XILINX.

Optimizing for Throughput

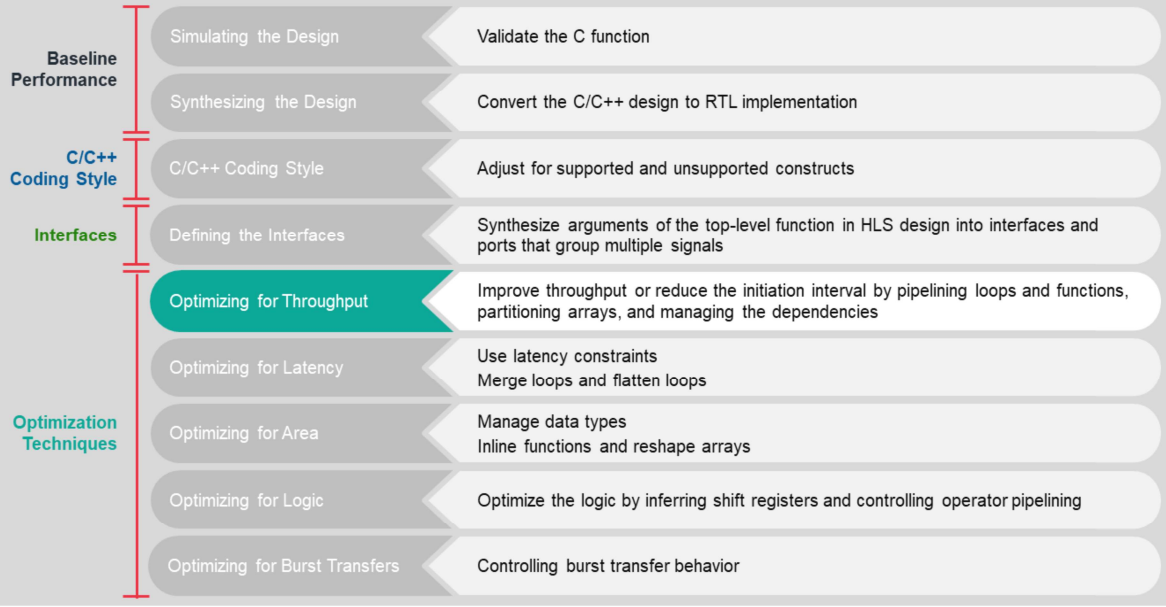
2021.1

Objectives

After completing this module, you will be able to:

- Describe arrays and their performance limitations in C
- Identify some of the techniques that optimize array performance

HLS Design Methodology



XILINX.

After the initial optimization step, the design is pipelined for performance.

HLS Design Methodology

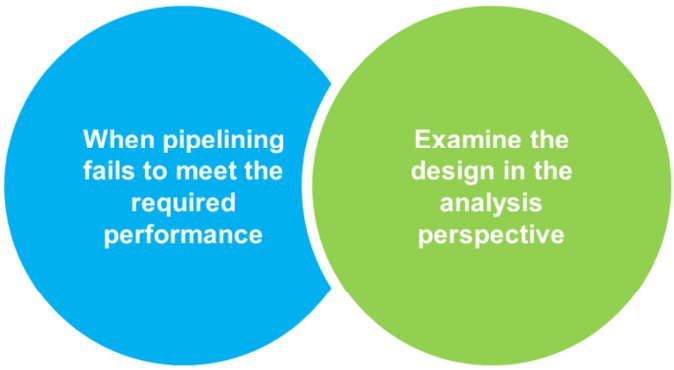
C code can contain descriptions that prevent a function or a loop from being pipelined with the required performance → these issues can be addressed by using other optimization directives

XILINX.

But sometimes C code can contain descriptions that prevent a function or a loop from being pipelined with the required performance.

In some cases, this might require a code modification but, in most cases, these issues can be addressed by using other optimization directives.

Optimize for Throughput



Issues that might be encountered while pipelining the functions and loops:

→ Addressed using optimization directives and configurations

→ Reduce bottlenecks in data structures

XILINX.

When pipelining fails to meet the required performance, the key to addressing the issue is to examine the design in the analysis perspective.

The issues that might be encountered while pipelining the functions and loops are likely to address using the optimization directives and configurations by helping to reduce the bottlenecks in data structures.

Optimize for Throughput

Directives and Configurations	Description
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers to improve access to data and remove block RAM bottlenecks
DEPENDENCE	Used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals)
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead
UNROLL	Unroll for loops to create multiple independent operations rather than a single collection of operations

The table shows the different directives and configurations.

- The ARRAY_PARTITION directive partitions the large arrays into multiple smaller arrays or individual registers, to improve the access to data and remove block RAM bottlenecks.
- The DEPENDENCE directive provides additional information that can overcome loop-carry dependencies and is used to remove the implied dependencies when pipelining the loops
- The INLINE directive removes the function boundaries. This can be used to bring the logic or loops up one level of a hierarchy, thus improving the latency by reducing the function call overhead.
- The UNROLL directive is used where a loop cannot be pipelined with the required initiation interval. It unrolls loops to create multiple independent operations (more logic) rather than a single collection of operations. It helps in removing a potential bottleneck.

Optimize for Throughput

Directives and Configurations	Description
CONFIG ARRAY PARTITION	Determines how arrays are partitioned, including global arrays and if the partitioning impacts array ports
CONFIG COMPILE	Controls synthesis-specific optimizations such as the automatic loop pipelining and floating-point math optimizations
CONFIG SCHEDULE	Determines the effort level to use during the synthesis scheduling phase and the verbosity of the output messages
CONFIG_UNROLL	Allows all loops below the specified number of loop iterations to be automatically unrolled

- The Config Array Partition configuration determines how arrays are partitioned, including global arrays and if the partitioning impacts the array ports.
- The Config Compile configuration controls synthesis specific optimizations such as the automatic loop pipelining and floating-point math optimizations.
- The Config Schedule determines the effort level to use during the synthesis scheduling phase, the verbosity of the output messages, and specify if II should be relaxed in pipelined tasks to achieve timing.
- The CONFIG_UNROLL allows all loops below the specified number of loop iterations to be automatically unrolled.

Arrays: Performance Bottlenecks

Arrays are intuitive and useful software constructs

Allow the C algorithm to be easily captured and understood

During the synthesis process, the array is implemented as a RAM

- **Single-port RAM:** Impossible to pipeline for loop to process a new loop iteration every clock cycle
- **Dual-port RAM:** Allows only two accesses per clock cycle

Three reads are required to calculate the value of the sum, and so three accesses per clock cycle are required to pipeline the loop with a new iteration every clock cycle

Thus, the array in this code limits performance and creates a bottleneck

Let's look at the performance bottlenecks for arrays.

Arrays are intuitive and useful software constructs. They allow the C algorithm to be easily captured and understood.

The example code here shows a case in which accesses to an array can limit performance in the final RTL design.

There are three accesses to the array mem[N] to create a summed result. During the synthesis process, the array is implemented as a RAM.

If the RAM is specified as a single-port RAM, it is impossible to pipeline for a loop to process a new loop iteration every clock cycle. If the RAM is specified as a dual-port RAM, it allows only two accesses per clock cycle.

Three reads are required to calculate the value of the sum, and so three accesses per clock cycle are required to pipeline the loop with a new iteration every clock cycle.

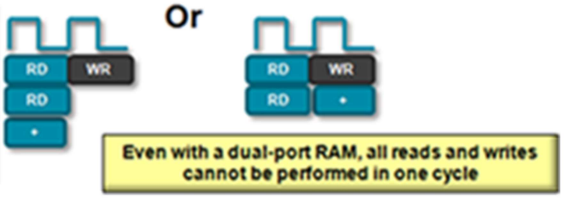
Thus, the array in this code limits performance and creates a bottleneck.

Arrays: Performance Bottlenecks

Arrays are intuitive and useful software constructs

Allow the C algorithm to be easily captured and understood

```
void foo_top (...) {
  ...
  for (i = 2; i < N; i++)
    mem[i] = mem[i-1] +mem[i-2];
  ...
}
```



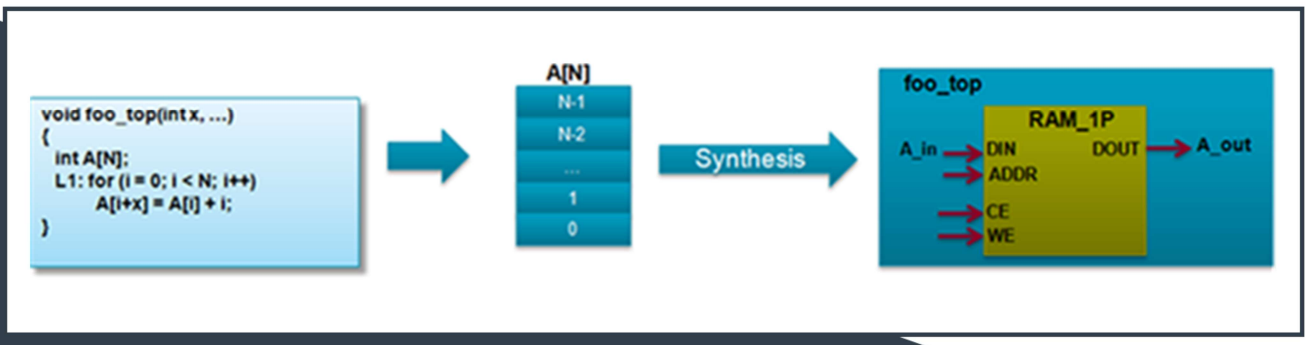
Solution:

- Array can be partitioned and reshaped to produce higher data bandwidth
- Allows more optimal configuration of the array
- Provides a better implementation of the memory resource

XILINX.

To avoid this, the array can be partitioned and reshaped to produce higher data bandwidth. It also allows more optimal configuration of the array and provides a better implementation of the memory resource.

Arrays in HLS



Arrays in the C code → Block RAM elements in RTL

To use a FIFO instead of a block RAM → STREAM directive

Arrays are automatically specified as streaming:

- If an array is set as interface type ap_fifo, axis, or ap_hs, etc
- If the arrays are used in a region where the DATAFLOW optimization is applied

All other arrays must be specified as streaming using the STREAM directive if a FIFO is required for the implementation

XILINX.

Let's understand how arrays function in HLS.

By default, all the arrays in the C code are implemented as block RAM elements in RTL, unless complete partitioning reduces them to individual registers. To use a FIFO instead of a block RAM, the array must be specified as streaming using the STREAM directive.

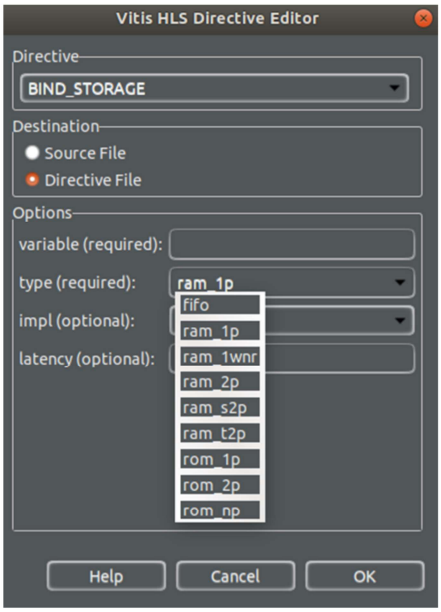
The arrays are automatically specified as streaming:

- If an array on the top-level function interface is set as interface type ap_fifo, axis, or ap_hs, etc.
- If the arrays are used in a region where the DATAFLOW optimization is applied

All other arrays must be specified as streaming using the STREAM directive if a FIFO is required for the implementation.

Arrays in HLS

Any memory resource in the library can be targeted while using the arrays

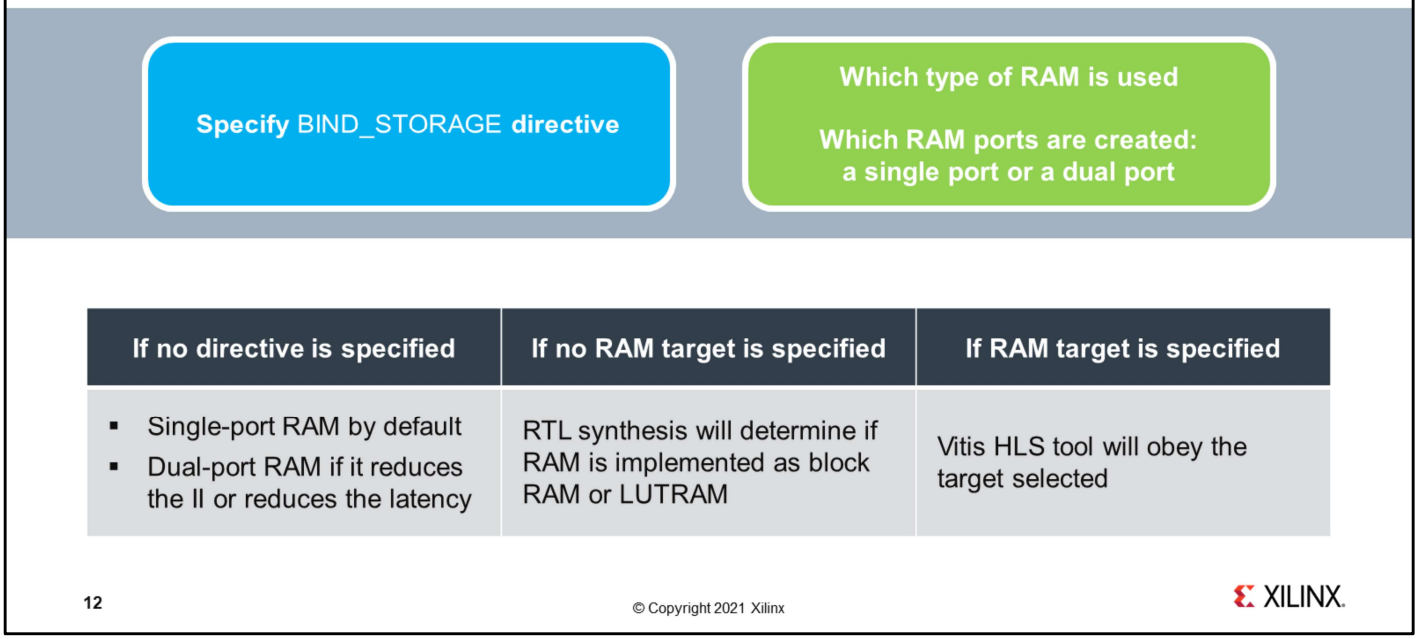


XILINX.

Any memory resource in the library can be targeted while using the arrays.

For example, array A is targeted to single port distributed RAM here from the list of available cores. The library model defines the ports and sequential operation.

Array and RAM Selection



If no directive is specified	If no RAM target is specified	If RAM target is specified
▪ Single-port RAM by default ▪ Dual-port RAM if it reduces the II or reduces the latency	RTL synthesis will determine if RAM is implemented as block RAM or LUTRAM	Vitis HLS tool will obey the target selected

Let's now understand how Array and RAM selection is done.

The BIND_STORAGE directive can explicitly specify which type of RAM is used, and therefore which RAM ports are created: a single port or a dual port.

If no directive is specified, the Vitis HLS tool uses:
- A single-port RAM by default
- A dual-port RAM if it reduces the initiation interval or reduces the latency

If no RAM target is specified, RTL synthesis will determine if RAM is implemented as block RAM or LUTRAM.

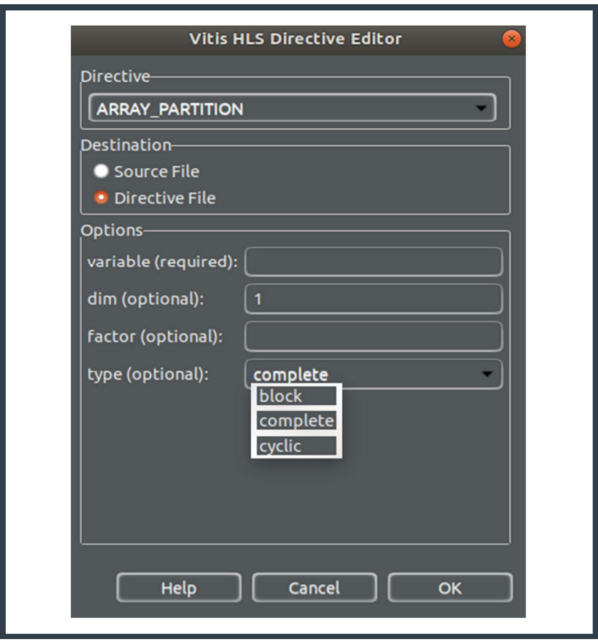
If the user specifies the RAM target, the Vitis HLS tool will obey the target selected.

Array Partitioning

Partitions the large arrays into multiple smaller arrays or individual registers to improve parallel access to data and remove block RAM bottlenecks

Types of array partitioning:

- **Block:** Original array is split into equally sized blocks of consecutive elements of the original array
- **Complete:** Default operation is to split the array into its elements. This corresponds to resolving a memory into registers
- **Cyclic:** Original array is split into equally sized blocks, interleaving the elements of the original array



XILINX.

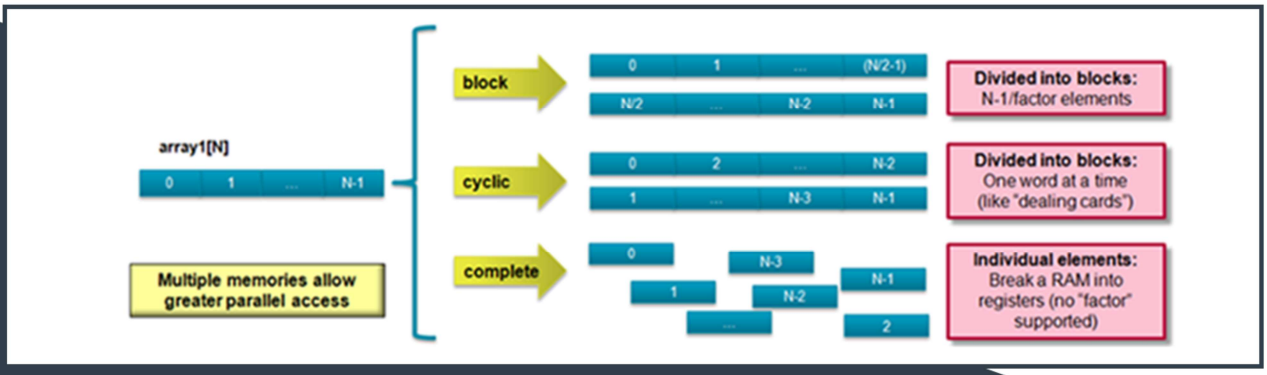
Let's understand array partitioning in detail.

Arrays are partitioned using the ARRAY_PARTITION directive. ARRAY_PARTITION directive partitions the large arrays into multiple smaller arrays or individual registers to improve parallel access to data and remove block RAM bottlenecks.

The Vitis HLS tool provides three types of array partitioning: block, complete, and cyclic.

- In block partitioning, the original array is split into equally sized blocks of consecutive elements of the original array.
- In complete partitioning, the default operation is to split the array into its elements. This corresponds to resolving a memory into registers.
- In cyclic partitioning, the original array is split into equally sized blocks, interleaving the elements of the original array.

Array Partitioning



For block and cyclic partitioning:

- Factor option specifies the number of arrays that are created
- Graphic here shows a factor of 2 is used; that is, the array is divided into two smaller arrays

For complete partitioning:

- Factor option is not supported, and a RAM is broken into registers

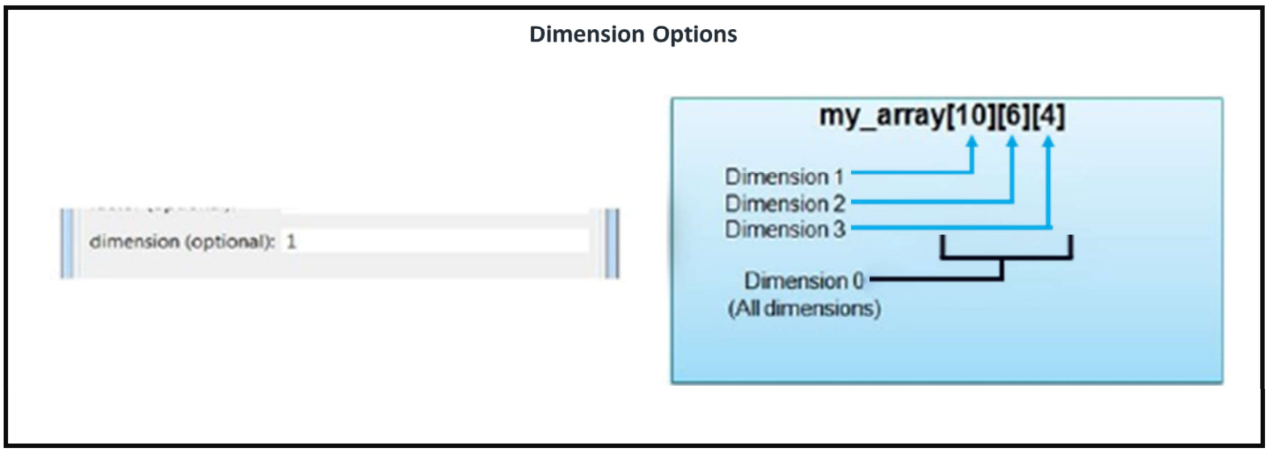
XILINX.

For block and cyclic partitioning, the factor option specifies the number of arrays that are created. The graphic here shows a factor of 2 is used; that is, the array is divided into two smaller arrays.

For complete partitioning, the factor option is not supported, and a RAM is broken into registers.

Array Dimensions

How can we specify which dimension is partitioned in multi-dimensional partitioning?

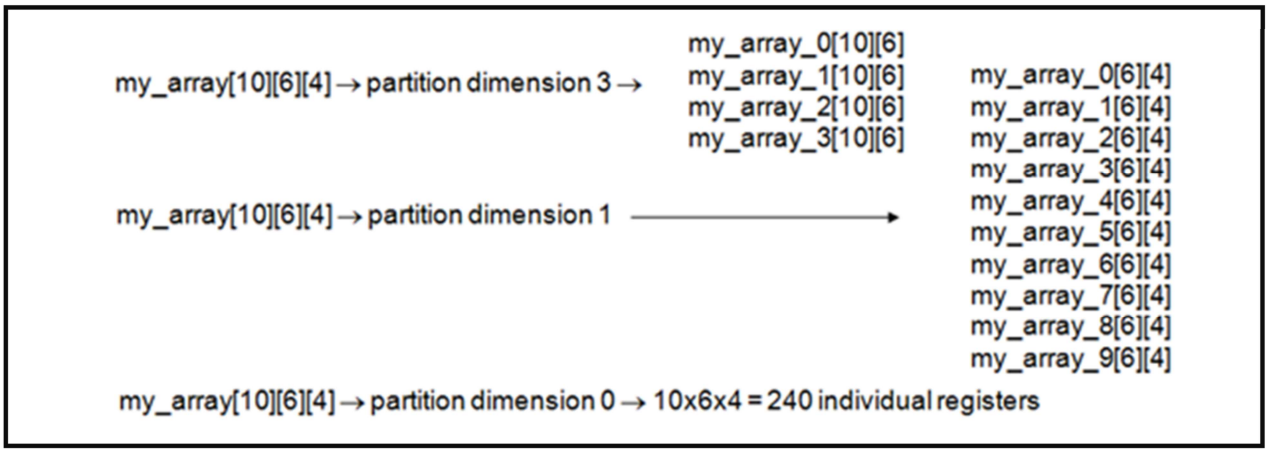


XILINX.

Now the question arises: how we can specify which dimension is partitioned in multi-dimensional partitioning?

The dimension option is used to specify which dimension is partitioned. If you have an array named my_array [10] [6] [4] as shown, then dimension 1 is 10, the dimension 2 is 6, and dimension 3 is 4.

Array Dimensions



16

XILINX.

The examples here demonstrate:

- How partitioning dimension 3 results in 4 separate arrays.
- Partitioning dimension 1 results in 10 separate arrays.
- If zero is specified as the dimension, all dimensions are partitioned.

config_array_partition

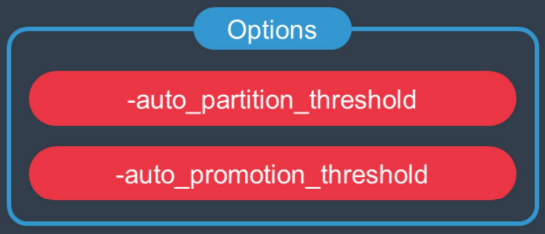
Use the config_array_partition command to improve throughput

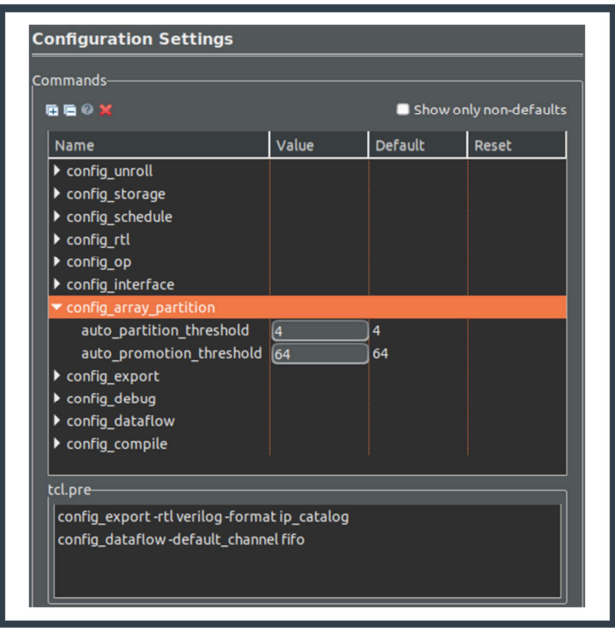
Syntax: config_array_partition [OPTIONS]

Specifies the default behavior for array partitioning

Determines how the arrays are automatically partitioned based on the number of elements

Solution > Solution Settings > General > Add > config_array_partition





XILINX.

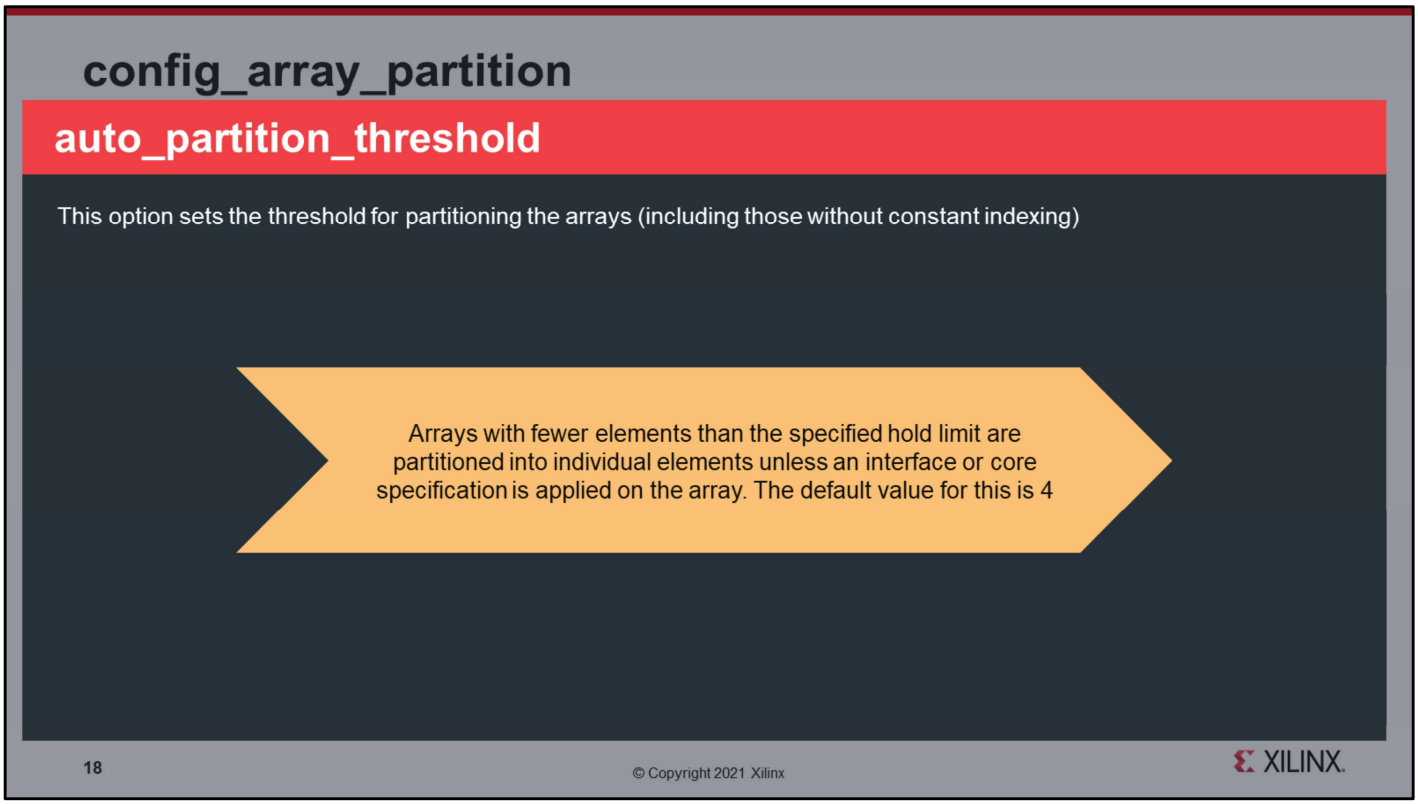
The Vitis HLS tool partitions the arrays automatically by using the config_array_partition command to improve the throughput.

Its Syntax is config_array_partition [OPTIONS].

The config_array_partition configuration specifies the default behavior for array partitioning and determines how the arrays are automatically partitioned based on the number of elements. This configuration is accessed through the menu Solution > Solution Settings > General > Add > config_array_partition.

The following are the options under Config_array_partition:

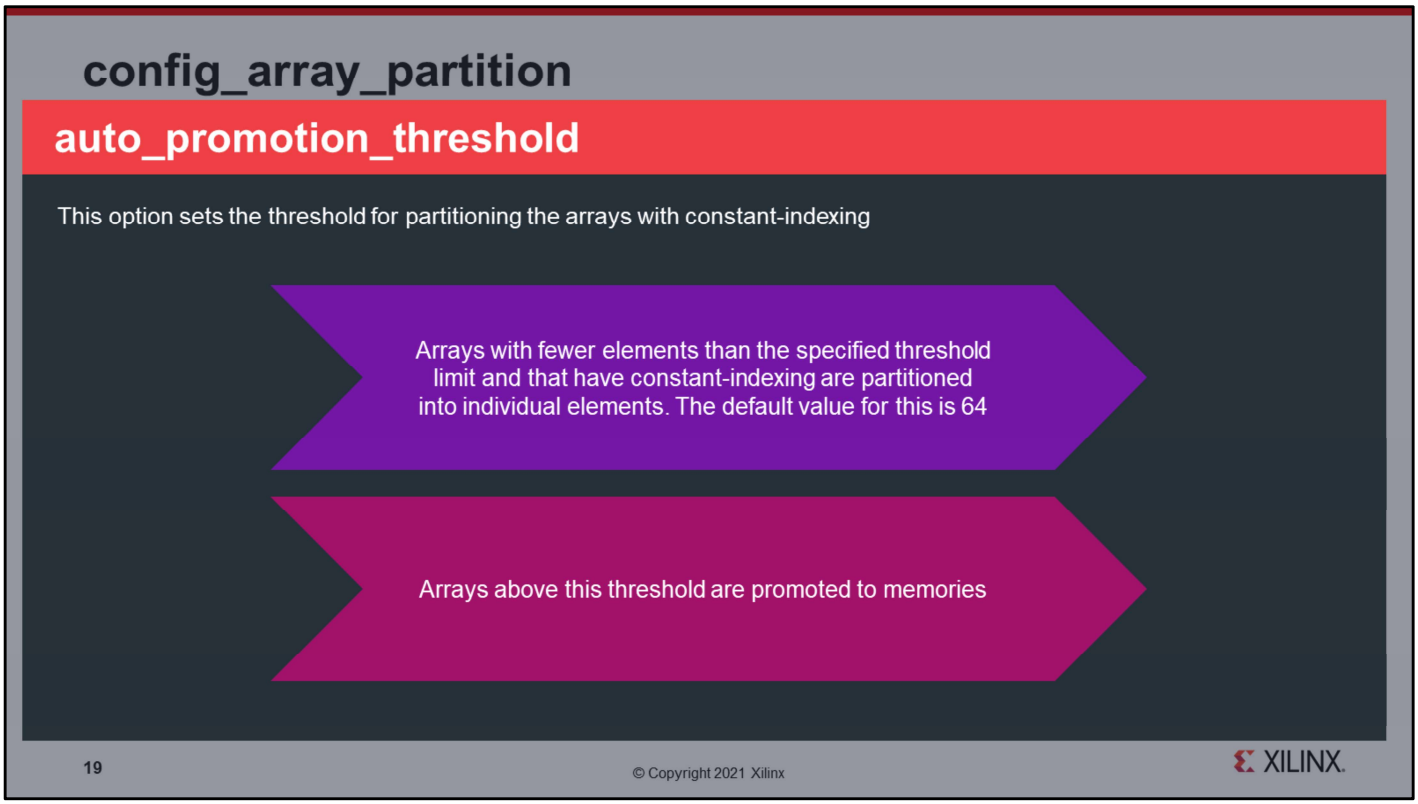
- auto_partition_threshold
- auto_promotion_threshold



auto_partition_threshold

The -auto_partition_threshold option in the config_array_partition command sets the threshold for partitioning the arrays (including those without constant indexing).

Arrays with fewer elements than the specified threshold limit are partitioned into individual elements unless an interface or core specification is applied on the array. The default value for this is 4.



auto_promotion_threshold
The -auto_promotion_threshold option in the config_array_partition command sets the threshold for partitioning the arrays with constant-indexing.

Arrays with fewer elements than the specified threshold limit and that have constant-indexing are partitioned into individual elements. The default value for this is 64. Arrays above this threshold are promoted to memories.

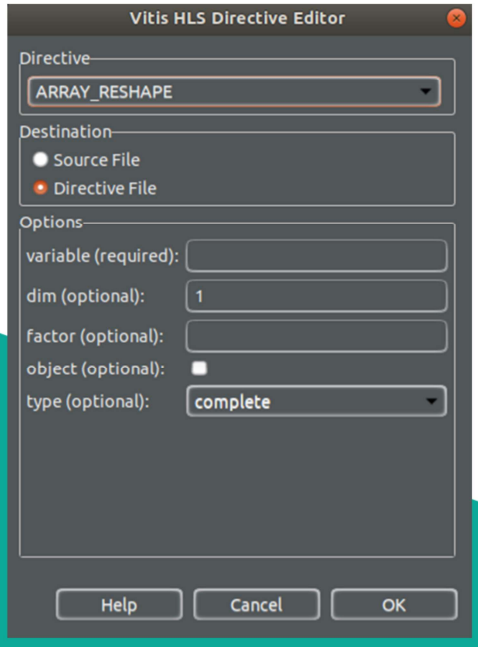
Array Reshaping

ARRAY_RESHAPE directive allows more data to be accessed in a single clock cycle

Reduces the number of block RAMs while still allowing the beneficial attributes of partitioning—parallel access to the data

Vitis HLS tool may automatically unroll any loops consuming this data and will try to improve the throughput

Reshaping recombines partitioned arrays back into a single array



XILINX.

Let's discuss the concept of array reshaping.

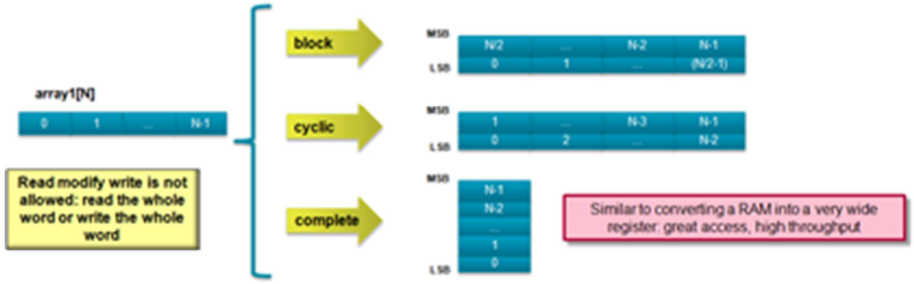
The ARRAY_RESHAPE directive allows more data to be accessed in a single clock cycle. It is used to reduce the number of block RAMs while still allowing the beneficial attributes of partitioning–parallel access to the data.

In cases where more data can be accessed in a single clock cycle, the Vitis HLS tool may automatically unroll any loops consuming this data and will try to improve the throughput. Sometimes, reshaping recombines partitioned arrays back into a single array.

Array Reshaping

ARRAY_RESHAPE directive transforms the arrays into the form shown here

```
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
...
}
```



XILINX.

The example code here shows how the ARRAY_RESHAPE directive is used and how it transforms the arrays into the form shown here.

Reshaping vs. Partitioning

Reshaping	Partitioning
Partitions large arrays into multiple smaller arrays	Partitions large arrays into multiple smaller arrays
Useful for increasing memory or data bandwidth	Useful for increasing memory or data bandwidth
Increases the width of the data word and does not increase the number of memory ports	Increases the number of memory ports; thus, increasing the number of I/Os to deal with

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

22

XILINX.

Let's now look at the difference between array reshaping and array partitioning.

Both the ARRAY_RESHAPE and the ARRAY_PARTITION directives partition large arrays into multiple smaller arrays. Both these directives are useful for increasing memory or data bandwidth.

The reshaping increases the width of the data word and does not increase the number of memory ports.

The partitioning increases the number of memory ports; thus, increasing the number of I/Os to deal with. This directive is used only if you have to use independent addressing.

The following is a common error message that a user can get due to using reshaping or partitioning.

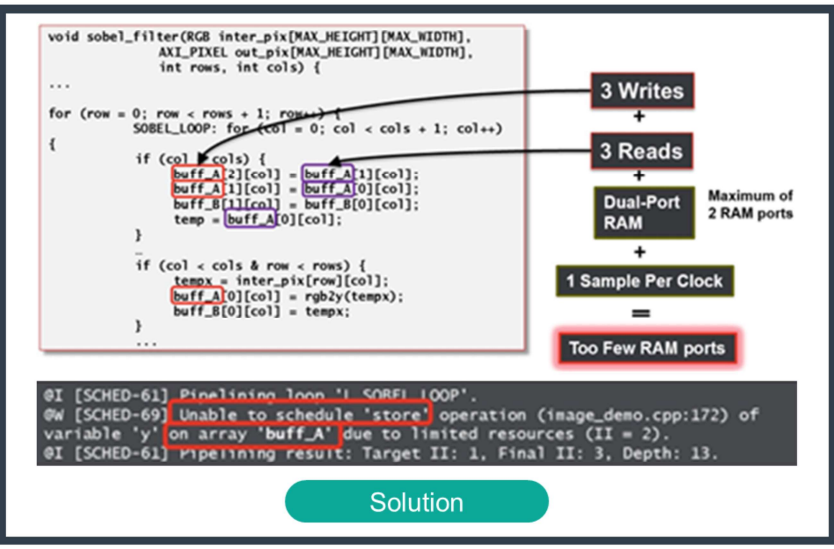
Bandwidth Issues

Array accesses used in the code can be bottlenecks inside the functions or loops
Sometimes it is difficult to achieve II = 1 even after applying the PIPELINE and DATAFLOW directives

Even if dual-port RAM is used here, still it is difficult to achieve II = 1 as there are very few RAM ports available for parallel access

Final II for this design is equal to 3

Warning:
"Unable to store operation of variable y on array buff_A due to limited resources."



XILINX.

Let's discuss the bandwidth issues that arise while using arrays.

The array accesses used in the code can be bottlenecks inside the functions or loops. Sometimes it is difficult to achieve II = 1 even after applying the PIPELINE and DATAFLOW directives.

For example, the sobel filter code shown here performs three reads and three writes.

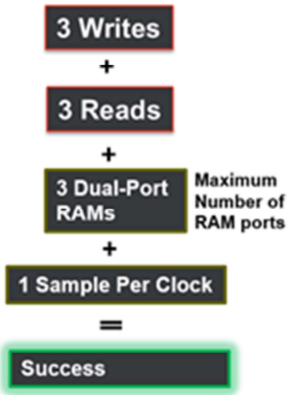
Even if the dual port RAM is used here, still it is difficult to achieve II = 1, as there are very few RAM ports available for parallel access.

The final II for this design is equal to 3, and the HLS tool throws a warning saying, "Unable to store operation of variable y on array buff_A due to limited resources."

Solution: Partitioning Arrays

Partitioning

```
void sobel_filter(RGB inter_pix[MAX_HEIGHT][MAX_WIDTH],
                  AXI_PIXEL out_pix[MAX_HEIGHT][MAX_WIDTH],
                  int rows, int cols) {
...
#pragma HLS ARRAY_PARTITION variable=buff_A dim=1 complete
    for (row = 0; row < rows + 1; row++) {
            SOBEL_LOOP: for (col = 0; col < cols + 1; col++)
{
            if (col < cols) {
                 buff_A[2][col] = buff_A[1][col];
                 buff_A[1][col] = buff_A[0][col];
                 buff_B[1][col] = buff_B[0][col];
                 temp = buff_A[0][col];
            }
            _
            if (col < cols & row < rows) {
                 tempx = inter_pix[row][col];
                 buff_A[0][col] = rgb2y(tempx);
                 buff_B[0][col] = tempx;
            }
            ...
```



In the same example, array buff_A is partitioned using ARRAY_PARTITION directive

This creates three dual-port RAMs for performing three reads and three writes

All the operations can now run parallelly, giving II = 1

XILINX.

The solution to the issue mentioned in the previous example code is partitioning. Partitioning improves the availability of data.

In the same example, the array buff_A is partitioned using ARRAY_PARTITION directive. This creates 3 dual port RAMs for performing 3 reads, and 3 writes.

All the operations can now run parallelly, giving II = 1, as shown.

Apply Your Knowledge

Multiple Choice Question

Arrays can often cause performance bottlenecks due to _____.

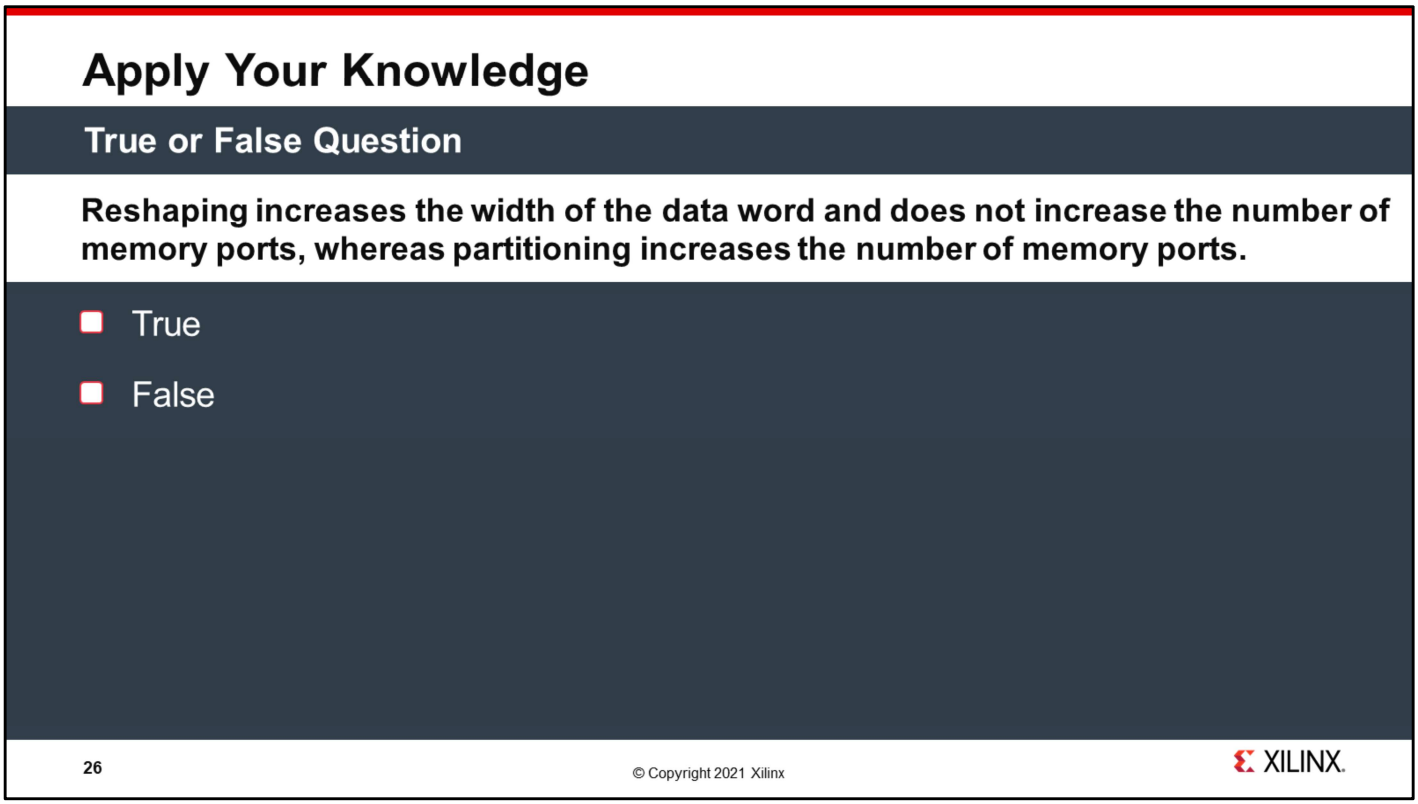
- Large number of accesses to the array
- Arrays are targeted to default RAM
- Fewer number of RAM ports available
- All of the above

Correct answer:
All of the above

Correct feedback:
You selected the correct answer

Incorrect feedback:
The correct answer is “All of the above”.

Try again feedback:
Hint: Refer to the “Arrays: Performance Bottlenecks” slide.



Correct answer:
True

Correct feedback:
You selected the correct answer.

Incorrect feedback:
The correct answer is “False”.

Summary

- The Vitis HLS tool will determine which type of RAM to implement depending on the types of access and requirements
  - Use the BIND_STORAGE directive to explicitly state which RAM to use
- Arrays are performance limiting
  - Arrays can create performance bottlenecks if not handled properly
  - Use array partitioning and reshaping directives to achieve throughput
- ARRAY_RESHAPE directive allows more data to be accessed in a single clock cycle