every clock cycle. You can also specify the initiation interval through the use of the II option for the pragma.

Pipelining a loop allows the operations of the loop to be implemented in a concurrent manner as shown in the following figure. In this figure, (A) shows the default sequential operation where there are 3 clock cycles between each input read (II=3), and it requires 8 clock cycles before the last output write is performed.

Figure 4: **Loop Pipeline**



IMPORTANT!: *Loop pipelining can be prevented by loop carry dependencies. You can use the `DEPENDENCE` pragma to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).*

If Vivado HLS cannot create a design with the specified II, it:

- Issues a warning.
- Creates a design with the lowest possible II.

You can then analyze this design with the warning message to determine what steps must be taken to create a design that satisfies the required initiation interval.

Syntax

Place the pragma in the C source within the body of the function or loop.

```
#pragma HLS pipeline II=<int> enable_flush rewind
```

Where:

- `II=<int>`: Specifies the desired initiation interval for the pipeline. Vivado HLS tries to meet this request. Based on data dependencies, the actual result might have a larger initiation interval. The default II is 1.
- `enable_flush`: An optional keyword which implements a pipeline that will flush and empty if the data valid at the input of the pipeline goes inactive.

TIP: *This feature is only supported for pipelined functions: it is not supported for pipelined loops.*

- `rewind`: An optional keyword that enables rewinding, or continuous loop pipelining with no pause between one loop iteration ending and the next iteration starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:
  - Is considered as initialization.
  - Is executed only once in the pipeline.
  - Cannot contain any conditional operations (if-else).

TIP: *This feature is only supported for pipelined loops: it is not supported for pipelined functions.*

Example 1

In this example function `foo` is pipelined with an initiation interval of 1:

```
void foo { a, b, c, d} {
  #pragma HLS pipeline II=1
  ...
}
```

***Note*:** The default value for II is 1, so II=1 is not required in this example.

See Also

- pragma HLS dependence
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

- xcl_pipeline_loop
- *SDAccel Environment Optimization Guide* (UG1207)

pragma HLS protocol

Description

The `PROTOCOL` pragma specifies a region of the code to be a protocol region, in which no clock operations are inserted by Vivado HLS unless explicitly specified in the code. A protocol region can be used to manually specify an interface protocol to ensure the final design can be connected to other hardware blocks with the same I/O protocol.

Note: See "Specifying Manual Interface"in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information.

Vivado HLS does not insert any clocks between the operations, including those that read from, or write to, function arguments, unless explicitly specified in the code. The order of read and writes are therefore obeyed in the RTL.

A clock operation may be specified:

- In C by using an `ap_wait()` statement (include `ap_utils.h`).
- In C++ and SystemC designs by using the `wait()` statement (include `systemc.h`).

The `ap_wait` and `wait` statements have no effect on the simulation of C and C++ designs respectively. They are only interpreted by Vivado HLS.

To create a region of C code:

1. Enclose the region in braces, `{}`,
2. Optionally name it to provide an identifier.

For example, the following defines a region called `io_section`:

```
io_section:{
...
}
```

Syntax

Place the pragma inside the boundaries of a region to define the protocol for the region.

```
#pragma HLS protocol <floating | fixed>
```

Where:

- `floating`: Protocol mode that allows statements outside the protocol region to overlap with the statements inside the protocol region in the final RTL. The code within the protocol region remains cycle accurate, but other operations can occur at the same time. This is the default protocol mode.
- `fixed`: Protocol mode that ensures that there is no overlap of statements inside or outside the protocol region.

IMPORTANT!: *If no protocol mode is specified, the default of floating is assumed.*

Example 1

This example defines region `io_section` as a fixed protocol region. Place the pragma inside region:

```
io_section:{
  #pragma HLS protocol fixed
  ...
}
```

See Also

- pragma HLS array_map
- pragma HLS array_reshape
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
- xcl_array_partition
- *SDAccel Environment Optimization Guide* (UG1207)

pragma HLS reset

Description

Adds or removes resets for specific state variables (global or static).

The reset port is used in an FPGA to restore the registers and block RAM connected to the reset port to an initial value any time the reset signal is applied. The presence and behavior of the RTL reset port is controlled using the `config_rtl` configuration file. The reset settings include the ability to set the polarity of the reset, and specify whether the reset is synchronous or asynchronous, but more importantly it controls, through the reset option, which registers are reset when the reset signal is applied. See Clock, Reset, and RTL Output in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information.

Greater control over reset is provided through the `RESET` pragma. If a variable is a static or global, the `RESET` pragma is used to explicitly add a reset, or the variable can be removed from the reset by turning `off` the pragma. This can be particularly useful when static or global arrays are present in the design.

Syntax

Place the pragma in the C source within the boundaries of the variable life cycle.

```
#pragma HLS reset variable=<a> off
```

Where:

- `variable=<a>`: Specifies the variable to which the pragma is applied.
- `off`: Indicates that reset is not generated for the specified variable.

Example 1

This example adds reset to the variable `a` in function `foo` even when the global reset setting is `none` or `control`:

```
void foo(int in[3], char a, char b, char c, int out[3]) {
#pragma HLS reset variable=a
```

Example 2

Removes reset from variable `a` in function `foo` even when the global reset setting is `state` or `all`.

```
void foo(int in[3], char a, char b, char c, int out[3]) {
#pragma HLS reset variable=a off
```

See Also

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

pragma HLS resource

Description

Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL. If the RESOURCE pragma is not specified, Vivado HLS determines the resource to use.

Vivado HLS implements the operations in the code using hardware cores. When multiple cores in the library can implement the operation, you can specify which core to use with the `RESOURCE` pragma. To generate a list of available cores, use the `list_core` command.

TIP: *The `list_core` command is used to obtain details on the cores available in the library. The `list_core` can only be used in the Vivado HLS Tcl command interface, and a Xilinx device must be specified using the `set_part` command. If a device has not been selected, the `list_core` command does not have any effect.*

For example, to specify which memory element in the library to use to implement an array, use the `RESOURCE` pragma. This lets you control whether the array is implemented as a single or a dual-port RAM. This usage is important for arrays on the top-level function interface, because the memory type associated with the array determines the ports needed in the RTL.

You can use the `latency=` option to specify the latency of the core. For block RAMs on the interface, the `latency=` option allows you to model off-chip, non-standard SRAMs at the interface, for example supporting an SRAM with a latency of 2 or 3. See Arrays on the Interface in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information. For internal operations, the `latency=` option allows the operation to be implemented using more pipelined stages. These additional pipeline stages can help resolve timing issues during RTL synthesis.

IMPORTANT!: *To use the `latency=` option, the operation must have an available multi-stage core. Vivado HLS provides a multi-stage core for all basic arithmetic operations (add, subtract, multiply and divide), all floating-point operations, and all block RAMs.*

For best results, Xilinx recommends that you use `-std=c99` for C and `-fno-builtin` for C and C++. To specify the C compile options, such as `-std=c99`, use the Tcl command `add_files` with the `-cflags` option. Alternatively, use the **Edit CFLAGs** button in the Project Settings dialog box. See Creating a New Synthesis Project in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Syntax

Place the pragma in the C source within the body of the function where the variable is defined.

```
#pragma HLS resource variable=<variable> core=<core>\
latency=<int>
```

Where:

- `variable=<variable>`: A required argument that specifies the array, arithmetic operation, or function argument to assign the `RESOURCE` pragma to.
- `core=<core>`: A required argument that specifies the core, as defined in the technology library.

- `latency=<int>`: Specifies the latency of the core.

Example 1

In the following example, a 2-stage pipelined multiplier is specified to implement the multiplication for variable `c` of the function `foo`. It is left to Vivado HLS which core to use for variable d.

```
int foo (int a, int b) {
int c, d;
#pragma HLS RESOURCE variable=c latency=2
c = a*b;
d = a*c;
return d;
}
```

Example 2

In the following example, the variable `coeffs[128]` is an argument to the top-level function `foo_top`. This example specifies that `coeffs` be implemented with core RAM_1P from the library:

```
#pragma HLS resource variable=coeffs core=RAM_1P
```

TIP: *The ports created in the RTL to access the values of `coeffs` are defined in the RAM_1P core.*

See Also

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

pragma HLS stream

Description

By default, array variables are implemented as RAM:

- Top-level function array parameters are implemented as a RAM interface port.
- General arrays are implemented as RAMs for read-write access.
- In sub-functions involved in DATAFLOW optimizations, the array arguments are implemented using a RAM pingpong buffer channel.
- Arrays involved in loop-based DATAFLOW optimizations are implemented as a RAM pingpong buffer channel

If the data stored in the array is consumed or produced in a sequential manner, a more efficient communication mechanism is to use streaming data as specified by the `STREAM` pragma, where FIFOs are used instead of RAMs.

IMPORTANT!: *When an argument of the top-level function is specified as* INTERFACE *type* `ap_fifo`*, the array is automatically implemented as streaming.*

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS stream variable=<variable> depth=<int> dim=<int> off
```

Where:

- `variable=<variable>`: Specifies the name of the array to implement as a streaming interface.
- `depth=<int>`: Relevant only for array streaming in DATAFLOW channels. By default, the depth of the FIFO implemented in the RTL is the same size as the array specified in the C code. This options lets you modify the size of the FIFO and specify a different depth.

  When the array is implemented in a DATAFLOW region, it is common to the use the `depth=` option to reduce the size of the FIFO. For example, in a DATAFLOW region when all loops and functions are processing data at a rate of II=1, there is no need for a large FIFO because data is produced and consumed in each clock cycle. In this case, the `depth=` option may be used to reduce the FIFO size to 1 to substantially reduce the area of the RTL design.

TIP: *The* `config_dataflow -depth` *command provides the ability to stream all arrays in a* DATAFLOW *region. The* `depth=` *option specified here overrides the* `config_dataflow` *command for the assigned* `variable`*.*

- `dim=<int>`: Specifies the dimension of the array to be streamed. The default is dimension 1. Specified as an integer from 0 to *N*, for an array with *N* dimensions.
- `off`: Disables streaming data. Relevant only for array streaming in dataflow channels.

TIP: *The* `config_dataflow -default_channel fifo` *command globally implies a* `STREAM` *pragma on all arrays in the design. The* `off` *option specified here overrides the* `config_dataflow` *command for the assigned* `variable`*, and restores the default of using a RAM pingpong buffer based channel.*

Example 1

The following example specifies array `A[10]` to be streaming, and implemented as a FIFO:

```
#pragma HLS STREAM variable=A
```

Example 2

In this example array `B` is set to streaming with a FIFO depth of 12:

```
#pragma HLS STREAM variable=B depth=12
```

Example 3

Array C has streaming disabled. It is assumed to be enabled by `config_dataflow` in this example:

```
#pragma HLS STREAM variable=C off
```

See Also

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

pragma HLS top

Description

Attaches a name to a function, which can then be used with the `set_top` command to synthesize the function and any functions called from the specified top-level. This is typically used to synthesize member functions of a class in C/C++.

Specify the pragma in an active solution, and then use the `set_top` command with the new name.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS top name=<string>
```

Where:

- `name=<string>`: Specifies the name to be used by the `set_top` command.

Examples

Function `foo_long_name` is designated the top-level function, and renamed to `DESIGN_TOP`. After the pragma is placed in the code, the `set_top` command must still be issued from the Tcl command line, or from the top-level specified in the GUI project settings.

```
void foo_long_name () {
  #pragma HLS top name=DESIGN_TOP
  ...
}

set_top DESIGN_TOP
```

See Also

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

pragma HLS unroll

Description

Unroll loops to create multiple independent operations rather than a single collection of operations. The `UNROLL` pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel.

Loops in the C/C++ functions are kept rolled by default. When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations might also be impacted by logic inside the loop body (for example, `break` conditions or modifications to a loop exit variable). Using the `UNROLL` pragma you can unroll loops to increase data access and throughput.

The `UNROLL` pragma allows the loop to be fully or partially unrolled. Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently. Partially unrolling a loop lets you specify a factor N, to create N copies of the loop body and reduce the loop iterations accordingly. To unroll a loop completely, the loop bounds must be known at compile time. This is not required for partial unrolling.

Partial loop unrolling does not require *N* to be an integer factor of the maximum loop iteration count. Vivado HLS adds an exit check to ensure that partially unrolled loops are functionally identical to the original loop. For example, given the following code:

```
for(int i = 0; i < X; i++) {
  pragma HLS unroll factor=2
  a[i] = b[i] + c[i];
}
```

Loop unrolling by a factor of 2 effectively transforms the code to look like the following code where the `break` construct is used to ensure the functionality remains the same, and the loop exits at the appropriate point:

```
for(int i = 0; i < X; i += 2) {
  a[i] = b[i] + c[i];
  if (i+1 >= X) break;
  a[i+1] = b[i+1] + c[i+1];
}
```

Because the maximum iteration count *X* is a variable, Vivado HLS may not be able to determine its value and so adds an exit check and control logic to partially unrolled loops. However, if you know that the specified unrolling factor, 2 in this example, is an integer factor of the maximum iteration count *X*, the `skip_exit_check` option lets you remove the exit check and associated logic. This helps minimize the area and simplify the control logic.

TIP: *When the use of pragmas like DATA_PACK, ARRAY_PARTITION, or ARRAY_RESHAPE, let more data be accessed in a single clock cycle, Vivado HLS automatically unrolls any loops consuming this data, if doing so improves the throughput. The loop can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle. This feature is controlled using the `config_unroll` command. See `config_unroll` in the Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more information.*

Syntax

Place the pragma in the C/C++ source within the body of the loop to unroll.

```
#pragma HLS unroll factor=<N> region skip_exit_check
```

Where:

- `factor=<N>`: Specifies a non-zero integer indicating that partial unrolling is requested. The loop body is repeated the specified number of times, and the iteration information is adjusted accordingly. If `factor=` is not specified, the loop is fully unrolled.
- `region`: An optional keyword that unrolls all loops within the body (region) of the specified loop, without unrolling the enclosing loop itself.

- `skip_exit_check`: An optional keyword that applies only if partial unrolling is specified with `factor=`. The elimination of the exit check is dependent on whether the loop iteration count is known or unknown:
  - Fixed (known) bounds: No exit condition check is performed if the iteration count is a multiple of the factor. If the iteration count is not an integer multiple of the factor, the tool:
    1. Prevents unrolling.
    2. Issues a warning that the exit check must be performed to proceed.
  - Variable (unknown) bounds: The exit condition check is removed as requested. You must ensure that:
    1. The variable bounds is an integer multiple of the specified unroll factor.
    2. No exit check is in fact required.

Example 1

The following example fully unrolls `loop_1` in function `foo`. Place the pragma in the body of `loop_1` as shown:

```
loop_1: for(int i = 0; i < N; i++) {
  #pragma HLS unroll
  a[i] = b[i] + c[i];
}
```

Example 2

This example specifies an unroll factor of 4 to partially unroll `loop_2` of function `foo`, and removes the exit check:

```
void foo (...) {
  int8 array1[M];
  int12 array2[N];
  ...
  loop_2: for(i=0;i<M;i++) {
    #pragma HLS unroll skip_exit_check factor=4
    array1[i] = ...;
    array2[i] = ...;
    ...
  }
  ...
}
```

Example 3

The following example fully unrolls all loops inside `loop_1` in function `foo`, but not `loop_1` itself due to the presence of the `region` keyword:

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {
  int temp1[N];
  loop_1: for(int i = 0; i < N; i++) {
    #pragma HLS unroll region
    temp1[i] = data_in[i] * scale;
      loop_2: for(int j = 0; j < N; j++) {
        data_out1[j] = temp1[j] * 123;
      }
      loop_3: for(int k = 0; k < N; k++) {
        data_out2[k] = temp1[k] * 456;
      }
  }
}
```

See Also

- pragma HLS loop_flatten
- pragma HLS loop_merge
- pragma HLS loop_tripcount
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
- opencl_unroll_hint
- *SDAccel Environment Optimization Guide* (UG1207)

Appendix C

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 webhelp:

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. *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821)
9. *Zynq UltraScale+ MPSoC Software Developer Guide* (UG1137)
10. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* (UG850)
11. *ZCU102 Evaluation Board User Guide* (UG1182)
12. *PetaLinux Tools Documentation: Workflow Tutorial* (UG1156)

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

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

15. Vivado® Design Suite Documentation

Please Read: Important Legal Notices

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

AUTOMOTIVE APPLICATIONS DISCLAIMER

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

Copyright

© Copyright 2018 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries.OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.All other trademarks are the property of their respective owners.