

ACCELERATED CHIP DESIGN CYCLE WORKSHOP: HIGH-LEVEL SYNTHESIS

This workshop is intended to provide the participants with the necessary skills to create high-level synthesis IPs using the Vitis HLS tool flow. Various techniques and directives inherent to Vitis HLS to improve the design performance are also introduced in detail

The labs have been developed and tested on Windows 10 professional and Ubuntu 20.04 machines and using Vivado HLS 2018.2 and Vitis HLS 2022.1 version tools. Linux machine to run the Vitis HLS tool is more recommended though.

Labs Overview:

Lab1: Vitis HLS Design Flow

This lab provides a basic introduction to high-level synthesis using the Vitis HLS tool flow. You will use Vitis HLS in GUI mode to create a project. You will simulate, synthesize, and implement the provided design.

Lab2: Improving Performance

This lab introduces various techniques and directives which can be used in Vitis HLS to improve design performance. The design under consideration accepts an image in a (custom) RGB format, converts it to the Y'UV color space, applies a filter to the Y'UV image and converts it back to RGB.

Lab3: Improving Area and Resource Utilization

This lab introduces various techniques and directives which can be used in Vitis HLS to improve design performance as well as area and resource utilization. The design under consideration performs discrete cosine transformation (DCT) on an 8x8 block of data.

Lab2: Improving Performance

Introduction:

This lab introduces various techniques and directives which can be used in Vitis HLS to improve design performance. The design under consideration accepts an image in a (custom) RGB format, converts it to the Y'UV color space, applies a filter to the Y'UV image and converts it back to RGB.

Objectives:

After completing this lab, you will be able to:

- Add directives in your design
- Understand the effect of INLINE directive
- Improve performance using PIPELINE directive
- Distinguish between DATAFLOW directive and Configuration Command functionality

Steps:

STEP 1: Create a Vitis HLS Project from Command Line

[This part pertains ONLY to Linux-based setup. For windows-based setup, open the provided .tcl file, try to understand the steps and perform them by using the Vitis GUI. Once done, jump directly to STEP 2.](#)

Validate your design using terminal. Create a new Vitis HLS project from the terminal

1. Change directory to {labs}/lab2.
A self-checking program (**yuv_filter_test.c**) is provided. Using that we can validate the design. A Makefile is also provided. Using the Makefile, the necessary source files can be compiled and the compiled program can be executed. You can examine the contents of these files and the project directory.
2. In the terminal, type **make** to compile and execute the program. (You might need to set up the system environment variable for make command)

```
gcc -ggdb -w -I/include -c -o yuv_filter.o yuv_filter.c
gcc -ggdb -w -I/include -c -o yuv_filter_test.o yuv_filter_test.c
gcc -ggdb -w -I/include -c -o image_aux.o image_aux.c
gcc -lm yuv_filter.o yuv_filter_test.o image_aux.o -o yuv_filter
./yuv_filter
Test passed!
```

Validating the design

Note that the source files (**yuv_filter.c**, **yuv_filter_test.c**, and **image_aux.c**) were compiled, then **yuv_filter** executable program was created, and then it was executed. The program tests the design and outputs **Test Passed** message.

A Vitis HLS tcl script file is provided and can be used to create the Vitis HLS project.

3. A Vitis HLS tcl script file (**pynq_yuv_filter.tcl**) is provided and can be used to create a Vivado HLS project. Type **vitis_hls -f pynq_yuv_filter.tcl** in the Vitis HLS Command Prompt window to create

the project targeting the xc7z020clg400-1 part. The project will be created and Vitis_hls.log file will be generated.

4. Open the **vitis_hls.log** file from **{lab}/lab2** using any text editor and observe the following sections:
- Creating directory and project called **yuv_filter.prj** within it, adding design files to the project,
setting solution name as solution1, setting target device, setting desired clock period of 10ns, and importing the design and testbench files.
 - Synthesizing (Generating) the design which involves scheduling and binding of each functions and sub-function
 - Generating RTL of each function and sub-function in SystemC, Verilog, and VHDL languages

```

**** VITIS HLS - High-Level Synthesis from C/C++ and OpenCL v2021.2 (64-bit)
**** Do Build 1367213 on Thu Oct 19 02:47:39 PDT 2022
**** IP Build 1606179 on Thu Oct 25 06:23:18 PDT 2022
** Copyright 2006-2021 Xilinx, Inc. All Rights Reserved.

source /tools/Vitis/Vitis_HLS/2021.2/scripts/vitis_hls.tcl -notrace
INFO: Applying HLS V2021.2 patch v1.2 for IP revision
INFO: [HLS 200-10] Running: /tools/Vitis/Vitis_HLS/2021.2/bin/unwrapped/lnx64/vitis_hls
INFO: [HLS 200-11] Run user: 'hazel' on host: 'hazel-100154191600' (Linux x86_64 version: 3.4.0-80-generic) on Tue Feb 14 00:12:11 CST 2022.
INFO: [HLS 200-12] Run as: /vitis 13.04.2 LTS [launcher-orig-vcr@vite 600]
INFO: [HLS 200-13] In directory: '/home/xup/hls/labs/lab2'
Sourcing Tcl script 'pygen_yuv_filter.tcl'
INFO: [HLS 200-1310] Running: open_project -reset yuv_filter.prj
INFO: [HLS 200-13] Opening and resetting project: '/home/xup/hls/labs/lab2/yuv_filter.prj'.
WARNING: [HLS 200-48] No /home/xup/hls/labs/lab2/yuv_filter.prj/solution/solution1.xpr file found.
INFO: [HLS 200-1320] Running: add_files yuv_filter.c
INFO: [HLS 200-13] adding design file 'yuv_filter.c' to the project
INFO: [HLS 200-1330] Running: add_files -is image_xxx.c
INFO: [HLS 200-13] Adding test bench file 'image_xxx.c' to the project
INFO: [HLS 200-1320] Running: add_files -is yuv_filter_test.c
INFO: [HLS 200-13] adding test bench file 'yuv_filter_test.c' to the project.
INFO: [HLS 200-1310] Running: add_files -is test_data
INFO: [HLS 200-13] Adding test bench file 'test_data' to the project
INFO: [HLS 200-1310] Running: set_top yuv_filter
INFO: [HLS 200-1310] Running: open_solution solution1
INFO: [HLS 200-13] Creating and opening solution: '/home/xup/hls/labs/lab2/yuv_filter.prj/solution1'.
INFO: [HLS 200-1250] Using default flow target: 'vivado'
Resolution: For help on HLS 200-1500 see www.xilinx.com/rgt-hls/docs/doc/v=2021.21-hls/guidance;doc=200-1500.html
INFO: [HLS 200-1310] Running: set part a7c7020cig600-0
INFO: [HLS 200-1011] Setting target device to 'a7c7020-cig600-0'
INFO: [HLS 200-1310] Running: create_clock -period 10
INFO: [CNS 201-292] Setting up clock 'default' with a period of 10ns.
INFO: [HLS 200-1310] Running: create_design
INFO: [HLS 200-1311] Finished file checks and directory preparation: CPU user time: 0 seconds, CPU system time: 0 seconds, Elapsed time: 0.01 seconds; current allocated memory: 261.382
INFO: [HLS 200-13] Analyzing design file 'yuv_filter.c' ...
WARNING: [HLS 307-3140] Missing argument for 'port' (yuv_filter.c:13:9).
INFO: [HLS 200-1311] Finished Source Code Analysis and Preprocessing: CPU user time: 0.34 seconds, CPU system time: 0.20 seconds, Elapsed time: 0.57 seconds; current allocated memory:
INFO: [HLS 200-777] Using Interface Defaults for 'Vivado' flow target.

```

```
INFO: [XPPM 283-541] Flattening a loop nest 'RGB2YUV_LOOP_X' (yuv_filter.c:119:18) in function 'yuv_filter'.
INFO: [XPPM 283-541] Flattening a loop nest 'YUV_SCALE_LOOP_X' (yuv_filter.c:119:18) in function 'yuv_filter'.
INFO: [XPPM 283-541] Flattening a loop nest 'YUV2RGB_LOOP_X' (yuv_filter.c:173:18) in function 'yuv_filter'.
INFO: [HLS 280-472] Inferring partial write operation for 'yuv.channels.ch1' (yuv_filter.c:161:34).
INFO: [HLS 280-472] Inferring partial write operation for 'yuv.channels.ch2' (yuv_filter.c:162:34).
INFO: [HLS 280-472] Inferring partial write operation for 'yuv.channels.ch3' (yuv_filter.c:163:34).
INFO: [HLS 280-472] Inferring partial write operation for 'scale.channels.ch1' (yuv_filter.c:181:34).
INFO: [HLS 280-472] Inferring partial write operation for 'scale.channels.ch2' (yuv_filter.c:182:34).
INFO: [HLS 280-472] Inferring partial write operation for 'scale.channels.ch3' (yuv_filter.c:183:34).
INFO: [HLS 280-111] Finished architecture synthesis: CPU user time: 0.09 seconds, CPU system time: 0.01 seconds, Elapsed time: 0.11 seconds; current allocated memory: 327.492 MB.
INFO: [HLS 280-18] Starting hardware synthesis ...
INFO: [HLS 280-18] Synthesizing 'yuv_filter' ...
INFO: [HLS 280-42] -- Implementing module 'yuv_filter_Pipeline_RGB2YUV_LOOP_X_RGB2YUV_LOOP_X'
INFO: [HLS 280-28] -----
INFO: [SCHD 284-13] Starting scheduling ...
INFO: [HLS 280-480] Changing DSP latency (front-end_in00) to 3 in order to utilize available DSP registers.
INFO: [HLS 280-480] Changing DSP latency (front-end_in01) to 3 in order to utilize available DSP registers.
INFO: [HLS 280-480] Changing DSP latency (front-end_in02) to 3 in order to utilize available DSP registers.
INFO: [SCHD 284-61] Pipelining loop 'RGB2YUV_LOOP_X_RGB2YUV_LOOP_X'.
INFO: [HLS 280-1470] Pipelining result: Target II = NA, Front II = 1, Depth = 3, loop 'RGB2YUV_LOOP_X_RGB2YUV_LOOP_X'.
INFO: [SCHD 284-13] Finished scheduling.
INFO: [HLS 280-111] Finished scheduling: CPU user time: 0.04 seconds, CPU system time: 0.04 seconds, Elapsed time: 0.21 seconds; current allocated memory: 329.621 MB.
INFO: [RIND 280-100] Starting micro-architecture generation ...
INFO: [RIND 280-100] Performing variable lifetime analysis.
INFO: [RIND 280-100] Exploring resource sharing.
INFO: [RIND 280-100] Binding ...
INFO: [RIND 280-100] Finished micro-architecture generation.
INFO: [HLS 280-111] Finished binding: CPU user time: 0.00 seconds, CPU system time: 0 seconds, Elapsed time: 0.07 seconds; current allocated memory: 329.621 MB.
INFO: [HLS 280-18] Starting hardware synthesis ...
INFO: [HLS 280-18] Synthesizing 'yuv_filter' ...
INFO: [HLS 280-42] -- Implementing module 'yuv_filter_Pipeline_YUV_SCALE_LOOP_X_YUV_SCALE_LOOP_X'
INFO: [HLS 280-18] -----
INFO: [SCHD 284-13] Starting scheduling ...
INFO: [SCHD 284-61] Pipelining loop 'YUV_SCALE_LOOP_X_YUV_SCALE_LOOP_X'.
INFO: [HLS 280-1470] Pipelining result: Target II = NA, Front II = 1, Depth = 2, loop 'YUV_SCALE_LOOP_X_YUV_SCALE_LOOP_X'.
INFO: [SCHD 284-13] Finished scheduling.
INFO: [HLS 280-111] Finished scheduling: CPU user time: 0.11 seconds, CPU system time: 0 seconds, Elapsed time: 0.11 seconds; current allocated memory: 330.140 MB.
INFO: [RIND 280-100] Starting micro-architecture generation ...
INFO: [RIND 280-100] Performing variable lifetime analysis.
INFO: [RIND 280-100] Exploring resource sharing.
INFO: [RIND 280-100] Binding ...
INFO: [RIND 280-100] Finished micro-architecture generation.
INFO: [HLS 280-111] Finished binding: CPU user time: 0.03 seconds, CPU system time: 0 seconds, Elapsed time: 0.04 seconds; current allocated memory: 330.140 MB.
INFO: [HLS 280-18] Starting hardware synthesis ...
INFO: [HLS 280-18] Synthesizing 'yuv_filter' ...
INFO: [HLS 280-42] -- Implementing module 'yuv_filter_Pipeline_YUV2RGB_LOOP_X_YUV2RGB_LOOP_X'
INFO: [HLS 280-28] -----
INFO: [SCHD 284-13] Starting scheduling ...
INFO: [HLS 280-480] Changing DSP latency (front-end_in03) to 3 in order to utilize available DSP registers.
INFO: [HLS 280-480] Changing DSP latency (front-end_in04) to 3 in order to utilize available DSP registers.
INFO: [HLS 280-480] Changing DSP latency (front-end_in05) to 3 in order to utilize available DSP registers.
INFO: [SCHD 284-61] Pipelining loop 'YUV2RGB_LOOP_X_YUV2RGB_LOOP_X'.
INFO: [HLS 280-1470] Pipelining result: Target II = NA, Front II = 1, Depth = 11, loop 'YUV2RGB_LOOP_X_YUV2RGB_LOOP_X'.
INFO: [SCHD 284-13] Finished scheduling.

Synthesizing (Generating) the design

INFO: [RTN 280-198] Finished creating RTL model for 'yuv_filter'.
INFO: [HLS 280-111] Finished creating RTL model: CPU user time: 0.21 seconds, CPU system time: 0 seconds, Elapsed time: 0.21 seconds; current allocated memory: 338.355 MB.
INFO: [WPG 210-270] Implementing memory 'yuv_filter_yuv_channels.ch1_RAM_AUTO_IPSR_RAM' using auto RAMs.
INFO: [HLS 280-111] Finished generating all RTL models: CPU user time: 0.52 seconds, CPU system time: 0 seconds, Elapsed time: 0.50 seconds; current allocated memory: 341.008 MB.
INFO: [HLS 280-111] Finished updating report files: CPU user time: 0.2 seconds, CPU system time: 0.01 seconds, Elapsed time: 0.24 seconds; current allocated memory: 345.824 MB.
INFO: [VHDL 280-350] Generating VHDL RTL for 'yuv_filter'.
INFO: [VLOG 280-350] Generating Verilog RTL for 'yuv_filter'.
INFO: [HLS 280-790] *** Loop Constraint Status: All loop constraints were satisfied.
INFO: [HLS 280-790] *** Estimated Fmax: 143.87 MHz.
INFO: [HLS 280-111] Finished Command 'synth_design' CPU user time: 3.48 seconds, CPU system time: 0.05 seconds, Elapsed time: 0.11 seconds; current allocated memory: 341.227 MB.
INFO: [HLS 280-112] Total CPU user time: 9.36 seconds, Total CPU system time: 1.04 seconds, Total elapsed time: 22.97 seconds; peak allocated memory: 346.185 MB.
INFO: [Common 17-100] Exiting vitis_hls at Mon Feb 16 00:33:37 2022...
```

5. Open the created project (in GUI mode) from the Vitis HLS Command Prompt window, by typing **vitis_hls -p yuv_filter.prj**.

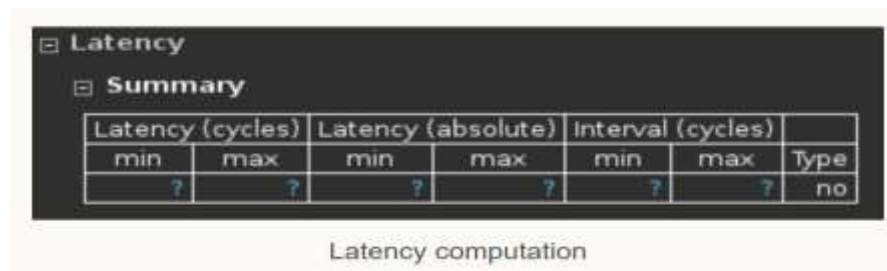
The Vitis HLS will open in GUI mode and the project will be opened.

STEP 2: Analyze the Created Project and Results

Open the source file and note that three functions are used. Look at the results and observe that the latencies are undefined (represented by ?).

1. In Vitis HLS GUI, expand the **source** folder in the Explorer view and double-click **yuv_filter.c** to view the content.
 - The design is implemented in 3 functions: **rgb2yuv**, **yuv_scale** and **yuv2rgb**.
 - Each of these filter functions iterates over the entire source image (which has maximum dimensions specified in image_aux.h), requiring a single source pixel to produce a pixel in the result image.
 - The scale function simply applies individual scale factors, supplied as top-level arguments to the Y'UV components.
 - Notice that most of the variables are of user-defined (typedef) and aggregate (e.g. structure, array) types.

- Also notice that the original source used malloc() to dynamically allocate storage for the internal image buffers. While appropriate for such large data structures in software, malloc() is not synthesizable and is not supported by Vitis HLS.
 - A viable workaround is conditionally compiled into the code, leveraging the `__SYNTHESIS__` macro. Vitis HLS automatically defines the `__SYNTHESIS__` macro when reading any code. This ensures the original malloc() code is used outside of synthesis but Vitis HLS will use the workaround when synthesizing.
2. Expand the **syn>report** folder in the Explorer view and double-click **yuv_filter_csynth.rpt** entry to open the synthesis report.
 3. Each of the loops in this design has variable bounds – the width and height are defined by members of input type image_t. When variables bounds are present on loops the total latency of the loops cannot be determined: this impacts the ability to perform analysis using reports. Hence, “?” is reported for various latencies.



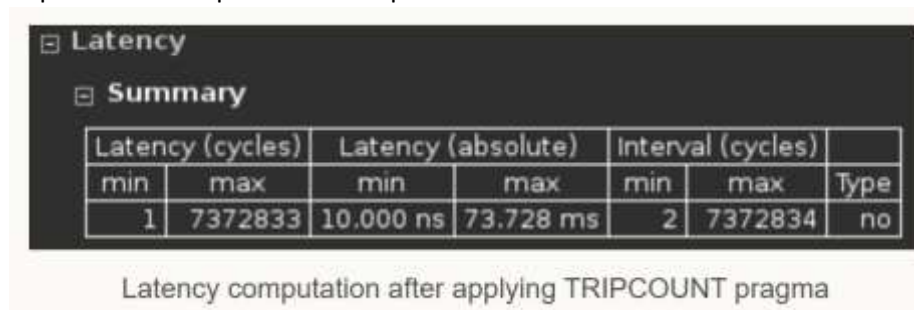
Latency computation

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
?	?	?	?	?	?	no

STEP 3: Apply TRIPCOUNT Pragma

Open the source file and uncomment pragma lines, re-synthesize, and observe the resources used as well as estimated latencies. Answer the questions listed in the detailed section of this step.

1. To assist in providing loop-latency estimates, Vitis HLS provides a TRIPCOUNT directive which allows limits on the variables bounds to be specified by the user. In this design, such directives have been embedded in the source code, in the form of #pragma statements.
2. Uncomment the **#pragma** lines (**50, 53, 90, 93, 130, 133**) to define the loop bounds and save the file.
3. Synthesize the design by selecting **Solution > Run C Synthesis > Active Solution**. View the synthesis report when the process is completed.



Latency computation after applying TRIPCOUNT pragma

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1	7372833	10.000 ns	73.728 ms	2	7372834	no

Looking at the report, and answer the following question.

Question 1

Estimated clock period:

Worst case latency:

Number of DSP48E used:

Number of BRAMs used:

Number of FFs used:

Number of LUTs used:

- Expand the **Module & Loop** in the “Synthesis Summary” and note the latency and trip count numbers for the yuv_scale function. Note that the iteration latency of *YUV_SCALE_LOOP_X_YUV_SCALE_LOOP_Y* is 6x the specified TRIPCOUNT, implying that 6 cycles are used for each of the iteration of the loop.

Module & Loops	Issue Type	Violation Type	Distance	Block	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
▼ yuv_filter	-	-	-	-	7372833	7.372834	-	7372834	-	no
▶ yuv_filter_Pipeline_RGB2YUV_LOOP_X_RGB2YUV_LOOP_Y	-	-	-	-	2457606	2.458E7	-	2457606	-	no
▼ yuv_filter_Pipeline_YUV_SCALE_LOOP_X_YUV_SCALE_LOOP_Y	-	-	-	-	2457606	2.458E7	-	2457606	-	no
▶ yuv_filter_Pipeline_YUV_SCALE_LOOP_X_YUV_SCALE_LOOP_Y	-	-	-	-	2457604	2.458E7	6	1	2457600	yes
▶ yuv_filter_Pipeline_YUV2RGB_LOOP_X_YUV2RGB_LOOP_Y	-	-	-	-	2457610	2.458E7	-	2457610	-	no

Loop latency

Note that *YUV_SCALE_LOOP_X_YUV_SCALE_LOOP_Y* is already pipelined. By default, Vitis HLS automatically pipelines loops with a tripcount of 64, or greater. If this option is applied, the innermost loop with a tripcount higher than the threshold is pipelined, or if the tripcount of the innermost loop is less than or equal to the threshold, its parent loop is pipelined. If the innermost loop has no parent loop, the innermost loop is pipelined regardless of its tripcount.

- In the report tab, expand and click on the **yuv_filter_Pipeline_RGB2YUV_LOOP_X_RGB2YUV_LOOP_Y** entry to open the report.
- Answer the following question pertaining to rgb2yuv function.

Question 2

Estimated clock period:

Worst case latency:

Number of DSP48E used:

Number of FFs used:

Number of LUTs used:

- Similarly, open the *yuv2rgb* function and answer the following questions pertaining to yuv2rgb function.

Question 3

Estimated clock period:

Worst case latency:

Number of DSP48E used:

Number of FFs used:

Number of LUTs used:

STEP 4: Remove the pipeline optimization done by Vitis HLS automatically by adding pipeline off pragma

- Select **Project > New Solution**
- A *Solution Configuration* dialog box will appear. Note that the check boxes of *Copy existing directives and constraints from solution* are checked with *solution1* selected. Click the **Finish** button to create a new solution with the default settings.

Solution Wizard @maxwell-OptiPlex-7080

Solution Configuration

Create Vitis HLS solution for selected technology

Solution Name:

Clock:
Period: Uncertainty:

Part Selection:
Part:

Options:
☒ Copy directives and constraints from solution:

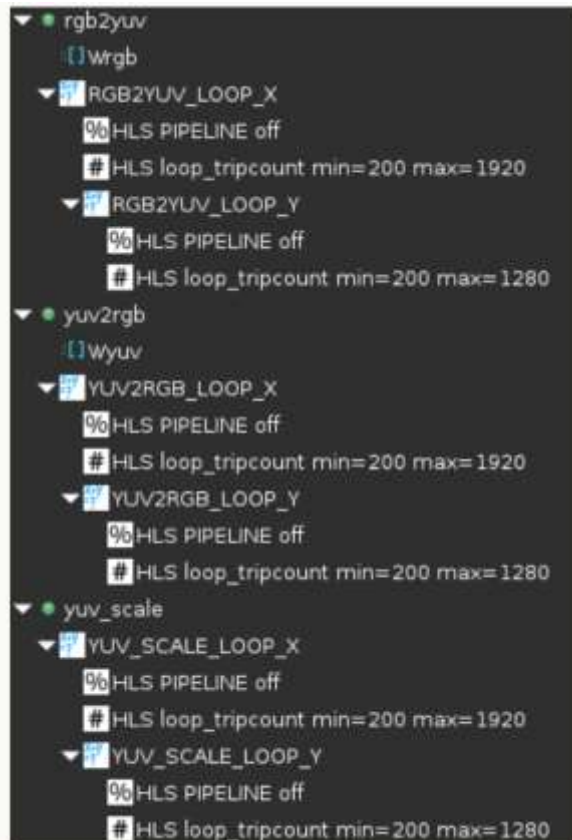
Flow Target:
 Configure [several options](#) for the selected flow target

Creating a new Solution after copying the existing solution

3. Make sure that the **yuv_filter.c** source is opened and visible in the information pane, and click on the **Directive** tab.
4. Select function **RGB2YUV_LOOP_X** in the directives pane, right-click on it and select **Insert Directive...**
5. Click on the drop-down button of the *Directive* field. A pop-up menu shows up listing various directives. Select **PIPELINE** directive.
6. In the *Vitis HLS Directive Editor* dialog box, click on the **off** option to turn OFF the automatic pipelining. Make sure that the Directive File is selected as destination. Click **OK**.



7. Similarly, apply the **PIPELINE off** directive to **YUV2RGB_LOOP_X**, **YUV2RGB_LOOP_Y**, **YUV_SCALE_LOOP_X**, **YUV_SCALE_LOOP_Y** and **RGB2YUV_LOOP_Y** objects. At this point, the Directive tab should look like as follows.



PIPELINE off directive applied

8. Click on the **Synthesis** button.
9. When the synthesis is completed, report shows the performance and area without the automatic optimization of Vitis HLS.

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	8.651 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trips Count	Pipelined	BRAM	DSP	FF	LUT	URAM
yuv_8btr	-	-	-	-	44248323	4.420E9	-	44248324	-	no	12288	7	518	1230	0
RGB2YUV_LOOP_X	-	-	-	-	14789440	1.470E9	7682	-	1920	no	-	-	-	-	-
YUV_SCALE_LOOP_X	-	-	-	-	9834240	9.834E7	5122	-	1920	no	-	-	-	-	-
YUV2RGB_LOOP_X	-	-	-	-	19668640	1.970E9	10242	-	1920	no	-	-	-	-	-

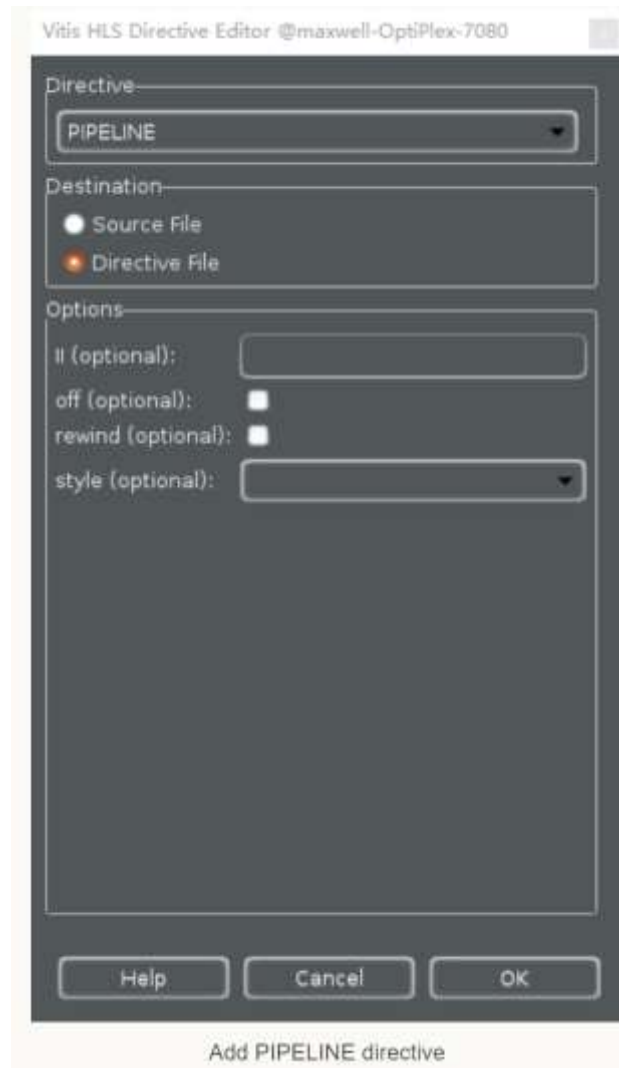
Performance after applying PIPELINE off directive

STEP 5: Apply PIPELINE Directive

Create a new solution by copying the previous solution settings. Apply PIPELINE directive. Generate the solution and understand the output.

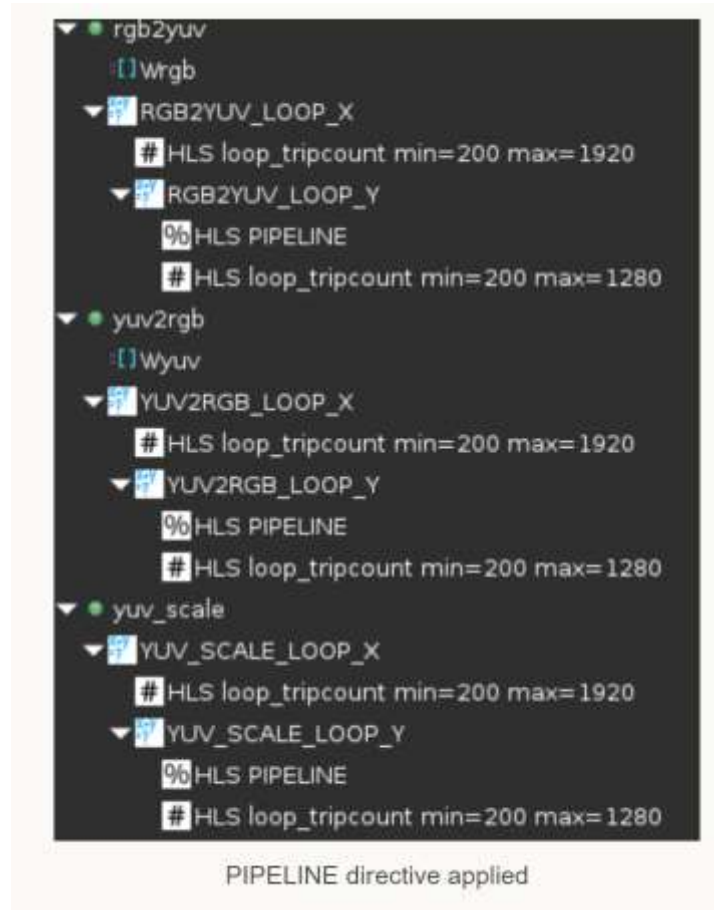
1. Select **Project > New Solution**

2. A *Solution Configuration* dialog box will appear. Click the **Finish** button (with copy from Solution2 selected).
3. Make sure that the **yuv_filter.c** source is opened and visible in the information pane and click on the **Directive** tab.
4. Select the pragma *HLS PIPELINE off* of **RGB2YUV_LOOP_Y** in the directives pane, right-click on it, and select **Modify Directive**.
5. In the *Vitis HLS Directive Editor* dialog box, click the **off** option to turn on the pipelining. Make sure that the *Directive File* is selected as destination. Click **OK**.



- When an object (function or loop) is pipelined, all the loops below it, down through the hierarchy, will be automatically unrolled.
- In order for a loop to be unrolled it must have fixed bounds: all the loops in this design have variable bounds, defined by an input argument variable to the top-level function.
- Note that the TRIPCOUNT directive on the loops only influences reporting, it does not set bounds for synthesis.
- Neither the top-level function nor any of the sub-functions are pipelined in this example.

- The pipeline directive must be applied to the inner-most loop in each function – the innermost loops have no variable-bounded loops inside of them which are required to be unrolled and the outer loop will simply keep the inner loop fed with data.
6. Leave // (Initiation Interval) blank as Vitis HLS will try for an II=1, one new input every clock cycle.
 7. Click **OK**.
 8. Similarly, apply the **PIPELINE** directive to **YUV2RGB_LOOP_Y** and **YUV_SCALE_LOOP_Y** objects, but remove the **PIPELINE** directive of **YUV2RGB_LOOP_X**, **YUV_SCALE_LOOP_X** and **RGB2YUV_LOOP_X** objects. At this point, the *Directive* tab should look like as follows.



9. Click on the **Synthesis** button.
10. When the synthesis is completed, select **Project > Compare Reports...** to compare the two solutions.
11. Select *Solution2* and *Solution3* from the **Available Reports**, and click on the **Add>>** button.
12. Observe that the latency is reduced.

Performance Estimates			
Timing			
Clock		solution2	solution3
ap_clk	Target	10.00 ns	10.00 ns
	Estimated	6.651 ns	6.960 ns
Latency			
		solution2	solution3
Latency (cycles)	min	1	1
	max	44248323	7372833
Latency (absolute)	min	10.000 ns	10.000 ns
	max	0.442 sec	73.728 ms
Interval (cycles)	min	2	2
	max	44248324	7372834

Performance comparison after pipelining

In Solution2, the total loop latency of the inner-most loop was $\text{loop_body_latency} \times \text{loop iteration count}$, whereas in Solution3 the new total loop latency of the inner-most loop is $\text{loop_body_latency} + \text{loop iteration count}$.

13. Scroll down in the comparison report to view the resources utilization. Observe that the FFs, LUTs, and DSP48E utilization increased whereas BRAM remained same.

Utilization Estimates		
	solution2	solution3
BRAM_18K	12288	12288
DSP	7	8
FF	518	932
LUT	1230	1728
URAM	0	0

Resources utilization after pipelining

STEP 6: Apply DATAFLOW Directive and Configuration Command

Create a new solution by copying the previous solution (Solution3) settings. Apply DATAFLOW directive. Generate the solution and understand the output.

1. Select **Project > New Solution**
2. A *Solution Configuration* dialog box will appear. Click the **Finish** button (with copy from Solution3 selected).
3. Close all inactive solution windows by selecting **Project > Close Inactive Solution Tabs**.
4. Make sure that the **yuv_filter.c** source is opened in the information pane and select the Directive tab.
5. Select function **yuv_filter** in the *Directives* pane, right-click on it and select **Insert Directive...**
6. A pop-up menu shows up listing various directives. Select **DATAFLOW** directive and click **OK**.
7. Click on the **Synthesis** button.
8. When the synthesis is completed, the synthesis report is automatically opened.

9. Observe additional information, **Dataflow** Type, in the Performance Estimates section is mentioned.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.271 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
120041	7372841	1.200 ms	73.728 ms	40015	2457615	dataflow

Performance estimate after DATAFLOW directive applied

- The Dataflow pipeline throughput indicates the number of clocks cycles between each set of inputs reads. If this throughput value is less than the design latency it indicates the design can start processing new inputs before the currents input data are output.
 - While the overall latencies haven't changed significantly, the dataflow throughput (interval) is showing that the design can achieve close to the theoretical limit ($1920 \times 1280 = 2457600$) of processing one pixel every clock cycle.
10. Scrolling down into the Utilization Estimates, observe that the number of BRAMs required has doubled. This is due to the default dataflow ping-pong buffering.

Utilization Estimates					
Summary					
Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	60	-
FIFO	-	-	693	476	-
Instance	-	11	1175	1706	-
Memory	24576	-	0	0	0
Multiplexer	-	-	-	108	-
Register	-	-	12	-	-
Total	24576	11	1880	2350	0
Available	280	220	106400	53200	0
Utilization (%)	8777	5	1	4	0

Resource estimate with DATAFLOW directive applied

- When **DATAFLOW** optimization is performed, memory buffers are automatically inserted between the functions to ensure the next function can begin operation before the

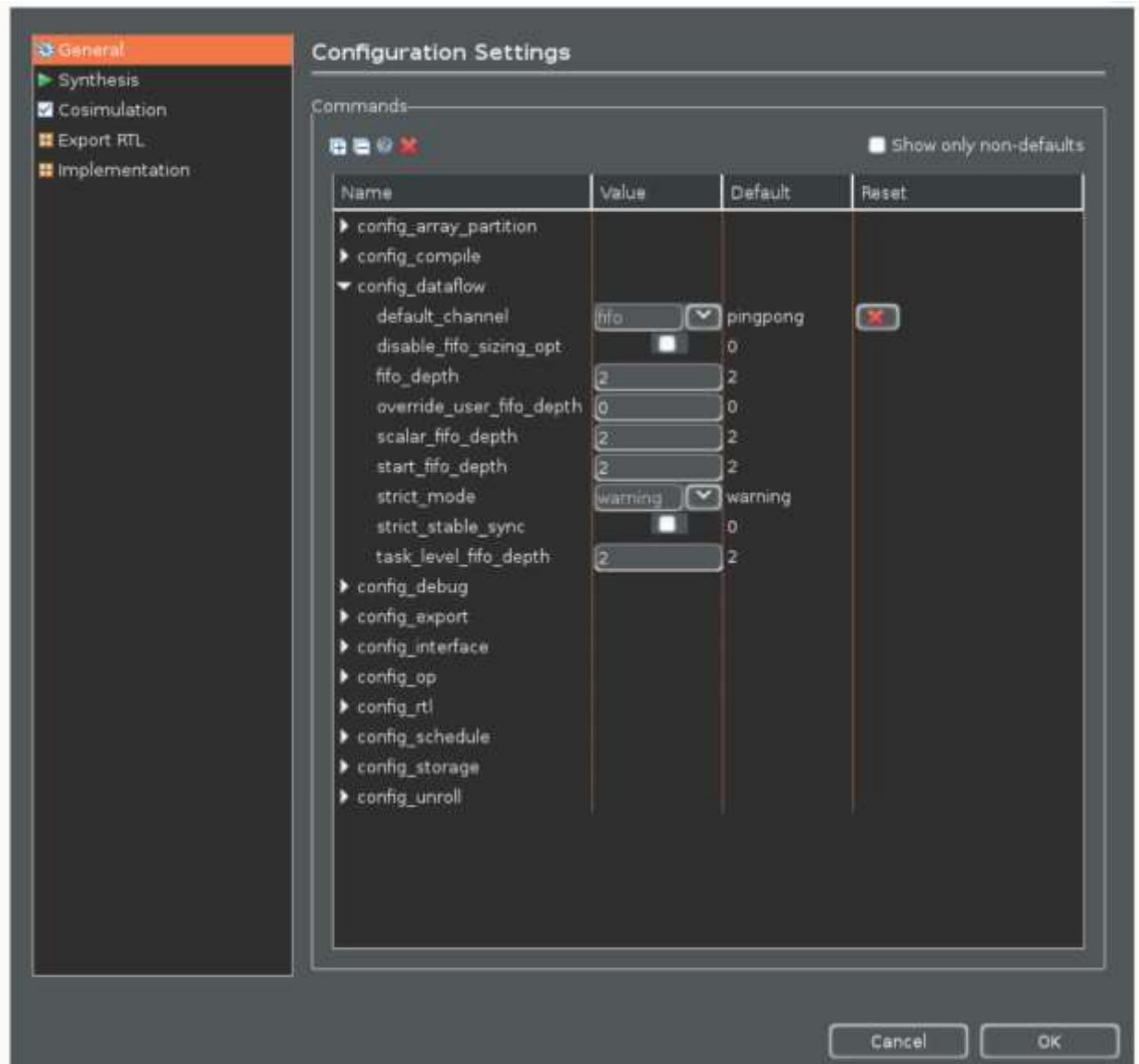
previous function has finished. The default memory buffers are ping-pong buffers sized to fully accommodate the largest producer or consumer array.

- Vitis HLS allows the memory buffers to be the default **ping-pong** buffers or **FIFOs**. Since this design has data accesses which are fully sequential, FIFOs can be used. Another advantage to using FIFOs is that the size of the FIFOs can be directly controlled (not possible in ping-pong buffers where random accesses are allowed).

11. The memory buffers type can be selected using Vitis HLS Configuration command.

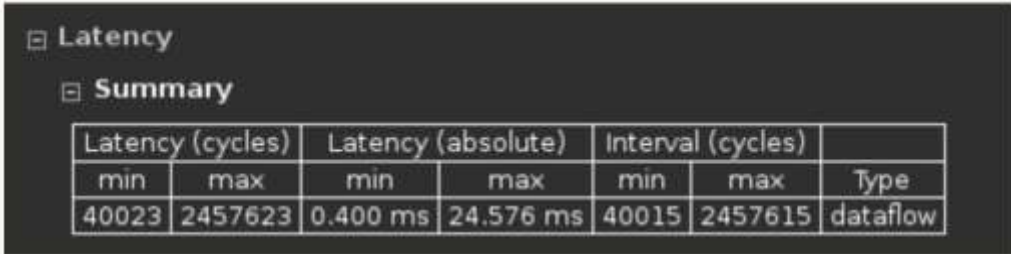
STEP 7: Apply Dataflow configuration command, generate the solution, and observe the improved resources utilization.

1. Select **Solution > Solution Settings...** to access the configuration command settings.
2. In the *Configuration Settings* dialog box, expand **config_dataflow** folder.
3. Set **fifo** as the default_channel. Enter **2** as the fifo_depth. Click **OK**.



Selecting Dataflow configuration command and FIFO as buffer

4. Click **OK** again.
5. Click on the **Synthesis** button.
6. When the synthesis is completed, the synthesis report is automatically opened.
7. Note that the latency has reduced. Since this design has data accesses which are fully sequential, the data can flow to next function without waiting for all pixels to be processed.



The screenshot shows a 'Latency' section with a 'Summary' table. The table has columns for Latency (cycles), Latency (absolute), and Interval (cycles), each with min and max values, and a Type column. The values are: Latency (cycles) min=40023, max=2457623; Latency (absolute) min=0.400 ms, max=24.576 ms; Interval (cycles) min=40015, max=2457615; Type=dataflow.

Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
min	max	min	max	min	max	
40023	2457623	0.400 ms	24.576 ms	40015	2457615	

Latency estimation after Dataflow configuration command

CONCLUSION

In this lab, you learned that even though this design could not be pipelined at the top-level, a strategy of pipelining the individual loops and then using dataflow optimization to make the functions operate in parallel was able to achieve the same high throughput, processing one pixel per clock. When DATAFLOW directive is applied, the default memory buffers (of ping-pong type) are automatically inserted between the functions. Using the fact that the design used only sequential (streaming) data accesses allowed the costly memory buffers associated with dataflow optimization to be replaced with simple 2 element FIFOs using the Dataflow command configuration.

References:

https://xilinx.github.io/xup_high_level_synthesis_design_flow/Lab2.html