HLS Component Development

Using the Vitis Unified IDE 2024.1

Lab-2: Performance Optimization

Mar-2025

Abstract

This lab introduces various techniques and optimization directives that can be used to improve the performance of the design. The analysis includes a comparison of a methodology that optimizes at the loop level with one that optimizes at the function level.

Objectives

After completing this lab, you will be able to:

- Add optimization directives in your design
- Understand the effect of PIPELINE directive
- Understanding performance using analysis viewer
- Distinguish between affect of different optimization directives
- Improvements on the source code

Introduction

This exercise uses a matrix multiplication design to show how you can fully optimize a design heavily based on loops. The design goal is to read one sample per clock cycle using a FIFO interface, while minimizing the area.
This lab is based on matrix multiplication algorithm already created in Lab-1.

Lab Steps

Step 1: Open Lab-1

In this step, you will launch the Vitis Unified IDE tool and open the Lab-1.

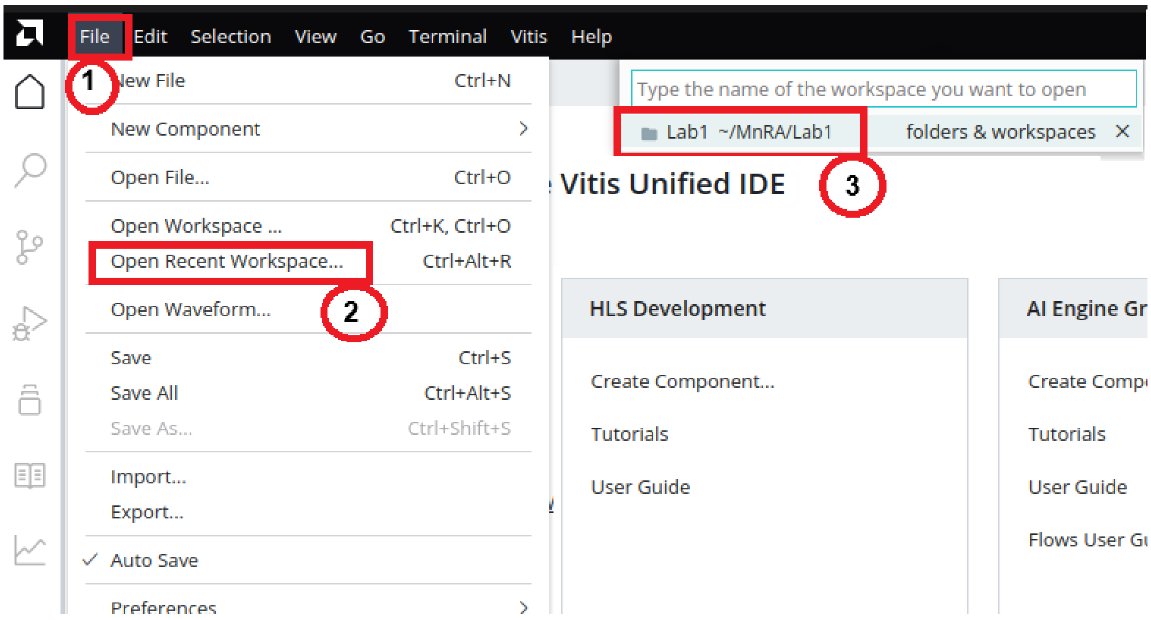
a. Click the ***Vitis Unified IDE 2024.1*** icon ( ) from the desktop to launch the tool.

b. Navigate to the toolbar and click **File** (1).

c. Click on **Open Recent Workspaces** (2).

d. In the Information pane a search bar with names of recent workspaces will appear (3).

e. Select **Lab1**.



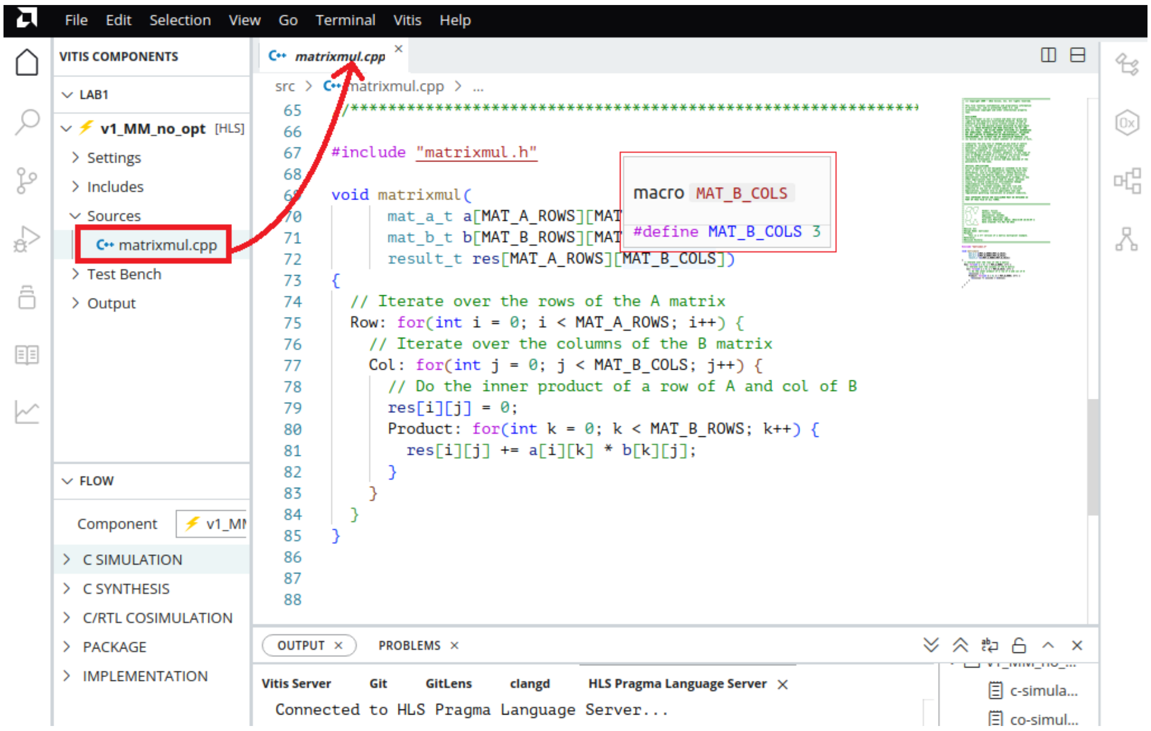
It will open your component, created during Lab-1.

a. Expand the **Sources** folder in the **Vitis Component** pane under the name of you component (`v1_MM_no_opt`).

b. Click **matrixmul.cpp** to open the file in the Information pane.

c. Review the code structures.

Scroll down the file to see that the source code has two input arrays, a and b, and output array res. Hold the mouse over the macros (as seen in following figure) to see that each is `three-by-three` for a total of `nine` elements.



```
//*****************************************************************

#include "matrixmul.h"

void matrixmul(
      mat_a_t a[MAT_A_ROWS][MAT
      mat_b_t b[MAT_B_ROWS][MAT
      result_t res[MAT_A_ROWS][MAT_B_COLS])
{
  // Iterate over the rows of the A matrix
  Row: for(int i = 0; i < MAT_A_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Col: for(int j = 0; j < MAT_B_COLS; j++) {
      // Do the inner product of a row of A and col of B
      res[i][j] = 0;
      Product: for(int k = 0; k < MAT_B_ROWS; k++) {
        res[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

```
macro MAT_B_COLS
#define MAT_B_COLS 3
```

```
Connected to HLS Pragma Language Server...
```

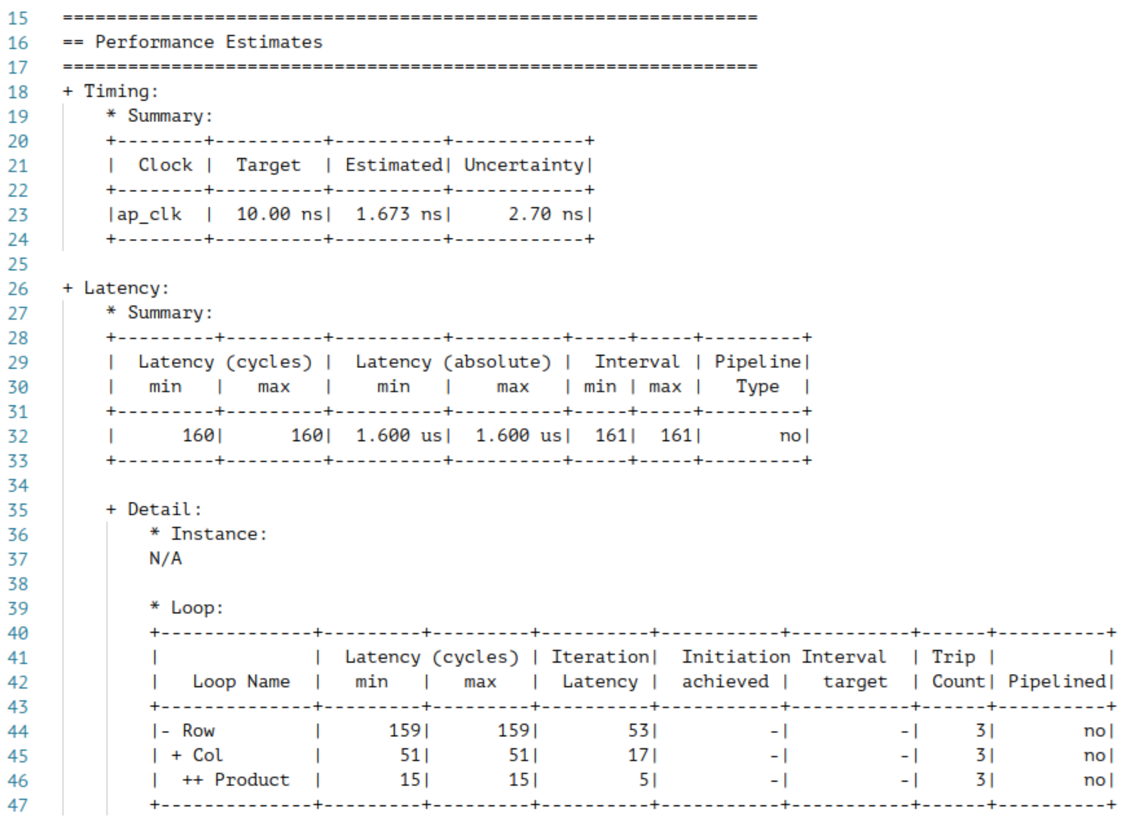
It can be seen that the design consists of three nested loops. The `Product` loop is the inner most loop performing the actual Matrix elements product and sum. The `Col` loop is the outer-loop which feeds the next column element data with the passed row element data to the Product loop. Finally, `Row` is the outer-most loop. The `res[i][j]=0` (line 79) resets the result every time a new row element is passed and new column element is used.

Step 2: Synthesize and Analyze the Design

Click the **Run** under **C SYNTHESIS** in the **Flow** Navigator to synthesize the design to RTL.
When synthesis successfully completes open the synthesis report to analyze performance. You can find Synthesis Report at following places.

- In Flow Navigator under `C Synthesis > Reports > Synthesis`.
- In Vitis Component Pane under `v1_MM_no_opt > Output > Syn > report > matrixmul_csynth.rpt`

The detailed report of the matrixmul synthesize, and the Performance estimates appears.



```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+----------+----------+------------+
    |  Clock |  Target  | Estimated| Uncertainty|
    +--------+----------+----------+------------+
    |ap_clk  |  10.00 ns|  1.673 ns|     2.70 ns|
    +--------+----------+----------+------------+

+ Latency:
    * Summary:
    +---------+---------+----------+----------+-----+-----+---------+
    |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
    |   min   |   max   |    min   |    max   | min | max |   Type  |
    +---------+---------+----------+----------+-----+-----+---------+
    |      160|      160|  1.600 us|  1.600 us|  161|  161|       no|
    +---------+---------+----------+----------+-----+-----+---------+

    + Detail:
        * Instance:
        N/A

        * Loop:
        +--------------+---------+---------+----------+-----------+-----------+------+----------+
        |              |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
        |   Loop Name  |   min   |   max   |  Latency |  achieved |   target  | Count| Pipelined|
        +--------------+---------+---------+----------+-----------+-----------+------+----------+
        |- Row         |      159|      159|        53|          -|          -|     3|        no|
        | + Col        |       51|       51|        17|          -|          -|     3|        no|
        |  ++ Product  |       15|       15|         5|          -|          -|     3|        no|
        +--------------+---------+---------+----------+-----------+-----------+------+----------+
```

- The interval is 161 clock cycles. Because there are nine elements in each input array, the design takes approximately nine cycles per input read.
- The interval is one cycle longer than the latency, so there is no parallelism in the hardware at this point.
- The latency/interval is due to nested loops.

Product loop (innermost):

- Has a latency of 5 clock cycles.
- Has 15 clock cycles total for all 3x iterations.

Col loop:

- It requires 1 clock to enter loop Product and 1 clock to exit
- It takes 17 clock cycles for each iteration (1+15+1).
- Has 51 cycles for all 3x iterations to complete.

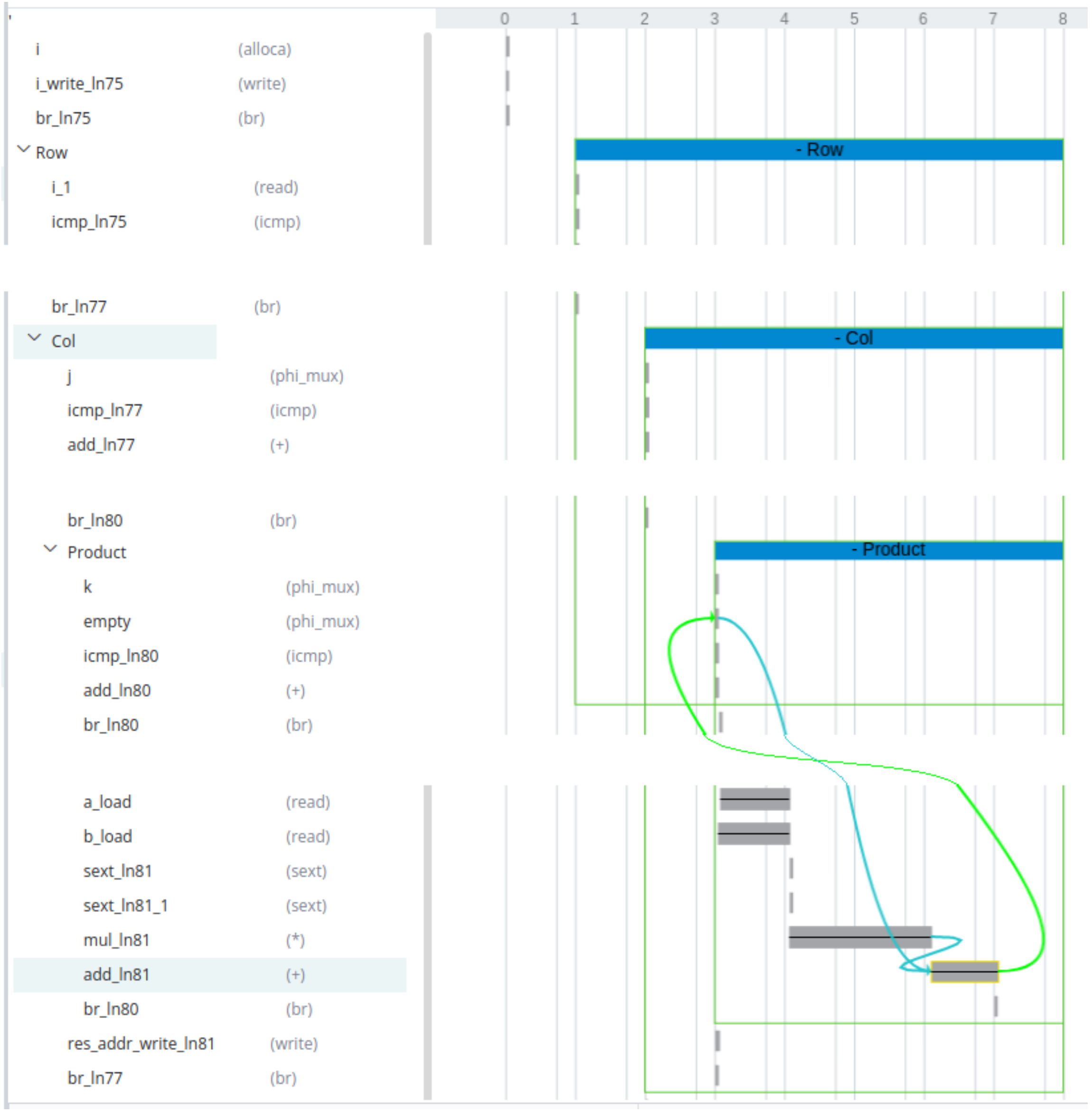
Row loop (top-level loop):

- It requires 1 clock to enter Col loop and 1 clock to exit
- It has a latency of 53 (1+51+1) clock cycles per iteration.
- It takes 161 clock cycles in total for all 3x iterations of the loop.

You can do one of two things to improve the initiation interval: Pipeline the loops or Pipeline the entire function.You begin by pipelining the loops and then compare those results to pipelining the entire function.
When pipelining loops, the initiation interval of the loops is the important metric to monitor. As seen in this exercise, even when the design reaches the stage at which the loop can process a sample every clock cycle, the initiation interval of the function is still reported as the time it takes for the loops contained within the function to finish processing all data for the function.

Step 3: Pipeline the Product Loop

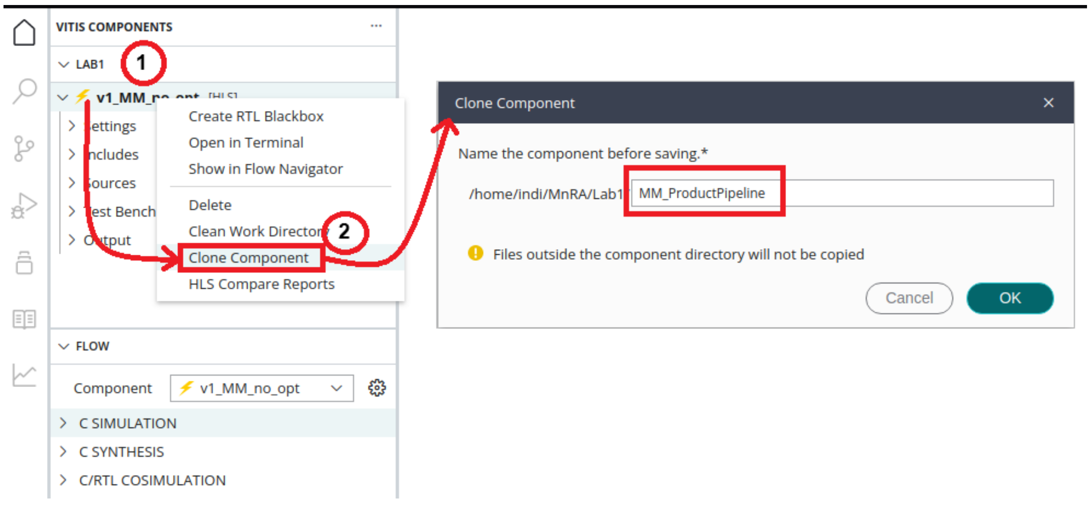
If we look at the schedule viewer from last lab, we can see the following:



Notice that the operations to `load` the two variables **a** and **b**, `multiply` them and finally `sum` the result happens across multiple states. To improve the latency, we could pipeline these operations. This means that while one iteration is multiplying the two variables, the variables for the next multiply operation are already loaded. The same happens when summing the results together: the variables of the next iteration are already multiplied and the variables of the iterations after that are already read.

3.1 Create New Implementation of Component

- Create a new implementation by Cloning the component `v1_MM_no_opt`.
- Right click on the component name `v1_MM_no_opt` in Vitis Component pane.
- Select **Clone Component**.
- Type the component name in `MM_ProductPipeline` in the dialogue that appeared.

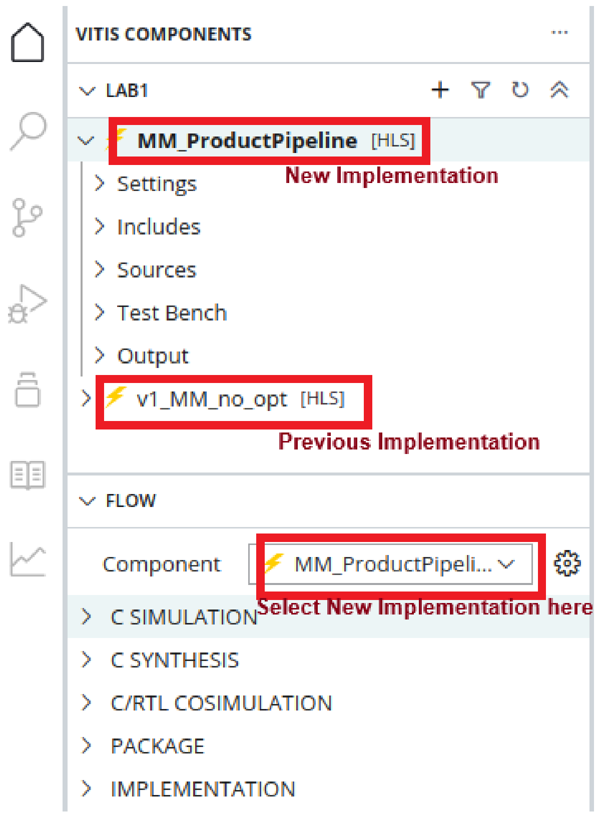


Name the component before saving.*

/home/indi/MnRA/Lab1 MM_ProductPipeline

Files outside the component directory will not be copied

A copy of the component will be created with all the same configuration settings as the original component, including the clock, part, and optimization directives.

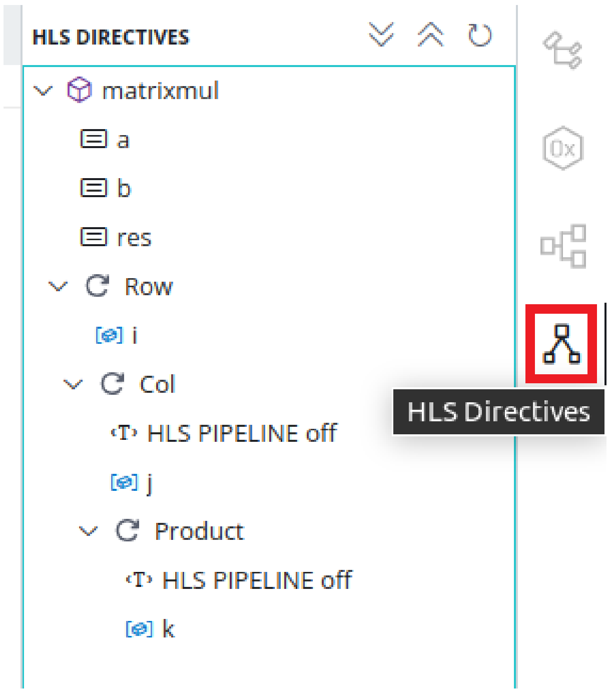


3.2 Applying Optimization Directives

Apply **pipeline** directive on the **Product** loop as below:

a. Ensure the C source code `matixmul.cpp` is visible in the Information pane.

b. On the rightmost auxiliary pane, click on **HLS Directives**.

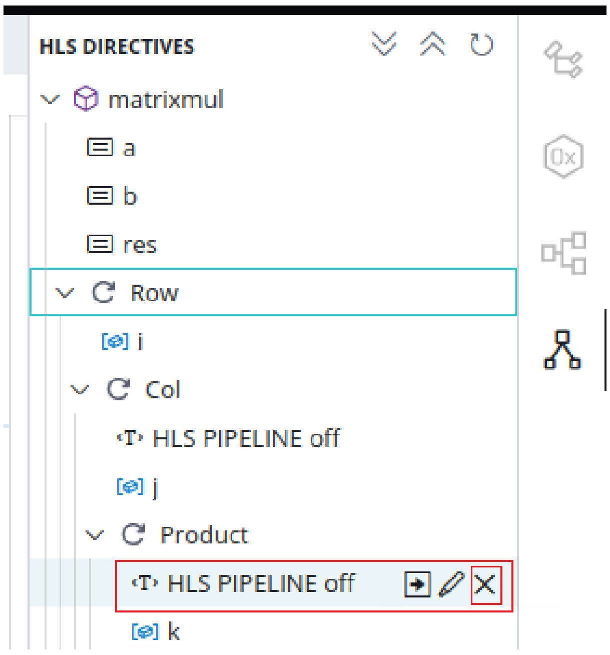


Please remember that during 1st lab we have turned OFF auto pipelining of `Col` and `Product` loop by adding following lines in `hls_config.cfg` file.

- `syn.directive.pipeline=matrixmul/Col off`
- `syn.directive.pipeline=matrixmul/Product off`

You can observe this from the from *`HLS PIPELINE OFF`* text under the `Col` and `Product` loop in the **HLS Directives** hierarchy.

c. In the HLS Directives tab first remove these two directives. You can hover over the text `HLS PIPELINE OFF` and click on the cross (X) symbol.



d. Confirm the removal of these lines by opening the `hls_config.cfg` file under **settings** and switching to **Source Editor** view.

e. Now, Select **Product** loop.

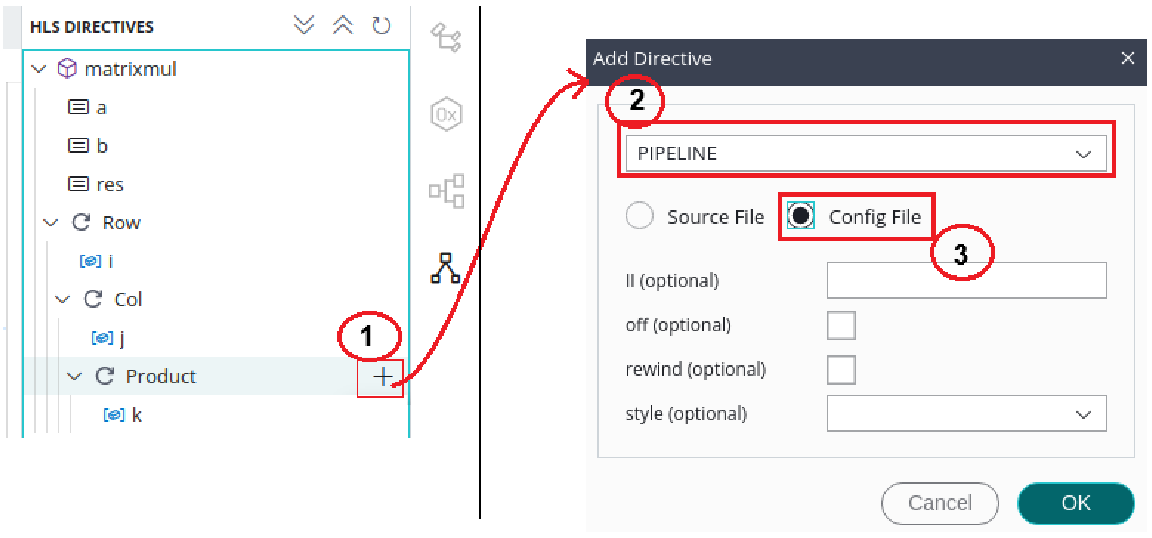
f. Click on plus (+) symbol (1).

g. Chose the **PIPELINE** from drop down menu (2).

h. Select **Config File** (3).
Optimization directives can be inserted either in the `source file` or in the `config file`. If the optimization is added in the source file, it will affect all implementations since the same source code is shared by all implementations. However, if the optimization is added in the config file, it will affect only the current implementation, as each implementation has its own config file (e.g., `hls_config.cfg`).

i. Keep all other options unchecked and click Ok.

j. Notice `HLS PIPELINE` under `Product` Loop on **HLS Directives** Tab



PIPELINE

Source File [✓ Config File]

II (optional)

off (optional)

rewind (optional)

style (optional)

This will apply **pipelining** to the **Product** loop with a default initiation interval **(II) of 1**, meaning one new loop iteration per clock cycle.

Since the innermost loop is pipelined, the tool will automatically attempt to flatten the outer loops. To observe the effect on the pipelining of the Product loop specifically, we will keep the other loops un-flattened.

k. Select loop **Row**.

l. Click on plus (+) symbol.

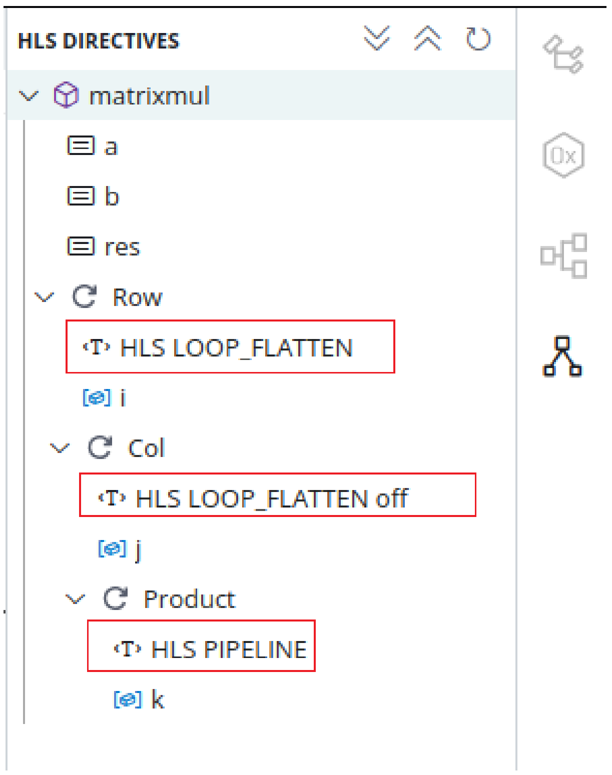
m. Chose the **LOOP_FLATTEN** from drop down menu.

n. Select **Config File**.

o. Keep all other options as default and click Ok.

p. Do the same for the loop **Col**.

The **HLS Directive** pane should show the following optimization directives:



```
HLS DIRECTIVES
matrixmul
    a
    b
    res
    Row
        HLS LOOP_FLATTEN
        i
        Col
            HLS LOOP_FLATTEN off
            j
            Product
                HLS PIPELINE
                k
```

Loop flatten is disabled for now to help read the report. Loop flatten merges nested loops together to form one bigger loop.

3.3 Synthesize the design and inspect the schedule viewer

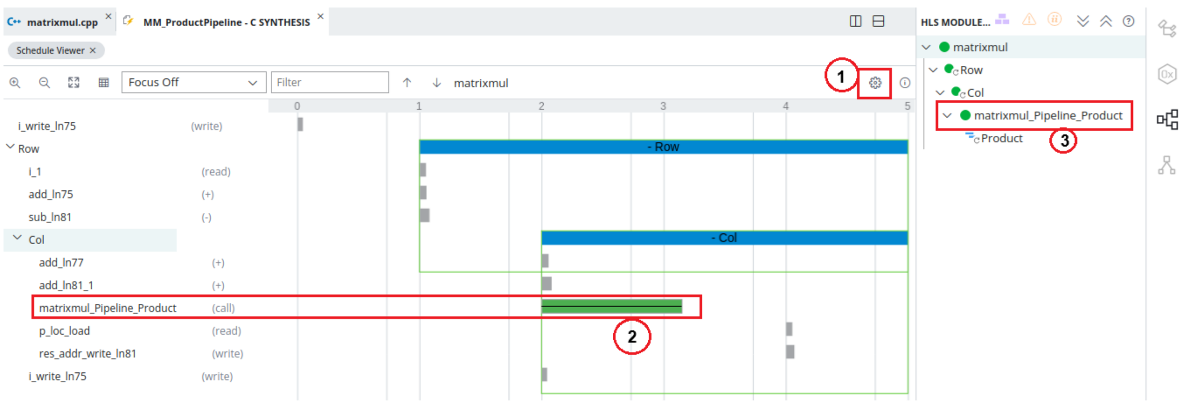
a. Click the Run under C Synthesis in the Flow Navigator to synthesize the design to RTL.

b. After synthesis, open the schedule viewer.

c. On the right side, click on the gear icon (1), to open the filters.

d. Deselect everything except: '+', '-', 'read', 'write', and call.

e. Expend both loops, notice the operation **matrixmul_Pipeline_Product**, which is represented as a big green rectangle in the schedule viewer (2). The same operation can be found on the left under the Module Hierarchy tab (3).

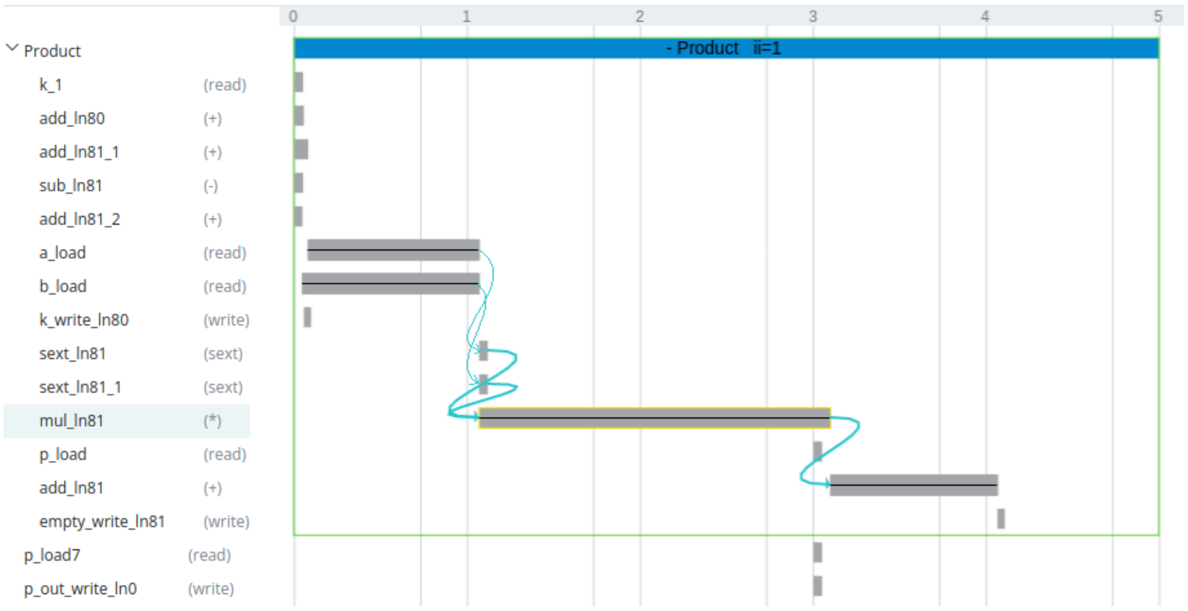


```
i_write_ln75                (write)
Row
  i_1                       (read)
  add_ln75                  (+)
  sub_ln81                  (-)
  Col
    add_ln77                (+)
    add_ln81_1              (+)
    matrixmul_Pipeline_Product (call)
    p_loc_load              (read)
    res_addr_write_ln81     (write)
  i_write_ln75              (write)
```

The compiler has wrapped the operation in function, and as a result, we have two schedule viewers to investigate.

f. Double click on **matrixmul_Pipeline_Product** to open the second schedule viewer.

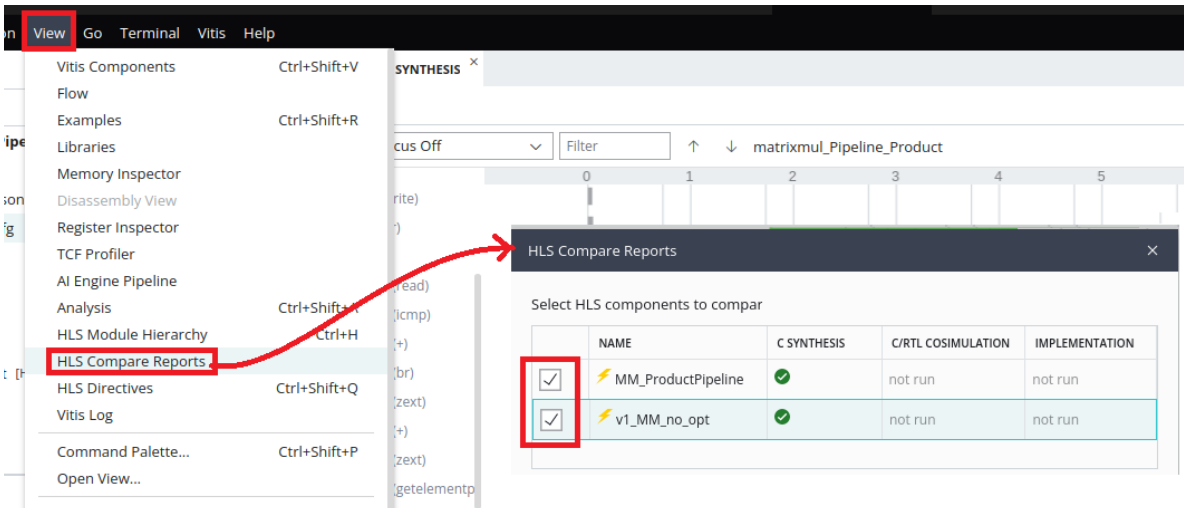
g. Apply the same filters as before, but also include the 'sext' and '*' operation. The schedule viewer looks like this:



Although this schedule viewer looks very similar to the one without optimizations, there is a major difference. The product loop is pipelined, which means that the states in the loop are now executed in parallel.

3.4 Compare the reports

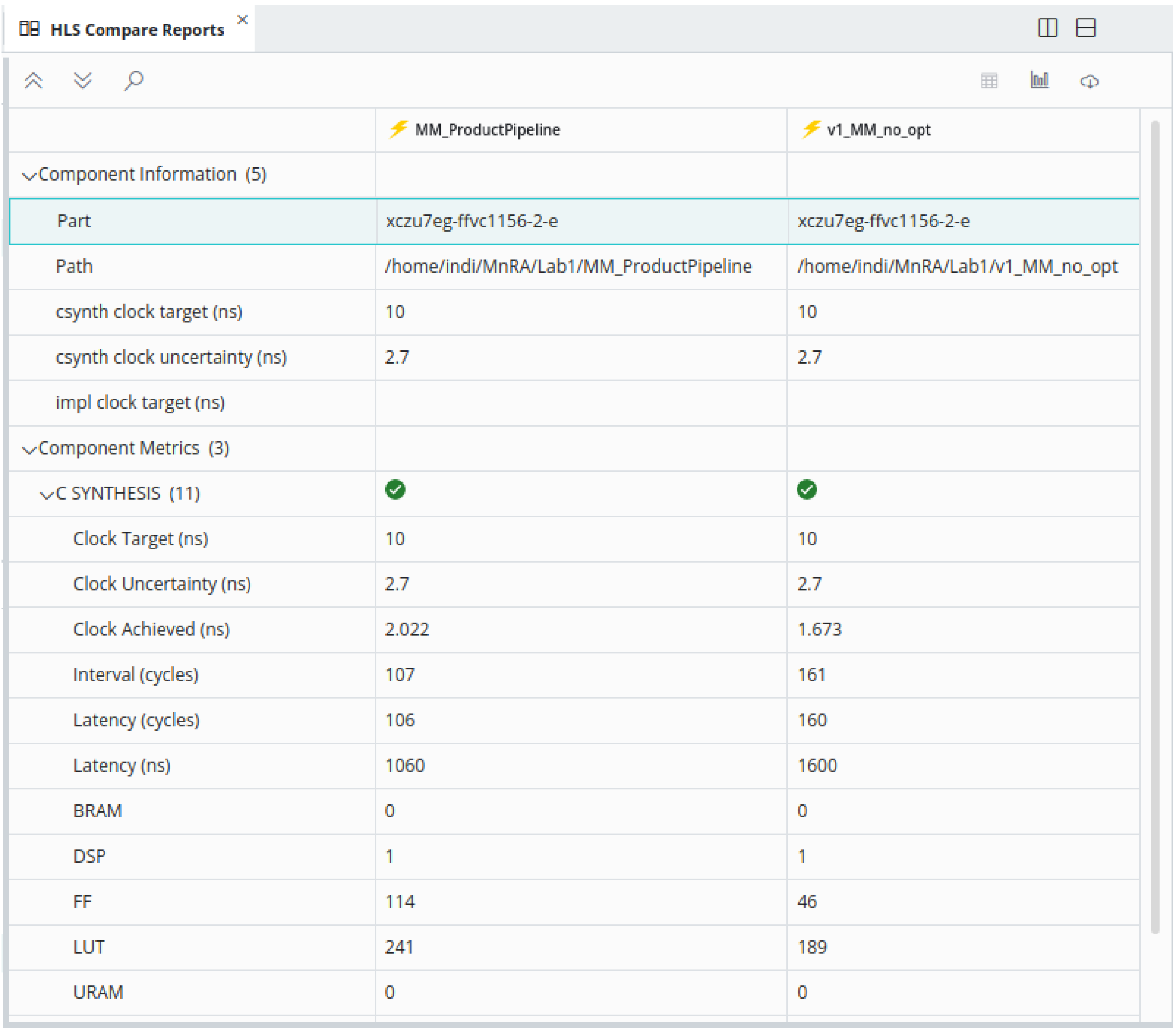
As the goal of Vitis HLS is to optimize the code, one has to compare multiple reports to find the solution that fits the design requirements. This is facilitated by a build in tool.
Under the View menu select HLS Compare Reports as shown in following figure to compare both implementations.



Select HLS components to compar

	NAME	C SYNTHESIS	C/RTL COSIMULATION	IMPLEMENTATION
✓	MM_ProductPipeline	✓	not run	not run
✓	v1_MM_no_opt	✓	not run	not run

This will open the comparison reports where you can observe the performance and resource consumption of both implementations as shown below:



	MM_ProductPipeline	v1_MM_no_opt
Component Information (5)		
Part	xczu7eg-ffvc1156-2-e	xczu7eg-ffvc1156-2-e
Path	/home/indi/MnRA/Lab1/MM_ProductPipeline	/home/indi/MnRA/Lab1/v1_MM_no_opt
csynth clock target (ns)	10	10
csynth clock uncertainty (ns)	2.7	2.7
impl clock target (ns)		
Component Metrics (3)		
C SYNTHESIS (11)	✓	✓
Clock Target (ns)	10	10
Clock Uncertainty (ns)	2.7	2.7
Clock Achieved (ns)	2.022	1.673
Interval (cycles)	107	161
Latency (cycles)	106	160
Latency (ns)	1060	1600
BRAM	0	0
DSP	1	1
FF	114	46
LUT	241	189
URAM	0	0

By comparing both reports, it is clear that the pipelining helped to improve the latency and interval. However, this came at a cost: more Flip Flops (FF) and Lookup Tables (LUT) are used.

Step 4: Pipeline the Col Loop

Although the previous solution already had a lower latency than before, it still is quite high. Pipelining the product loop only improved the latency a bit, but this loop is still executed nine times. So, lets pipeline the Col loop.

4.1 Create New Implementation of Component

- Create a new implementation by Cloning the component `v1_MM_no_opt`.
- Right click on the component name `v1_MM_no_opt` in Vitis Component pane.
- Select **Clone Component**.
- Type the component name in `MM_ColPipeline` in the dialogue that appeared.

A copy of the component will be created with all the same configuration settings as the original component, including the clock, part, and optimization directives.

In the HLS Directives tab first remove the following two directives.

- `syn.directive.pipeline=matrixmul/Col off`
- `syn.directive.pipeline=matrixmul/Product off`

You can hover over the text `HLS PIPELINE OFF` and click on the cross (X) symbol.

4.2 Applying Optimization Directives

Apply **pipeline** directive on the **Col** loop as below:

a. Ensure the C source code `matixmul.cpp` is visible in the Information pane.

b. On the rightmost auxiliary pane, click on **HLS Directives**.

c. Now, Select **Col** loop.

d. Click on plus (+) symbol.

e. Chose the **PIPELINE** from drop down menu.

f. Select **Config File**.

g. Keep all other options unchecked and click Ok.

Apply **INTERFACE** directive on the input **a** and **b** loop as below:

h. Now, Select variable **a**.

i. Click on plus (+) symbol.

j. Chose the **INTERFACE** from drop down menu.

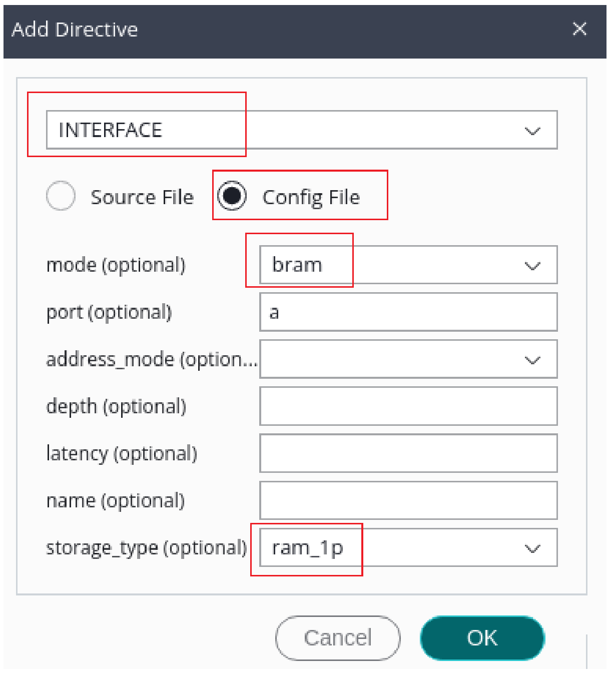
k. Select **Config File**.

l. Insert the following options

- mode (optional): BRAM
- Storage type (optional): ram_1p

m. Keep all other options at default and click Ok.

n. Repeat the same for variable **b**.


Add Directive

INTERFACE

Source File [✓ Config File]

mode (optional)	bram
port (optional)	a
address_mode (option...	
depth (optional)	
latency (optional)	
name (optional)	
storage_type (optional)	ram_1p

The BRAM directive tells Vitis HLS what type of BRAM it needs to use for the variables. BRAM_1P means that it is BRAM that can only be accessed for one variable at a time, similarly BRAM_2P can read 2 variables at the same time.

4.3 Synthesize the design and inspect the schedule viewer

Synthesize the design to RTL. During synthesis, the information reported in the Output tab shows that the **Product** loop is **completely unrolled**. This is due the pipelining of its outer Col loop. Pipelining a loop results in complete unrolling of inner loops in the loop nest. Moreover loop flattening was performed on loop **Row** and that the default initiation internal target of 1 could not be achieved on loop **Col** due to a dependency. Some script from output tab is pasted below:

INFO: [HLS 214-291] Loop 'Product' is marked as complete unroll implied by the pipeline pragma (../src/matrixmul.cpp:80:16)

INFO: [XFORM 203-541] Flattening a loop nest 'Row' (../src/matrixmul.cpp:75:8) in function 'matrixmul'.

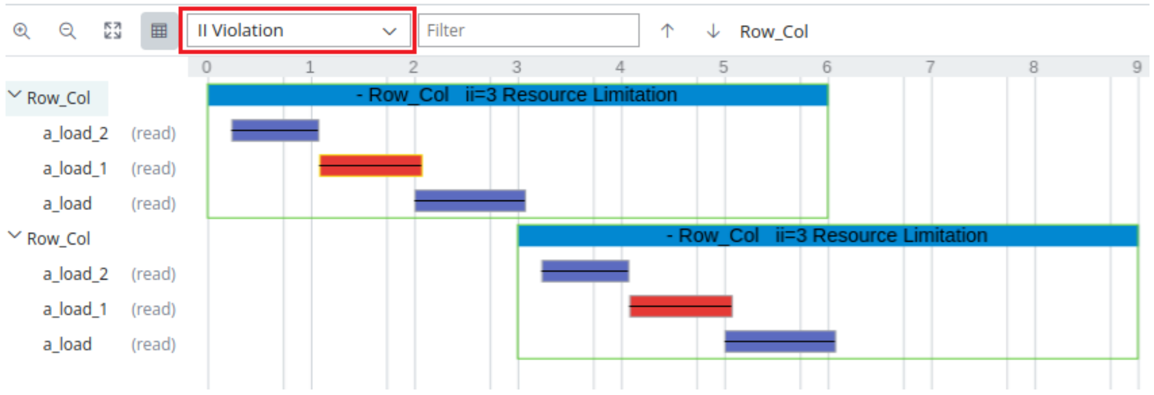
INFO: [SCHED 204-61] Pipelining loop 'Row_Col'.

WARNING: [HLS 200-885] The II Violation in module 'matrixmul' (loop 'Row_Col'): Unable to schedule 'load' operation 8 bit ('a_load_2', ../src/matrixmul.cpp:75) on array 'a' due to limited memory ports (II = 1). Please consider using a memory core with more ports or partitioning the array 'a'.

WARNING: [HLS 200-885] The II Violation in module 'matrixmul' (loop 'Row_Col'): Unable to schedule 'load' operation 8 bit ('a_load_1', ../src/matrixmul.cpp:75) on array 'a' due to limited memory ports (II = 2). Please consider using a memory core with more ports or partitioning the array 'a'.
INFO: [HLS 200-1470] Pipelining result : Target II = NA, Final II = 3, Depth = 6, loop 'Row_Col'

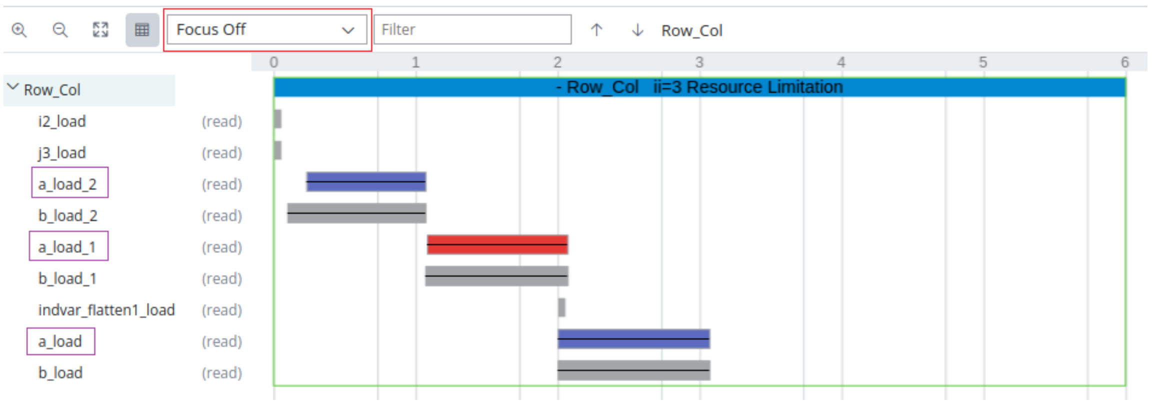
Reviewing the synthesis report shows, as noted above, that the interval for loop Row Col is only three: the target is to process one sample every cycle. Once again, you can use the Schedule viewer perspective to highlight why the initiation target was not achieved.

Open the schedule viewer.
On the top, select **Focus Off** and change it to **II Violation**. This filters everything away and creates a view to help identify the root problem why the requested II was not achieved.



From the view, it is clear that reading the three values from variable b out of the BRAM takes three states. This in turn means that it also takes three states before the fourth value can be read from BRAM. Hence, a second loop can't be started until the first three values are read.

Change the view back to **Focus Off**, and use the filter (from gear symbol) to only show the **read** operations.



Looking at this view, it is clear that even when the issue with port **b** is resolved, the same issue occurs with port **a**: it also has to perform 3 reads.

```
Note: High-Level Synthesis can only report one schedule error or warning at a time, because,
as soon as the first issue occurs, the actions to create an achievable schedule invalidates
any other infeasible schedules.
```

Step 5: Partition the Arrays

High-Level Synthesis allows arrays to be partitioned, mapped together and re-shaped. These techniques allow the access to array to be modified without changing the source code. Partitioning these arrays creates multiple number of arrays and hence multiple number of ports. An array partitioned by a factor n will have n number of ports, hence n number of elements can be accessed at a time.

In our matrixmul algorithm, the loop index for the **Product** loop is **k**, both arrays should be partitioned along their respective **k** dimension: the design needs to access more than two values of k in each clock cycle. For array **a**, this is dimension **2** because its access patterns is `a[i][k]`; for array **b**, this is dimension **1** because its access pattern is `b[k][j]`. Partitioning these arrays creates k arrays - in this case, k number ports.

Alternatively, we can use re-shape instead of partition allowing one wide array (port) to be created instead of k ports. So, Partitioning creates multiple, smaller sections of BRAM, while reshaping changes the layout of the array by placing multiple variables in one memory address.

5.1 Create New Implementation of Component and Apply optimization Directives

Create a new implementation from `MM_ColPipeline` and name it as **MM_ArrayPartition**.

In the HLS Directive tab:

a. Select variable **a**.

b. Click on plus (+) symbol.

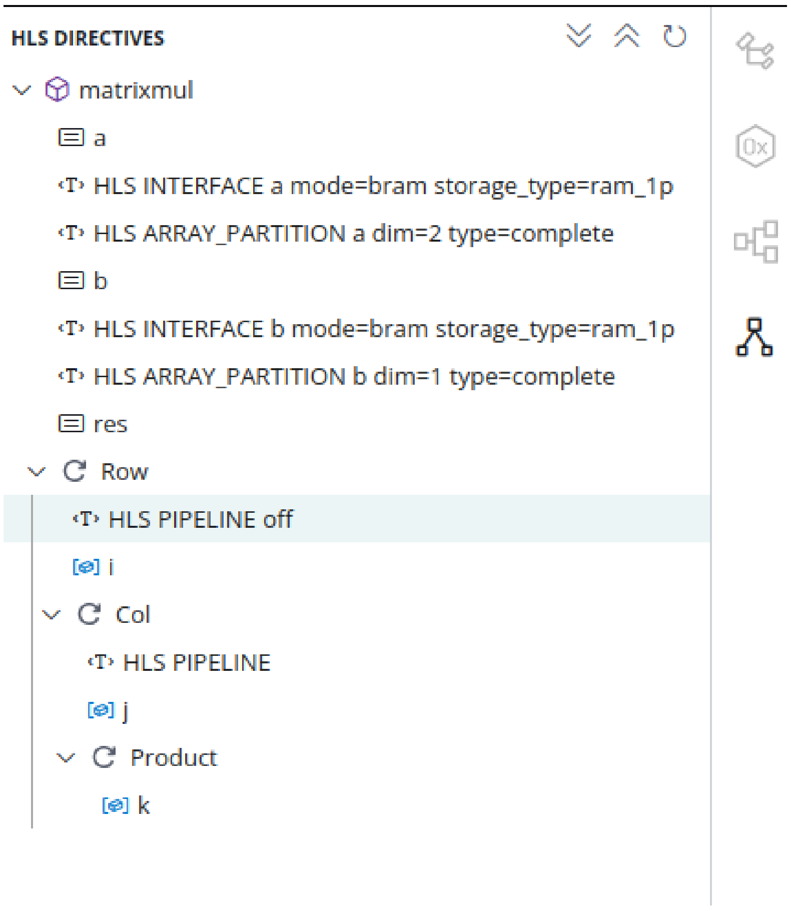
c. In the Directives Editor dialog box activate the Directive drop-down menu at the top and select ARRAY_PARTITION.

d. Set the dimension to 2 and click OK.

e. Repeat this process for variable b, but set the dimension to 1.

f. Select Row loop and turn the PIPELINE directive OFF.

The Directive pane should show the following optimization directives.

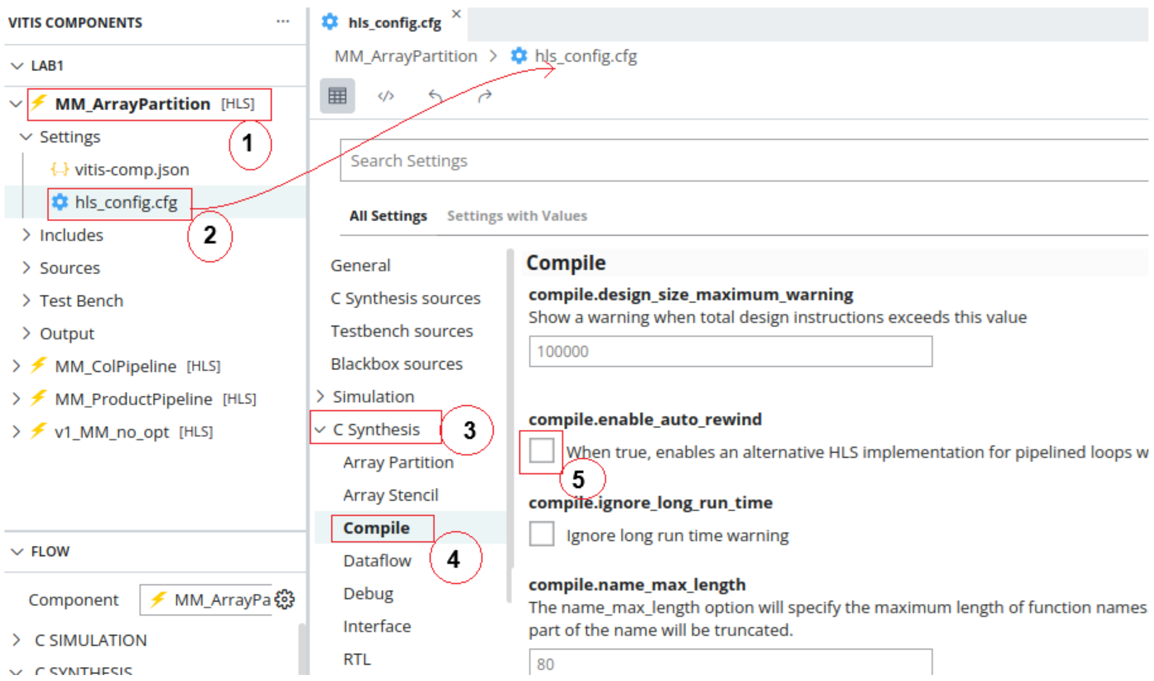

HLS DIRECTIVES

```
matrixmul
  a
  HLS INTERFACE a mode=bram storage_type=ram_1p
  HLS ARRAY_PARTITION a dim=2 type=complete
  b
  HLS INTERFACE b mode=bram storage_type=ram_1p
  HLS ARRAY_PARTITION b dim=1 type=complete
  res
  Row
    HLS PIPELINE off
    i
    Col
      HLS PIPELINE
      j
      Product
        k
```

The tool will auto-rewind the matrixmul function to continually feed inner pipelined loop. To see the affect without auto-rewind option perform following steps.

g. Open the configurations file `hls_config.cfg` under Settings in Vitis component view.

h. Now under `C Synthesis > Compile` **un-check** the box `compile.enable_auto_rewind`.



VITIS COMPONENTS

LAB1
- MM_ArrayPartition [HLS]
  - Settings
    - vitis-comp.json
    - hls_config.cfg
  - Includes
  - Sources
  - Test Bench
  - Output
- MM_ColPipeline [HLS]
- MM_ProductPipeline [HLS]
- v1_MM_no_opt [HLS]

FLOW

Component: MM_ArrayPa

C SIMULATION

C SYNTHESIS

hls_config.cfg

MM_ArrayPartition > hls_config.cfg

Search Settings

All Settings | Settings with Values

General, C Synthesis sources, Testbench sources, Blackbox sources, Simulation, C Synthesis (Array Partition, Array Stencil, Compile, Dataflow, Debug, Interface, RTL)

Compile

compile.design_size_maximum_warning
Show a warning when total design instructions exceeds this value
100000

compile.enable_auto_rewind
☐ When true, enables an alternative HLS implementation for pipelined loops w

compile.ignore_long_run_time
☐ Ignore long run time warning

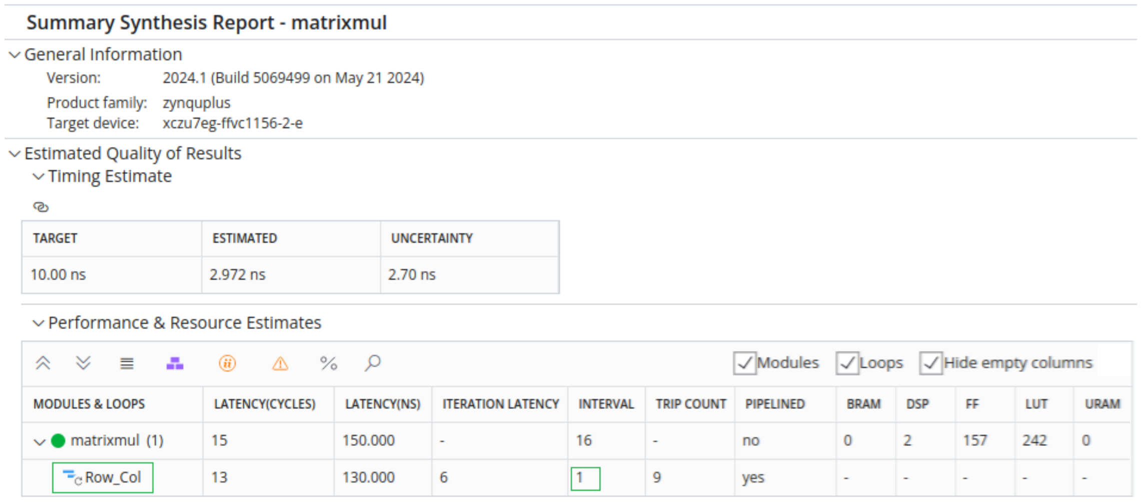
compile.name_max_length
The name_max_length option will specify the maximum length of function names
part of the name will be truncated.
80

i. Click the Run under C Synthesis in the Flow Navigator to synthesize the design to RTL.

j. Open the Synthesis report.

5.2 Analyze the Latency and Interval

The report shows the top-level loop Row_Col is now processing data at 1 sample per clock period.



Summary Synthesis Report - matrixmul

General Information

Version:	2024.1 (Build 5069499 on May 21 2024)
Product family:	zynquplus
Target device:	xczu7eg-ffvc1156-2-e

Estimated Quality of Results

Timing Estimate

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	2.972 ns	2.70 ns

Performance & Resource Estimates

✓ Modules ✓ Loops ✓ Hide empty columns

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM	DSP	FF	LUT	URAM
● matrixmul (1)	15	150.000	-	16	-	no	0	2	157	242	0
Row_Col	13	130.000	6	1	9	yes	-	-	-	-	-

- The top-level Row_Col module takes 13 clock cycles to complete.
- It reads 1 sample per cycle (Initiation Interval II=1).
- The Row_Col loop outputs the first sample after 6 cycles (iteration latency = 6), and subsequently, it outputs 1 sample per cycle.
- After 9 cycles (iterations or Trip count), it finishes reading all the samples.
- The last result will be output after 15 cycles (6+9).
- The function completes the processing first set of inputs after 15 clock and returns to start processing the next set of data, making the interval 16 cycles.

5.3 Loop Pipeline with rewind option

However, note that our pipelined design completes the read operation by the 9th clock cycle, but we must wait until the 15th clock cycle before reading the next set of inputs. The pipelined execution does not accept a new set of inputs until it has processed the previous set of data, creating a bubble at the top-level loop.

To eliminate this bubble, we can enable the rewind option at the top-level loop. This allows the next set of data to be read immediately after the 9th clock cycle, reducing the initiation interval to 9 cycles.

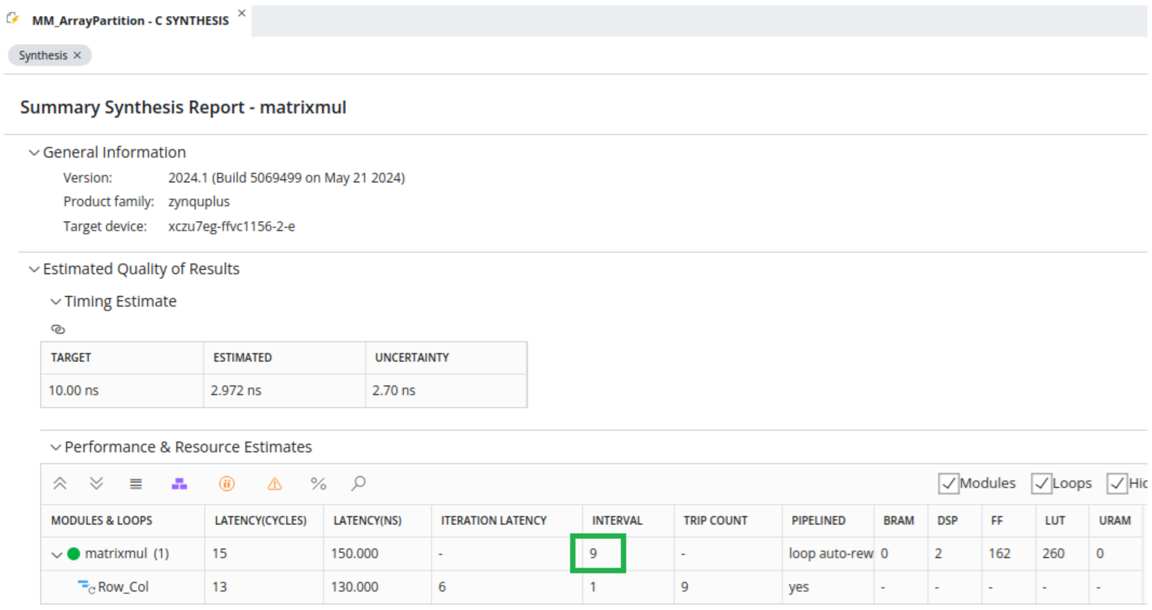
To see the affect rewind on interval perform following steps.

a. Open the configurations file `hls_config.cfg` under Settings in Vitis component view.

b. Now under `C Synthesis > Compile` **check** the box `compile.enable_auto_rewind`.

c. Click the Run under C Synthesis in the Flow Navigator to synthesize the design to RTL.

d. Open the Synthesis report.



MM_ArrayPartition - C SYNTHESIS ×

Synthesis ×

Summary Synthesis Report - matrixmul

General Information

Version:	2024.1 (Build 5069499 on May 21 2024)
Product family:	zynquplus
Target device:	xczu7eg-ffvc1156-2-e

Estimated Quality of Results

Timing Estimate

TARGET	ESTIMATED	UNCERTAINTY
10.00 ns	2.972 ns	2.70 ns

Performance & Resource Estimates

✓Modules ✓Loops ✓Hic

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM	DSP	FF	LUT	URAM
● matrixmul (1)	15	150.000	-	9	-	loop auto-rew	0	2	162	260	0
Row_Col	13	130.000	6	1	9	yes	-	-	-	-	-

Please Note that Interval of matrixmul module has reduced from 16 to 9 cycles.

Step 6: Function Pipelining Instead of Loop Pipelining

It is worth pipelining the function instead of the loops to observe the difference in the two approaches.

6.1 Create New Implementation of Component and Apply optimization Directives

Create a new implementation from `MM_ArrayPartition` and name it as **MM_FunctionPipeline**.

In the HLS Directive tab:

a. Under **col** loop, hover over the PIPELINE directive and remove it by clicking on cross (x) symbol.

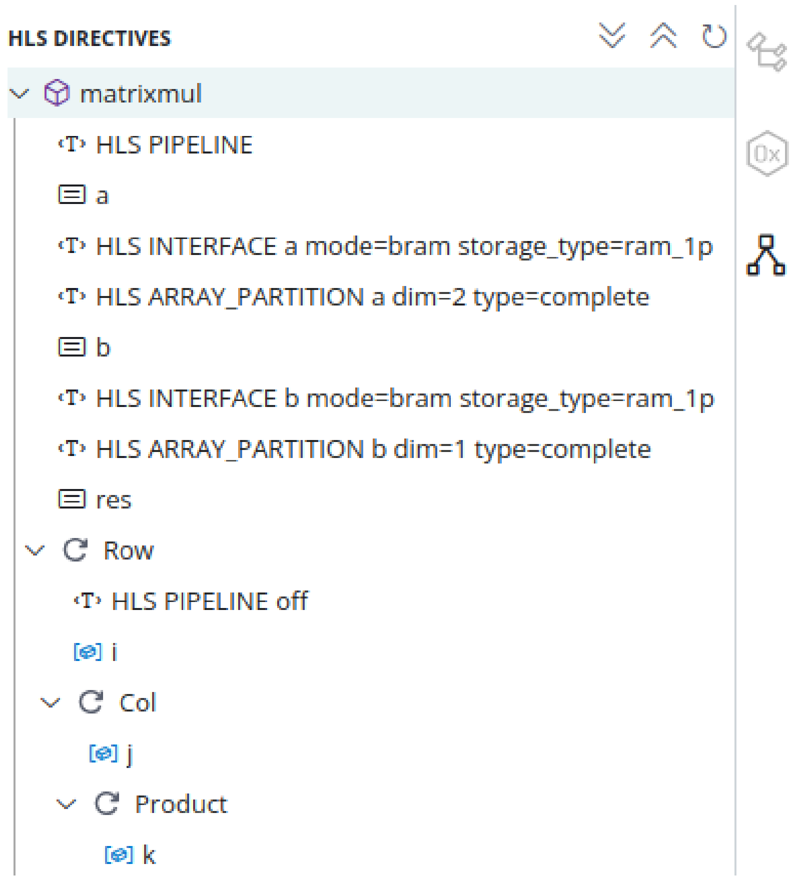
b. To add PIPELINE directive on function, select **matrixmul**.

c. Click on plus (+) symbol.

d. Chose the **PIPELINE** from drop down menu.

e. Select **Config File** and click OK.

The Directive pane should show the following optimization directives.

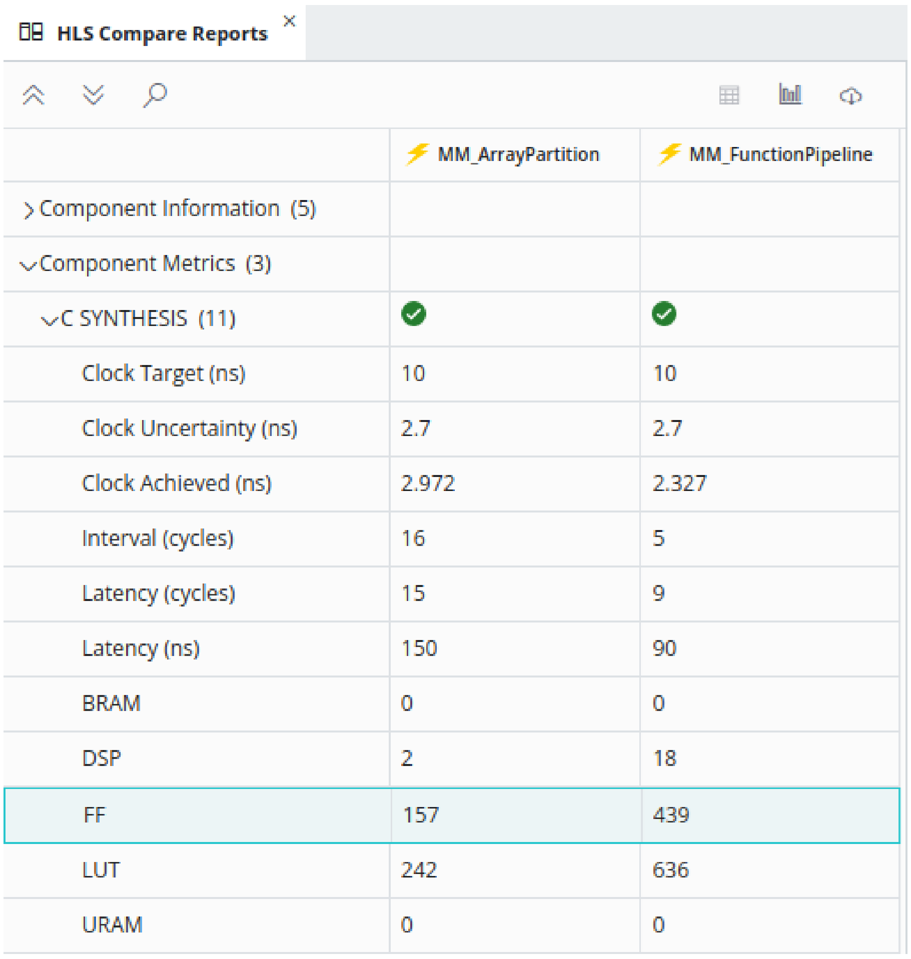


HLS DIRECTIVES

```
matrixmul
    HLS PIPELINE
    a
    HLS INTERFACE a mode=bram storage_type=ram_1p
    HLS ARRAY_PARTITION a dim=2 type=complete
    b
    HLS INTERFACE b mode=bram storage_type=ram_1p
    HLS ARRAY_PARTITION b dim=1 type=complete
    res
  Row
      HLS PIPELINE off
      i
    Col
        j
      Product
          k
```

Question: Check the log from output tab and report which loops are completely unrolled?

The comparison of solutions `MM_ArrayReshape` and `MM_FunctionPipeline` is shown below:


HLS Compare Reports

	MM_ArrayPartition	MM_FunctionPipeline
> Component Information (5)		
⌄ Component Metrics (3)		
⌄ C SYNTHESIS (11)	✓	✓
Clock Target (ns)	10	10
Clock Uncertainty (ns)	2.7	2.7
Clock Achieved (ns)	2.972	2.327
Interval (cycles)	16	5
Latency (cycles)	15	9
Latency (ns)	150	90
BRAM	0	0
DSP	2	18
FF	157	439
LUT	242	636
URAM	0	0

The design now completes in fewer clocks and can start a new transaction every 5 clock cycles. However, the area and resources have increased substantially because all the loops in the design were unrolled. Pipelining loops allows the loops to remain rolled, thus providing a good means of controlling the area. When pipelining a function, all loops contained in the function are unrolled, which is a requirement for pipelining.
The pipelined function design can process a new set of 9 samples every 5 clock cycles. This exceeds the requirement of 1 sample per clock because the default behavior of High-Level Synthesis is to produce a design with the highest performance. The pipelined function results in the best performance. However, if it exceeds the required performance, it might take multiple additional directives to slow the design down.
Pipelining loops gives you an easy way to control resources, with the option of partially unrolling the design to meet performance.

Step 7: Apply streaming (FIFO) Interfaces

7.1 Create New Implementation of Component and Apply optimization Directives

Create a new implementation from `MM_ArrayPartition` and name it as **MM_FifoInterface**.

In the HLS Directive tab:

a. Under variable **a** hover over the INTERFACE directive (`HLS INTERFACE a mode=bram storage_type=ram_1p`)

b. Click on edit directive (pencil symbol) to open Edit Directive dialog. You can also open it by double click.

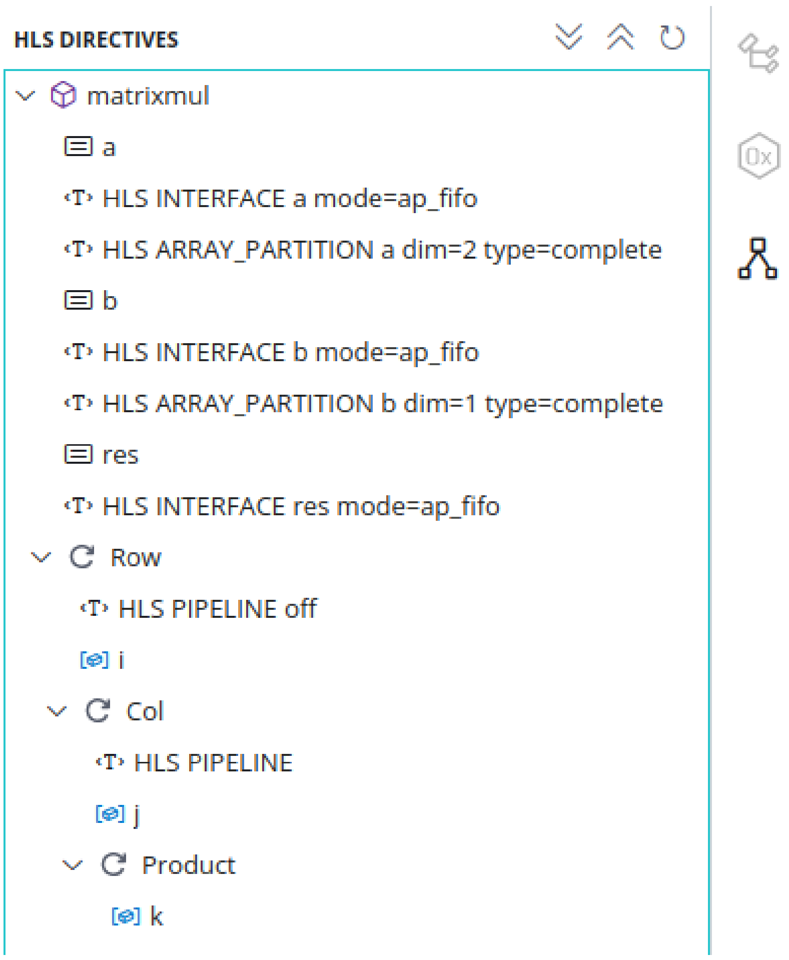
c. Click the `Storage_type (optional)` drop-down menu to select the `empty` entry.

d. Click the **mode** drop-down menu to select **ap_fifo** and click OK.

e. Repeat this process for variable **b**.

f. Also add same INTERFACE directive (ap_fifo) for variable **res**.

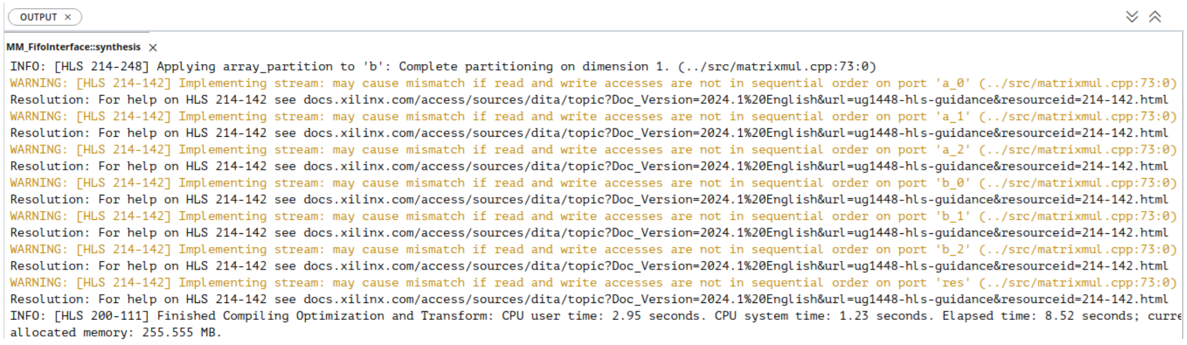
The Directive pane should show the following optimization directives.



```
HLS DIRECTIVES
matrixmul
    a
    HLS INTERFACE a mode=ap_fifo
    HLS ARRAY_PARTITION a dim=2 type=complete
    b
    HLS INTERFACE b mode=ap_fifo
    HLS ARRAY_PARTITION b dim=1 type=complete
    res
    HLS INTERFACE res mode=ap_fifo
  Row
    HLS PIPELINE off
    i
    Col
      HLS PIPELINE
      j
      Product
        k
```

7.2 Synthesize the design and analyze warnings in Output tab

Click the Run under C Synthesis in the Flow Navigator to synthesize the design to RTL. Observe the Warnings in output tab.



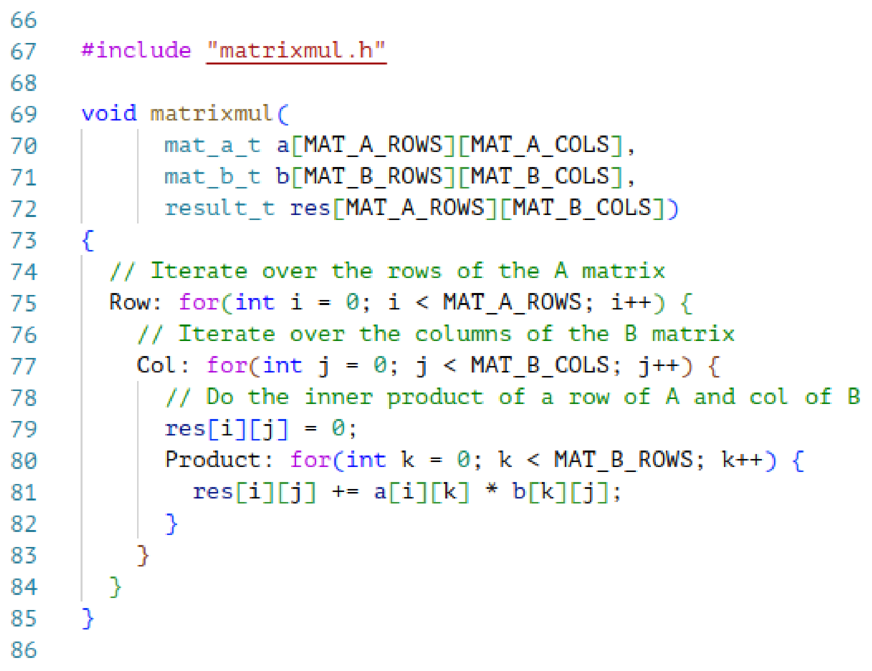
```
OUTPUT ×
MM_FifoInterface::synthesis ×
INFO: [HLS 214-248] Applying array_partition to 'b': Complete partitioning on dimension 1. (../src/matrixmul.cpp:73:0)
WARNING: [HLS 214-142] Implementing stream: may cause mismatch if read and write accesses are not in sequential order on port 'a_0' (../src/matrixmul.cpp:73:0)
Resolution: For help on HLS 214-142 see docs.xilinx.com/access/sources/dita/topic?Doc_Version=2024.1%20English&url=ug1448-hls-guidance&resourceid=214-142.html
WARNING: [HLS 214-142] Implementing stream: may cause mismatch if read and write accesses are not in sequential order on port 'a_1' (../src/matrixmul.cpp:73:0)
Resolution: For help on HLS 214-142 see docs.xilinx.com/access/sources/dita/topic?Doc_Version=2024.1%20English&url=ug1448-hls-guidance&resourceid=214-142.html
WARNING: [HLS 214-142] Implementing stream: may cause mismatch if read and write accesses are not in sequential order on port 'a_2' (../src/matrixmul.cpp:73:0)
Resolution: For help on HLS 214-142 see docs.xilinx.com/access/sources/dita/topic?Doc_Version=2024.1%20English&url=ug1448-hls-guidance&resourceid=214-142.html
WARNING: [HLS 214-142] Implementing stream: may cause mismatch if read and write accesses are not in sequential order on port 'b_0' (../src/matrixmul.cpp:73:0)
Resolution: For help on HLS 214-142 see docs.xilinx.com/access/sources/dita/topic?Doc_Version=2024.1%20English&url=ug1448-hls-guidance&resourceid=214-142.html
WARNING: [HLS 214-142] Implementing stream: may cause mismatch if read and write accesses are not in sequential order on port 'b_1' (../src/matrixmul.cpp:73:0)
Resolution: For help on HLS 214-142 see docs.xilinx.com/access/sources/dita/topic?Doc_Version=2024.1%20English&url=ug1448-hls-guidance&resourceid=214-142.html
WARNING: [HLS 214-142] Implementing stream: may cause mismatch if read and write accesses are not in sequential order on port 'b_2' (../src/matrixmul.cpp:73:0)
Resolution: For help on HLS 214-142 see docs.xilinx.com/access/sources/dita/topic?Doc_Version=2024.1%20English&url=ug1448-hls-guidance&resourceid=214-142.html
WARNING: [HLS 214-142] Implementing stream: may cause mismatch if read and write accesses are not in sequential order on port 'res' (../src/matrixmul.cpp:73:0)
Resolution: For help on HLS 214-142 see docs.xilinx.com/access/sources/dita/topic?Doc_Version=2024.1%20English&url=ug1448-hls-guidance&resourceid=214-142.html
INFO: [HLS 200-111] Finished Compiling Optimization and Transform: CPU user time: 2.95 seconds. CPU system time: 1.23 seconds. Elapsed time: 8.52 seconds; curre
allocated memory: 255.555 MB.
```

Whenever a FIFO interface is applied the tool will warn you to ensure the sequential access. If you have ensured the sequential you can ignore the warnings otherwise re-check you code for access patterns.

Let us analyze our code for access patterns. From the code shown in Figure below, array **res** performs writes in the following sequence (MAT_B_COLS = MAT_B_ROWS = 3):

1. Write to [0][0] on line-79.
2. Then a write to [0][0] on line-81.
3. Then a write to [0][0] on line-81.
4. Then a write to [0][0] on line-81.
5. Write to [0][1] on line-79 (after index j increments).
6. Then a write to [0][1] on line 81.
7. Etc.

Four consecutive writes to address [0][0] does not constitute a streaming access pattern; this is random access. Streaming access differs from random access. Streaming involves sequential access to memory locations/addresses and doesn't permit accessing the same memory location twice before completing the sequence.



```
66
67  #include "matrixmul.h"
68
69  void matrixmul(
70        mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
71        mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
72        result_t res[MAT_A_ROWS][MAT_B_COLS])
73  {
74    // Iterate over the rows of the A matrix
75    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
76      // Iterate over the columns of the B matrix
77      Col: for(int j = 0; j < MAT_B_COLS; j++) {
78        // Do the inner product of a row of A and col of B
79        res[i][j] = 0;
80        Product: for(int k = 0; k < MAT_B_ROWS; k++) {
81          res[i][j] += a[i][k] * b[k][j];
82        }
83      }
84    }
85  }
86
```

Examining the code in above figure reveals that there are similar issues reading arrays **a** and **b**. It is impossible to use a FIFO interface for data access with the code as written. To use a FIFO interface, the optimization directives available in High-Level Synthesis are inadequate because the code currently enforces a certain order of reads and writes.

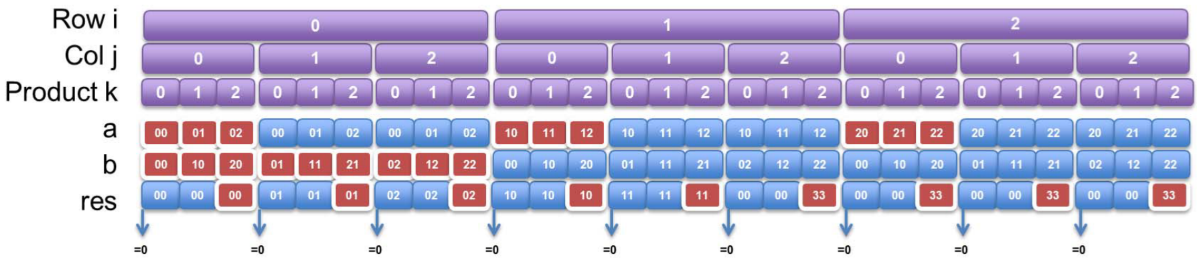
The nature of the C code, which specified multiple accesses to the same addresses, prevents streaming interfaces being applied.

- In a streaming interface, the values must be accessed in sequential order.
- In the code, the accesses were also port accesses, which High-Level Synthesis is unable to move around and optimize. The C code specified writing the value zero to port res at the start of every product loop.

This may be part of the intended behavior. HLS cannot simply decide to change the specification of the algorithm.

The code intuitively captured the behavior of a matrix multiplication, but it prevented a required behavior in the hardware: streaming accesses.

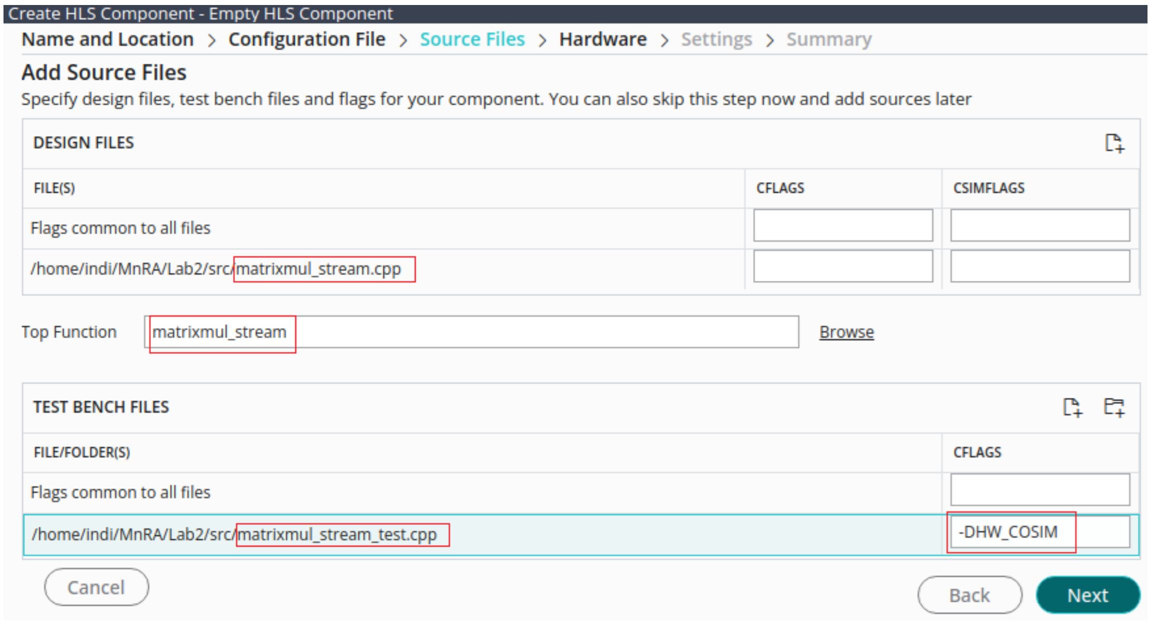
Further optimization requires a re-write of the code.
Next exercise uses an updated version of the C code you worked. The following figure explains how the C code was updated. It shows the I/O access pattern for the new code. Out of necessity the address values are shown in a small font. As variables i, j and k iterate from 0 to 3, the lower part of Figure shows the addresses generated to read **a**, **b** and write to **res**. In addition, at the start of each **Product** loop, **res** is set to the value zero.


Row i: 0 1 2
Col j: 0 1 2 0 1 2 0 1 2
Product k: 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
a: 00 01 02 00 01 02 00 01 02 10 11 12 10 11 12 10 11 12 20 21 22 20 21 22 20 21 22
b: 00 10 20 01 11 21 02 12 22 00 10 20 01 11 21 02 12 22 00 10 20 01 11 21 02 12 22
res: 00 00 00 01 01 01 02 02 02 10 10 10 11 11 11 00 00 33 00 00 33 00 00 33 00 00 33
=0 =0 =0 =0 =0 =0 =0 =0 =0

To have a hardware design with sequential streaming accesses, the ports accesses can only be those shown highlighted in red. For the read ports, the data must be **cached** internally to ensure the design does not have to re-read the port. For the write port **res**, the data must be saved into a temporary variable and only written to the port in the cycles shown in red.
This requires modification of source code. Since we need to modify the source code, we can't create a new implementation as it will affect other implementations. Therefore, we'll create a new component.

7.3 Create a New Component using terminal

Close the current workspace from `File > Close Workspace`. this will re-launch the Vitis-IDE. Open new Workspace as `/home/indi/MnRA/Lab2`. Create new HLS component as below. Don't forget to add CFLAG (-DHW_COSIM) with test-bench file.


Create HLS Component - Empty HLS Component

Name and Location > Configuration File > Source Files > Hardware > Settings > Summary

Add Source Files

Specify design files, test bench files and flags for your component. You can also skip this step now and add sources later

DESIGN FILES

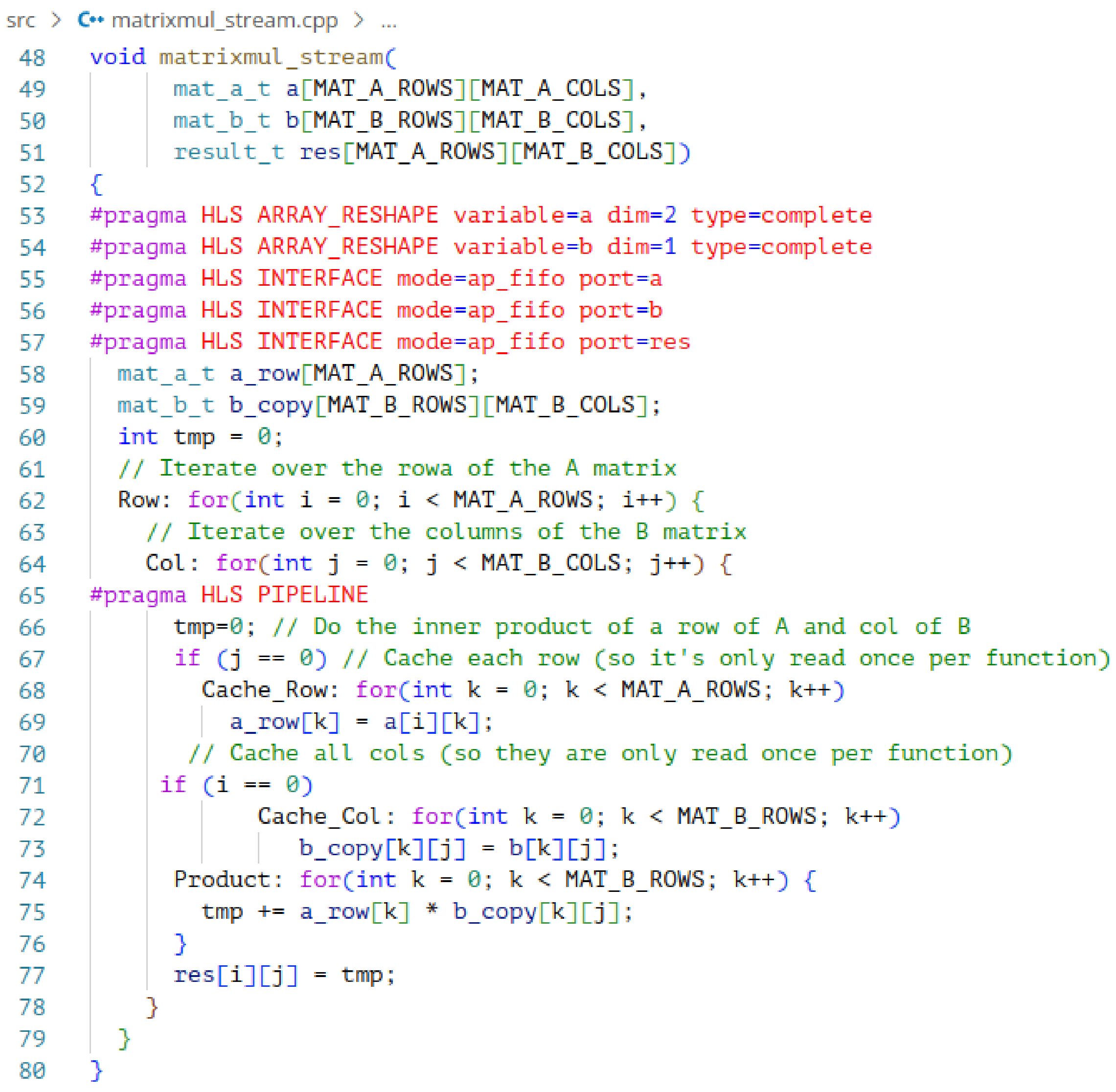
FILE(S)	CFLAGS	CSIMFLAGS
Flags common to all files		
/home/indi/MnRA/Lab2/src/matrixmul_stream.cpp		

Top Function: matrixmul_stream Browse

TEST BENCH FILES

FILE/FOLDER(S)	CFLAGS
Flags common to all files	
/home/indi/MnRA/Lab2/src/matrixmul_stream_test.cpp	-DHW_COSIM

After the component is created, Open the Source folder in vitis component view and double-click `matrixmul_stream.cpp` to open the code as shown below:


src > matrixmul_stream.cpp > ...

```
void matrixmul_stream(
      mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
      mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
      result_t res[MAT_A_ROWS][MAT_B_COLS])
{
#pragma HLS ARRAY_RESHAPE variable=a dim=2 type=complete
#pragma HLS ARRAY_RESHAPE variable=b dim=1 type=complete
#pragma HLS INTERFACE mode=ap_fifo port=a
#pragma HLS INTERFACE mode=ap_fifo port=b
#pragma HLS INTERFACE mode=ap_fifo port=res
  mat_a_t a_row[MAT_A_ROWS];
  mat_b_t b_copy[MAT_B_ROWS][MAT_B_COLS];
  int tmp = 0;
  // Iterate over the rowa of the A matrix
  Row: for(int i = 0; i < MAT_A_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Col: for(int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS PIPELINE
      tmp=0; // Do the inner product of a row of A and col of B
      if (j == 0) // Cache each row (so it's only read once per function)
        Cache_Row: for(int k = 0; k < MAT_A_ROWS; k++)
          a_row[k] = a[i][k];
       // Cache all cols (so they are only read once per function)
     if (i == 0)
          Cache_Col: for(int k = 0; k < MAT_B_ROWS; k++)
             b_copy[k][j] = b[k][j];
      Product: for(int k = 0; k < MAT_B_ROWS; k++) {
        tmp += a_row[k] * b_copy[k][j];
      }
      res[i][j] = tmp;
    }
  }
}
```

Review the code and confirm the following:

- The directives from before, including the FIFO interfaces, are specified in the code as `pragmas`.
- For-loops have been added to cache the row and column reads.
- A temporary variable is used for the accumulation and port res is only written to when the final result is computed for each value.
- Because the for-loops to cache the row and column would require multiple cycles to perform the reads, the pipeline directive has been applied to the Col for-loop, ensuring these cache for-loops are automatically unrolled.

7.4 Synthesize the design and verify the RTL using co-simulation

a. Disable auto-rewind option as described in section 5.2.

b. Click the Run under C Synthesis in the Flow Navigator to synthesize the design to RTL.

c. When synthesis completes, use the click Run under C/RTL Cosimulation in the Flow Navigator to launch the Cosimulation Dialog box.

d. Click OK to start RTL verification.

The design has been now been fully synthesized to read one sample every clock cycle using streaming FIFO interfaces.

Summary

In this tutorial, you learned:

- How to analyze pipelined loops and understand exactly which limitations prevent optimizations targets from being achieved.
- The advantages and disadvantages of function versus loop pipelining.
- How unintended dependencies in the code can prevent hardware design goals from being realized and how they can be overcome by modifications to the source code.