SDx Pragma Reference Guide

UG1253 (v2019.1) June 5, 2019

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/05/2019 Version 2019.1	
pragma SDS data mem_attribute	Changed AXI_DMA to AXIDMA.
pragma HLS clock	Added pragma.
pragma HLS stable	Added pragma.
01/24/2019 Version 2018.3	
Chapter 4: HLS Pragmas	Removed pragma HLS Protocol
General updates	Editorial updates.
12/05/2018 Version 2018.3	
Chapter 2: OpenCL Attributes	• Added xcl_latency. • Added xcl_loop_tripcount. • Added <II_number> to xcl_pipeline_loop.
Chapter 3: SDS Pragmas	Added FASTDMA to pragma SDS data data_mover.
10/02/2018 Version 2018.2.xdf	
	No changes for release.
07/02/2018 Version 2018.2	
pragma HLS array_map	Added discussion of clock cycle expectations for horizontal and vertical array mapping.
pragma HLS interface	Added details of specifying burst mode, and added `latency` option.
06/06/2018 Version 2018.2	
Chapter 3: SDS Pragmas	Removed `pragma SDS PARTITION`
Chapter 4: HLS Pragmas	Removed `pragma HLS CLOCK`
04/04/2018 Version 2018.1	
Entire document	Minor editorial updates for 2018.1 release.

Table of Contents

Introduction

The Xilinx® SDx™ tools, including the SDAccel™ environment, the SDSoC™ environment, and the Vivado® High-Level Synthesis (HLS) tool, provide an out-of-the-box experience for system programmers looking to partition elements of a software application to run in an FPGA-based hardware kernel, and having that hardware work seamlessly with the rest of the application running in a processor or embedded processor.

The out-of-the-box experience provides adequate results for many applications. However, you might also need to optimize the hardware logic to extract the best quality of results from the hardware partition; to improve the performance of the kernel, the data throughput, reduce the latency, or reduce the resources used by the kernel. In this case, certain attributes, directives, or pragmas, can be used to direct the compilation and synthesis of the hardware kernel, or to optimize the function of the data mover operating between the processor and the hardware logic.

This guide describes the different forms and types of pragmas available for use in the OpenCL™ C language, or in standard C/C++ language definitions of a system-level application in the SDx development environment.

- OpenCL attributes are defined in the OpenCL language standard, and apply optimizations to the hardware kernel.
- Xilinx provides additional OpenCL attributes that are named starting with xcl_.
- SDS pragmas are defined for use with C or C++ language, and apply to the interface, data mover, and hardware kernel in SDSoC design projects.
- HLS pragmas are defined for use with C or C++ language and can be used in the SDx flow to apply optimizations to the hardware kernel in the Vivado HLS tool.
- Directives are the Vivado HLS tool Tcl commands that can be applied to the hardware partition, like HLS pragmas, but are not discussed in any detail here. Refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information.

The goal of kernel optimization is to create processing logic that can consume all the data as soon as it arrives at the kernel interfaces. This is generally achieved by expanding the processing code to match the data path with techniques, such as function pipelining, loop unrolling, array partitioning, dataflowing, etc. The attributes and pragmas described here are provided to assist your optimization effort.

You can apply optimization pragmas and attributes to the following objects and scopes:

- Interfaces: When you apply pragmas to an interface, the SDx development environment applies the directive to the top-level function, because the top-level function is the scope that contains the interface.
- Functions: When you apply pragmas to functions, SDx development environment applies the directive to all objects within the scope of the function. The effect of any directive stops at the next level of function hierarchy. The only exception is a directive that supports or uses a recursive option, such as the PIPELINE pragmas that recursively unrolls all loops in the hierarchy.
- Loops: When you apply pragmas to loops, SDx development environment applies the directive to all objects within the scope of the loop. For example, if you apply a LOOP_MERGE directive to a loop, SDx development environment applies the pragmas to any sub-loops within the loop but not to the loop itself.
- Arrays: When you apply pragmas to arrays, SDx development environment applies the directive to the scope that contains the array.
- Regions: When you apply pragmas to regions, SDx development environment applies the directive to the entire scope of the region. A region is any area enclosed within two braces, as shown in the following example:

```
{
the scope between these braces is a region
}
```

TIP: *Apply optimizations to a region in the same way you apply them to functions and loops.*

- You can label loops and regions to make it easier to identify them in your code. The following example shows labeled and unlabeled loops and regions:

```
// Example of a loop with a label
My_For_Loop:for(i=0; i<3;i++ {
printf("This loop has the label My_For_Loop \n");
}

// Example of a region with no label
{
printf("The scope between these braces has NO label");
}

// Example of a NAMED region
My_Region:{
printf("The scope between these braces HAS the label My_Region");
}
```

- When specifying values for arguments of attributes or pragmas, you can use literal values (for example, 1, 55, 3.14), or pass a macro using #define. The following example shows an attribute and pragma with literal values:

```
__attribute__((xcl_array_partition(cyclic,5,1)));

#pragma HLS ARRAY_PARTITION variable = k_matrix_val  cyclic   factor=5
```

  This example uses defined macros:

```
#define E 5
#pragma HLS ARRAY_PARTITION variable = k_matrix_val  cyclic   factor=E

__attribute__((xcl_array_partition(cyclic,E,1)));
```

IMPORTANT! *For both attributes and pragmas, do not use declared constants to specify values as they are not supported.*

OpenCL Attributes

Optimizations in OpenCL

This section describes OpenCL™ attributes that can be added to source code to assist system optimization by the SDAccel™ development environment, and Vivado® High-Level Synthesis (HLS) tool synthesis.

The SDx™ environment 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 usage and DDR bandwidth usage. 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 1: **OpenCL Attributes by Type**

Type	Attributes
Kernel Optimization	• reqd_work_group_size • vec_type_hint • work_group_size_hint • xcl_latency • 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

Table 1: **OpenCL Attributes by Type** *(cont'd)*

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

TIP: *The SDAccel compiler also supports many of the standard attributes supported by* `gcc`*, such as:*

- ALWAYS_INLINE
- NOINLINE
- UNROLL
- 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 tool compilers.

TIP: *The NOINLINE attribute is also a standard feature of GCC, and is also supported by SDx tool 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 might 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, 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 API 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 the SDAccel development environment. 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.

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 following code:

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

reqd_work_group_size

Description

When OpenCL API 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. After the work-group size (local size) is 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 attribute 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 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 hints to the OpenCL compiler representing the computational width of the kernel, providing a basis for calculating processor bandwidth usage when the compiler is looking to auto-vectorize 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 runtime to handle cases where the compiler decides to merge or separate workitems. This means that these 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 might 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 2: **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. Therefore, < 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 Profiling and 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 API 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 Profiling and 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 because 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 Profiling and 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.*

An advantage of using 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 XCL_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 tool 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 environment 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 the SDAccel compiler 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 the SDAccel environment 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)));
```

Because the size of the block is four, the SDAccel compiler 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 the SDAccel compiler has to build around each register to ensure functional equivalence with the usage in the original code. As a general best practice guideline for the SDx environment, 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 Profiling and Optimization Guide* (UG1207)
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

xcl_array_reshape

Description

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

This attribute combines array partitioning with vertical array mapping.

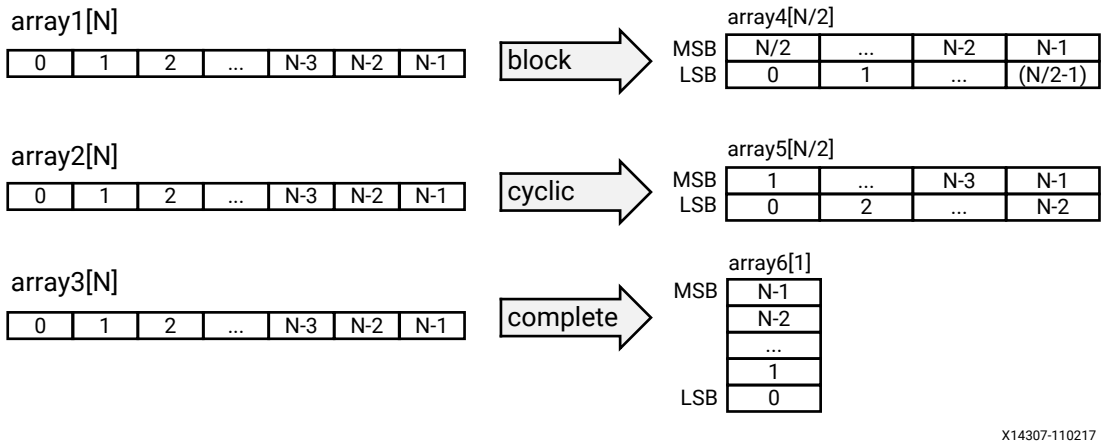
The XCL_ARRAY_RESHAPE attribute combines the effect of XCL_ARRAY_PARTITION, breaking an array into smaller arrays, and concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing parallel access to the data. This attribute creates a new array 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 environment 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. As a result, the 17 elements are reshaped into an array of five 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 Profiling and Optimization Guide* (UG1207)
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

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 register transfer level (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`), the Vivado High-Level Synthesis (HLS) tool 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, the HLS tool analyzes the dataflow between sequential functions or loops and create channels (based on ping-pong 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, the HLS tool attempts to minimize the initiation interval and start operation as soon as data is available.

TIP: *The HLS tool 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 the HLS tool 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, the HLS tool 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 XCL_DATAFLOW attribute before the function definition or the loop definition:

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

Examples

Specifies dataflow optimization within function `foo`.

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

See Also

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

xcl_latency

Description

The XCL_LATENCY attribute 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 or region latency is the number of clock cycles required for the code 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).

TheVivado High-Level Synthesis (HLS) tool always tries to minimize latency in the design. When the XCL_LATENCY attribute is specified, the tool behavior is as follows:

- When latency is greater than the minimum, or less than the maximum: The constraint is satisfied. No further optimizations are performed.
- When latency is less than the minimum: If the HLS tool can achieve less than the minimum specified latency, it extends the latency to the specified value, potentially increasing sharing.
- When latency is greater than the maximum: If the HLS tool 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 XCL_LATENCY attribute to limit the efforts of the tool to find a 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

Assign the XCL_LATENCY attribute before the body of the function, loop, or region:

```
__attribute__((xcl_latency(min, max)))
```

Where:

- <min>: Specifies the minimum latency for the function, loop, or region of code.

- <max>: Specifies the maximum latency for the function, loop, or region of code.

Example 1

The `for` loop in the `test` function is specified to have a minimum latency of 4 and a maximum latency of 8:

```
__kernel void test(__global float *A, __global float *B, __global float *C,
int id)
{
  for (unsigned int i = 0; i < id; i++)
__attribute__((xcl_latency(4, 12))) {
   C[id] = A[id] * B[id];
 }
}
```

See Also

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

xcl_loop_tripcount

Description

The XCL_LOOP_TRIPCOUNT attribute can be applied to a loop to manually specify the total number of iterations performed by the loop.

IMPORTANT! *The XCL_LOOP_TRIPCOUNT attribute is for analysis only, and does not impact the results of synthesis.*

The Vivado High-Level Synthesis (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, the HLS tool 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 calculated, the XCL_LOOP_TRIPCOUNT attribute lets you specify minimum, maximum, and average 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 attribute in the OpenCL source before the loop declaration:

```
__attribute__((xcl_loop_tripcount(<min>, <max>, <average>)))
```

Where:

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

Examples

In this example the WHILE loop in function `f` is specified to have a minimum tripcount of 2, a maximum tripcount of 64, and an average tripcount of 33:

```
__kernel void f(__global int *a) {
unsigned i = 0;
__attribute__((xcl_loop_tripcount(2, 64, 33)))
  while(i < 64) {
    a[i] = i;
    i++;
  }
}
```

See Also

- pragma HLS loop_tripcount
- *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 the SDx environment by the `reqd_work_group_size` attribute. SDx environment supports work size larger than 4096 with the XCL_MAX_WORK_GROUP_SIZE attribute.

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 (for example, 64x64). That is, iterating over an entire workgroup will fully add the input matrices, a and b, and output the result. 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 environment supports with the standard OpenCL attribute REQD_WORK_GROUP_SIZE. SDAccel environment 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 Profiling and 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 API specification for loop pipelining using the XCL_PIPELINE_LOOP attribute.

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(<II_number>)))
```

Where:

- <II_number>: Specifies the desired initiation interval (II) for the pipeline. The Vivado High-Level Synthesis (HLS) tool tries to meet this request, however, based on data dependencies the loop might have a larger initiation interval. When the II is not specified, the default is 1.

Example 1

The following example specifies an II target of 3 for the `for` loop in the specified function:

```
__kernel void f(__global int *a) {
  __attribute__((xcl_pipeline_loop(3)))
  for (unsigned i = 0; i < 64; ++i)
    a[i] = i;
}
```

See Also

- pragma HLS pipeline
- *SDAccel Environment Profiling and 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 API 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 tool 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 Profiling and 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 framework 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 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 Profiling and 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 Profiling and Optimization Guide* (UG1207)

SDS Pragmas

Optimizations in SDSoC

This section describes pragmas for the SDSoC™ system compilers, `sdscc` and `sds++` to assist system optimization.

The SDSoC environment system compilers target a base platform and invoke the Vivado® High-Level Synthesis (HLS) tool to compile synthesizeable C/C++ functions into programmable logic. Using the SDSoC IDE, or sdscc/sds++ command line options, you select functions from your source program to run in hardware, specify accelerator and system clocks, and set properties on data transfers.

In the SDSoC environment, you control the system generation process by structuring hardware functions and calls to hardware functions to balance communication and computation, and by inserting pragmas into your source code to guide the system compiler. The SDSoC compiler automatically chooses the best possible system port to use for any data transfer, but allows you to override this selection by using pragmas. You can also specify pragmas to select different data movers for your hardware function arguments, and use pragmas to control the number of data elements that are transferred to/from the hardware function.

All pragmas specific to the SDSoC environment are prefixed with `#pragma SDS` and should be inserted into C/C++ source code, either immediately prior to a function declaration or at a function call site for optimization of a specific function call.

```
#pragma SDS data access_pattern(in_a:SEQENTIAL, out_b:SEQUENTIAL)
void f1(int in_a[20], int out_b[20]);
```

The SDS pragmas include the types specified below:

Table 3: **SDS Pragmas by Type**

Type	Pragmas
Data Access Patterns	• pragma SDS data access_pattern
Data Transfer Size	• pragma SDS data copy • pragma SDS data zero_copy
Memory Attributes	• pragma SDS data mem_attribute

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

Type	Pragmas
Data Mover Type	• pragma SDS data data_mover
SDSoC Platform Interfaces to External Memory	• pragma SDS data sys_port
Hardware Buffer Depth	• pragma SDS data buffer_depth
Asynchronous Function Execution	• pragma SDS async • pragma SDS wait
Specifying Resource Binding	• pragma SDS resource
Hardware/Software Tracing	• pragma SDS trace

pragma SDS async

Description

The ASYNC pragma must be paired with the WAIT pragma to support manual control of the hardware function synchronization.

The ASYNC pragma is specified immediately preceding a call to a hardware function, directing the compiler not to automatically generate the wait based on data flow analysis. The WAIT pragma must be inserted at an appropriate point in the program to direct the CPU to wait until the associated `ASYNC` function call with the same ID has completed.

In the presence of an ASYNC pragma, the SDSoC system compiler does not generate an `sds_wait()` in the stub function for the associated call. The program must contain the matching `sds_wait(ID)` or `#pragma SDS wait(ID)` at an appropriate point to synchronize the controlling thread running on the CPU with the hardware function thread. An advantage of using the `#pragma SDS wait(ID)` over the `sds_wait(ID)` function call is that the source code can then be compiled by compilers other than the SDSoC compiler, like `gcc`, that does not interpret either ASYNC, or WAIT pragmas.

Syntax

Place the pragma in the C source immediately before the function call:

```
#pragma SDS async(<ID>)
...
#pragma SDS wait(<ID>)
```

Where:

- `<ID>`: Is a user-defined ID for the `ASYNC/WAIT` pair specified as a compile time unsigned integer constant.

Example 1

The following code snippet shows an example of using these pragmas with different IDs:

```
{
    #pragma SDS async(1)
    mmult(A, B, C);
    #pragma SDS async(2)
    mmult(D, E, F);
    ...
    #pragma SDS wait(1)
    #pragma SDS wait(2)
}
```

The program running on the hardware first transfers `A` and `B` to the mmult hardware and returns immediately. Then the program transfers `D` and `E` to the mmult hardware and returns immediately. When the program later executes to the point of `#pragma SDS wait(1)`, it waits for the output `C` to be ready. When the program later executes to the point of `#pragma SDS wait(2)`, it waits for the output `F` to be ready.

Example 2

The following code snippet shows an example of using these pragmas with the same `ID` to pipeline the data transfer and accelerator execution:

```
for (int i = 0; i < pipeline_depth; i++) {
    #pragma SDS async(1)
    mmult_accel(A[i%NUM_MAT], B[i%NUM_MAT], C[i%NUM_MAT]);
}
for (int i = pipeline_depth; i < NUM_TESTS; i++) {
    #pragma SDS wait(1)
    #pragma SDS async(1)
    mmult_accel(A[i%NUM_MAT], B[i%NUM_MAT], C[i%NUM_MAT]);
}
for (int i = 0; i < pipeline_depth; i++) {
    #pragma SDS wait(1)
}
```

In the above example, the first loop ramps up the pipeline with a depth of `pipeline_depth`, the second loop executes the pipeline, and the third loop ramps down the pipeline. The hardware buffer depth (pragma SDS data buffer_depth) should be set to the same value as `pipeline_depth`. The goal of this pipeline is to transfer data to the accelerator for the next execution while the current execution is not finished. Refer to "Increasing System Parallelism and Concurrency" in *SDSoC Environment Profiling and Optimization Guide* (UG1235) for more information.

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

pragma SDS data access_pattern

Description

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration.

This pragma specifies the data access pattern in the hardware function. The SDSoC compiler checks the value of this pragma to determine the hardware interface to synthesize. If the access pattern is `SEQUENTIAL`, a streaming interface (such as `ap_fifo`) will be generated. Otherwise, with `RANDOM` access pattern, a RAM interface will be generated. Refer to Data Motion Network Generation in SDSoC in the *SDSoC Environment Profiling and Optimization Guide* (UG1235) for more information on the use of this pragma in data motion network generation.

Syntax

The syntax for this pragma is:

```
#pragma SDS data access_pattern(ArrayName:<pattern>)
```

Where:

- `ArrayName`: Specifies one of the formal parameters of the function to assign the pragma to.
- <pattern>: can be either `SEQUENTIAL` or `RANDOM`. The default is `RANDOM`.

Example 1

The following code snippet shows an example of using this pragma for the array argument (`A`):

```
#pragma SDS data access_pattern(A:SEQUENTIAL)
void foo(int A[1024], int B[1024]);
```

In the example shown above, a streaming interface will be generated for argument `A`, while a RAM interface will be generated for argument `B`. The access pattern for argument `A` must be A[0], A[1], A[2], ... , A[1023], and all elements must be accessed only once. On the other hand, argument `B` can be accessed in a random fashion, and each element can be accessed zero or more times.

Example 2

The following code snippet shows an example of using this pragma for a pointer argument:

```
#pragma SDS data access_pattern(A:SEQUENTIAL)
#pragma SDS data copy(A[0:1024])
void foo(int *A, int B[1024]);
```

In the above example, if argument `A` is intended to be a streaming port, the two pragmas shown must be applied. Without these, SDSoC tool synthesizes argument `A` as a register (IN, OUT, or INOUT based on the usage of `A` in function `foo`).

Example 3

The following code snippet shows the combination of the ZERO_COPY pragma and the ACCESS_PATTERN pragma:

```
#pragma SDS data zero_copy(A)
#pragma SDS data access_pattern(A:SEQUENTIAL)
void foo(int A[1024], int B[1024]);
```

In the above example, the ACCESS_PATTERN pragma is ignored. After the ZERO_COPY pragma is applied to an argument, an AXI Master interface will be synthesized for that argument. Refer to Zero Copy Data Mover in the *SDSoC Environment Profiling and Optimization Guide* (UG1235) for more information.

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

pragma SDS data buffer_depth

Description

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration, and applies to all the callers of the function.

This pragma only applies to arrays that map to block RAM or FIFO interfaces. For a block RAM-mapped array, the <BufferDepth> value specifies hardware multi-buffer depth. For a FIFO-mapped array, the <BufferDepth> value specifies the depth of the hardware FIFO allocated for the array. For this pragma, the following must be true:

- BRAM: 1 ≤ <BufferDepth> ≤ 4, and 2 ≤ ArraySize ≤ 16384.
- FIFO: <BufferDepth> = 2^n, where 4 ≤ n ≤ 20.

TIP: *When the pragma is not specified, the default <buffer_depth> is 1.*

Syntax

The syntax of this pragma is:

```
#pragma SDS data buffer_depth(ArrayName:<BufferDepth>)
```

Where:

- `ArrayName`: Specifies one of the formal parameters of the function to assign the pragma to.
- <BufferDepth>: must be a compile-time constant value.
- Multiple arrays can be specified as a comma separated list in one pragma. For example:

```
#pragma SDS data buffer_depth(ArrayName1:BufferDepth1,
ArrayName2:BufferDepth2)
```

Example 1

This example specifies a multi-buffer of size 4 used for the RAM interface of argument `a`:

```
#pragma SDS data buffer_depth(a:4)
void foo(int a[1024], b[1024]);
```

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

pragma SDS data copy

Description

The `pragma SDS data copy | zero_copy` must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration.

IMPORTANT! *The COPY pragma and the `ZERO_COPY` pragma are mutually exclusive and should not be specified together on the same object.*

The COPY pragma implies that data is explicitly copied between the host processor memory and the hardware function. A suitable data mover performs the data transfer. See "Improving System Performance" in *SDSoC Environment Profiling and Optimization Guide* (UG1235) for more information.

The ZERO_COPY means that the hardware function accesses the data directly from shared memory through an AXI master bus interface.

IMPORTANT! *By default, the SDSoC compiler assumes the COPY pragma for an array argument, meaning the data is explicitly copied from the processor to the accelerator via a data mover.*

Syntax

The syntax for this pragma is:

```
#pragma SDS data copy|zero_copy(ArrayName[<offset>:<length>])
```

Where:

- `ArrayName[<offset>:<length>]`: specifies the function parameter or argument to assign the pragma to, and the array dimension and data transfer size.
- `ArrayName`: must be one of the formal parameters of the function definition, not from the prototype (where parameter names are optional) but from the function definition.
- `<offset>`: Optionally specifies the number of elements from the first element in the array. It must be specified as a compile-time constant.

IMPORTANT! *The <offset> value is currently ignored, and should be specified as 0.*

- `<length>`: Specifies the number of elements transferred from the array for the specified dimension. It can be an arbitrary expression as long as the expression can be resolved at runtime inside the function.

TIP: *As shown in the examples below, <length> can be a C arithmetic expression involving other scalar arguments of the same function.*

- For a multi-dimensional array, each dimension should be separately specified. For example, for a two-dimensional array, use:

```
pragma SDS data copy(ArrayName[offset_dim1:length1][offset_dim2:length2])
```

- Multiple arrays can be specified in the same pragma, using a comma separated list. For example, use:

```
pragma SDS data copy(ArrayName1[offset1:length1],
ArrayName2[offset2:length2])
```

- The `[<offset>:<length>]` argument is optional, and is only needed if the data transfer size for an array cannot be determined at compile time. When this is not specified, the `COPY` or `ZERO_COPY` pragma is only used to select between copying the memory to/from the accelerator through a data mover versus directly accessing the processor memory by the accelerator. To determine the array size, the SDSoC compiler analyzes the callers to the accelerator function to determine the transfer size based on the memory allocation APIs for the array, for example, `malloc` or `sds_alloc`. If the analysis fails, it checks the argument type to see if the argument type has a compile-time array size and uses that size as the data transfer size. If the data transfer size cannot be determined, the compiler generates an error message so that you can specify the data size with `[<offset_dim>:<length>]`. If the data size is different between the caller and the callee, or different between multiple callers, the compiler also generates an error message so that you can correct the source code or use this pragma to override the compiler analysis.

Example 1

The following example applies the COPY pragma to both the "A" and "B" arguments of the accelerator function `foo` right before the function declaration. Notice the `<length>` option is specified as an expression, `size*size`:

```
#pragma SDS data copy(A[0:size*size], B[0:size*size])
void foo(int *A, int *B, int size);
```

The SDSoC system compiler will replace the body of the function `foo` with accelerator control, data transfer, and data synchronization code. The following code snippet shows the data transfer part:

```
void _p0_foo_0(int *A, int *B, int size)
{
    ...
    cf_send_i(&(_p0_swinst_foo_0.A), A, (size*size) * 4, &_p0_request_0);
    cf_receive_i(&(_p0_swinst_foo_0.B), B, (size*size) * 4, &_p0_request_1);
    ...
}
```

As shown above, the offset value `size*size` is used to tell the SDSoC runtime the number of elements of arrays "A" and "B."

TIP: *The `cf_send_i` and `cf_receive_i` functions require the number of bytes to transfer, so the compiler will multiply the number of elements specified by `<length>` with the number of bytes for each element (4 in this case).*

Example 2

The following code snippet shows an example of applying the ZERO_COPY pragma, instead of the COPY pragma above:

```
#pragma SDS data zero_copy(A[0:size*size], B[0:size*size])
void foo(int *A, int *B, int size);
```

The body of function `foo` becomes:

```
void _p0_foo_0(int *A, int *B, int size)
{
    ...
    cf_send_ref_i(&(_p0_swinst_foo_0.A), A, (size*size) * 4,
&_p0_request_0);
    cf_receive_ref_i(&(_p0_swinst_foo_0.B), B, (size*size) * 4,
&_p0_request_1);
    ...
}
```

The `cf_send_ref_i` and `cf_receive_ref_i` functions only transfer the reference or pointer of the array to the accelerator, and the accelerator accesses the processor memory directly.

Example 3

The following example shows a ZERO_COPY pragma with multiple arrays specified to generate a direct memory interface with DDR and the hardware function:

```
#pragma SDS data zero_copy(in1[0:mat_dim*mat_dim], in2[0:mat_dim*mat_dim],
out[0:mat_dim*mat_dim])
void matmul_partition_accel(int *in1,   // Read-Only Matrix 1
                            int *in2,   // Read-Only Matrix 2
                            int *out,   // Output Result
                            int mat_dim); //  Matrix Dim (assumed only
square matrix)
```

Example 4

A DATA COPY pragma instructs the compiler to insert the transfer size expression into the corresponding send/receive call within stub function body. As a result, it is essential that the argument names used in the function declaration match the argument names in the function definition. The following code snippet illustrates a common mistake: using an argument name in the function declaration that is different from the argument name used in the function definition:

```
"foo.h"
#pragma SDS data copy(in_A[0:1024])
void foo(int *in_A, int *out_B);

"foo.cpp"
#include "foo.h"
void foo(int *A, int *B)
{
...
}
```

Any C/C++ compiler will ignore the argument name in the function declaration, because the C/C ++ standard makes the argument name in the function declaration optional. Only the argument name in the function definition is used by the compiler. However, the SDSoC compiler will issue a warning when trying to apply the pragma:

```
WARNING: [SDSoC 0-0] Cannot find argument in_A in accelerator function
foo(int *A, int *B)
```

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

pragma SDS data data_mover

Description

IMPORTANT! *This pragma is not recommended for normal use. Only use this pragma if the compiler-generated data mover type does not meet the design requirement.*

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration. This pragma applies to all the callers of the bound function.

By default, the SDSoC compiler chooses the type of the data mover automatically by analyzing the code. The DATA_MOVER pragma can be used to override the compiler default. This pragma specifies the HW IP type, or `DataMover`, used to transfer an array argument.

The FASTDMA data mover supports a wider data-width to support higher bandwidth for data transfer. For Zynq® UltraScale+™ MPSoC the data-width is from 64-bits to 256-bits. For Zynq-7000 the data-width is 64-bits.

The SDSoC™ compiler automatically assigns an instance of the data mover HW IP to use for transferring the corresponding array. The `:id` can be specified to assign a specific data mover instance for the associated formal parameter. If more than two formal parameters have the same `DataMover` and the same `id`, they will share the same data mover HW IP instance.

IMPORTANT! *An additional requirement for using the AXIDMA_SIMPLE data mover is that the corresponding array must be allocated using `sds_alloc()`.*

Syntax

The syntax for this pragma is:

```
#pragma SDS data data_mover(ArrayName:DataMover[:id])
```

Where:

- `ArrayName`: Specifies one of the formal parameters of the function to assign the pragma to.
- `DataMover`: Must be one of the following:
  - AXIFIFO: used for non-contiguous memory, <300 bytes.
  - AXIDMA_SIMPLE: used for contiguous memory, <32MB.
  - AXIDMA_SG: can be used for either contiguous or non-contiguous memory, >300 bytes.
  - FASTDMA: contiguous memory only. The pragma is required when FASTDMA is desired.
- `:id`: is optional, but must be specified as a positive integer when it is used.
- Multiple arrays can be specified in one pragma, separated by a comma (,). For example:

```
#pragma SDS data data_mover(ArrayName1:DataMover[:id],
ArrayName2:DataMover[:id])
```

Example 1

The following code snippet shows an example of specifying the data mover ID in the pragma:

```
#pragma SDS data data_mover(A:AXIDMA_SG:1, B:AXIDMA_SG:1)
void foo(int A[1024], int B[1024]);
```

In the example above, the same instance of the AXIDMA_SG IP is shared to transfer data for arguments `A`, and `B`, because the same data mover ID has been specified.

Example 2

The following example uses the FASTDMA data mover:

```
#pragma SDS data
data_mover(A:FASTDMA,B:FASTDMA,C:FASTDMA,D:AXIDMA_SIMPLE,E:AXIDMA_SIMPLE)
void foo(float A[1024], float B[1024], float C[1024], int D[1024], int
E[1024]);
```

The compiler will transfer arrays A, B, and C with individual FASTDMA data movers, and arrays D, E with individual AXIDMA_SIMPLE data movers.

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

pragma SDS data mem_attribute

Description

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration. This pragma applies to all the callers of the function.

For an operating system like Linux that supports virtual memory, user-space allocated memory is paged, which can affect system performance. The SDSoC runtime also provides an API to allocate physically contiguous memory. The pragmas in this section can be used to tell the compiler whether the arguments have been allocated in physically contiguous memory.

Syntax

The syntax for this pragma is:

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

Where:

- `ArrayName`: Specifies one of the formal parameters of the function to assign the pragma to.
- `Contiguity`: Must be specified as either `PHYSICAL_CONTIGUOUS` or `NON_PHYSICAL_CONTIGUOUS`. The default value is `NON_PHYSICAL_CONTIGUOUS`:
  - `PHYSICAL_CONTIGUOUS` means that all memory corresponding to the associated `ArrayName` is allocated using `sds_alloc`.

  - `NON_PHYSICAL_CONTIGUOUS` means that all memory corresponding to the associated `ArrayName` is allocated using `malloc` or as a free variable on the stack. This helps the SDSoC compiler select the optimal data mover.
- Multiple arrays can be specified in one pragma, separated by a comma (,). For example:

```
#pragma SDS data mem_attribute(ArrayName:contiguity, ArrayName:contiguity)
```

Example 1

The following code snippet shows an example of specifying the `contiguity` attribute:

```
#pragma SDS data mem_attribute(A:PHYSICAL_CONTIGUOUS)
void foo(int A[1024], int B[1024]);
```

In the example above, the user tells the SDSoC compiler that array `A` is allocated in the memory block that is physically contiguous. The SDSoC compiler then chooses `AXIDMA_SIMPLE` instead of `AXIDMA_SG`, because the former is smaller and faster for transferring physically contiguous memory.

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

pragma SDS data sys_port

Description

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration, and applies to all the callers of the function.

This pragma overrides the SDSoC compiler default choice of memory port. If the SYS_PORT pragma is not specified for an array argument, the interface to the external memory is automatically determined by the SDSoC system compilers (sdscc/sds++) based on array memory attributes (cacheable or non-cacheable), array size, data mover used, etc.

The Zynq®-7000 device provides a cache coherent interface (S_AXI_ACP) between programmable logic and external memory, and high-performance ports (S_AXI_HP) for non-cache coherent access. The Zynq® UltraScale+™ MPSoC provides a cache coherent interface (S_AXI_HPCn_FPD), and non-cache coherent interface called (S_AXI_HPn_FPD).

Syntax

The syntax for this pragma is:

```
#pragma SDS data sys_port(<param_name>:<port>)
```

Where:

- <param_name>: Specifies one of the formal parameters of the function to assign the pragma to.
- <port>: The SDSoC compiler recognizes predefined memory port types: ACP for Zynq-7000 devices only, HPC, HP, or MIG, which represent cache coherent access (ACP, HPC), high speed non-coherent access (HP), or memory accessible through a soft memory controller implemented in PL logic (MIG). You can also use a specific platform port name for the <port>, but this is not recommended unless the compiler does not select the correct port, which could occur for a stream port in the platform. To get a list of platform ports, in a terminal shell, run `sds++ -sds-pf-info <platform>`.

  For example, the `sds++ -sds-pfm-info zcu102` command returns the following under System Ports:

```
System Ports

Use the system port name in a sysport pragma, for example
#pragma SDS data sys_port(parameter_name:system_port_name)

System Port Name (Vivado BD instance name, Vivado BD port name)
ps_e_S_AXI_HPC0_FPD (ps_e, S_AXI_HPC0_FPD)
ps_e_S_AXI_HPC1_FPD (ps_e, S_AXI_HPC1_FPD)
ps_e_S_AXI_HP0_FPD (ps_e, S_AXI_HP0_FPD)
ps_e_S_AXI_HP1_FPD (ps_e, S_AXI_HP1_FPD)
ps_e_S_AXI_HP2_FPD (ps_e, S_AXI_HP2_FPD)
ps_e_S_AXI_HP3_FPD (ps_e, S_AXI_HP3_FPD)
```

  In this case, the SYS_PORT pragma could be defined as:

```
#pragma SDS data sys_port(Array1:ps_e_S_AXI_HPC0_FPD)
```

  If <port> is defined using the HPC shortcut, then the argument, Array1, could be assigned to either HPC0, or HPC1, by the SDSoC compiler.

  When the platform is created, the designer could specify a shortcut for a specific platform port, using the `PFM.AXI_PORT` property. Refer to *SDSoC Environment Platform Development Guide* (UG1146) for more information on `PFM` properties. For example:

```
set_property PFM.AXI_PORT {M_AXIS {type "M_AXIS" sptag "Counter"}} \
[get_bd_cells /stream_fifo]
```

  This defines a SYS_PORT tag, "Counter", which can be specified in the pragma as:

```
#pragma SDS data sys_port(Array1:Counter)
```

This would be the same as declaring the following:

```
#pragma SDS data sys_port(Array1:stream_fifo_M_AXIS)
```

- Multiple arguments can be specified in one pragma, separated by commas:

```
#pragma SDS data sys_port(param1:port, param2:port)
```

Example 1

The following code snippet shows an example of using this pragma:

```
#pragma SDS data sys_port(A:HP)
void foo(int A[1024], int B[1024]);
```

In the above example, if the caller passes an array (A) allocated with cache coherent calls, such as `malloc`, or `sds_alloc`, the SDSoC compiler uses an HP platform interface even though this might not be the best choice.

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

pragma SDS data zero_copy

Description

TIP: *Refer to pragma SDS data copy for a complete description of the ZERO_COPY pragma.*

The COPY pragma implies that data is explicitly copied between the host processor memory and the hardware function, using a suitable data mover for the transfer. The ZERO_COPY pragma means that the hardware function accesses the data directly from shared memory through an AXI master bus interface.

IMPORTANT! *The COPY pragma and the ZERO_COPY pragma are mutually exclusive and should not be specified together on the same object.*

Example 1

The following example shows a ZERO_COPY pragma with multiple arrays specified to generate a direct memory interface with DDR and the hardware function:

```
#pragma SDS data zero_copy(in1[0:mat_dim*mat_dim], in2[0:mat_dim*mat_dim],
out[0:mat_dim*mat_dim])
void matmul_partition_accel(int *in1,  // Read-Only Matrix 1
                            int *in2,  // Read-Only Matrix 2
                            int *out,  // Output Result
                            int mat_dim); //  Matrix Dim (assumed only
square matrix)
```

IMPORTANT! *The array argument passed to a ZERO_COPY data mover must be physically contiguous. Passing a malloc'd buffer to a ZERO_COPY data mover will result in undefined behavior.*

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)

pragma SDS resource

Description

This pragma can be used at function call sites to manually specify resource binding.

The RESOURCE pragma is specified immediately preceding a call to a hardware function, directing the compiler to bind the caller to a specified accelerator instance. The SDSoC compiler identifies when multiple resource IDs have been specified for a function, and automatically generates a hardware accelerator and data motion network realizing the hardware functions in programmable logic.

Syntax

The syntax of the pragma is:

```
#pragma SDS resource(<ID>)
```

Where:

- <ID>: Must be a compile time unsigned integer constant. For the same function, each unique ID represents a unique instance of the hardware accelerator.

Example 1

The following code snippet shows an example of using this pragma with a different `ID`:

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

In the previous example, the first call to function `mmult` will be bound to an accelerator with an ID of 1, and the second call to `mmult` will be bound to another accelerator with an ID of 2.

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

pragma SDS trace

Description

The SDSoC environment tracing feature provides a detailed view of what is happening in the system during execution of an application, through the use of hardware/software event tracing. See the *SDSoC Environment User Guide* (UG1027) for more information.

This pragma specifies the trace insertion for the accelerator with granularity at the function level or the argument level, to let you monitor the activity on the accelerator for debug purposes. When tracing is enabled, tracing instrumentation is automatically inserted into the software code, and hardware monitors are inserted into the hardware system during implementation of the hardware logic. You can monitor either the complete accelerator function, or an individual parameter of the function.

The type of trace can be `SW`, `HW`, or both. `HW` trace means the "start" and "stop" of the corresponding hardware component, such as the start and stop of the hardware accelerator, or the start and stop of data transfer of the specified argument. This lets you monitor activity moving onto, and off of, the hardware. The `SW` trace lets you observe the software stub for the hardware accelerated function, to monitor the function, and arguments on the software side of the transaction. You can also monitor both the hardware, and software transactions.

Syntax

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration.

```
#pragma SDS trace(<var1>[:SW|:HW][,<var2>[:SW|:HW]])
```

Where:

- <var>: Specifies either the function name, or one of the parameters of the function.
- `[:SW|:HW]`: Specifies either HW tracing or SW tracing. The absence of this option indicates that both HW and SW traces are inserted.

Example 1

The following example traces the specified function, `foo`:

```
#pragma SDS monitor trace(foo)
void foo(int a, int b);
```

TIP: *The absence of either `:HW` or `:SW` indicates that both traces are inserted for the accelerator.*

Example 2

The following example demonstrates using this pragma to trace multiple arguments of the function.

```
#pragma SDS monitor trace(a, b:SW, c:HW)
void foo(int a, int b, int *c);
```

In the previous example, both `HW` and `SW` traces are inserted for argument `a`. Only the `SW` trace is inserted for argument `b`. For argument `c`, only the `HW` trace is inserted.

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

pragma SDS wait

Description

TIP: *Refer to the ASYNC pragma for more information.*

The WAIT pragma must be paired with the ASYNC pragma to support manual control of the hardware function synchronization.

The ASYNC pragma is specified immediately preceding a call to a hardware function, directing the compiler not to automatically generate the wait based on data flow analysis. The WAIT pragma must be inserted at an appropriate point in the program to direct the CPU to wait until the associated ASYNC function call with the same ID has completed.

See Also

- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDSoC Environment User Guide* (UG1027)

HLS Pragmas

Optimizations in Vivado HLS

In both SDAccel™ and SDSoC™ development environments, the hardware kernel must be synthesized from the OpenCL™, C, or C++ language into the register transfer level (RTL) that can be implemented into the programmable logic of a Xilinx® device. The Vivado® High-Level Synthesis (HLS) tool synthesizes RTL from the OpenCL, C, and C++ language descriptions.

The HLS tool is intended to work with your SDAccel or SDSoC development environment project without interaction. However, the HLS tool also provides pragmas that can be used to optimize the design: reduce latency, improve throughput performance, and reduce area and device resource usage 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 Profiling and Optimization Guide (UG1235) for more information.*

The HLS pragmas include the optimization types specified below:

Table 4: **Vivado HLS Pragmas by Type**

Type	Attributes
Kernel Optimization	• pragma HLS allocation • 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

Table 4: **Vivado HLS Pragmas by Type** *(cont'd)*

Type	Attributes
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 register transfer level (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 used 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 the Vivado High-Level Synthesis (HLS) tool.
  - `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. Remapping the original N arrays will require N cycles with 1 port block RAM, or ceiling (N/2) cycles with a 2 port block RAM.
  - Vertical mapping concatenates the array to form a new array with longer words. Remapping the original N arrays is similar to the horizontal mapping above except when the same index is used: this will require only 1 cycle.
- `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, the Vivado High-Level Synthesis (HLS) tool 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 and provides the following:

- 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 Profiling and Optimization Guide* (UG1207)

pragma HLS array_reshape

Description

Combines array partitioning with vertical array mapping.

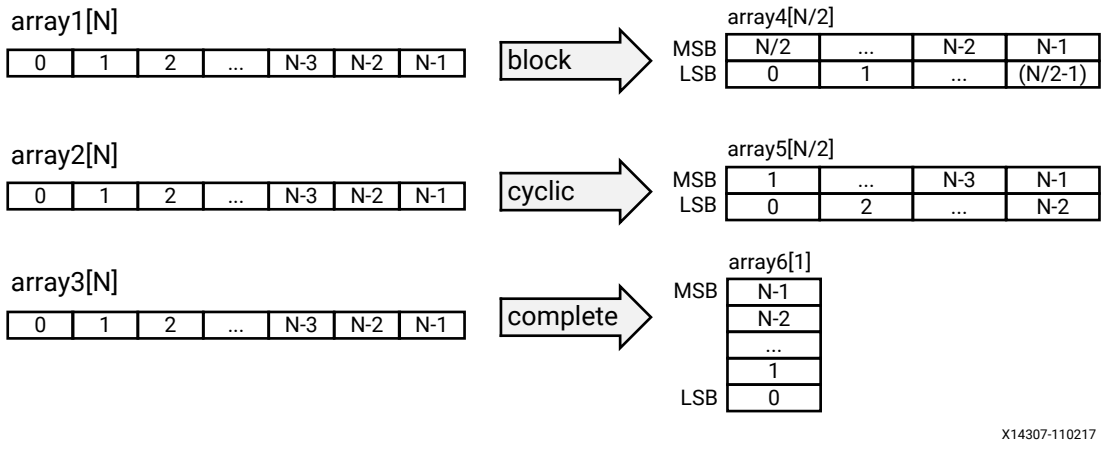
The ARRAY_RESHAPE pragma combines the effect of ARRAY_PARTITION, breaking an array into smaller arrays, with the effect of the vertical type of ARRAY_MAP, concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing the primary benefit of partitioning: 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 that applies to all functions in the design. However, SystemC designs (SC_MODULES) support multiple clocks. Multiple named clocks can be specified using the `create_clock` command, and specific clocks can be applied to different functions using the CLOCK pragma. Each SC_MODULE is synthesized using the specified clock.

Syntax

Place the pragma in the C source within the boundaries of a function.

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

Where:

- `domain=<clock>`: Specifies the name of a clock as defined by the `create_clock` command.

Examples

The following example shows a SystemC design where the top-level function, `foo_top`, has two clocks: `fast_clock` and `slow_clock`. The top-level function uses the `fast_clock`, and the sub-function `foo_sub` uses the `slow_clock`:

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

The HLS command `create_clock` defines the clocks as follows:

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

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. The Vivado High-Level Synthesis (HLS) tool 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, and 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, etc.). 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)

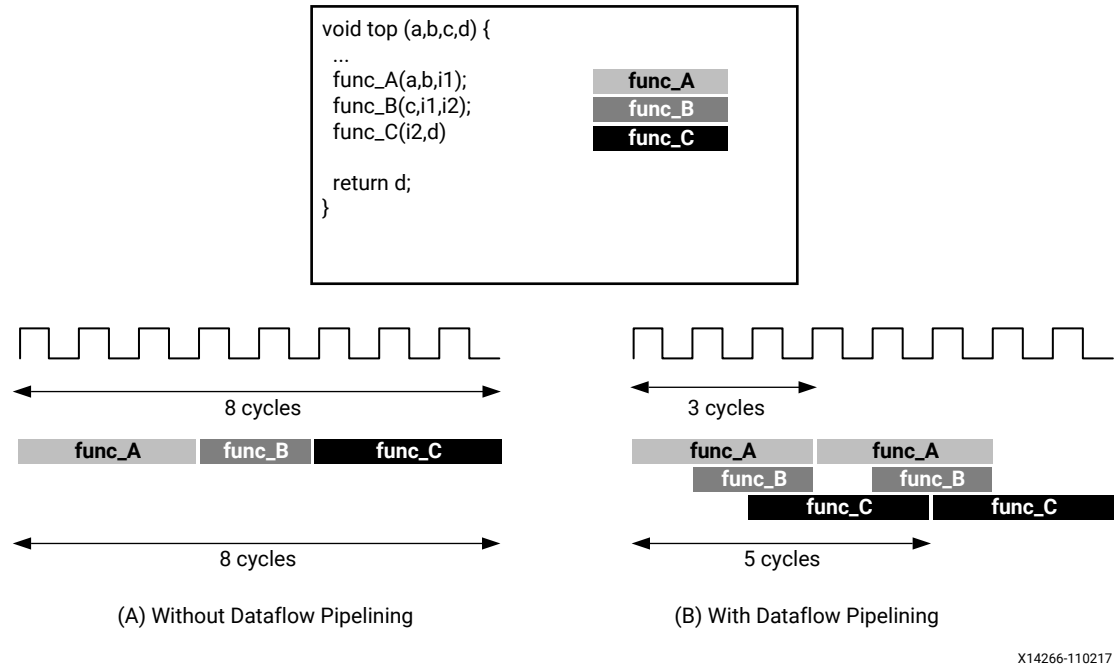
pragma HLS dataflow

Description

The DATAFLOW pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the register transfer level (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`), the Vivado High-Level Synthesis (HLS) tool 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, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels (based on ping pong 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, the HLS tool 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 the HLS tool 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, the HLS tool issues a message and does not perform DATAFLOW optimization.*

You can use the STABLE pragma to mark variables within DATAFLOW regions to be stable to avoid concurrent read or write of variables.

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

See Also

- pragma HLS allocation
- pragma HLS stable
- xcl_dataflow
- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
- *SDAccel Environment Profiling and 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).

The Vivado High-Level Synthesis (HLS) tool automatically detects the following 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 some 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, the HLS tool might 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 the HLS tool 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, the HLS tool 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 the HLS tool 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 the HLS tool 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 Profiling and 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, the Vivado High-Level Synthesis (HLS) tool 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 register transfer level (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 register transfer level (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: *The Vivado High-Level Synthesis (HLS) tool 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 a register transfer level (RTL) design, these same input and output operations must be performed through a port in the design interface and typically operate using a specific input/output (I/O) 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 the following:

- Any function-level protocol that is specified: 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.

- Function arguments: 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 I/O protocols are used to sequence data into and out of the block.
- Global variables accessed by the top-level function, and defined outside its scope:
    - If a global variable is accessed, but all read and write operations are local to the function, the resource is created in the RTL 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: *The Vivado High-Level Synthesis (HLS) tool automatically determines the I/O protocol used by any sub-functions. You cannot control these ports except to specify whether the port is registered.*

Specifying Burst Mode

When specifying burst-mode for interfaces, using the `max_read_burst_length` or `max_write_burst_length` options (as described in the Syntax section) there are limitations and related considerations that are derived from the AXI standard:

1. The burst length should be less than, or equal to 256 words per transaction, because ARLEN & AWLEN are 8 bits; the actual burst length is AxLEN+1.
2. In total, less than 4 KB is transferred per burst transaction.
3. Do not cross the 4 KB address boundary.
4. The bus width is specified as a power of 2, between 32-bits and 512-bits (i.e. 32, 64, 128, 256, 512 bits) or in bytes: 4, 8, 16, 32, 64.

With the 4KB limit, the max burst length for a bus width of:

- 32-bits is 256 words transferred in a single burst transaction. In this case, the total bytes transferred per transaction would be 1024.
- 64-bits is 256 words transferred in a single burst transaction. The total bytes transferred per transaction would be 2048.
- 128-bits is 256 words transferred in a single burst transaction. The total bytes transferred per transaction would be 4096.
- 256-bits is 128 words transferred in a single burst transaction. The total bytes transferred per transaction would be 4096.

- 512-bits is 64 words transferred in a single burst transaction. The total bytes transferred per transaction would be 4096.

***Note*:** The IP generated by the HLS tool might not actually perform the maximum burst length as this is design dependent. For example, pipelined accesses from a for-loop of 100 iterations when `max_read_burst_length` or `max_write_burst_length` is set to 128, will not fill the max burst length.

However, if the design is doing longer accesses in the source code than the specified maximum burst length, the access will be split into smaller bursts. For example, a pipelined for-loop with 100 accesses and `max_read_burst_length` or `max_write_burst_length` set to 64, will be split into 2 transactions, one of the max burst length (or 64) and one transaction of the remaining data (burst of length 36 words).

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. The HLS tool 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! *The HLS tool 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 the 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 the 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. The HLS tool 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 the HLS tool 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, the HLS tool 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>.

IMPORTANT! *When specifying the `bundle=` name you should use all lower-case characters.*

- `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 the 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`. AXI4-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 the HLS tool 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 AXI4-Lite interface clock is the same clock as the system clock. This option is used to specify a separate clock for the AXI4-Lite (`s_axilite`) interface.

TIP: *If the `bundle` option is used to group multiple top-level function arguments into a single AXI4-Lite interface, the clock option need only be specified on one of the bundle members.*

- `latency=<value>`: When `mode` is `m_axi`, this specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request a number of cycles (latency) before the read or write is expected. If this figure it too low, the design will be ready too soon and may stall waiting for the bus. If this figure is too high, bus access may be granted but the bus may stall waiting on the design to start the access.
- `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.

TIP: *If the port is a read-only port, then set the `num_write_outstanding=1` and `max_write_burst_length=2` to conserve memory resources. For write-only ports, set the `num_read_outstanding=1` and `max_read_burst_length=2`.*

- `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

- *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:** Number of clock cycles required to produce an output.
- **Function latency:** Number of clock cycles required for the function to compute all output values, and return.

- **Loop latency:** Number of cycles to execute all iterations of the loop.

See "Performance Metrics Example" of *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

The Vivado High-Level Synthesis (HLS) tool always tries to 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 the HLS tool 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 HLS tool 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 register transfer level (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 register transfer level (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, the Vivado High-Level Synthesis (HLS) tool 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 the HLS tool 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 LOOP_TRIPCOUNT pragma can be applied to a loop to manually specify the total number of iterations performed by a loop.

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

The Vivado High-Level Synthesis (HLS) tool 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, the HLS tool 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 calculated, the LOOP_TRIPCOUNT pragma lets you specify minimum and maximum iterations for a loop. This allows 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 the following 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 two 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:
  - <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

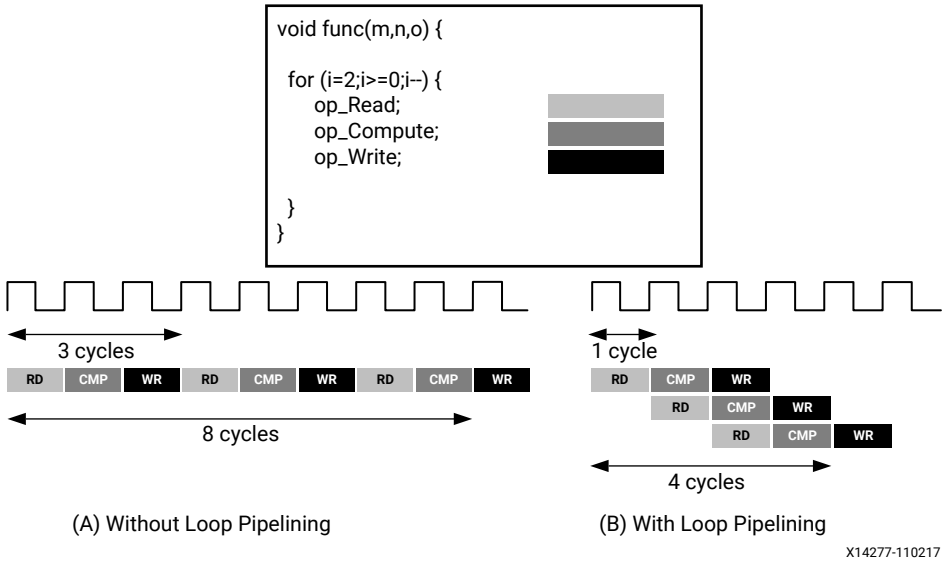
Description

The PIPELINE pragma reduces the initiation interval (II) 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 II of the loop or function. The default II 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 the following 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 the Vivado High-Level Synthesis (HLS) tool cannot create a design with the specified II, it issues a warning and 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. The HLS tool 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`: Optional keyword that 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`: 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

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

pragma HLS reset

Description

The RESET pragma 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 register transfer level (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

The RESOURCE pragma specifies that a specific library resource (core) is used to implement a variable (array, arithmetic operation, or function argument) in the register transfer level (RTL). If the RESOURCE pragma is not specified, the Vivado High-Level Synthesis (HLS) tool determines the resource to use. The HLS tool 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 obtains details on the cores available in the library. The `list_core` can only be used in the HLS tool 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 allows you control whether the array is implemented as a single or a dual-port RAM, or in UltraRAM. 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. 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.

For more information, see "Arrays on the Interface" in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

IMPORTANT! *To use the `latency=` option, the operation must have an available multi-stage core. The HLS tool 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, select 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 two-stage pipelined multiplier is specified to implement the multiplication for variable <c> of the function `foo`. The HLS tool selects the 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 <coeffs[128]>variable is an argument to the top-level function `foo_top`. This example specifies that `coeffs` is 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 stable

Description

The STABLE pragma marks variables within a DATAFLOW region as being stable. The STABLE pragma applies to both scalar and array variables whose content can be either written or read by the process inside the DATAFLOW region. This pragma eliminates the extra synchronization involved for the DATAFLOW region.

Syntax

Place the pragma in the C source within the boundaries of a region, function, or loop that is also marked for DATAFLOW.

```
#pragma HLS stable variable=<name>
```

Where:

- `variable=<name>`: Specifies the name of a variable within the DATAFLOW region.

Examples

The following example specifies the array A as stable. If A is read by proc2, then it will not be written by another process while the DATAFLOW region is being executed.

```
void foo(int A[...], int B[…] ...
#pragma HLS dataflow
#pragma HLS stable variable=A
   proc1(...);
   proc2(A, ...);
```

See Also

- pragma HLS dataflow
- *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 ping pong buffer channel.
- Arrays involved in loop-based DATAFLOW optimizations are implemented as a RAM ping pong 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 option 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 ping pong 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. In the below example, it is assumed to be enabled by `config_dataflow`:

```
#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 register transfer level (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. The Vivado High-Level-Synthesis (HLS) tool 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, the HLS tool might 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, the HLS tool 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 Profiling and Optimization Guide* (UG1207)

XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

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

Documentation Navigator and Design Hubs

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

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

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

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

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

References

These documents provide supplemental material useful with this guide:

SDAccel Documents

1. *SDAccel Environment User Guide* (UG1023)
2. *SDAccel Environment Profiling and Optimization Guide* (UG1207)
3. *SDAccel Environment Getting Started Tutorial* (UG1021)
4. *SDAccel Environment Debugging Guide* (UG1281)

SDSoC Documents

1. *SDSoC Environment User Guide* (UG1027)
2. *SDSoC Environment Profiling and Optimization Guide* (UG1235)
3. *SDSoC Environment Tutorial: Introduction* (UG1028)
4. *SDSoC Environment Platform Development Guide* (UG1146)

Additional Documents

1. *SDx Pragma Reference Guide* (UG1253)
2. *Xilinx OpenCV User Guide* (UG1233)
3. *Platform Cable USB II Data Sheet* (DS593)

More Resources

1. Xilinx® licensing website: https://www.xilinx.com/getproduct
2. SDSoC Developer Zone: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.
3. SDAccel Developer Zone: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html
4. *Xilinx End-User License Agreement* (UG763)
5. *Third Party End-User License Agreement* (UG1254)

Please Read: Important Legal Notices

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