XILINX.

Pipeline for Performance: DATAFLOW

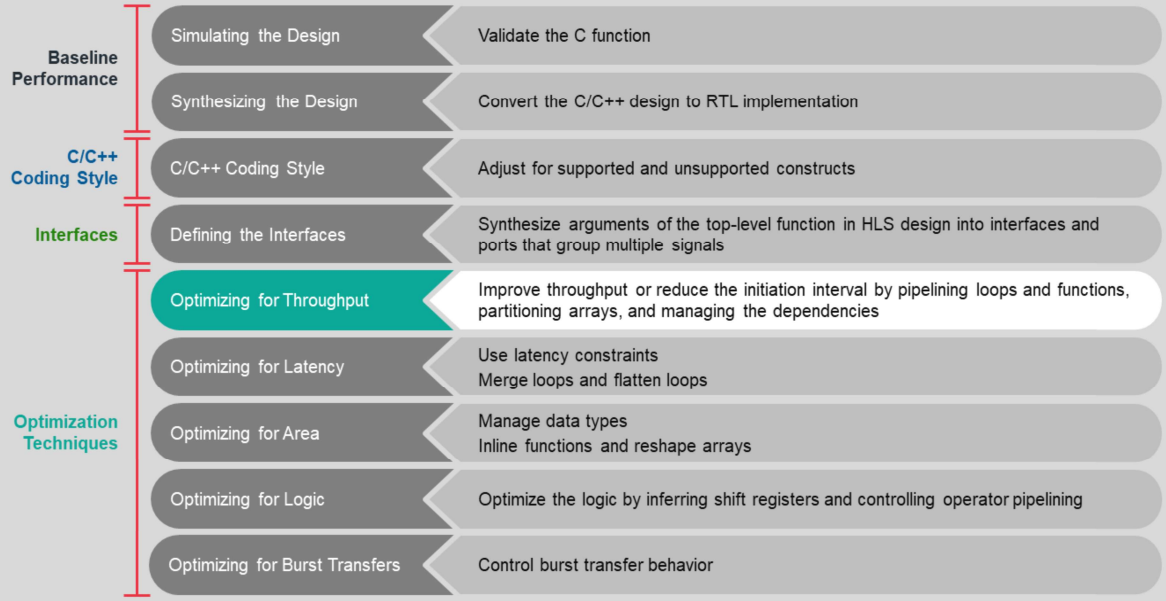
2021.1

Objectives

After completing this module, you will be able to:

- Describe the dataflow technique that improves the throughput of a design
- Identify some of the bottlenecks that impact design performance

Vitis HLS Design Methodology



XILINX.

Once the interfaces are defined, the next step in creating a high-performance design is to pipeline the functions, loops, and tasks.

Pipeline for Performance

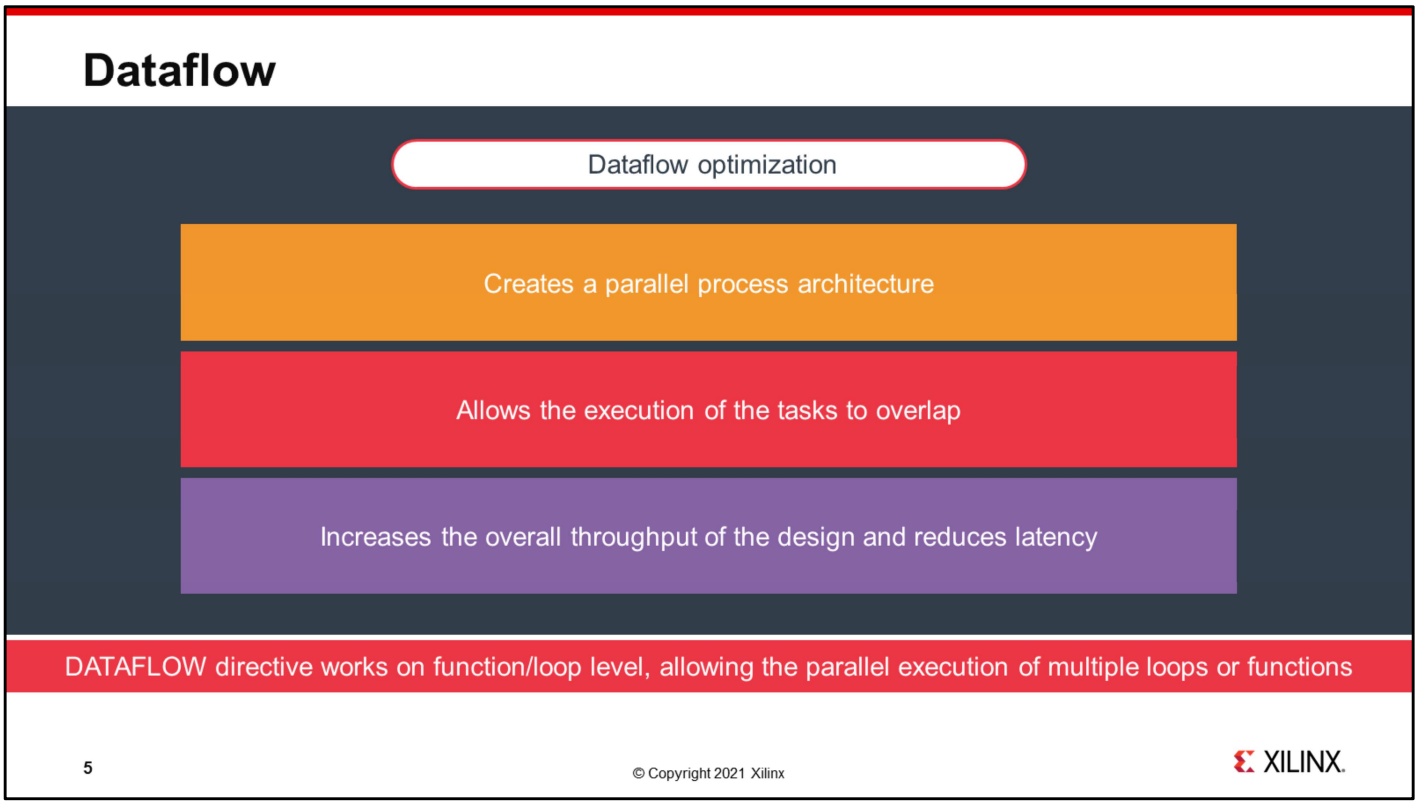
Create as much concurrent operation as possible

Directives and Configurations	Description
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function
DATAFLOW	Enables task-level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval
BIND_OP	Defines a specific implementation for an operation in the RTL
BIND_STORAGE	Defines a specific implementation for a storage element, or memory, in the RTL
Config Compile	Allows loops to be automatically pipelined based on their iteration count

At this stage of the optimization process, you want to create as much concurrent operation as possible.

The table shows the directives that can be used for pipelining.

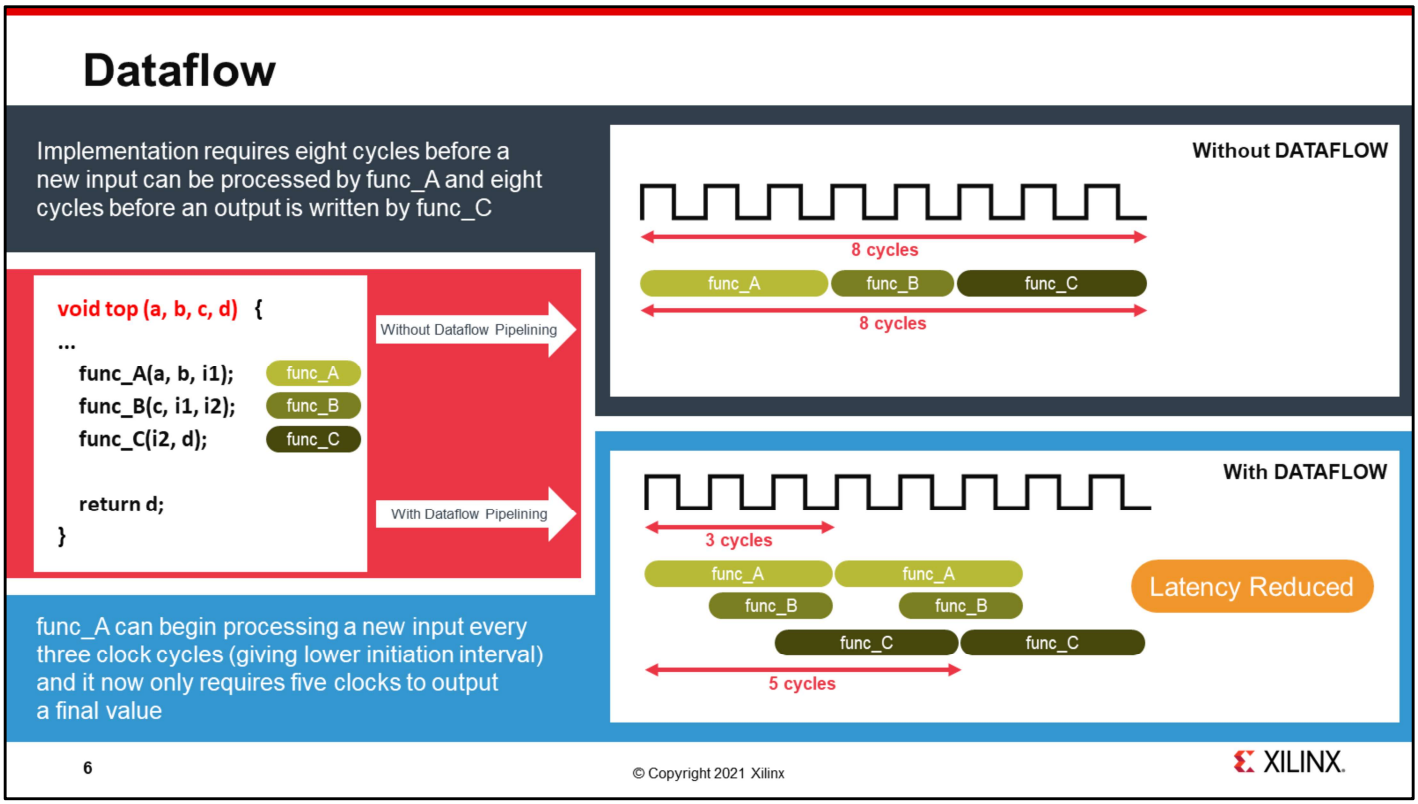
- The PIPELINE directive applied to functions and loops reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
- The DATAFLOW directive applied at the level that contains the functions and loops, makes them work in parallel.
- The BIND_OP directive defines a specific implementation for an operation in the RTL.
  The BIND_STORAGE directive defines a specific implementation for a storage element, or memory, in the RTL.
- The Config Compile configuration allows loops to be automatically pipelined based on their iteration count.



Let’s understand the concept of dataflow.

Dataflow optimization creates a parallel process architecture. It allows the execution of the tasks to overlap, increasing the overall throughput of the design and reducing latency.

The Vitis HLS tool provides a DATAFLOW directive, which is same as the PIPELINE directive, but it works on a function/loop level, allowing the parallel execution of multiple loops or functions.



Code with three functions calls to func_A, func_B, and func_C is shown here.

Without DATAFLOW:

- In the without DATAFLOW pipelining example, the implementation requires eight cycles before a new input can be processed by func_A and eight cycles before an output is written by func_C.

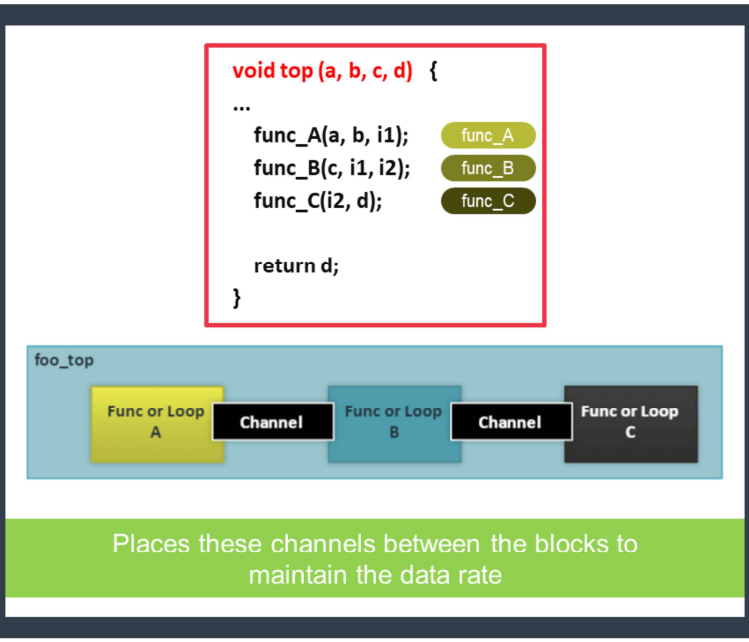
With DATAFLOW:

- In the with DATAFLOW pipelining example, func_A can begin processing a new input every three clock cycles (lower initiation interval) and it now only requires five clocks to output a final value, which shows the latency for the design has reduced significantly.

Configuring the Dataflow Channel

Vitis HLS tool analyzes the function or a loop body

Creates individual channels that model the dataflow to store the results of each task in the dataflow region



Places these channels between the blocks to maintain the data rate

XILINX.

Let's look at the process of configuring the dataflow channel.

When a particular region, such as a function body or a loop body, is identified as a region to apply the dataflow optimization, the Vitis HLS tool analyzes the function or a loop body and **creates individual channels** that model the dataflow to store the results of each task in the dataflow region.

It places these channels between the blocks to maintain the data rate.

Configuring the Dataflow Channel

These channels can be either:

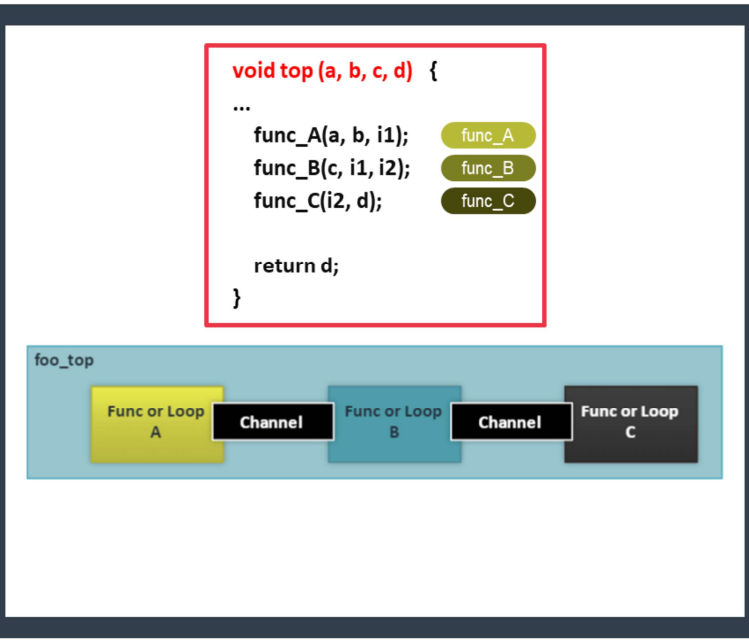
- Simple FIFOs with handshake signals for scalar variables
- Ping-pong buffers with memory elements, for non-scalar variables like arrays

Channels contain handshaking signals to indicate when the FIFO or the ping-pong buffer is full or empty

Dataflow optimization has an area overhead

Individual buffers allow Vitis HLS tool to free each task to execute at its own pace

Throughput is only limited by the availability of the input and output buffers



XILINX.

These channels can be either simple FIFOs with handshake signals for scalar variables, or ping-pong buffers, which includes the memory elements, for non-scalar variables like arrays. Each of these channels also contains handshaking signals to indicate when the FIFO or the ping-pong buffer is full or empty.

Dataflow optimization, therefore, has an area overhead, due to additional memory blocks added to the design.

By having these individual FIFOs and/or ping-pong buffers, the Vitis HLS tool frees each task to execute at its own pace, and the throughput is only limited by the availability of the input and output buffers.

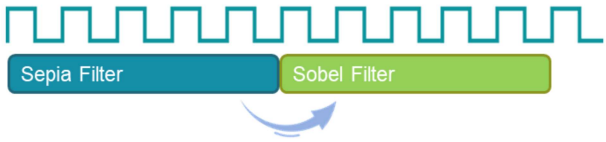
This is done at the cost of additional FIFO or block RAM registers for the ping-pong buffer.

FIFO vs. Ping-Pong RAM

```
//This memory is turned into a FIFO during optimization
rgb_pixel inter_pix [MAX_HEGHT] [MAX_WIDTH];

// Primary processing functions
sepia_filter (in_pix, inter_pix);
sobel_filter (inter_pix, out_pix2);
```

Sepia Filter

Sobel Filter



Whenever the code has arrays, these arrays are passed as single entities

XILINX.

Let's have a look at the difference between FIFO and ping-pong RAM.

Whenever the code has arrays, these arrays are passed as single entities, by default, as shown.

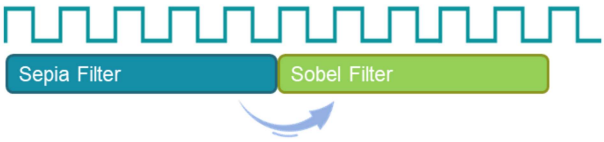
The example code has two primary functions sepia and sobel filter.

FIFO vs. Ping-Pong RAM

```
//This memory is turned into a FIFO during optimization
rgb_pixel inter_pix [MAX_HEGHT] [MAX_WIDTH];

// Primary processing functions
sepia_filter (in_pix, inter_pix);
sobel_filter (inter_pix, out_pix2);
```

Sepia Filter

Sobel Filter



Finish all writes to inter_pix[N]…

Then Sobel can start to access inter_pix[N]

- First, all the writes to the inter_pix [N] array of the sepia filter are done, and then the sobel filter starts accessing the inter_pix [N] array
- This is a default behavior of the design when dataflow is not used

XILINX.

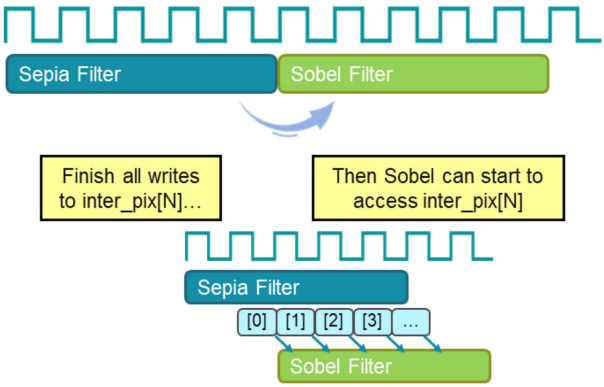
First, all the writes to inter_pix [N] array of the sepia filter are done, and then the sobel filter starts accessing the inter_pix [N] array. This is a default behavior of the design when dataflow is not used.

FIFO vs. Ping-Pong RAM

```
//This memory is turned into a FIFO during optimization
rgb_pixel inter_pix [MAX_HEGHT] [MAX_WIDTH];

// Primary processing functions
sepia_filter (in_pix, inter_pix);
sobel_filter (inter_pix, out_pix2);
```

Sepia Filter

Sobel Filter



When the DATAFLOW directive is used, the Vitis HLS tool creates and implements the channels between the tasks as either:

- Ping-pong buffers
- FIFO buffers

Depending on the:

- Access patterns of the producer and consumer of the data

XILINX.

When the DATAFLOW directive is used, the Vitis HLS tool creates and implements the channels between the tasks as either ping-pong buffers or FIFO buffers depending on the access patterns of the producer and consumer of the data.

FIFO vs. Ping-Pong RAM

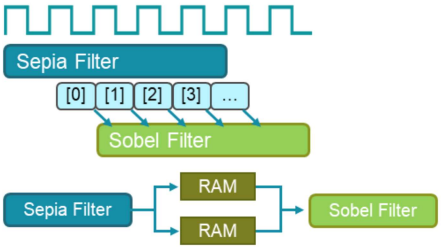
```
//This memory is turned into a FIFO during optimization
rgb_pixel inter_pix [MAX_HEGHT] [MAX_WIDTH];

// Primary processing functions
sepia_filter (in_pix, inter_pix);
sobel_filter (inter_pix, out_pix2);
```

Sepia Filter

Sobel Filter

Finish all writes to inter_pix[N]…

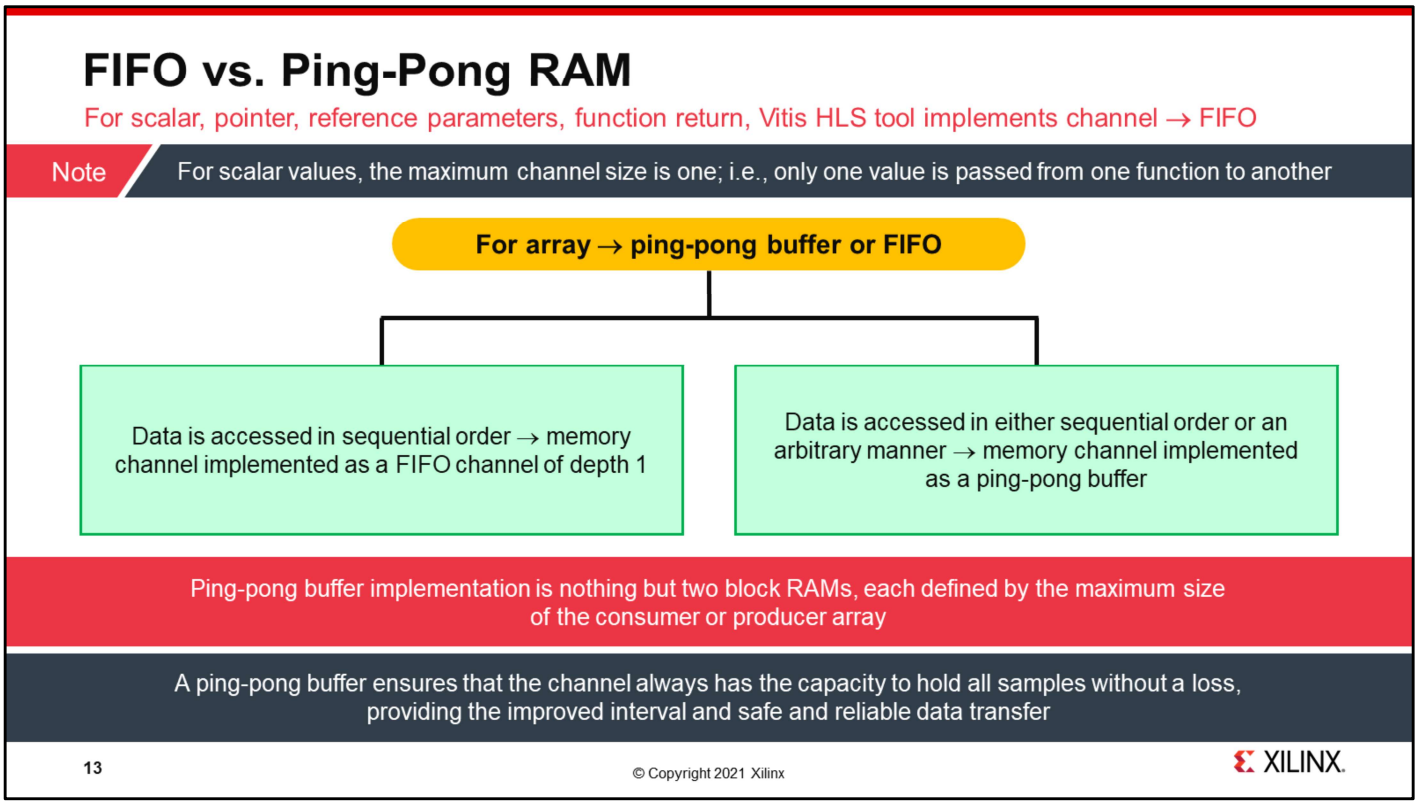
Then Sobel can start to access inter_pix[N]



These buffers are then implemented as the block RAM

XILINX.

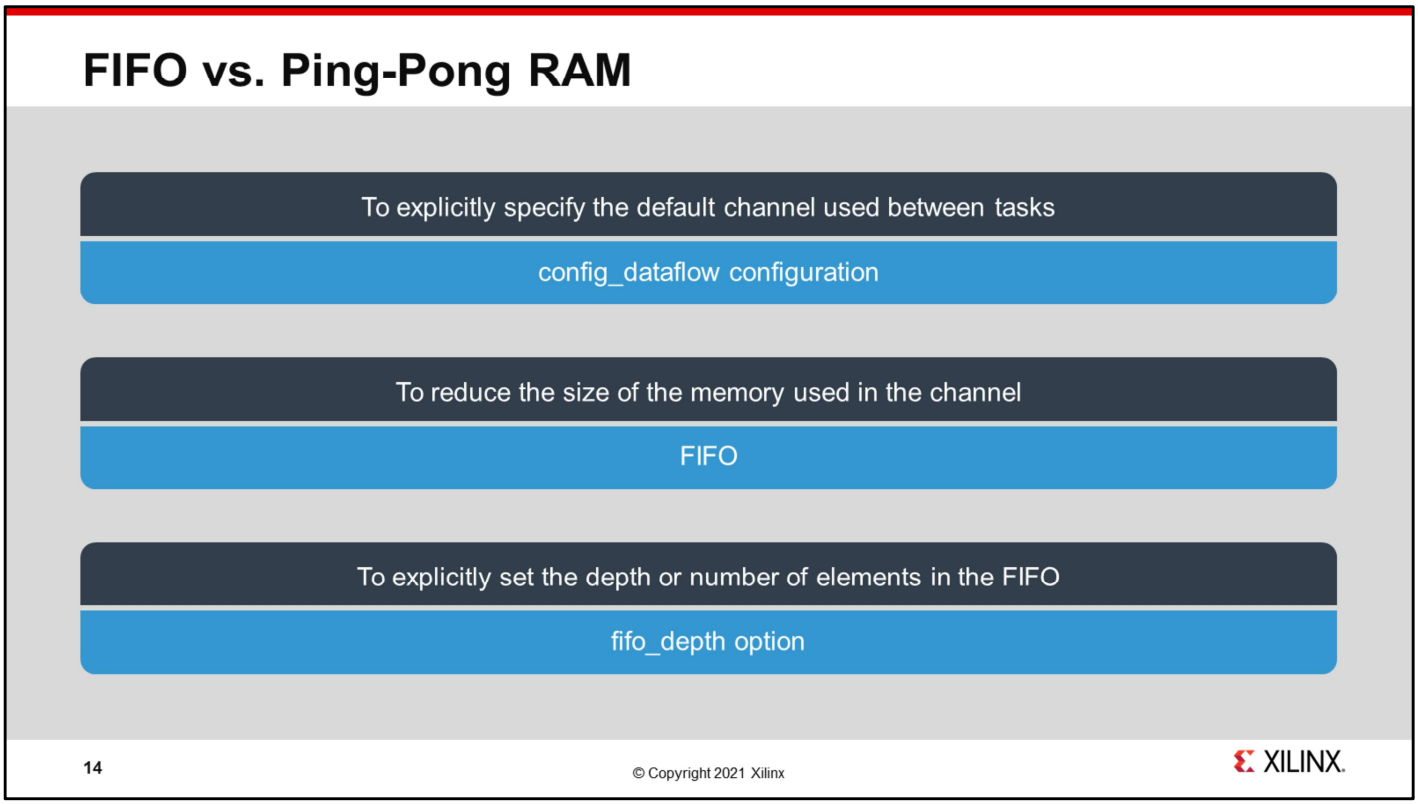
These buffers are then implemented as the block RAM as shown.



For a scalar, pointer, and reference parameters as well as the function return, the Vitis HLS tool implements the channel as a FIFO. It is to be noted that for scalar values, the maximum channel size is one; that is, only one value is passed from one function to another.

If the parameter (producer or consumer) is an array, the Vitis HLS tool implements the channel as a ping-pong buffer or a FIFO as follows:

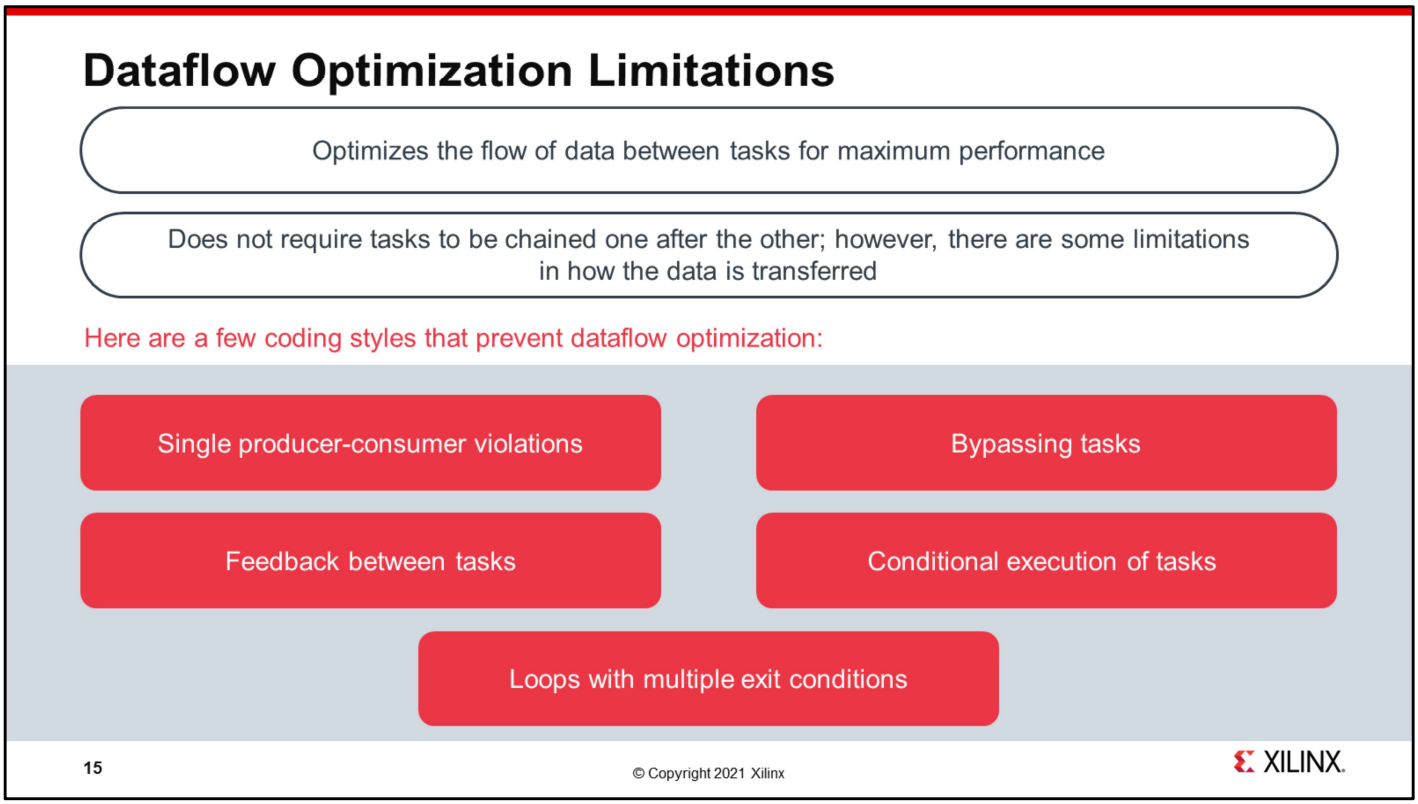
- If the HLS tool determines that the data is accessed in sequential order, it implements the memory channel as a FIFO channel of depth 1.
- If the HLS tool is unable to determine whether the data is accessed in either sequential order or an arbitrary manner, it implements the memory channel as a ping-pong buffer; that is, as two block RAMs each defined by the maximum size of the consumer or producer array.
- A ping-pong buffer ensures that the channel always has the capacity to hold all samples without a loss, providing the improved interval and safe and reliable data transfer.



To explicitly specify the default channel used between tasks, use the config_dataflow configuration. This configuration sets the default channel for all channels in a design.

To reduce the size of the memory used in the channel, you can use a FIFO.

To explicitly set the depth or number of elements in the FIFO, use the fifo_depth option.



Let's understand the limitations that dataflow optimization has.

Dataflow optimization optimizes the flow of data between tasks that is, between functions and loops which are pipelined for maximum performance.

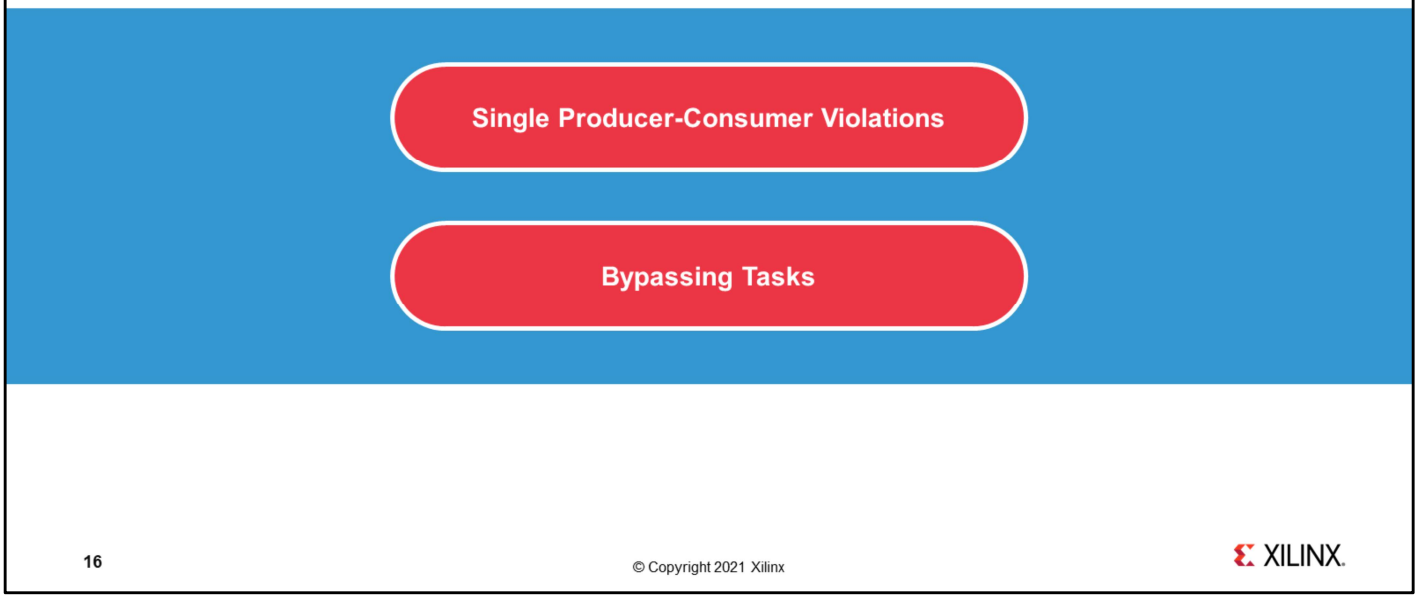
It does not require these tasks to be chained one after the other; however, there are some limitations in how the data is transferred.

Here are a few coding styles that prevent dataflow optimization:

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

Dataflow Optimization Limitations

Example



Let’s understand this in depth with the help of the examples of single producer-consumer violations and bypassing tasks.

Dataflow Optimization Limitations

Single Producer-Consumer Violations

- For the Vitis HLS tool to perform the dataflow optimization, all the elements passed between tasks must follow a single producer-consumer model
- Each variable must be driven from a single task and only be consumed by a single task

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

int temp1 [N];
Loop1: for(int i = 0; i <N; i++) {
      temp1 [i] = data_in[i] * scale;
}
Loop2: for(int j = 0; j < N; j++) {
      data_out1[j] = temp1 [j] * 123;
}
Loop3: for(int k = 0; k < N; K++) {
      data_out2[j] = temp1 [k] * 456;
}

}
```


XILINX.

Single Producer-Consumer Violations

- For the Vitis HLS tool to perform dataflow optimization, all the elements passed between tasks must follow a single producer-consumer model.
- Each variable must be driven from a single task and only be consumed by a single task.
- In the code example here, temp1 fans out and is consumed by both Loop2 and Loop3. This violates the single producer-consumer model.

Dataflow Optimization Limitations

Single Producer-Consumer Violations

Solution:

- A modified version of this code uses the function Split to create a single producer-consumer design. The function Split just copies the input to the two outputs
- In this case, data flows from Loop1 to the function Split and then to Loop2 and Loop3
- The data now flows between all four tasks, and the Vitis HLS tool can perform dataflow optimization

```
void Split (in[N], out1[N], out [N]) {
// Duplicated data
  L1:for (int i=1; i<N; i++) {
    out1 [i] = in [i];
    out2 [i] = in [i];
  }
}
void foo(int data_in[N], int scale, int data_out1 [N], int data_out2[N]) {

  int temp1[N], temp2[N] . temp3[N];
  Loop1: for (int i = 0; i < N; i++) {
    temp1[i] = data_in[i] * scale;
  }
  Split (temp1, temp2, temp3);
  Loop2: for(int j = 0; j < N; j++) {
    data_out1[j] = temp2[j] * 123;
  }
  Loop3: for(int k = 0; k < N; k++) {
    data_out2[j] = temp3[k] * 456;
  }
}
```


XILINX.

Let's look at the solution now.

- A modified version of this code uses the function Split to create a single producer-consumer design. The function Split just copies the input to the two outputs.
- In this case, data flows from Loop1 to the function Split and then to Loop2 and Loop3.
- The data now flows between all four tasks, and the Vitis HLS tool can perform dataflow optimization.

Dataflow Optimization Limitations

Bypassing Tasks

- User should not avoid the tasks, and this reduces the performance of the dataflow optimization
- Loop2 only uses the value of temp1
  - Value of temp2 is not consumed until after Loop2
- Therefore, temp2 bypasses the next task in the sequence, which limits the performance of the dataflow optimization

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

  int temp1(N), temp2(N). temp3(N);
  Loop1: for(int i = 0; i < N; i++) {
    temp1[i] = data_in[i] * scale;
    temp2[i] = data_in[i] >> scale;
  }
  Loop2: for(int j = 0; j < N; j++) {
    temp3[j] = temp1[j] + 123;
  }
  Loop3: for(int k = 0; k < N; k++) {
    data_out[j] = temp2[k] + temp3[k];
  }
}
```

19

XILINX.

Bypassing Tasks

- The user should not avoid tasks, and this reduces the performance of the dataflow optimization.
- In the code example here, Loop1 generates the values for temp1 and temp2.
- However, the next task, Loop2, only uses the value of temp1.
- The value of temp2 is not consumed until after Loop2.
- Therefore, temp2 bypasses the next task in the sequence, which limits the performance of the dataflow optimization.

Dataflow Optimization Limitations

Bypassing Tasks

Solution:

- Modify the code so that Loop2 consumes temp2 and produces temp4
- This ensures that the data flows from one task to the next

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

  int temp1[N], temp2[N]. temp3[N], temp4[N];
  Loop1: for(int i = 0; i < N; i++) {
    temp1[i] = data_in[i] * scale;
    temp2[i] - data_in[i] >> scale;
  }
  Loop2: for(int j = 0; j < N; j++) {
    temp3[j] = temp1[j] + 123;
    temp4[j] = temp2[j];
  }
  Loop3: for(int k = 0; k < N; k++) {
    data_out(j] = temp4[k] + temp3[k]
  }
}
```

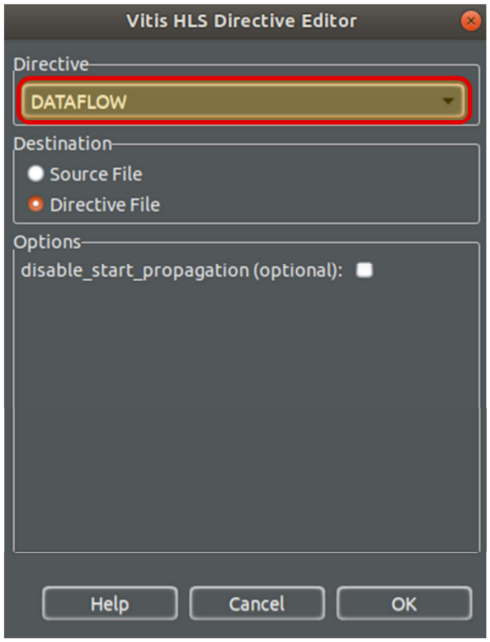

Let's look at the solution now.

- The solution is to modify the code so that Loop2 consumes temp2 and produces temp4 as follows.
- This ensures that the data flows from one task to the next.

Dataflow Optimization

The throughput rate of the design will be defined by the maximum throughput of the functions or loops

To improve the throughput rate:
- Pipeline the loops and functions
- Apply the dataflow on them



The Vitis HLS tool provides the DATAFLOW directive, through which the dataflow optimization can be set.

The throughput rate of the design will be defined by the maximum throughput of the functions or loops.

To improve the throughput rate, the first step is to pipeline the loops and functions, then apply the dataflow on them.

Dataflow Optimization

Vitis HLS tool uses the ping-pong memory buffers by default between the functions

- Memory size is defined by the maximum number of producer or consumer elements

Vitis HLS tool will determine if a FIFO can be used in place of a ping-pong buffer between the loops

- Memories can be specified to be FIFOs using the dataflow configuration
  - Select Solution > Solution Settings > General and expand config_dataflow
- With FIFOs, you can override the default size of the FIFO

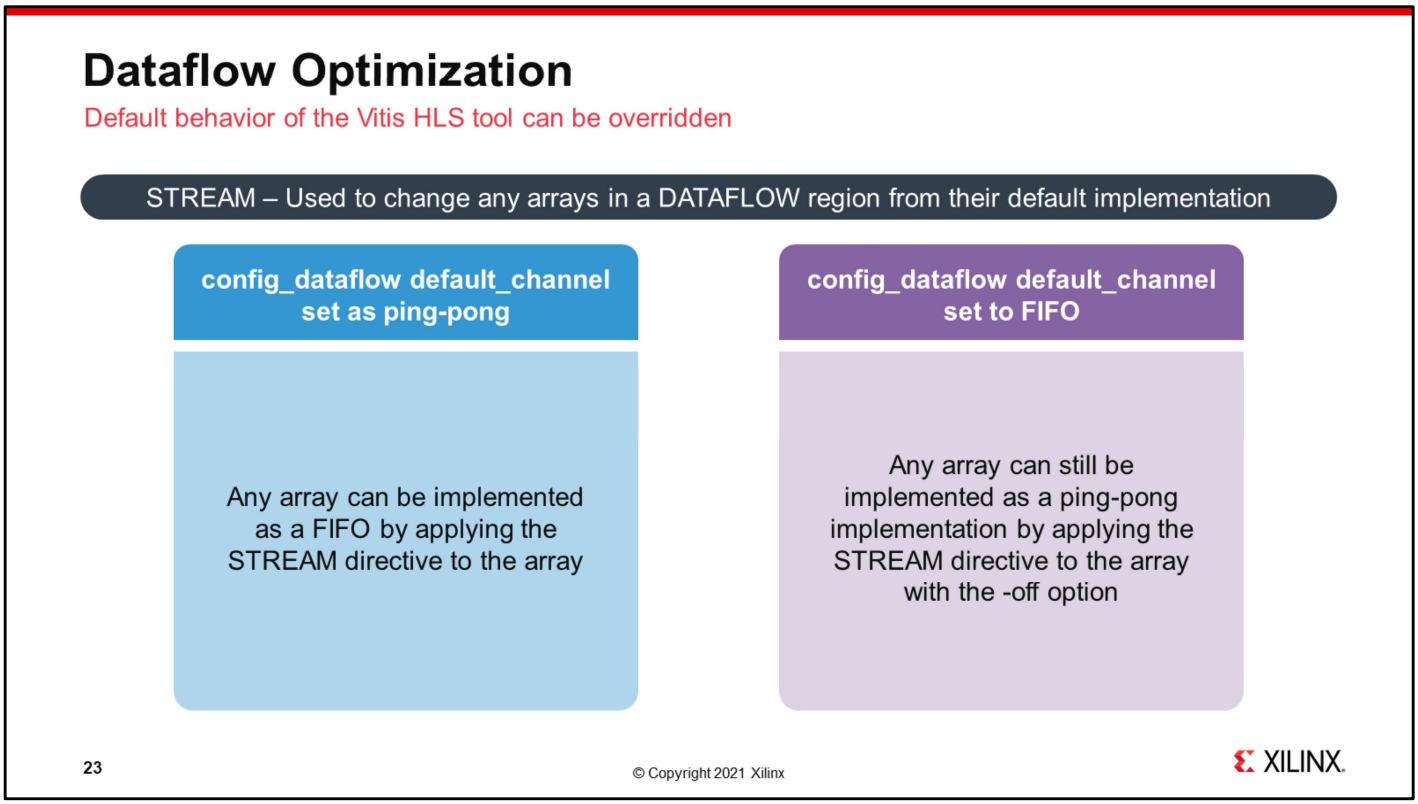
XILINX.

Generally, the Vitis HLS tool uses the ping-pong memory buffers by default, between the functions whose memory size is defined by the maximum number of producer or consumer elements.

Between the loops, the Vitis HLS tool will determine if a FIFO can be used in place of a ping-pong buffer where memories can be specified to be FIFOs using the dataflow configuration by selecting from the menu Solution > Solution Settings > General and expanding config_dataflow.

With FIFOs, you can override the default size of the FIFO.

Note: Setting the FIFO too small can result in an RTL verification failure.



The default behavior of the Vitis HLS tool can be overridden.

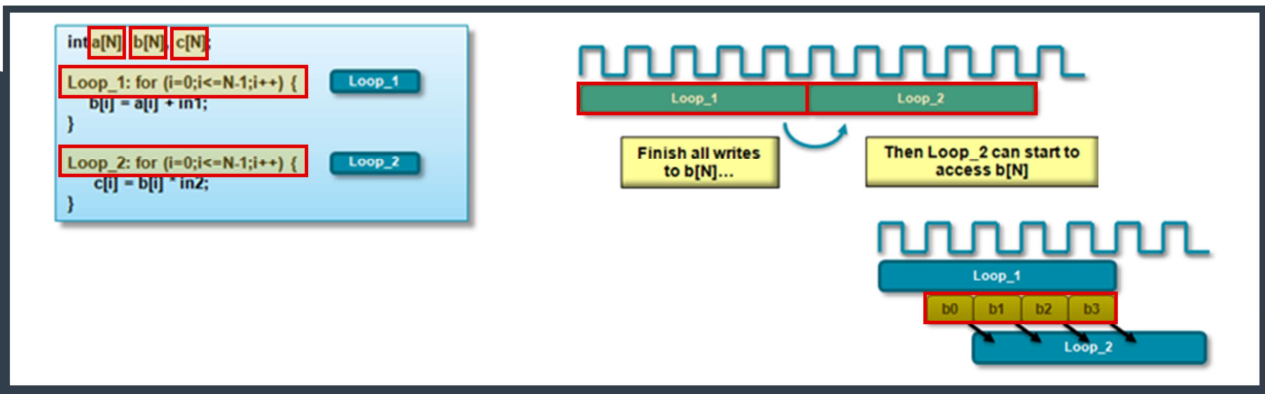
The STREAM directive is used to change any arrays in a DATAFLOW region from their default implementation.

- If config_dataflow default_channel is set as ping-pong, any array can be implemented as a FIFO by applying the STREAM directive to the array. Note that for a FIFO implementation to be used, the array must be accessed in a streaming manner.
- If config_dataflow default_channel is set to FIFO or the Vitis HLS tool has automatically determined that the data is accessed in a streaming manner, any array can still be implemented as a ping-pong implementation by applying the STREAM directive to the array with the -off option.

Dataflow: Ideal for Streaming Arrays and Multi-Rate Functions

Default behavior of the Vitis HLS tool can be overridden

Since the arrays are passed as single entities by default, the number of cycles required to complete an operation is high



Without DATAFLOW:

- Loop_2 starts when the Loop_1 finishes the writing of the data to b

With DATAFLOW:

- Loop_2 starts executing as soon as the data is ready, improving the throughput significantly

XILINX.

The dataflow directive allows the tasks to be overlapped, which makes it ideal for streaming arrays and multi-rate functions.

Since the arrays are passed as single entities by default, the number of cycles required to complete an operation is high.

The example code here has three arrays a, b, c and the two loops are used to assign values to arrays b and c.

Without DATAFLOW, each element of the array is accessed one by one, and the Loop_2 starts when the Loop_1 finishes the writing of the data to b.

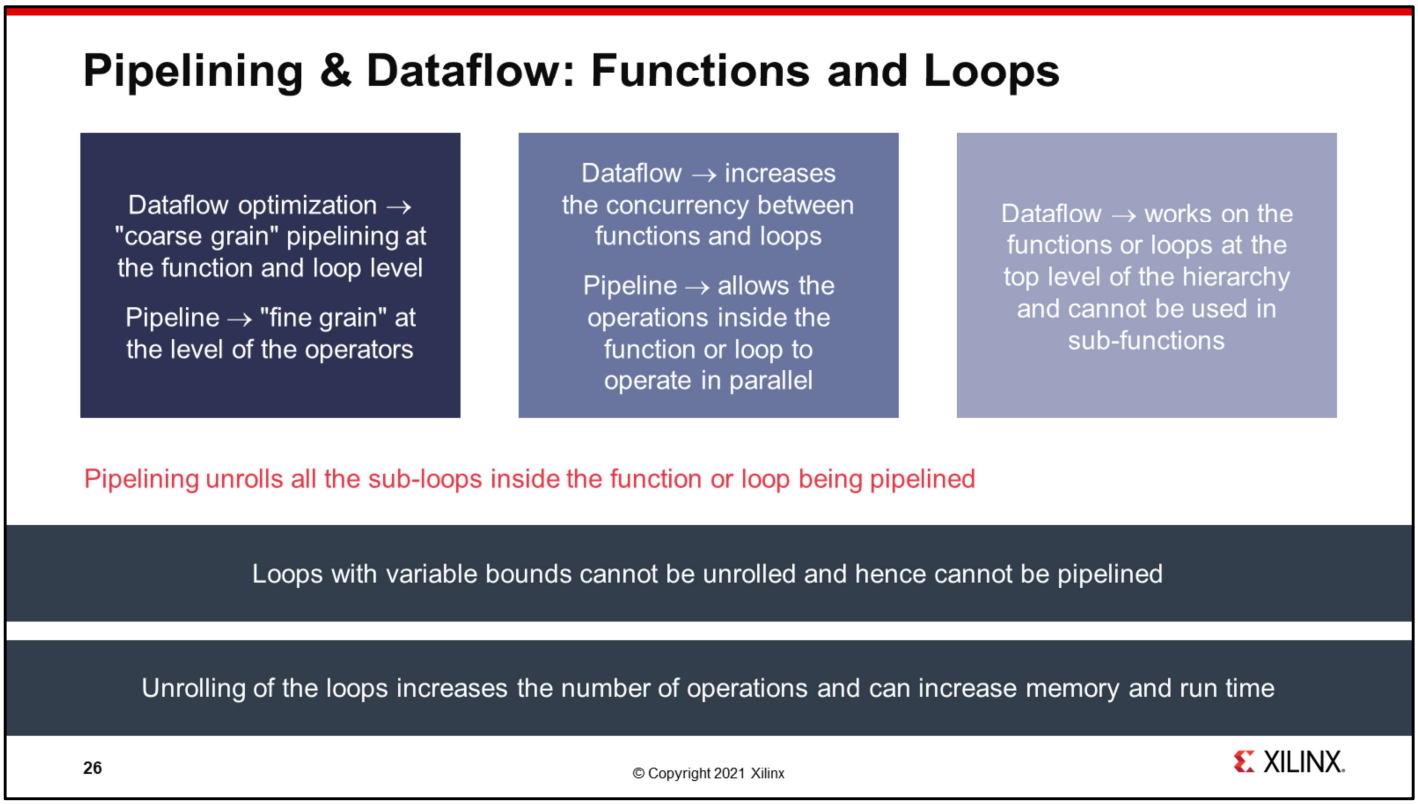
With DATAFLOW, the same two loops can be executed parallelly as shown here. Here, the Loop_2 starts executing as soon as the data is ready. This improves throughput significantly.

Dataflow: Ideal for Streaming Arrays and Multi-Rate Functions

In the case of multi-rate functions	To take maximum advantage of the dataflow in streaming designs
Dataflow buffers the data when one function or loop consumes or produces data a different rate from others	I/O interfaces at both ends of the data path should be streaming/handshake types (ap_hs or ap_fifo)

In the case of multi-rate functions, the dataflow buffers the data when one function or loop consumes or produces data a different rate from others.

To take maximum advantage of the dataflow in streaming designs, the I/O interfaces at both ends of the data path should be streaming/handshake types (ap_hs or ap_fifo).



Dataflow optimization is "coarse grain" pipelining at the function and loop level whereas the pipelining is "fine grain" at the level of the operators (*, +, >>, etc.).

The dataflow increases the concurrency between functions and loops and the pipelining allows the operations inside the function or loop to operate in parallel.

The dataflow only works on the functions or loops at the top level of the hierarchy and cannot be used in sub-functions.

The pipelining unrolls all the sub-loops inside the function or loop being pipelined.

- The loops with variable bounds cannot be unrolled and hence cannot be pipelined.
- Unrolling of the loops increases the number of operations and can increase memory and run time.

Apply Your Knowledge

Multiple Choice Question

The Vitis HLS tool implements a channel between sub-functions as _____. (Select all that apply)

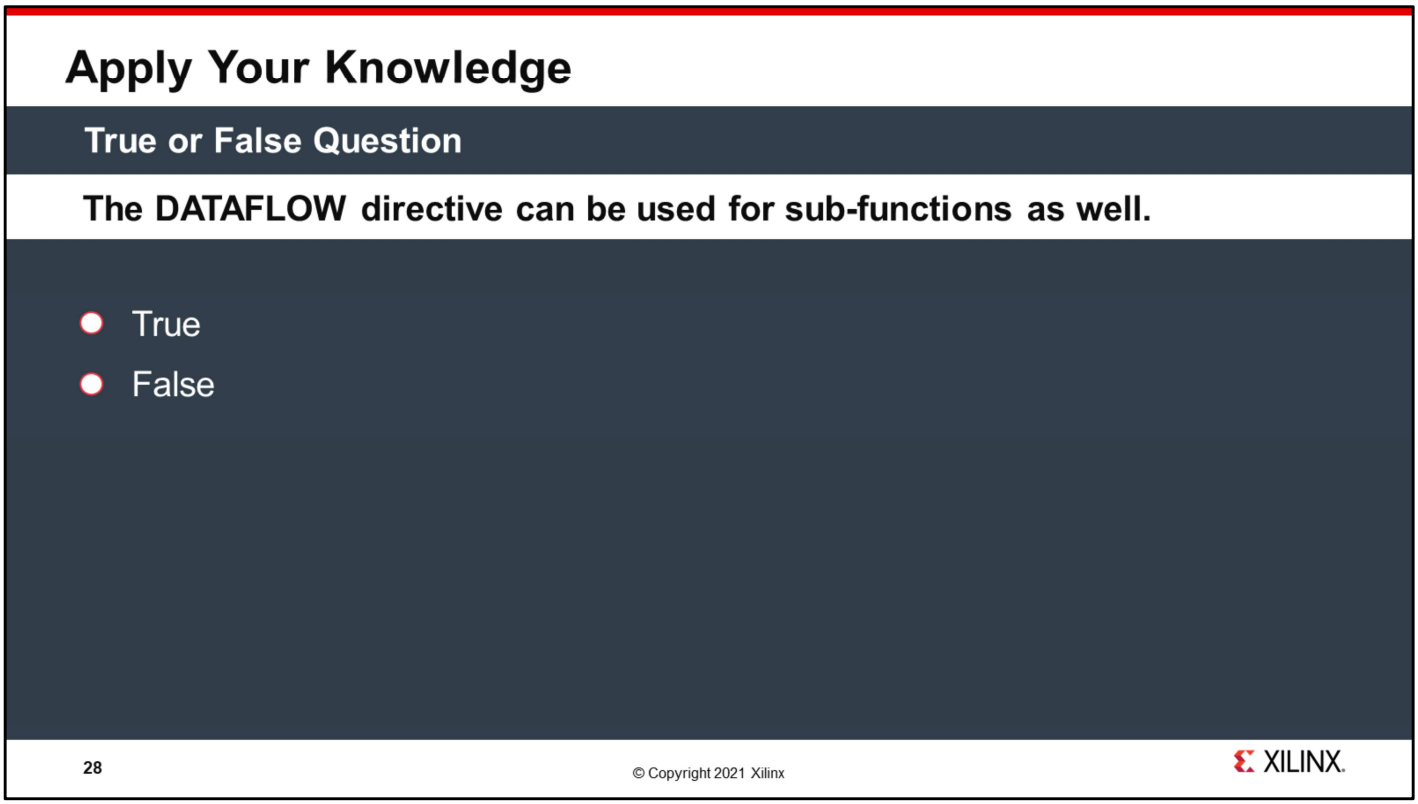
- ☐ LIFO
- ☐ FIFO
- ☐ Ping-pong
- ☐ None of the above

Correct answers:
"FIFO" & "Ping-pong"

Correct feedback:
You selected the correct answer.

Incorrect feedback:
The correct answers are "FIFO" & "Ping-pong"

Try again feedback:
Hint: Refer to the "Configuring the Dataflow Channel" slide.



Correct answer:
False

Correct feedback:
You selected the correct answer.

Incorrect feedback:
The DATAFLOW directive cannot be used in sub-functions.

Summary

- DATAFLOW directive: Enables task-level pipelining, allowing functions and loops to execute concurrently
  - Used to minimize interval
- HLS tool implements a channel between sub-modules as either a ping-pong buffer or FIFO buffers
  - Additional memory blocks are added to the design
- Coding styles that prevent the Vitis HLS tool from performing DATAFLOW optimization:
  - Single producer-consumer violations
  - Bypassing tasks