# SmartHLS™ Tutorial for Microchip PolarFire®:

# Sobel Filtering for Image Edge Detection

Revision 6.0

February 2023

# Contents

# 1 Revision History

| Revision | Date | Changes |
|---|---|---|
| 1.0 | December, 2020 | Initial Publication |
| 2.0 | March, 2021 | Updates for LegUp HLS EAP 9.2 release |
| 3.0 | June, 2021 | Updates for SmartHLS 2021.1.2 release |
| 4.0 | February, 2022 | Updates for SmartHLS 2021.3.1 release |
| 5.0 | October, 2022 | Updates for SmartHLS 2022.2 release |
| 6.0 | February, 2023 | Updates for SmartHLS 2022.3 release |

# 2 Requirements

Before beginning this tutorial, you should install the following software:

- SmartHLS™ 2022.3 or later
- Libero® SoC 2022.3 or later with Modelsim

This document uses the Windows versions of Libero® SoC 2022.3 and SmartHLS 2022.3. Depending on the version you use, the results generated from your Libero® SoC and SmartHLS could be slightly different from that presented in this document.

👆 We will use this cursor symbol throughout this tutorial to indicate sections where you need to perform actions to follow along.

# 3 Introduction

This tutorial will introduce you to high-level synthesis (HLS) concepts using SmartHLS. You will apply HLS to a real problem: synthesizing an image processing application from software written in the C++ programming language. Specifically, you will synthesize a circuit that performs one of the key steps of *edge detection* – a widely used transformation that identifies the edges in an input image and produces an output image showing just those edges. The step you will implement is called *Sobel* filtering. The computations in Sobel filtering are identical to those involved in convolutional layers of a convolutional neural network (CNN). Figure 1 shows an example of Sobel filtering applied to an image. This is the image that will be used as input data in this tutorial.

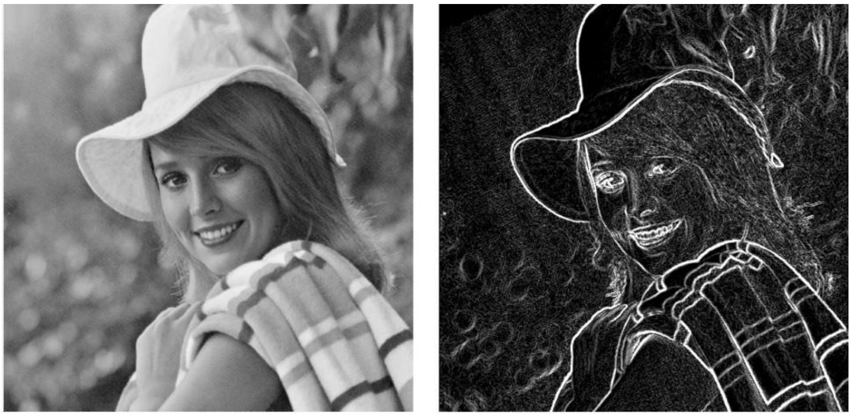


Figure 1: Sobel Filtering Before and After

Sobel filtering involves applying a pair of two 3 x 3 convolutional kernels (also called filters) to an image. The kernels are usually called $G_x$ and $G_y$ and they are shown below in the top of Figure 2. These two kernels "detect" the edges in the image in the horizontal and vertical directions. At each position in the input image, they are applied separately and then combined to produce a pixel value in the output image. The output value is approximated by:

$$G = |G_x| + |G_y|$$

Where the $G_x$ and $G_y$ terms are computed by multiplying each filter value by a pixel value and then summing the products together.

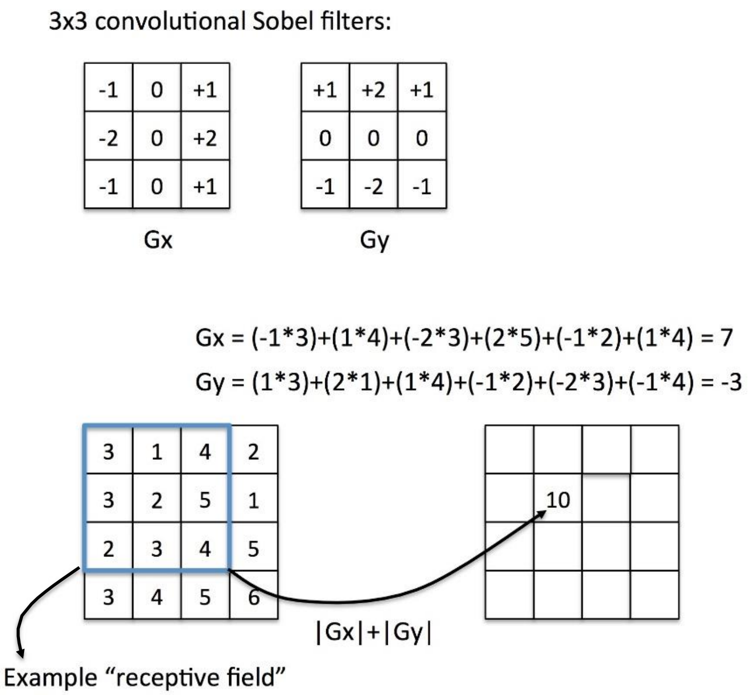

3x3 convolutional Sobel filters:
-1 0 +1
-2 0 +2
-1 0 +1
Gx
+1 +2 +1
0 0 0
-1 -2 -1
Gy
Gx = (-1*3)+(1*4)+(-2*3)+(2*5)+(-1*2)+(1*4) = 7
Gy = (1*3)+(2*1)+(1*4)+(-1*2)+(-2*3)+(-1*4) = -3
3 1 4 2
3 2 5 1
2 3 4 5
3 4 5 6
10
|Gx|+|Gy|
Example "receptive field"

Figure 2: Sobel filters and computational action to compute one pixel in the output image from an input image.

The bottom of Figure 2 shows an input image with 4 rows and 4 columns, where the value in each cell represents a pixel color. The figure illustrates the action of applying the Sobel filters at one position in the input image to compute one value in the output image. The input image pixels involved in the computation are often referred to as the *receptive field*.

A question that may occur to you is: what happens at the edges of the image? i.e., the locations where the placement of the 3 x 3 Sobel filters would "slide off" the edge of the input image. In this tutorial, our program/circuit will simply place 0s into the perimeter of the output image corresponding to such edge locations – this is referred to as *padding* and it is a commonly done technique.

The files for this tutorial can be found on github:

https://github.com/MicrochipTech/fpga-hls-examples/tree/main/sobel_tutorial

You can download all the files using this link:

https://github.com/MicrochipTech/fpga-hls-examples/archive/refs/heads/main.zip

Extract this zip file and navigate to the sobel_tutorial folder:

```
fpga-hls-examples-main\sobel_tutorial
```

# 4 Part 1: Basic Implementation

In this section, we will use SmartHLS to compile the Sobel filter to hardware, without any modifications to the C++ code.

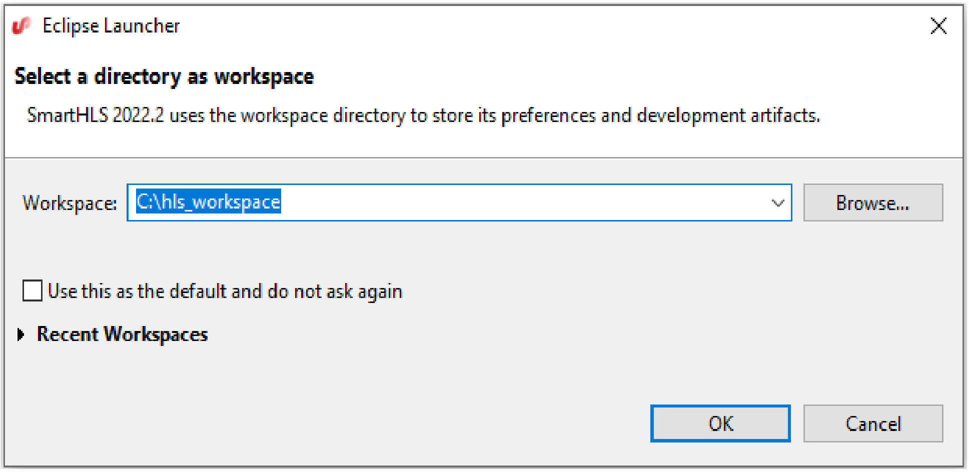
First start the SmartHLS IDE:

On Windows, this can be done by double-clicking on the SmartHLS shortcut either in the start menu or the desktop.

On Linux, make sure that `$(SMARTHLS_INSTALL_DIR)/SmartHLS/bin` is on your PATH and the SmartHLS IDE can be opened by running the following command:

```
shls_ide
```

You will first see a dialog box to select a workspace directory as shown in Figure 3 below. You can use the default workspace for all parts of this tutorial by clicking on *OK*.



**Select a directory as workspace**

SmartHLS 2022.2 uses the workspace directory to store its preferences and development artifacts.

Workspace: C:\hls_workspace

□ Use this as the default and do not ask again

Figure 3: Choosing a SmartHLS workspace.

**Warning:** Make sure there are no spaces in your workspace path. Otherwise, SmartHLS will error out when running synthesis.

Once the SmartHLS IDE opens, under the File menu, choose New and then SmartHLS C/C++ project as shown below in Figure 4.

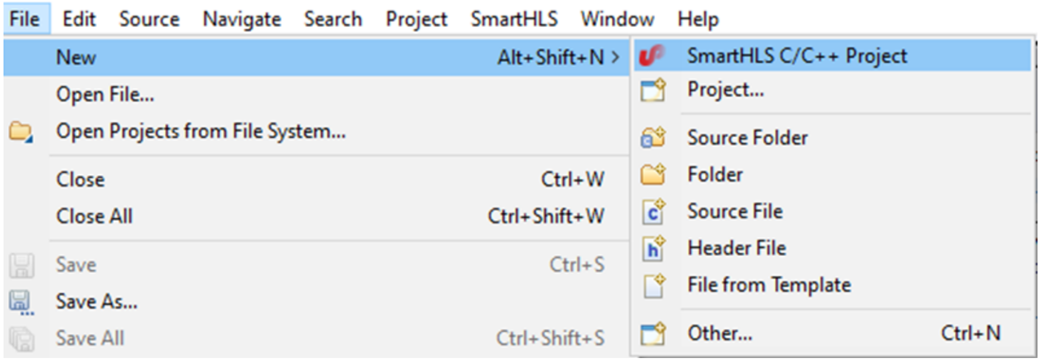
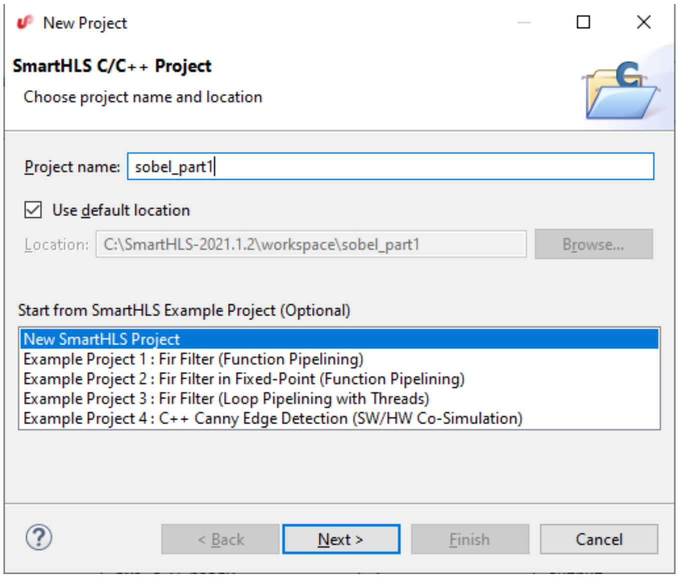


Figure 4: Create a new SmartHLS C/C++ project

For the project name, enter “`sobel_part1`” as shown in  Figure 5. Then click on *Next*.



**SmartHLS C/C++ Project**

Choose project name and location

Project name: sobel_part1

☑ Use default location

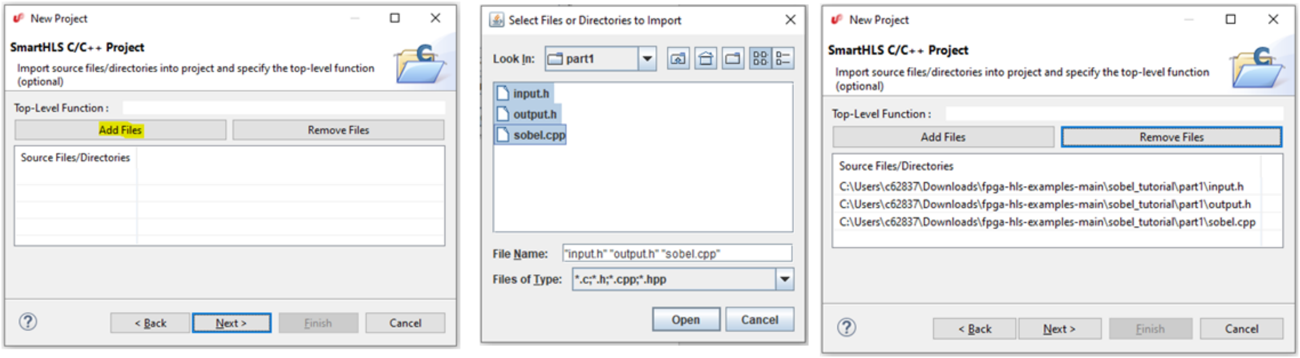
Location: C:\SmartHLS-2021.1.2\workspace\sobel_part1

Start from SmartHLS Example Project (Optional)

New SmartHLS Project
Example Project 1 : Fir Filter (Function Pipelining)
Example Project 2 : Fir Filter in Fixed-Point (Function Pipelining)
Example Project 3 : Fir Filter (Loop Pipelining with Threads)
Example Project 4 : C++ Canny Edge Detection (SW/HW Co-Simulation)

Figure 5: Creating a new SmartHLS project

As shown in Figure 6, you can leave the *Top-Level Function* blank since the `sobel_filter` function in sobel.cpp already has a pragma to indicate the top-level function. Now click on *Add Files* to import the source files for part 1 of this tutorial into the project. Navigate to where you have downloaded the tutorial files and go into the *part1* directory. You can hold shift to select all three source files: input.h, output.h and sobel.cpp. After you have added the source files to the project, click on *Next*.



SmartHLS C/C++ Project

Import source files/directories into project and specify the top-level function (optional)

Top-Level Function :

Look In: part1

File Name: "input.h" "output.h" "sobel.cpp"

Files of Type: *.c;*.h;*.cpp;*.hpp

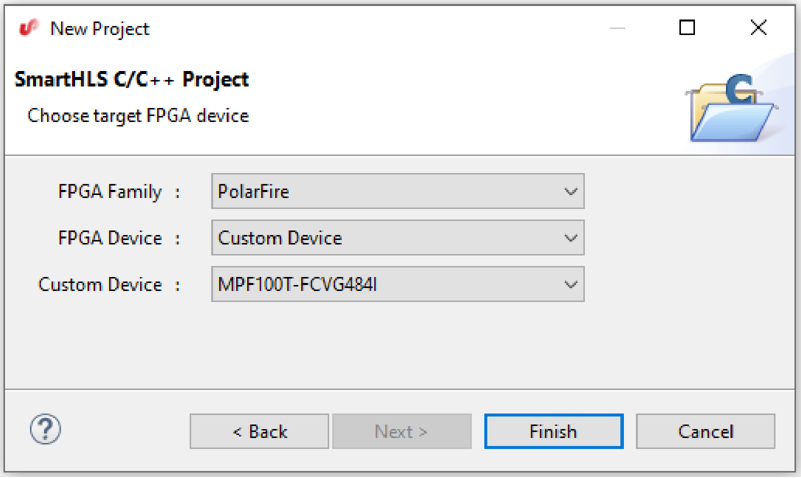
Source Files/Directories

C:\Users\c62837\Downloads\fpga-hls-examples-main\sobel_tutorial\part1\input.h
C:\Users\c62837\Downloads\fpga-hls-examples-main\sobel_tutorial\part1\output.h
C:\Users\c62837\Downloads\fpga-hls-examples-main\sobel_tutorial\part1\sobel.cpp

Figure 6: Adding part1 source files into the new SmartHLS project.

Next you will then see a dialog box where you can specify your own testbench, which are not needed for this part of the tutorial. So, click Next, without changing any of the options.

Finally, to complete the project creation, you will choose the FPGA device you intend to target. Use the selections shown in Figure 7, for *FPGA family* choose PolarFire®. For *FPGA Device*, you have an option to choose “MPF300TS-1FCG1152I on the MPF300 Board” or use another PolarFire device that is not listed. For this tutorial, we will use another PolarFire device, MPF100T-FCVG484I, which can be used with a Microsemi Libero® free *Silver* license (the bigger MPF300TS-1FCG1152I device requires the paid *Gold* license). To use MPF100T, choose *Custom Device* for the *FPGA Device* field, then choose MPF100T-FCVG484I for the *Custom Device* field. Click on *Finish* when you are done. It may take a few moments to create the project.



SmartHLS C/C++ Project

Choose target FPGA device

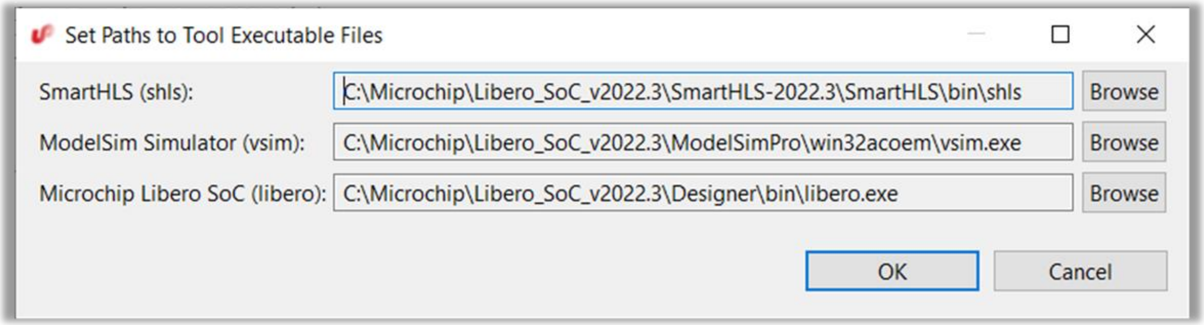
FPGA Family : PolarFire

FPGA Device : Custom Device

Custom Device : MPF100T-FCVG484I

Figure 7: Choose FPGA device

If this is the first time you are using SmartHLS, you will need to set up the paths to Modelsim (and Microsemi Libero® for later parts of this tutorial). To setup the paths, click on *SmartHLS* on the top menu bar, then click on *Tool Path Settings*. Once the dialog opens, set the paths for *ModelSim Simulator* and *Microsemi Libero® SoC* as shown in Figure 8 and click *OK*.



Set Paths to Tool Executable Files

SmartHLS (shls): C:\Microchip\Libero_SoC_v2022.3\SmartHLS-2022.3\SmartHLS\bin\shls

ModelSim Simulator (vsim): C:\Microchip\Libero_SoC_v2022.3\ModelSimPro\win32acoem\vsim.exe

Microchip Libero SoC (libero): C:\Microchip\Libero_SoC_v2022.3\Designer\bin\libero.exe

Figure 8: SmartHLS Tool Path Settings.

An important panel of the SmartHLS IDE is the *Project Explorer* on the left side of the window as shown in Figure 9. We will use the project explorer throughout this tutorial to view source files and synthesis reports.

Click on the small arrow icon to expand the `sobel_part1` project. You can now double click any of the source files, such as sobel.cpp, and you will see the source file appear in the main panel to the right of the *Project Explorer*.

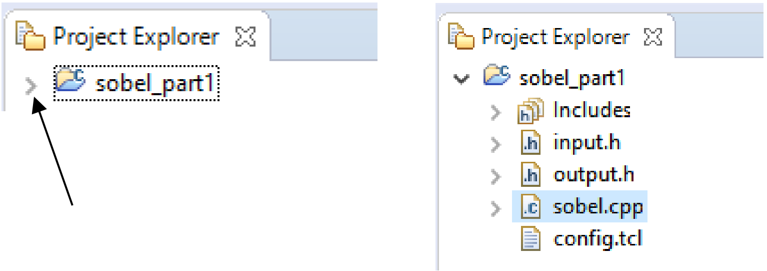


Figure 9: Project Explorer for browsing source files and reports.

Once a SmartHLS project is created, you should always open one of the source files (such as sobel.cpp) or double-click on the sobel_part1 directory in the Project Explorer pane. This will make sobel_part1 the active project.

When there are multiple projects open in the workspace, you need to click on the project in the Project Explorer pane or open a file from the project in order to make the project active before running any SmartHLS commands.

Towards the top of SmartHLS, you should find a toolbar, as shown in Figure 10, which you can use to execute the main features of the SmartHLS tool. Hover over each icon to find out their meanings.

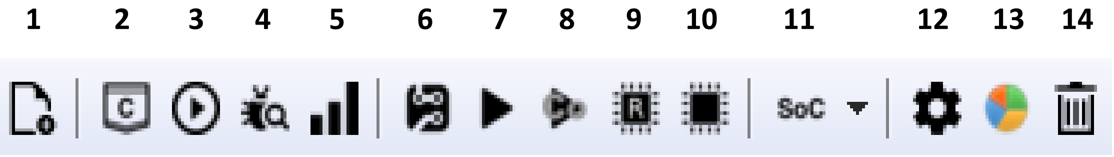


Figure 10: SmartHLS toolbar icons

Starting from the left of Figure 10, the icons are:

1) **Add Files to Project**

Then icons for the software development flow:

2) **Compile Software with GCC**
3) **Run Software that was compiled**
4) **Debug Software**
5) **Profile Software with gprof**

The hardware development flow icons are:

6) **Compile Software to Hardware**
7) **Simulate Hardware**
8) **Software/Hardware Co-simulation**
9) **Synthesize Hardware to FPGA – RTL synthesis only for resource results**
10) **Synthesize Hardware to FPGA – RTL synthesis, place and route for resource and timing results**
11) **Compile Software to Processor/Accelerator SoC**

With the last three icons, you can:

12) **Set HLS Constraints**
13) **Launch Schedule Viewer**
14) **Clean SmartHLS Project**

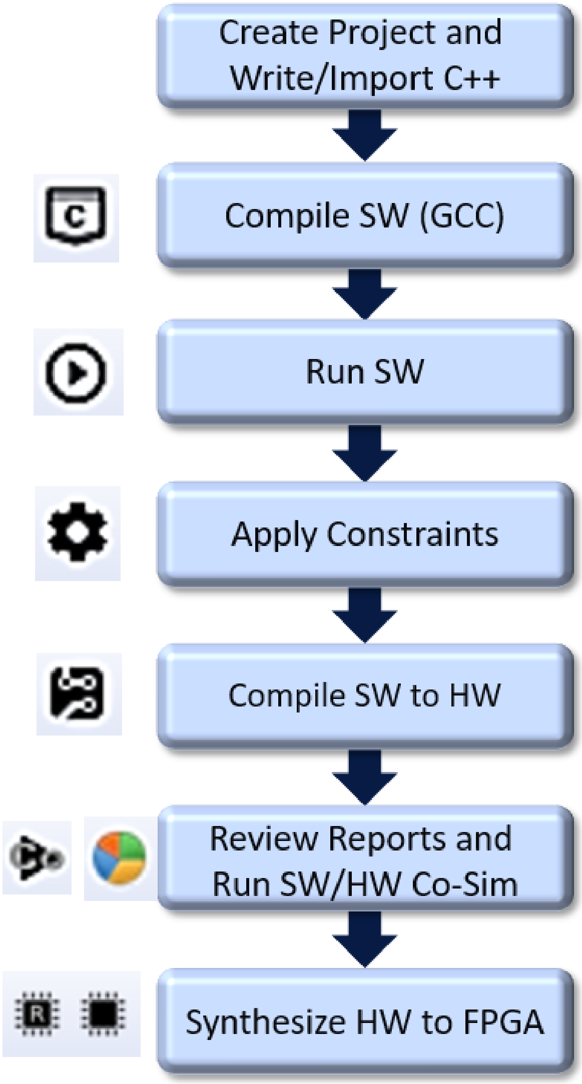


Figure 11: SmartHLS Design Flow Steps

These SmartHLS commands can also be run from the *SmartHLS* top bar menu.

Figure 11 summarizes the SmartHLS design flow steps. We initially create the SmartHLS project and follow a standard software development flow on the C++ (compile/run/debug). Then we apply HLS constraints (i.e., target clock period) and compile the software into hardware using SmartHLS. We can review reports about the generated hardware. Then we run software/hardware co-simulation to verify the generated hardware. Finally, we can synthesize the hardware to our target FPGA to report the hardware resource usage and Fmax.

We can now browse through the code in sobel.cpp. In the `sobel_filter` function, the first line "**#pragma** `HLS function top`" specifies that the `sobel_filter` function is the top-level function of the project. SmartHLS will only generate a hardware module for the top-level function and all descendent functions.

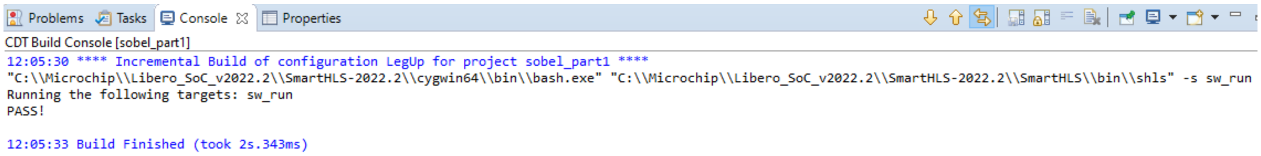
The `sobel_filter` top-level function contains a pair of nested loops that iterate over every pixel in the image. Inside the loop body is another pair of nested loops that iterate through the filter window at the current location in the image. For each pixel of the image that is not in the border, the 3x3 area centered on the pixel is convolved with $G_x$ and $G_y$, then its magnitude summed, to produce the final output pixel.

The `main` function is responsible for verifying the functionality of the `sobel_filter` function. The grayscale (8-bit) input image is stored in the 512x512 array `elaine_512_input` defined in input.h and the expected output image is stored in `elaine_512_golden_output` defined in output.h. The `main` function passes the input image to the `sobel_filter` function and prints "PASS!" if the computed output image matches the expected output image.

Before compiling to hardware, we should verify that the C++ program is correct by compiling and running the software. This is typical of HLS design, where the designer will verify that the design is functionally correct in software before compiling it to hardware.

Click on the *Compile Software* icon in the toolbar (Figure 10). This compiles the software with the GCC compiler. You will see the output from the compilation appearing in the bottom of the screen in the *Console* window of the IDE.

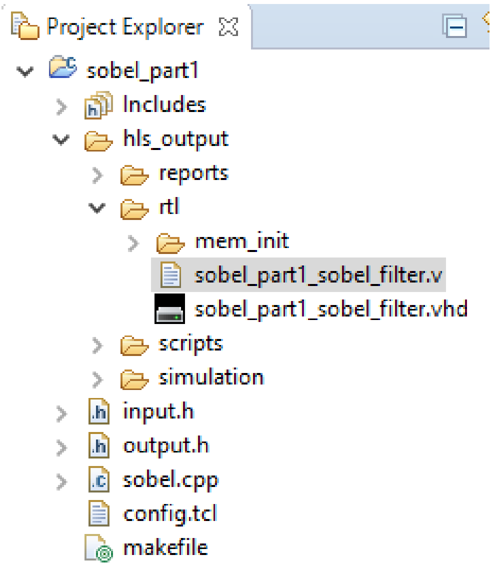
Now, execute the compiled software by clicking on the *Run Software* icon in the toolbar. You should see the message *PASS!* appearing in the *Console* window, as shown in Figure 12.



```
CDT Build Console [sobel_part1]
12:05:30 **** Incremental Build of configuration LegUp for project sobel_part1 ****
"C:\\Microchip\\Libero_SoC_v2022.2\\SmartHLS-2022.2\\cygwin64\\bin\\bash.exe" "C:\\Microchip\\Libero_SoC_v2022.2\\SmartHLS-2022.2\\SmartHLS\\bin\\shls" -s sw_run
Running the following targets: sw_run
PASS!

12:05:33 Build Finished (took 2s.343ms)
```

Figure 12: Console after running software execution.

Now we can compile the Sobel filter C++ software into hardware using SmartHLS by clicking on the toolbar icon to *Compile Software to Hardware*. This command invokes SmartHLS to compile the top-level `sobel_filter` function into hardware. If the top-level function calls descendant functions, all descendant functions are also compiled to hardware. You can find the generated Verilog code in sobel_part1_sobel_filter.v as shown in
Figure 13.



```
Project Explorer
sobel_part1
  Includes
  hls_output
    reports
    rtl
      mem_init
      sobel_part1_sobel_filter.v
      sobel_part1_sobel_filter.vhd
    scripts
    simulation
  input.h
  output.h
  sobel.cpp
  config.tcl
  makefile
```

Figure 13: Finding the SmartHLS-generated Verilog in the Project Explorer.

When the compilation finishes, a SmartHLS report file (summary.hls.sobel_filter.rpt) opens. The report shows the RTL interface of the top-level module corresponding to the top-level C++ function, the number of cycles scheduled for each basic block of the function as well as the memories that are used in the hardware. In this example, you will see the top-level RTL module has three interfaces, the standard *Control* interface that is used by any SmartHLS-generated circuit, and two *Memory* interfaces corresponding to the input and output array arguments of the top-level `sobel_filter` function. In the Memory Usage section of the report, there are no memories inside the generated hardware, as the input and output arrays are passed in as arguments into the top-level function. These input/output function arguments are listed as "I/O Memories" table.

You can visualize the schedule and control-flow of the hardware using the SmartHLS schedule viewer. Start the schedule viewer by clicking on the *Launch Schedule Viewer* icon in the toolbar (see Figure 10). In the left panel of the schedule viewer, you will see the names of the functions and basic blocks of each function. In this example, there is only one function that was compiled to hardware, `sobel_filter`. In the Explorer pane on the left, we see the `sobel_filter` function and 8 basic blocks within the function that are all prefixed by “`BB_`”.

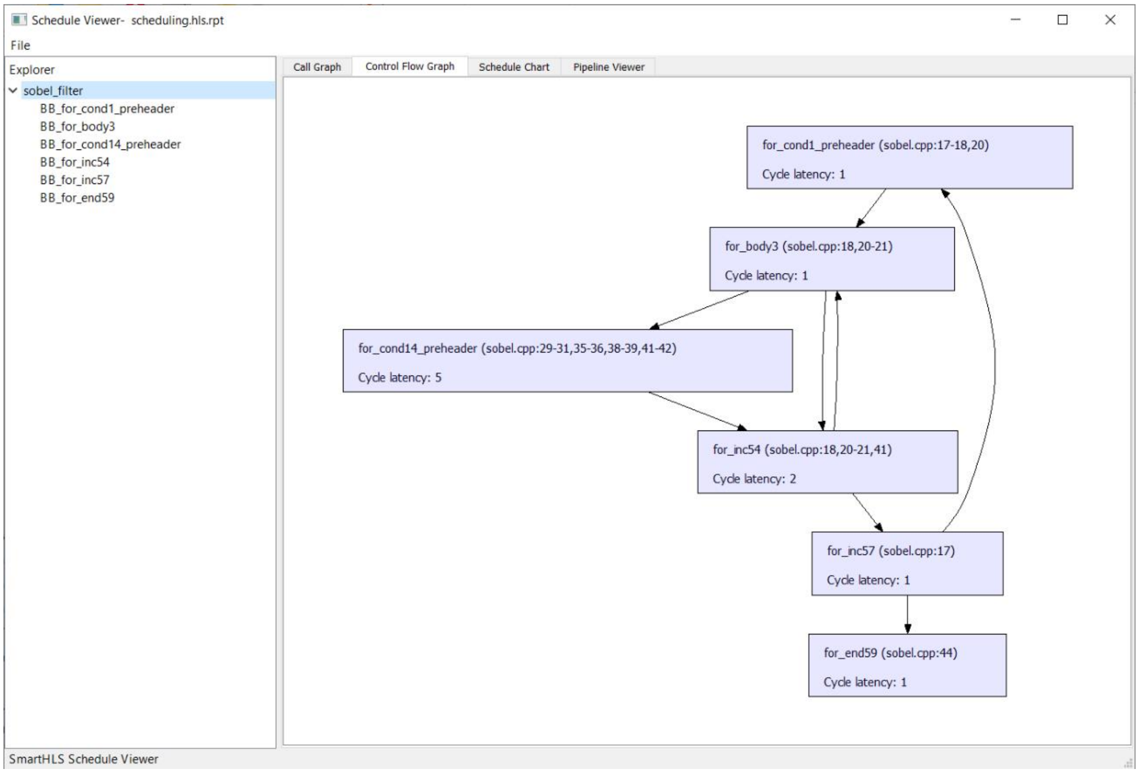
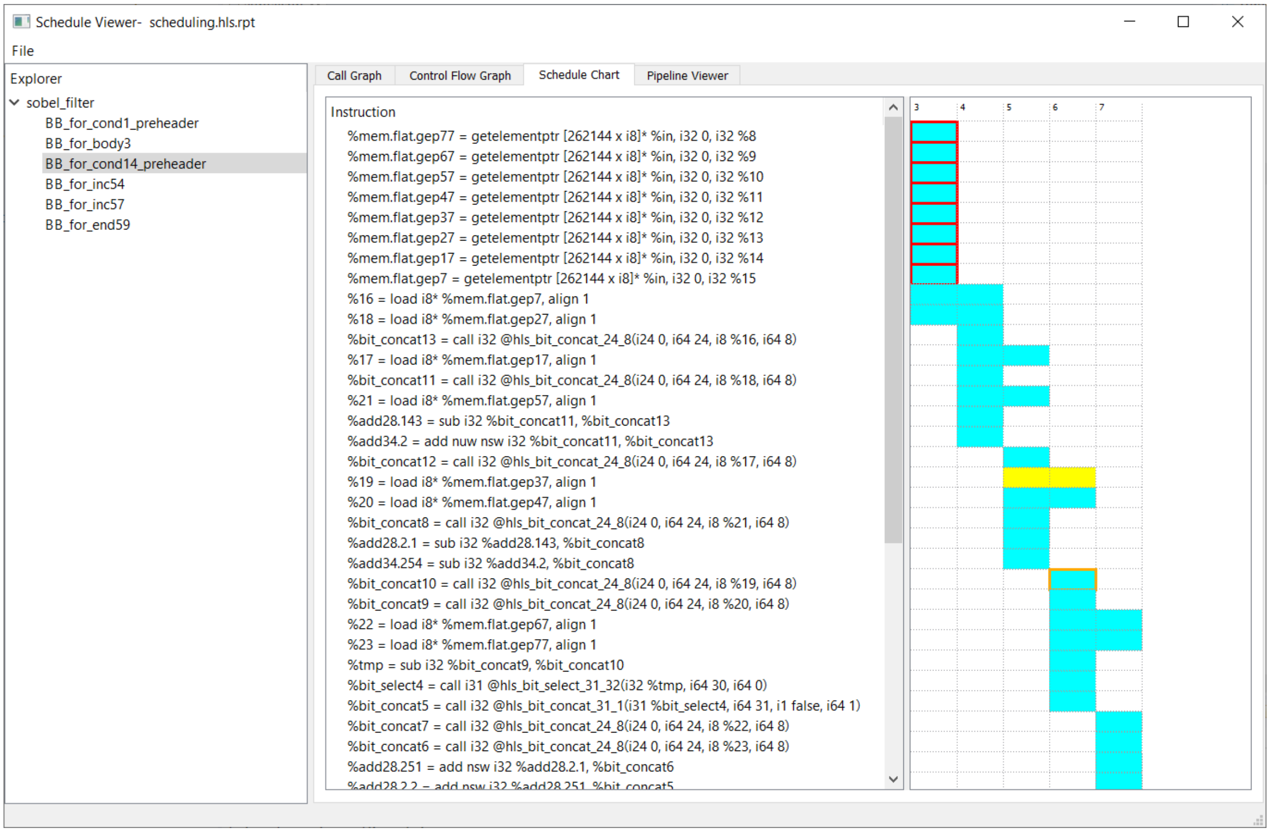


Figure 14: Control-Flow Graph for the Sobel filter.

Double-click on the `sobel_filter` function in the call-graph pane and you will see the control-flow graph for the function, similar to Figure 14. The names of the basic blocks in the program are prefixed with “`BB_`”, which is omitted in the control flow graph. Note that the basic block names may be slightly different depending on the version of SmartHLS you use. The basic block names are not easy to relate back to the original C++ code; however, you can observe that there are two loops in the control-flow graph, which correspond to the two outermost loops in the C++ code for the `sobel_filter` function. The inner loop contains basic blocks: `for_body3`, `for_cond14_preheader`, `for_body3_for_inc54_crit_edge`, and `for_inc54`. Try double clicking on `for_cond14_preheader` (if the basic block names are different from the figure, click on the left-most basic block).

Figure 15 shows the schedule for `BB_for_cond14_preheader`, which is the main part of the inner-most loop body. The middle panel shows the names of the instructions. The right-most panel shows how the instructions are scheduled into states (the figure shows that states 6 to 14 are scheduled for this basic block). Hold your mouse over top of some of the blue boxes in the schedule: you will see the inputs of the current instruction become red and outputs become orange. Look closely at the names of the instructions and try to connect the computations with those in the original C++ program. You will see that there are some loads, additions, subtractions, and shifts. After you are finished close the schedule viewer (File -> Exit).



Instruction

```
%mem.flat.gep77 = getelementptr [262144 x i8]* %in, i32 0, i32 %8
%mem.flat.gep67 = getelementptr [262144 x i8]* %in, i32 0, i32 %9
%mem.flat.gep57 = getelementptr [262144 x i8]* %in, i32 0, i32 %10
%mem.flat.gep47 = getelementptr [262144 x i8]* %in, i32 0, i32 %11
%mem.flat.gep37 = getelementptr [262144 x i8]* %in, i32 0, i32 %12
%mem.flat.gep27 = getelementptr [262144 x i8]* %in, i32 0, i32 %13
%mem.flat.gep17 = getelementptr [262144 x i8]* %in, i32 0, i32 %14
%mem.flat.gep7 = getelementptr [262144 x i8]* %in, i32 0, i32 %15
%16 = load i8* %mem.flat.gep7, align 1
%18 = load i8* %mem.flat.gep27, align 1
%bit_concat13 = call i32 @hls_bit_concat_24_8(i24 0, i64 24, i8 %16, i64 8)
%17 = load i8* %mem.flat.gep17, align 1
%bit_concat11 = call i32 @hls_bit_concat_24_8(i24 0, i64 24, i8 %18, i64 8)
%21 = load i8* %mem.flat.gep57, align 1
%add28.143 = sub i32 %bit_concat11, %bit_concat13
%add34.2 = add nuw nsw i32 %bit_concat11, %bit_concat13
%bit_concat12 = call i32 @hls_bit_concat_24_8(i24 0, i64 24, i8 %17, i64 8)
%19 = load i8* %mem.flat.gep37, align 1
%20 = load i8* %mem.flat.gep47, align 1
%bit_concat8 = call i32 @hls_bit_concat_24_8(i24 0, i64 24, i8 %21, i64 8)
%add28.2.1 = sub i32 %add28.143, %bit_concat8
%add34.254 = sub i32 %add34.2, %bit_concat8
%bit_concat10 = call i32 @hls_bit_concat_24_8(i24 0, i64 24, i8 %19, i64 8)
%bit_concat9 = call i32 @hls_bit_concat_24_8(i24 0, i64 24, i8 %20, i64 8)
%22 = load i8* %mem.flat.gep67, align 1
%23 = load i8* %mem.flat.gep77, align 1
%tmp = sub i32 %bit_concat9, %bit_concat10
%bit_select4 = call i31 @hls_bit_select_31_32(i32 %tmp, i64 30, i64 0)
%bit_concat5 = call i32 @hls_bit_concat_31_1(i31 %bit_select4, i64 31, i1 false, i64 1)
%bit_concat7 = call i32 @hls_bit_concat_24_8(i24 0, i64 24, i8 %22, i64 8)
%bit_concat6 = call i32 @hls_bit_concat_24_8(i24 0, i64 24, i8 %23, i64 8)
%add28.251 = add nsw i32 %add28.2.1, %bit_concat6
%add28.2.2 = add nsw i32 %add28.251, %bit_concat5
```

Figure 15: Schedule for the inner-most loop.

Now we can simulate the Verilog RTL hardware with ModelSim to find out the number of cycles needed to execute the circuit – the cycle latency. Close the schedule viewer first, then click the *SW/HW Co-Simulation* icon in the toolbar. SW/HW co-simulation will simulate the generated Verilog module, `sobel_filter_top`, in RTL using ModelSim, while running the rest of the program, `main`, in software. The co-simulation flow allows us to simulate and verify the SmartHLS-generated hardware without writing a custom RTL testbench.

In the *Console* window, you will see various messages printed by ModelSim related to loading simulation models for the hardware. The hardware may take a few minutes to simulate. We want to focus on the messages near the end of the simulation which will look like this:

```
...
# Cycle latency:     2087960
# ** Note: $finish    : ../simulation/cosim_tb.sv(325)
#    Time: 41759410 ns  Iteration: 1  Instance: /cosim_tb
# End time: 15:02:45 on Feb 28,2023, Elapsed time: 0:00:23
# Errors: 0, Warnings: 0
...
Info: Verifying RTL simulation
...
Retrieving hardware outputs from RTL simulation for sobel_filter
function call 1.
PASS!
...
Number of calls:            1
Cycle latency:     2,087,960
SW/HW co-simulation: PASS
```

We see that the co-simulation took 2,087,960 clock cycles to finish. The simulation printed "`SW/HW co-simulation: PASS!`" which indicates that the RTL generated by SmartHLS matches the software model.

The co-simulation flow uses the return value from the `main` software function to determine whether the co-simulation has passed. If the `main` function returns 0 then the co-simulation will PASS otherwise a non-zero return value will FAIL. Please make sure that your `main` function always follows this convention and returns 0 if the top-level function tests are all successful.

In the `main` function of sobel_part1, also called the software testbench, after calling the top-level function, we iterate over every pixel of the computed output image and verify the pixel against the expected value. A mismatch counter is incremented if a pixel is not as expected and this counter value is returned by the `main` function. If all values match, then the `main` function will return 0. Therefore, since co-simulation printed PASS (`main` returned 0) we have verified the generated hardware is correct.

We can also run co-simulation and launch Modelsim to show the Waveforms. From the *SmartHLS* top menu, select *SW/HW Co-Simulation with Waveforms* as shown in Figure 16.

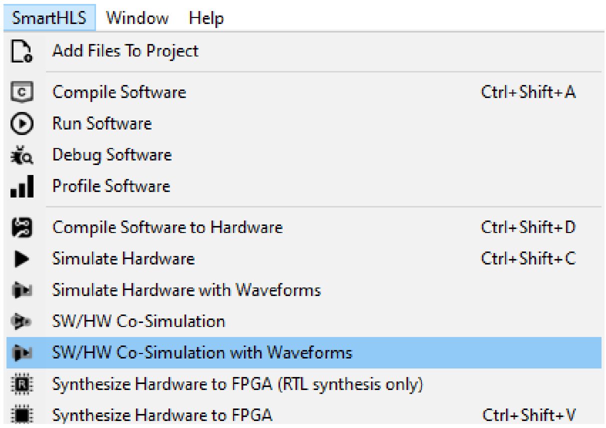

SmartHLS
Window
Help
Add Files To Project
Compile Software Ctrl+Shift+A
Run Software
Debug Software
Profile Software
Compile Software to Hardware Ctrl+Shift+D
Simulate Hardware Ctrl+Shift+C
Simulate Hardware with Waveforms
SW/HW Co-Simulation
SW/HW Co-Simulation with Waveforms
Synthesize Hardware to FPGA (RTL synthesis only)
Synthesize Hardware to FPGA Ctrl+Shift+V

Figure 16: Run SW/HW Co-Simulation with Waveforms

When Modelsim opens it will prompt “Are you sure you want to finish?”. Select “No”. Then you can view the signal waveforms as shown in Figure 17. After you are finished close Modelsim (File -> Quit).

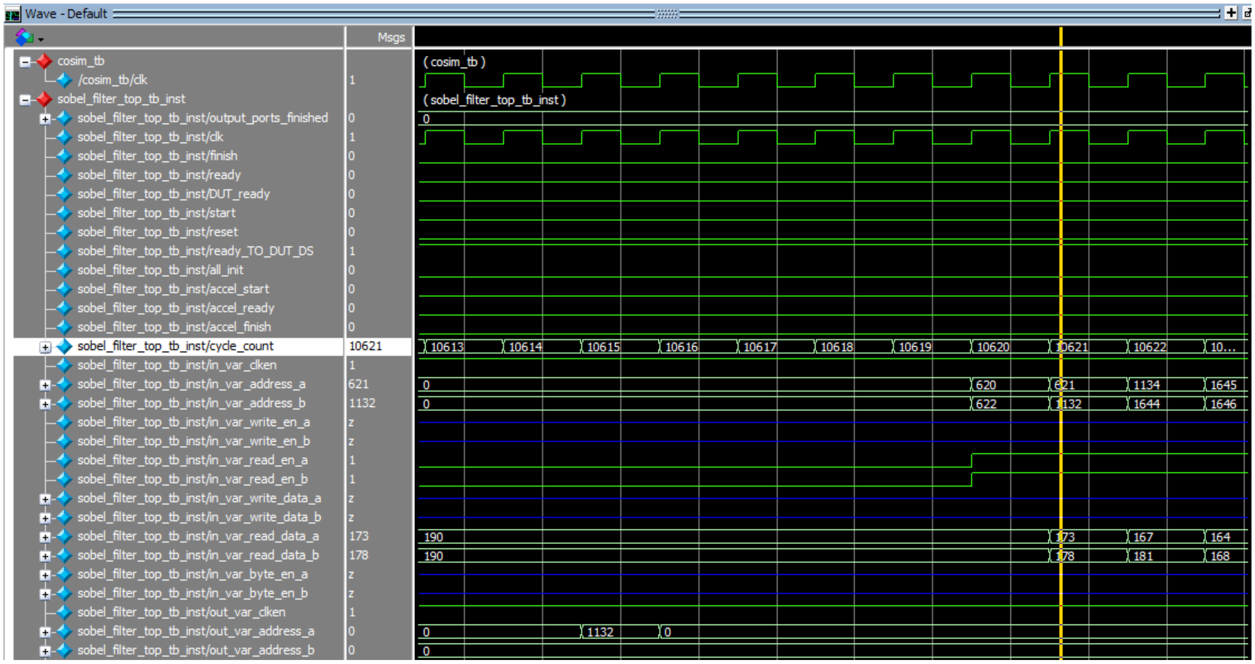

Wave - Default
Msgs
cosim_tb
/cosim_tb/clk 1
sobel_filter_top_tb_inst
sobel_filter_top_tb_inst/output_ports_finished 0
sobel_filter_top_tb_inst/clk 1
sobel_filter_top_tb_inst/finish 0
sobel_filter_top_tb_inst/ready 0
sobel_filter_top_tb_inst/DUT_ready 0
sobel_filter_top_tb_inst/start 0
sobel_filter_top_tb_inst/reset 0
sobel_filter_top_tb_inst/ready_TO_DUT_DS 1
sobel_filter_top_tb_inst/all_init 0
sobel_filter_top_tb_inst/accel_start 0
sobel_filter_top_tb_inst/accel_ready 0
sobel_filter_top_tb_inst/accel_finish 0
sobel_filter_top_tb_inst/cycle_count 10621
sobel_filter_top_tb_inst/in_var_clken 1
sobel_filter_top_tb_inst/in_var_address_a 621
sobel_filter_top_tb_inst/in_var_address_b 1132
sobel_filter_top_tb_inst/in_var_write_en_a z
sobel_filter_top_tb_inst/in_var_write_en_b z
sobel_filter_top_tb_inst/in_var_read_en_a 1
sobel_filter_top_tb_inst/in_var_read_en_b 1
sobel_filter_top_tb_inst/in_var_write_data_a z
sobel_filter_top_tb_inst/in_var_write_data_b z
sobel_filter_top_tb_inst/in_var_read_data_a 173
sobel_filter_top_tb_inst/in_var_read_data_b 178
sobel_filter_top_tb_inst/in_var_byte_en_a z
sobel_filter_top_tb_inst/in_var_byte_en_b z
sobel_filter_top_tb_inst/out_var_clken 1
sobel_filter_top_tb_inst/out_var_address_a 0
sobel_filter_top_tb_inst/out_var_address_b 0
(cosim_tb)
(sobel_filter_top_tb_inst)
0
10613 10614 10615 10616 10617 10618 10619 10620 10621 10622 10...
0 620 621 1134 1645
0 622 1132 1644 1646
190 173 167 164
190 178 181 168
0 1132 0
0

Figure 17: Modelsim waveforms shown during SW/HW Co-Simulation

Libero® is the name of Microsemi's synthesis, placement, routing, and timing analysis tool. SmartHLS can execute Libero® to synthesize, place and route the Verilog to the Microsemi PolarFire® FPGA to obtain information such as the resource usage and the Fmax of this design (i.e. the clock period).

Click the icon on the toolbar to *Synthesize Hardware to FPGA*. SmartHLS will automatically invoke Libero® to create a Libero® project and synthesize the SmartHLS design targeting the PolarFire® FPGA device. Libero® may take a while to finish.

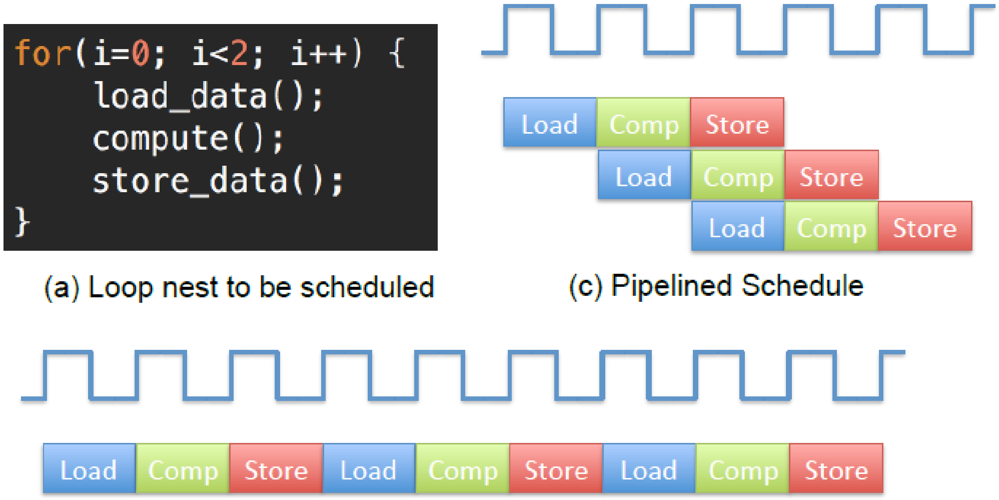
Once the command completes, SmartHLS will open the summary.results.rpt report file. SmartHLS will summarize the resource usage and Fmax results reported by Libero® after place and route. You should get similar results as what is shown below. Your numbers may differ slightly, depending on the version of SmartHLS and Libero® you are using. This tutorial used Libero® SoC v2022.3. The timing results and resource usage might also differ depending on the random seed used in the synthesis tool flow.

```
====== 2. Timing Result ======

+--------------+---------------+-------------+-------------+----------+-------------+
| Clock Domain | Target Period | Target Fmax | Worst Slack | Period   | Fmax        |
+--------------+---------------+-------------+-------------+----------+-------------+
| clk          | 10.000 ns     | 100.000 MHz | 6.822 ns    | 3.178 ns | 314.663 MHz |
+--------------+---------------+-------------+-------------+----------+-------------+

The reported Fmax is for the HLS core in isolation (from Libero's post-place-and-route
timing analysis).
When the HLS core is integrated into a larger system, the system Fmax may be lower
depending on the critical path of the system.

====== 3. Resource Usage ======

+--------------------------+---------------+--------+------------+
| Resource Type            | Used          | Total  | Percentage |
+--------------------------+---------------+--------+------------+
| Fabric + Interface 4LUT* | 680 + 0 = 680 | 108600 | 0.63       |
| Fabric + Interface DFF*  | 425 + 0 = 425 | 108600 | 0.39       |
| I/O Register             | 0             | 852    | 0.00       |
| User I/O                 | 0             | 284    | 0.00       |
| uSRAM                    | 0             | 1008   | 0.00       |
| LSRAM                    | 0             | 352    | 0.00       |
| Math                     | 0             | 336    | 0.00       |
+--------------------------+---------------+--------+------------+

* Interface 4LUTs and DFFs are occupied due to the uses of LSRAM, Math, and uSRAM.
  Number of interface 4LUTs/DFFs = (36 * #.LSRAM) + (36 * #.Math) + (12 * #.uSRAM) =
(36 * 0) + (36 * 0) + (12 * 0) = 0.
```

Wall-clock time is one of the key performance metrics for an FPGA design, computed as the product of the cycle latency and the clock period. In this case, our cycle latency was 2,087,960 and the clock period was 3.178 ns. The wall-clock time of our implementation is therefore 2,087,960 × 3.178 ns = 6.636 ms.

Now close the project by right clicking on the “sobel_part1” folder in the Project Explorer pane and click “Close Project”.

# 5 Part 2: Loop Pipelining



```
for(i=0; i<2; i++) {
    load_data();
    compute();
    store_data();
}
```

(a) Loop nest to be scheduled

(c) Pipelined Schedule

(b) Sequential Schedule

Figure 18: Loop pipelining Example.

In this section, you will use loop pipelining to improve the throughput of the hardware generated by SmartHLS. Loop pipelining allows a new iteration of the loop to be started before the current iteration has finished. By allowing the execution of the loop iterations to be overlapped, higher throughput can be achieved. The amount of overlap is controlled by the initiation interval (II). The II indicates how many cycles are required before starting the next loop iteration. Thus, an II of 1 means a new loop iteration can be started every clock cycle, which is the best we can achieve. The II needs to be larger than 1 in other cases, such as when there is a resource contention (multiple loop iterations need the same resource in the same clock cycle) or when there are loop-carried dependencies (the output of a previous iteration is needed as an input to the subsequent iteration). Resource contention commonly happens with memory accesses to dual-port block RAMs which can only perform two memory accesses per cycle.

Figure 18 shows an example of loop pipelining. Figure 18(b) shows the sequential loop, where a new loop iteration can start every 3 clock cycles (II=3), and the loop takes 9 clock cycles to finish the final write. Figure 18(c) shows the pipelined loop. In this example, there are no resource contentions or data dependencies. Therefore, the pipelined loop can start a new iteration every clock cycle (II=1) and takes only 5 clock cycles to finish the final write. As shown in this example, loop pipelining can significantly improve the performance of your circuit, especially when there are no data dependencies or resource contentions.

Follow the same procedure you used in the previous part of this tutorial (include the source files for part 2 and target MPF100T-FCVG484I), create a new SmartHLS project for part 2. Once the project is created, open the part 2 source file sobel.cpp.

With the Sobel filter, since each pixel of the output is dependent only on the input image and the constant matrices $G_x$ and $G_y$, we would like to pipeline the calculation of each pixel. The loop pipeline pragma in front of the loop, "`#pragma HLS loop pipeline`", tells SmartHLS to pipeline the loop and take advantage of loop parallelism:

```
#pragma HLS loop pipeline
        for (int i = 0; i < (HEIGHT - 2) * (WIDTH - 2); i++) {
```

You will notice that the pair of nested loops in part 1 (previously using i and j) have been flattened into one for loop. We have flattened the nested loops to allow us to apply loop pipelining on the entire loop body. Otherwise, when we apply loop pipelining on a nested outer loop, SmartHLS will automatically unroll the inner loops, which would not be possible with 512 iterations. See more details in the Appendix: Loop Pipelining in Part 1 vs Part 2.

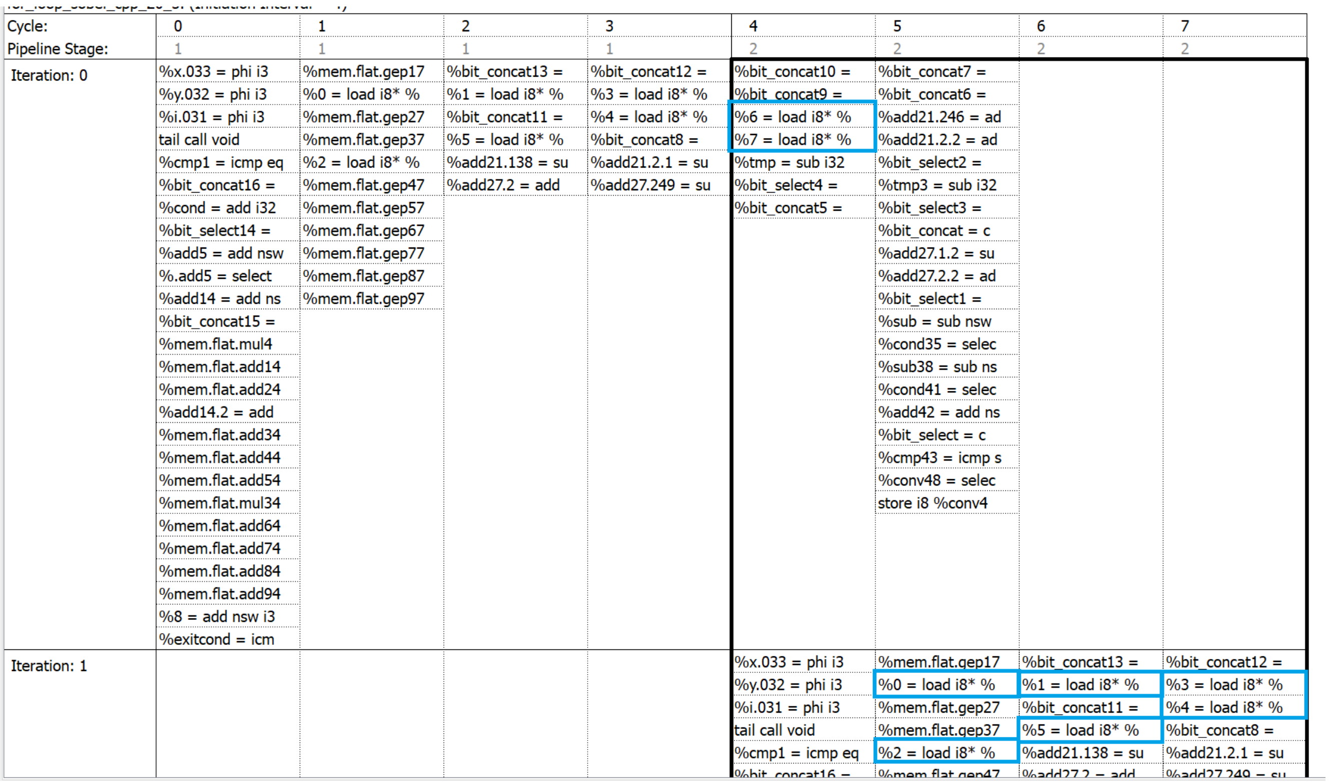
Now you can synthesize the design by clicking the *Compile Software to Hardware* icon in the toolbar. In the *Console* window, you should see messages like the following:

```
Info: Resource constraint limits initiation interval to 4.
      Resource 'in_external_memory_port' has 8 uses per cycle but only 2 units available.
      +---------------------------------------+----------------------------+----------------------+
      | Operation                             | Location                   | Competing Use Count |
      +---------------------------------------+----------------------------+----------------------+
      | 'load' (8b) operation for array 'in' | line 30 of sobel.cpp       | 1                    |
      | 'load' (8b) operation for array 'in' | line 30 of sobel.cpp       | 2                    |
      | 'load' (8b) operation for array 'in' | line 30 of sobel.cpp       | 3                    |
      | 'load' (8b) operation for array 'in' | line 30 of sobel.cpp       | 4                    |
      | 'load' (8b) operation for array 'in' | line 30 of sobel.cpp       | 5                    |
      | 'load' (8b) operation for array 'in' | line 30 of sobel.cpp       | 6                    |
      | 'load' (8b) operation for array 'in' | line 30 of sobel.cpp       | 7                    |
      | 'load' (8b) operation for array 'in' | line 30 of sobel.cpp       | 8                    |
      +---------------------------------------+----------------------------+----------------------+
      |                                       | Total # of Competing Uses | 8                    |
      +---------------------------------------+----------------------------+----------------------+
```

These messages indicate that SmartHLS cannot achieve an II of 1 (highest throughput) due to resource conflicts – there are 8 loads from the same RAM memory in the loop body. Since RAM blocks are dual ported on an FPGA, we need 4 cycles to perform 8 loads. Therefore, SmartHLS needs to schedule 4 cycles between successive iterations of the loop (an initiation interval of 4). Note that there are 8 loads from 'in', not 9, despite the receptive field being of size 3x3. This is because `gx[1][1]` and `gy[1][1] = 0`, so the multiplication and corresponding load are optimized away.

We can visualize the 8 memory loads in the pipeline using the SmartHLS schedule viewer. Click on the *Launch Schedule Viewer* icon 🥧. Double-click on `sobel_filter`, then in the Control Flow Graph, you will see a basic block called `BB_for_body`. Double-click `BB_For_body` to reveal the loop pipeline schedule, similar to that shown in Figure 19: Loop Pipeline Schedule for the Sobel Filter.. Horizontally from left to right shows the operations performed on successive clock cycles and vertically going down shows successive loop iterations. Here, you can see that the II of the loop is 4 and that a new loop iteration starts every 4 cycles. You can also see the instructions that are scheduled in each cycle for each loop iteration. All instructions that are shown in the same column are executed in the same cycle.

Now scroll to the far right in the schedule viewer. The dark black rectangle on the far right illustrates what the pipeline looks like in steady state. In steady state, two iterations of the loop are "in flight" at once. In steady state, you can see that there are two loads in cycle 4 from iteration 0 (the first row), and two loads each from cycle 5 to 7 from iteration 1. Thus the 8 loads are spread out over 4 cycles, making the Initiation Interval = 4.



| Cycle: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Pipeline Stage: | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| Iteration: 0 | %x.033 = phi i3 | %mem.flat.gep17 | %bit_concat13 = | %bit_concat12 = | %bit_concat10 = | %bit_concat7 = | | |
| | %y.032 = phi i3 | %0 = load i8* % | %1 = load i8* % | %3 = load i8* % | %bit_concat9 = | %bit_concat6 = | | |
| | %i.031 = phi i3 | %mem.flat.gep27 | %bit_concat11 = | %4 = load i8* % | %6 = load i8* % | %add21.246 = ad | | |
| | tail call void | %mem.flat.gep37 | %5 = load i8* % | %bit_concat8 = | %7 = load i8* % | %add21.2.2 = ad | | |
| | %cmp1 = icmp eq | %2 = load i8* % | %add21.138 = su | %add21.2.1 = su | %tmp = sub i32 | %bit_select2 = | | |
| | %bit_concat16 = | %mem.flat.gep47 | %add27.2 = add | %add27.249 = su | %bit_select4 = | %tmp3 = sub i32 | | |
| | %cond = add i32 | %mem.flat.gep57 | | | %bit_concat5 = | %bit_select3 = | | |
| | %bit_select14 = | %mem.flat.gep67 | | | | %bit_concat = c | | |
| | %add5 = add nsw | %mem.flat.gep77 | | | | %add27.1.2 = su | | |
| | %.add5 = select | %mem.flat.gep87 | | | | %add27.2.2 = ad | | |
| | %add14 = add ns | %mem.flat.gep97 | | | | %bit_select1 = | | |
| | %bit_concat15 = | | | | | %sub = sub nsw | | |
| | %mem.flat.mul4 | | | | | %cond35 = selec | | |
| | %mem.flat.add14 | | | | | %sub38 = sub ns | | |
| | %mem.flat.add24 | | | | | %cond41 = selec | | |
| | %add14.2 = add | | | | | %add42 = add ns | | |
| | %mem.flat.add34 | | | | | %bit_select = c | | |
| | %mem.flat.add44 | | | | | %cmp43 = icmp s | | |
| | %mem.flat.add54 | | | | | %conv48 = selec | | |
| | %mem.flat.mul34 | | | | | store i8 %conv4 | | |
| | %mem.flat.add64 | | | | | | | |
| | %mem.flat.add74 | | | | | | | |
| | %mem.flat.add84 | | | | | | | |
| | %mem.flat.add94 | | | | | | | |
| | %8 = add nsw i3 | | | | | | | |
| | %exitcond = icm | | | | | | | |
| Iteration: 1 | | | | | %x.033 = phi i3 | %mem.flat.gep17 | %bit_concat13 = | %bit_concat12 = |
| | | | | | %y.032 = phi i3 | %0 = load i8* % | %1 = load i8* % | %3 = load i8* % |
| | | | | | %i.031 = phi i3 | %mem.flat.gep27 | %bit_concat11 = | %4 = load i8* % |
| | | | | | tail call void | %mem.flat.gep37 | %5 = load i8* % | %bit_concat8 = |
| | | | | | %cmp1 = icmp eq | %2 = load i8* % | %add21.138 = su | %add21.2.1 = su |
| | | | | | %bit_concat16 = | %mem.flat.gep47 | %add27.2 = add | %add27.249 = su |

Figure 19: Loop Pipeline Schedule for the Sobel Filter.

Now, exit the schedule viewer and simulate the design in ModelSim by clicking the *SW/HW Co-Simulation* icon on the toolbar. You should see Console output similar to the following:

```
...
# Cycle latency:     1040409
# ** Note: $finish    : cosim_tb.sv(325)
#    Time: 20808390 ns  Iteration: 1  Instance: /cosim_tb
# End time: 11:58:20 on Feb 28,2023, Elapsed time: 0:00:18
# Errors: 0, Warnings: 0
...
Retrieving hardware outputs from RTL simulation for sobel_filter
function call 1.
PASS!
...
Number of calls: 1
Cycle latency: 1,040,409
SW/HW co-simulation: PASS
```

Observe that loop pipelining has dramatically improved the cycle latency for the design, reducing it from 2,087,960 cycles to 1,040,409 cycles in total.

Finally, use Microsemi’s Libero® to map the design onto the PolarFire® FPGA by clicking the *Synthesize Hardware to FPGA* icon on the toolbar. Once the synthesis run finishes, examine the FPGA speed (FMax) and area data from the summary.results.rpt report file. You should see results similar to the following:

```
====== 2. Timing Result ======

+--------------+---------------+-------------+-------------+----------+-------------+
| Clock Domain | Target Period | Target Fmax | Worst Slack | Period   | Fmax        |
+--------------+---------------+-------------+-------------+----------+-------------+
| clk          | 10.000 ns     | 100.000 MHz | 6.829 ns    | 3.171 ns | 315.358 MHz |
+--------------+---------------+-------------+-------------+----------+-------------+
...

====== 3. Resource Usage ======

+--------------------------+---------------+--------+------------+
| Resource Type            | Used          | Total  | Percentage |
+--------------------------+---------------+--------+------------+
| Fabric + Interface 4LUT* | 722 + 0 = 722 | 108600 | 0.66       |
| Fabric + Interface DFF*  | 363 + 0 = 363 | 108600 | 0.33       |
| I/O Register             | 0             | 852    | 0.00       |
| User I/O                 | 0             | 284    | 0.00       |
| uSRAM                    | 0             | 1008   | 0.00       |
| LSRAM                    | 0             | 352    | 0.00       |
| Math                     | 0             | 336    | 0.00       |
+--------------------------+---------------+--------+------------+
```

# 6 Part 3: Designing Streaming/Dataflow Hardware

The final hardware implementation we will cover in this tutorial is called a *streaming* implementation (also sometimes called a *dataflow* implementation). Streaming hardware can accept new inputs at a regular initiation interval (II), for example, every cycle. This bears some similarity to the loop pipelining part of the tutorial you completed above. While one set of inputs is being processed by the streaming hardware, new inputs can continue to be injected into the hardware every II cycles.

For example, a streaming hardware module might have a *latency* of 10 clock cycles and an II of 1 cycle. This would mean that, for a given set of inputs, the hardware takes 10 clock cycles to complete its work; however, the hardware can continue to receive new inputs every single cycle. Streaming hardware is thus very similar to a pipelined processor, where multiple different instructions are in flight at once, at intermediate stages of the pipeline. The word "streaming" is used because the generated hardware operates on a continuous stream of input data and produces a stream of output data. Image, audio and video processing are all examples of streaming applications.

In this part of the tutorial, we will synthesize a circuit that accepts a new input pixel of an image every cycle (the input stream) and produces a pixel of the output image every cycle (the output stream). Given this desired behavior, an approach that may spring to your mind is as follows: 1) Read in the entire input image, pixel by pixel. 2) Once the input image is stored, begin computing the Sobel-filtered output image. 3) Output the filtered image, pixel by pixel. While this approach is certainly possible, it suffers from several weaknesses. First, if the input image is 512x512 pixels, then it would take 262,144 cycles to input an image, pixel by pixel. This represents a significant wait before seeing any output. Second, we would need to store the entire input image in memory. Assuming 8-bit pixel values, this would require 262KB of memory. An alternative widely used approach to streaming image processing is to use *line buffers*.

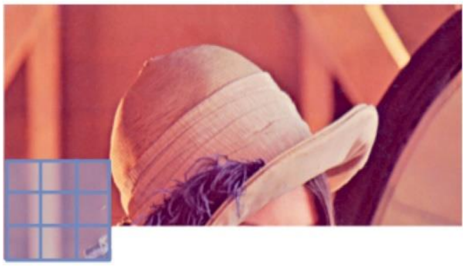


Figure 20: Motivation for use of line buffers.

Figure 20 shows the 3x3 Sobel filter sweeping across an input image. From this figure, we can make a key observation, namely, that to apply the Sobel filter, we do not need the *entire* input image. Rather, we only need to store the previous two rows of the input image, along with a few pixels from the current row being received (bottom row of pixels in the figure). Leveraging this observation, we can drastically reduce the amount of memory required to just two rows of the input image. The memory used to store the two rows are called "line buffers" and they can be efficiently implemented as block RAMs on the FPGA.

Create a new SmartHLS project for part 3 of the tutorial and include all the .cpp and .h files for part 3. Again, specify Microsemi's PolarFire® custom device (MPF100T-FCVG484I) and finish creating the project.

Examine the sobel.cpp file in the project viewer and you will find the following line:

```
static LineBuffer<unsigned char, WIDTH, 3> line_buffer;
```

This statement "instantiates" SmartHLS's LineBuffer template class from the <hls/image_processing.hpp> C++ library to create a line_buffer object. The template parameters specify the desired line buffer configuration: 1) use "unsigned char" 8-bit type to represent pixels, 2) set image width to WIDTH, and 3) set the filter size to 3. Inside the LineBuffer, there are internal arrays for storing the previous rows (two rows when filter size is 3), and an externally accessible 2D array named "`window`" to contain pixels in the current 3x3 receptive field. The line_buffer is declared as static so that its internal state and memory is retained between functions calls.

A few lines below you should see "`line_buffer.ShiftInPixel(input_pixel);`". Each call of the `ShiftInPixel()` function pushes in a new pixel into the line buffer and updates the line buffer's internal previous-row arrays as well as the receptive field window. In the subsequent nested loop, you will see the 3x3 receptive field is accessed by reading the "`window`" array, i.e., `line_buffer.window[m + 1][n + 1]`.

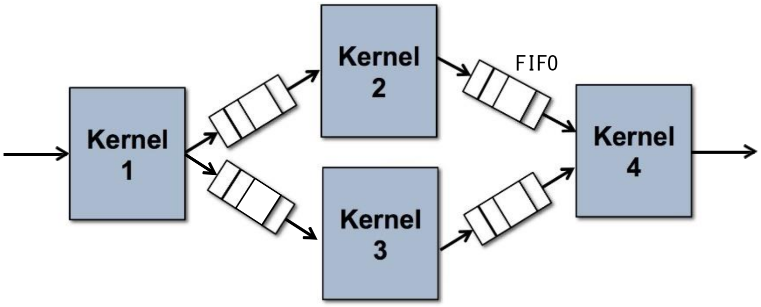


Figure 21: Streaming hardware circuit with FIFO queues between components.

Before going further, we will explain a common feature of streaming hardware called a FIFO (first-in first-out) queue. We use FIFO queues to interconnect the various streaming components, as shown in Figure 22. Here, we see a system with four streaming hardware modules, which are often called *kernels* (not to be confused with the convolutional kernels used in the Sobel filter!). The hardware kernels are connected with FIFO queues in between them. A kernel consumes data from its input FIFO queue(s) and pushes computed data into its output queue(s). If its input queue is empty, the kernel stalls (stops executing). Likewise, if the output queues are full, the unit stalls.

In the example in Figure 22, kernel 4 has two queues on its input, and consequently, kernel 4 commences once a data item is available in both of the queues.

The SmartHLS tool provides an easy-to-use FIFO data structure to interconnect streaming kernels, which is automatically converted into a hardware FIFO during circuit synthesis. Below is a snippet from the `sobel_filter` function in the sobel.cpp file. Observe that the input and output FIFOs are passed by reference to the function. A pixel value is read from the input FIFO via the read() function; later, a pixel is written to the output FIFO through the write() function. These functions are declared in the hls/streaming.hpp header file.

```
void sobel_filter(FIFO<unsigned char> &input_fifo,
                  FIFO<unsigned char> &output_fifo) {
    ...
    unsigned char input_pixel = input_fifo.read();
    ...
    output_fifo.write(outofbounds ? 0 : sum);
    ...
}
```

The rest of the `sobel_filter` function is very similar to the previous parts of this tutorial. An exception relates to the use of static variables so that data can be retained across calls to the function. A count variable tracks the number of times the function has been invoked and this is used to determine if the line buffers have been filled with data. Two static variables, i and j keep track of the row and column of the current input pixel being streamed into the function; this tracking allows the function to determine whether the pixel is out of bounds for the convolution operation (i.e. on the edge of the image).

The `sobel_filter` top-level function has an additional pragma:

```
#pragma HLS function pipeline
```

This pragma tells SmartHLS that the `sobel_filter` function is intended to be a streaming kernel.

In the main function in sobel.cpp, you will see that FIFOs are declared in the beginning. The FIFO class has a template parameter to specify the data type stored inside the FIFO. The FIFO constructor argument specifies the depth (how many elements can be stored). In this case, the FIFOs are declared to have the unsigned char data type to create 8-bit wide FIFOs.

In the main function, we see that the image input data (stored in input.h) is pushed into the input_fifo and the Sobel filter is invoked for *HEIGHT x WIDTH* times. Finally, the output values are checked for correctness and PASS or FAIL is reported. The main function returns 0 if the output values are correct.

Click the icons to compile C and run the software ▶, and you should see the computed and golden pixel values and the message “*PASS!*”.

Now generate the hardware with SmartHLS by clicking the *Compile Software to Hardware* icon . In the report file (summary.hls.sobel_filter.rpt) that opens, you should see the top-level RTL interface now includes an input AXI stream interface and an output AXI stream interface, corresponding to the input_fifo and output_fifo arguments of the top-level function. Under Pipeline Result you can see that the `sobel_filter` function is pipelined and has an initiation interval of 1.

```
====== 3. Pipeline Result ======

+--------------+--------------+-------------+------------------------+---------------------+-----------------+
| Label        | Function     | Basic Block | Location in Source Code | Initiation Interval | Pipeline Length |
+--------------+--------------+-------------+------------------------+---------------------+-----------------+
| sobel_filter | sobel_filter | init.check  | line 12 of sobel.cpp   | 1                   | 4               |
+--------------+--------------+-------------+------------------------+---------------------+-----------------+
```

This circuit has memories inside the hardware (see Local Memories under Memory Usage) due to the line buffers and the counters that are used. You can see there are two RAMs in the circuit both with 4096 bits, corresponding to the two line buffers each storing 512 x 8-bit pixels. Note we have removed other local memories from the report snippet below:

```
+-----------------------------------------------------------------------------------------------------------+
| Local Memories                                                                                            |
+----------------------------------------+---------------------+------+-------------+------------+-------+
| Name                                   | Accessing Function(s) | Type | Size [Bits] | Data Width | Depth |
+----------------------------------------+---------------------+------+-------------+------------+-------+
...                                                 ...                              ...
...                                                 ...                              ...
...                                                 ...                              ...
| sobel_filter_line_buffer_prev_row_a0_a0 | sobel_filter          | RAM  | 4096        | 8          | 512   |
| sobel_filter_line_buffer_prev_row_a1_a0 | sobel_filter          | RAM  | 4096        | 8          | 512   |
```

Now simulate the streaming hardware by clicking the *SW/HW Co-Simulation* icon . You will see scrolling output in the Console window, reporting the computed and expected pixel value at each clock cycle. After a few minutes the co-simulation will finish and in the Console you should see:

```
...
PASS!

...
Number of calls:       262,658
Cycle latency:       262,664
SW/HW co-simulation:  PASS
```

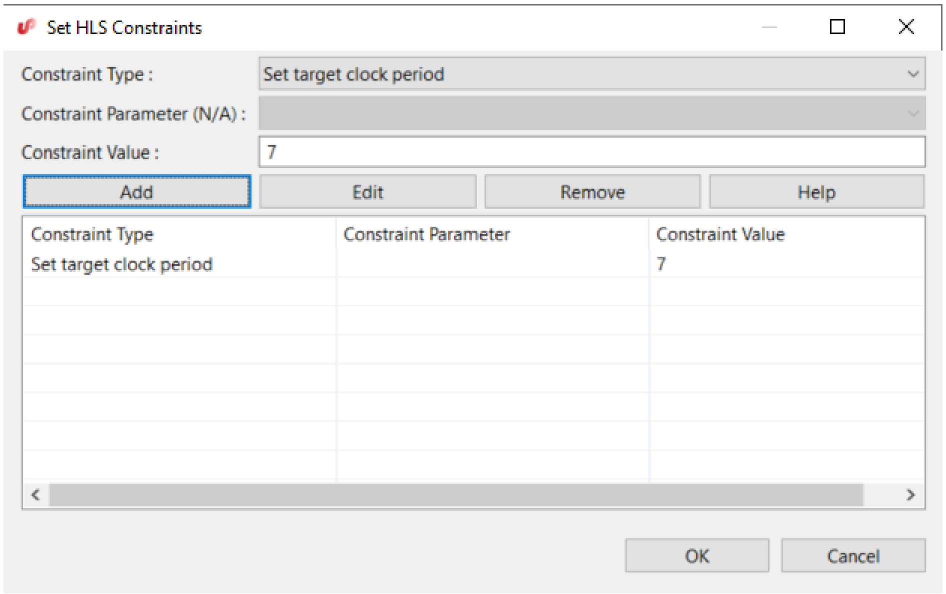
The total number of clock cycles is about 262,664, which is very close to 512 x 512 = 262,144. That is, the number of cycles for the streaming hardware is close to the total number of pixels computed, which confirms that we are processing 1 pixel every clock cycle (Initiation Interval is 1). At the end of the co-simulation, you should see that the co-simulation has passed.

Now, we can synthesize the circuit with Libero® targeting the PolarFire® FPGA by clicking the *Synthesize Hardware to FPGA* icon in the toolbar. You should see results similar to the following in the summary.results.rpt report file:

```
====== 2. Timing Result ======

+--------------+---------------+-------------+-------------+----------+-------------+
| Clock Domain | Target Period | Target Fmax | Worst Slack | Period   | Fmax        |
+--------------+---------------+-------------+-------------+----------+-------------+
| clk          | 10.000 ns     | 100.000 MHz | 4.938 ns    | 5.062 ns | 254.065 MHz |
+--------------+---------------+-------------+-------------+----------+-------------+
...

====== 3. Resource Usage ======

+--------------------------+----------------+--------+------------+
| Resource Type            | Used           | Total  | Percentage |
+--------------------------+----------------+--------+------------+
| Fabric + Interface 4LUT* | 435 + 72 = 497 | 108600 | 0.46       |
| Fabric + Interface DFF*  | 343 + 72 = 415 | 108600 | 0.38       |
| I/O Register             | 0              | 852    | 0.00       |
| User I/O                 | 0              | 284    | 0.00       |
| uSRAM                    | 0              | 1008   | 0.00       |
| LSRAM                    | 2              | 352    | 0.57       |
| Math                     | 0              | 336    | 0.00       |
+--------------------------+----------------+--------+------------+
```

SmartHLS also allows the user to give a target clock period constraint, which the compiler uses to schedule the operations in the program and insert registers so that the generated circuit can be implemented accordingly. It may not always be possible for SmartHLS to meet the user-provided target period precisely, due to the complexity of the circuit or the physical properties of the target FPGA device, but in general, a lower clock period constraint leads to higher Fmax. A lower clock period may cause larger circuit area due to SmartHLS inserting more registers, and a higher clock period constraint leads to lower Fmax but can also have less area.

Open the HLS Constraints dialog by clicking the icon ⚙ where we can change the target clock period constraint. As shown in Figure 22, select “Set target clock period” for *Constraint Type* and set *Constraint Value* to the desired clock period in nanoseconds: “7”. Then you must click the “Add” button. After pressing Add, the constraint will appear in the list of active HLS constraints. Then click OK.



Set HLS Constraints

Constraint Type : Set target clock period

Constraint Parameter (N/A) :

Constraint Value : 7

| Constraint Type | Constraint Parameter | Constraint Value |
|---|---|---|
| Set target clock period |  | 7 |

Figure 22: Setting the target clock period HLS constraint.

If the target clock period constraint is not provided by the user, as in this tutorial, SmartHLS will use the default target clock period constraint that has been set for each target FPGA device. The default clock period constraint is 10 ns for the Microsemi PolarFire® FPGA.

Now that we lowered the clock period constraint to 7 ns, we can recompile software to hardware by clicking the icon . You should see the pipeline length has increased from 4 to 5 cycles in the summary.hls.sobel_filter.rpt report file:

```
====== 3. Pipeline Result ======

+--------------+--------------+------------+-----------------------+---------------------+-----------------+
| Label        | Function     | Basic Block | Location in Source Code | Initiation Interval | Pipeline Length |
+--------------+--------------+------------+-----------------------+---------------------+-----------------+
| sobel_filter | sobel_filter | init.check | line 12 of sobel.cpp  | 1                   | 5               |
+--------------+--------------+------------+-----------------------+---------------------+-----------------+
```

The pipeline length increased because SmartHLS has added additional pipeline stages/registers to achieve the higher target Fmax. You can also synthesize the generated circuit with Libero® again to examine the impact of the clock period constraint on the generated circuit.

# 7 Instantiating SmartHLS IP Core in Libero® SmartDesign

After we use SmartHLS to design a hardware IP component, we will want to instantiate the component into Libero® SmartDesign and integrate this core into our larger system. When SmartHLS generates the hardware, SmartHLS will also generate a `create_hdl_plus.tcl` script to easily instantiate the SmartHLS-generated IP core into Libero® SmartDesign. You will see the Info message in the SmartHLS IDE console window which includes the full path to the script:

```
Info: Generating HDL+ Tcl script to be imported in SmartDesign:
C:\hls_workspace\sobel_part3\hls_output\scripts\create_hdl_plus.tcl.
```

Now open Libero® SoC, create a new Libero® Project by selecting from the top menu: Project -> New Project. Choose any project name and select the target PolarFire® FPGAs. In the *Design Flow* panel of the new Libero® project, create a new SmartDesign by double clicking "Create SmartDesign" as shown in Figure 23 below. Choose any name in the *Create New SmartDesign* dialog.

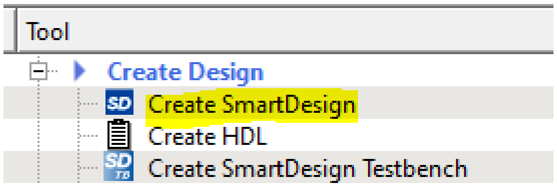


Figure 23 : Create SmartDesign in Libero® SoC.

In the *Design Hierarchy* panel, right click the SmartDesign (SD) icon and click *Set As Root* as shown in Figure 24 below.

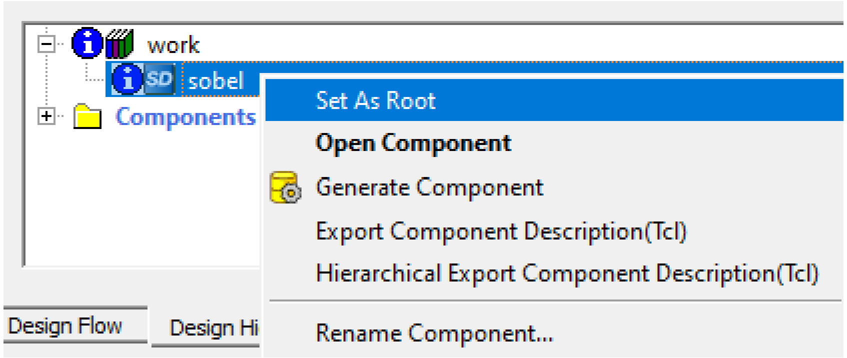
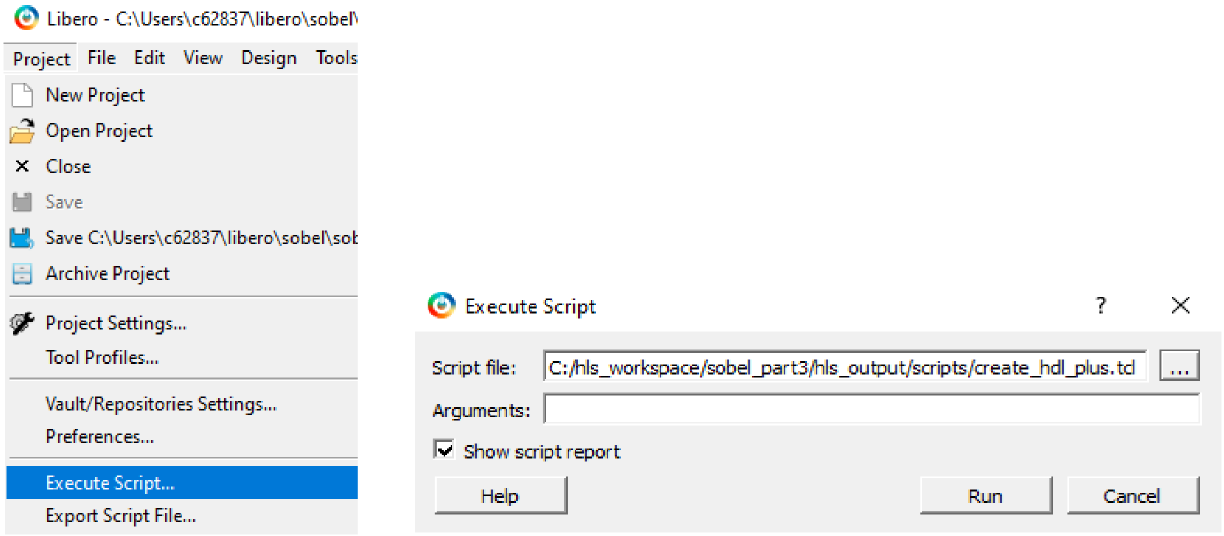


Figure 24: Select the SmartDesign module as root in Libero® SoC.

We now go to the Libero® *Project* menu and select *Execute Script* and give the path to the generated `create_hdl_plus.tcl` script as shown in Figure 25. Then click *Run*.



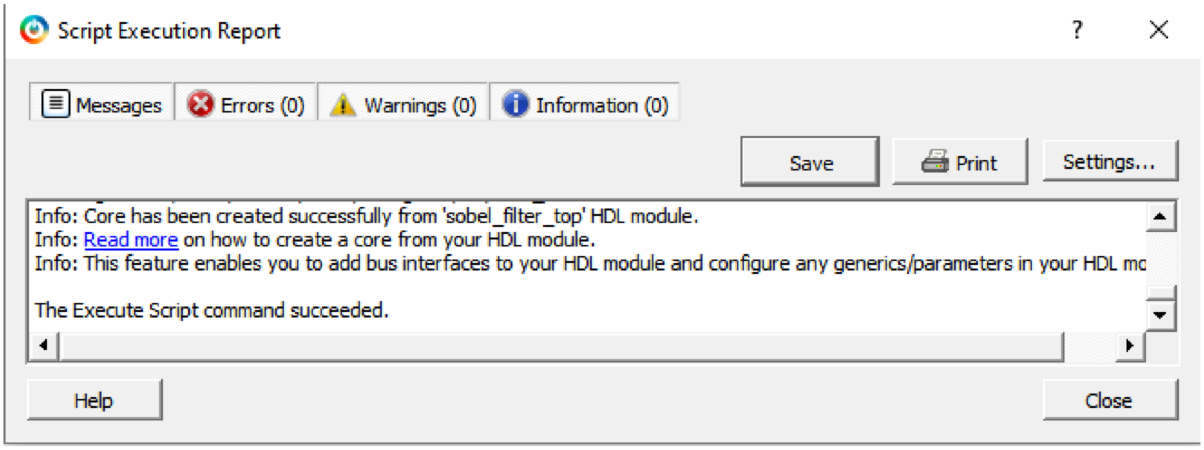
Script file: C:/hls_workspace/sobel_part3/hls_output/scripts/create_hdl_plus.tcl

Arguments:

☑ Show script report

Figure 25: Execute Tcl script to instantiate SmartHLS IP component into SmartDesign.

Running the Tcl script will add the SmartHLS-generated HDL+ component `sobel_filter_top` and all required Verilog files, memory initialization files, and other dependencies to the Libero® project. You should see the Execute Script command succeeded as shown in Figure 26.



```
Info: Core has been created successfully from 'sobel_filter_top' HDL module.
Info: Read more on how to create a core from your HDL module.
Info: This feature enables you to add bus interfaces to your HDL module and configure any generics/parameters in your HDL mo

The Execute Script command succeeded.
```

Figure 26: Script Execution Report after running SmartHLS SmartDesign Tcl script.

As shown in Figure 27, we can now instantiate the component in SmartDesign by right-clicking on the `sobel_filter_top` HDL+ component in the *Design Hiearchy* panel on the left and selecting the *Instantiate in <SmartDesign Name>* (*Instantiate in system* in Figure 27 as we named our SmartDesign *system*). In the SmartDesign system we will now see the new `sobel_filter_top_0` IP component.

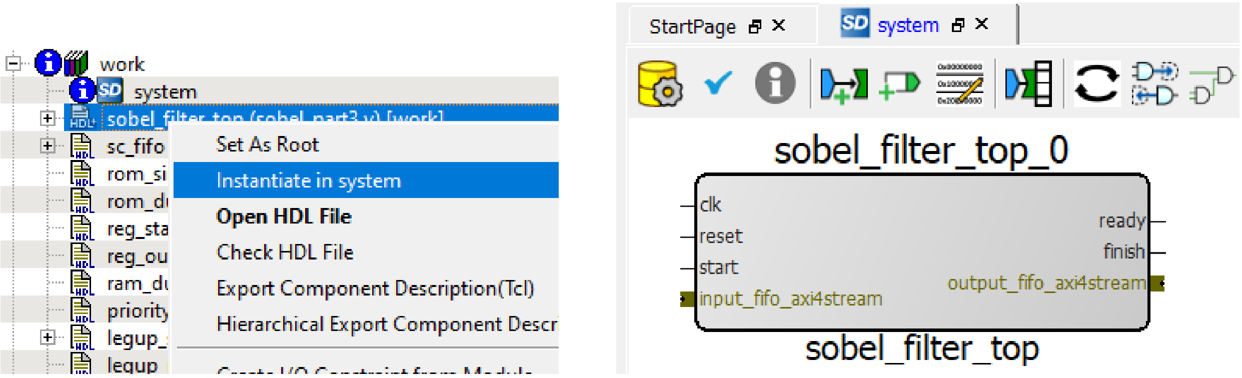
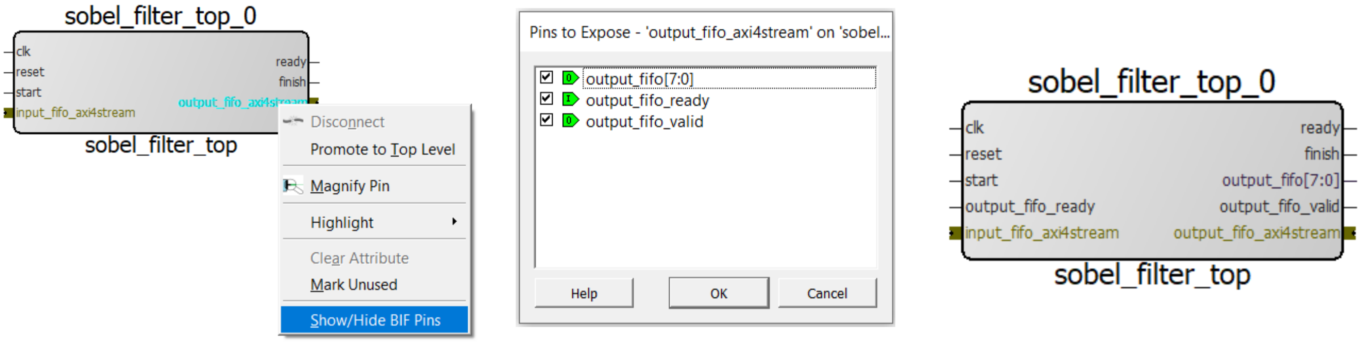


Figure 27: SmartHLS IP Component instantiated inside SmartDesign

In this case, since the `sobel_filter_top` IP component used SmartHLS FIFOs as top-level arguments, SmartHLS has automatically grouped the `output_fifo` and `input_fifo` data/ready/valid ports as AXI4-Stream bus interfaces.

We also have the option to expose the sub-signals under the AXI4-Stream bus. This will allow us to connect individual ports instead of the entire bus. To do this, right click on the AXI4-Stream bus on the SmartHLS-generated IP component and choose *Show/Hide BIF Pins*. Then choose the sub-signals as appropriate.

For example, as shown in Figure 29, we can right click the *output_fifo_axi4stream* bus and choose to *Show/Hide BIF Pins*, then we select all 3 pins and press OK. We will now see that the *sobel_filter_top_0* IP component has an input pin for *output_fifo_ready*, an output pin for *output_fifo[7:0]*, and an output pin for *output_fifo_valid*.



Pins to Expose - 'output_fifo_axi4stream' on 'sobel...

☑ output_fifo[7:0]
☑ output_fifo_ready
☑ output_fifo_valid

Figure 28: Expose the individual pins contained in the *output_fifo* AXI4-Stream bus

# 8 Summary

High-level synthesis allows hardware to be designed at a higher level of abstraction, lowering design time and cost. In this tutorial, you have gained experience with several key high-level synthesis concepts in SmartHLS, including loop pipelining and streaming/function pipelining, as applied to a practical example: edge detection in images. These key techniques can allow you to create a high-performance circuit from software.

For any questions, please contact us at SmartHLS@microchip.com.

# 9 Appendix: Loop Pipelining in Part 1 vs Part 2

In Part 2 of the tutorial, we noted that the nested loops from Part 1 were manually flattened into a single for loop (called “loop flattening”). As shown in sobel.cpp from Part 2 below:

```
#pragma HLS loop pipeline
    for (int i = 0; i < (HEIGHT - 2) * (WIDTH - 2); i++) {
        // increment row when column reaches end of row
        y = (x == WIDTH - 2) ? y + 1 : y;
        // increment column until end of row
        x = (x == WIDTH - 2) ? 1 : x + 1;
```

We flattened the nested loops because SmartHLS does not support loop pipelining nested loops without unrolling the inner loops. For example, if we open sobel.cpp from Part 1 and add a pragma to pipeline the outer loop of the nested loop:

```
#pragma HLS loop pipeline
    for (int i = 0; i < HEIGHT; i++) {
      for (int j = 0; j < WIDTH; j++) {
           // Set output to 0 if the 3x3 receptive field is out of bound.
           if ((i < 1) | (i > HEIGHT - 2) | (j < 1) | (j > WIDTH - 2)) {
               out[i][j] = 0;
               continue;
           }
```

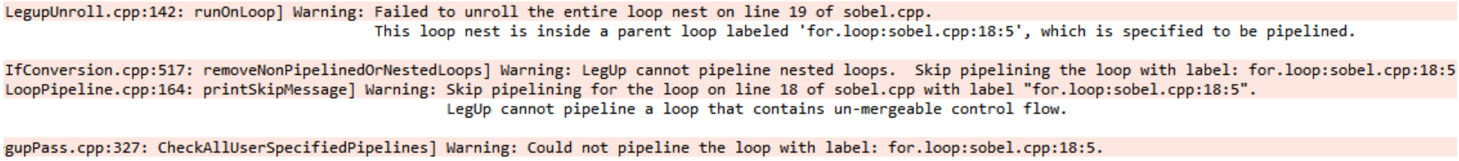
Then SmartHLS will try to fully unroll the innermost loop (j index) but SmartHLS will give a warning in the Console output since the loop has too many iterations:

```
Warning: Failed to unroll the entire loop nest on line 19 of sobel.c.
```

And since the innermost loop has not been unrolled, then the loop cannot be pipelined:

```
Warning: SmartHLS cannot pipeline nested loops.
```

See screenshot from the SmartHLS IDE console below:



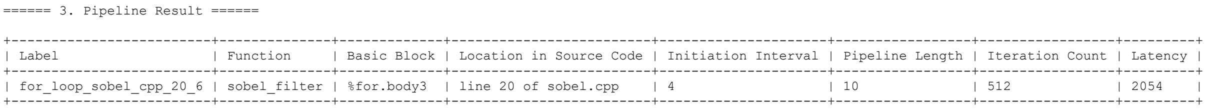
```
LegupUnroll.cpp:142: runOnLoop] Warning: Failed to unroll the entire loop nest on line 19 of sobel.cpp.
                                         This loop nest is inside a parent loop labeled 'for.loop:sobel.cpp:18:5', which is specified to be pipelined.

IfConversion.cpp:517: removeNonPipelinedOrNestedLoops] Warning: LegUp cannot pipeline nested loops.  Skip pipelining the loop with label: for.loop:sobel.cpp:18:5
LoopPipeline.cpp:164: printSkipMessage] Warning: Skip pipelining for the loop on line 18 of sobel.cpp with label "for.loop:sobel.cpp:18:5".
                                                 LegUp cannot pipeline a loop that contains un-mergeable control flow.

gupPass.cpp:327: CheckAllUserSpecifiedPipelines] Warning: Could not pipeline the loop with label: for.loop:sobel.cpp:18:5.
```

If you just pipeline the innermost loop, then the hardware will be less efficient than flattening the nested loop into a single loop, because for each outer loop iteration, we will need to stop and wait for the innermost loop pipeline to finish. If the nested loops are flattened and everything is pipelined then all the iterations can be overlapped and we never need to wait.

For example, if we open sobel.cpp from Part 1 and pipeline the innermost loop by adding the loop pipeline pragma:

```
        for (int i = 0; i < HEIGHT; i++) {
#pragma HLS loop pipeline
            for (int j = 0; j < WIDTH; j++) {
                // Set output to 0 if the 3x3 receptive field is out of bound.
                if ((i < 1) | (i > HEIGHT - 2) | (j < 1) | (j > WIDTH - 2)) {
                    out[i][j] = 0;
                    continue;
                }
```

When we re-run Software to Hardware by clicking the icon. The initiation interval of the innermost loop is 4 as shown in the summary report:



```
====== 3. Pipeline Result ======

+-------------------------+--------------+-------------+------------------------+---------------------+------------------+------------------+---------+
| Label                   | Function     | Basic Block | Location in Source Code | Initiation Interval | Pipeline Length | Iteration Count | Latency |
+-------------------------+--------------+-------------+------------------------+---------------------+------------------+------------------+---------+
| for_loop_sobel_cpp_20_6 | sobel_filter | %for.body3  | line 20 of sobel.cpp   | 4                   | 10               | 512              | 2054    |
+-------------------------+--------------+-------------+------------------------+---------------------+------------------+------------------+---------+
```

Then we run co-simulation and see the following Console output:

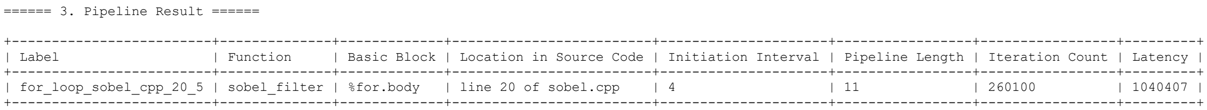
```
Number of calls:          1
Cycle latency:     1,052,677
SW/HW co-simulation: PASS
make[1]: Leaving directory '.../workspace/sobel_part1'

15:51:35 Build Finished (took 1m:51s.138ms)
```

This cycle latency roughly corresponds to 512 (outer loop iterations) x 2054 (latency of innermost loop pipeline) = 1,051,648 cycles. There are some extra cycles for the hardware running before and after the pipelined loop.

We can compare this latency to sobel.cpp in Part 2 when we pipelined the flattened loop:

```
#pragma HLS loop pipeline
    for (int i = 0; i < (HEIGHT - 2) * (WIDTH - 2); i++) {
        // increment row when column reaches end of row
        y = (x == WIDTH - 2) ? y + 1 : y;
        // increment column until end of row
        x = (x == WIDTH - 2) ? 1 : x + 1;
```

When we run compile software to hardware and we look at the summary report:



```
====== 3. Pipeline Result ======

+-------------------------+--------------+------------+-----------------------+----------------------+-----------------+-----------------+---------+
| Label                   | Function     | Basic Block | Location in Source Code | Initiation Interval | Pipeline Length | Iteration Count | Latency |
+-------------------------+--------------+------------+-----------------------+----------------------+-----------------+-----------------+---------+
| for_loop_sobel_cpp_20_5 | sobel_filter | %for.body  | line 20 of sobel.cpp  | 4                    | 11              | 260100          | 1040407 |
+-------------------------+--------------+------------+-----------------------+----------------------+-----------------+-----------------+---------+
```

The initiation interval of the flattened loop is still 4. But the pipeline length/depth is now 1 cycle longer (11 cycles instead of 10 cycles).

Now when we run co-simulation we see the Console output below:

```
Number of calls:           1
Cycle latency:     1,040,413
SW/HW co-simulation: PASS
make[1]: Leaving directory '.../workspace/sobel_part2'

15:59:45 Build Finished (took 1m:40s.823ms)
```

This cycle latency roughly corresponds to the 1,040,407 latency reported in the last column “Latency” in the pipeline summary report.

Flattening the loop improves the cycle latency from: 1,052,677 to 1,040,413 (1% improvement).

In this case, there is not much improvement by loop flattening. But depending on the loop nest there can be a big impact.