LegUp HLS Tutorial for Intel Arria 10
Sobel Filtering for Image Edge Detection

1 Introduction

This tutorial will introduce you to high-level synthesis (HLS) concepts using LegUp. 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) as discussed in previous days of the course. 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.

Sobel filtering involves applying a pair of two 3×3 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.

Figure 1: Sobel filtering example.

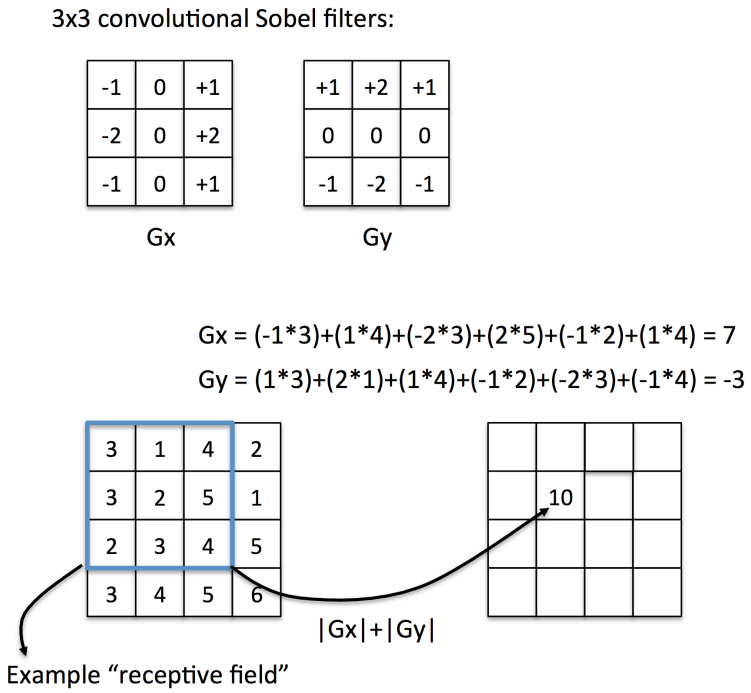

Figure 2: Sobel filters and computational action to compute one pixel in the output image from an input image.

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 together 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.

The bottom of Figure 2 shows an input image with 4 rows and 4 columns, where the value in each cell represents a pixel colour. 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 3×3 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.

Download the example files from `www.legupcomputing.com/static/downloads/tutorials/intel/sobel_tutorial.zip` and unzip the file.

2 Initial Implementation

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

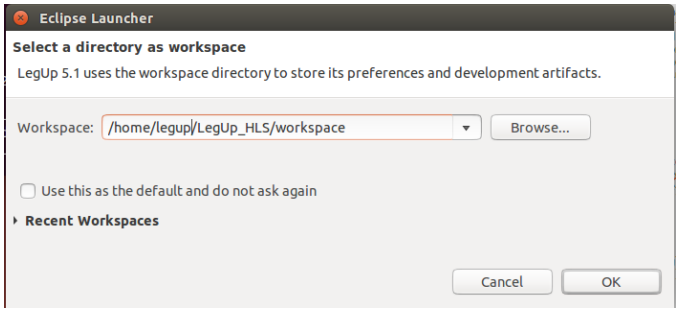

Figure 3: Choosing a workspace.

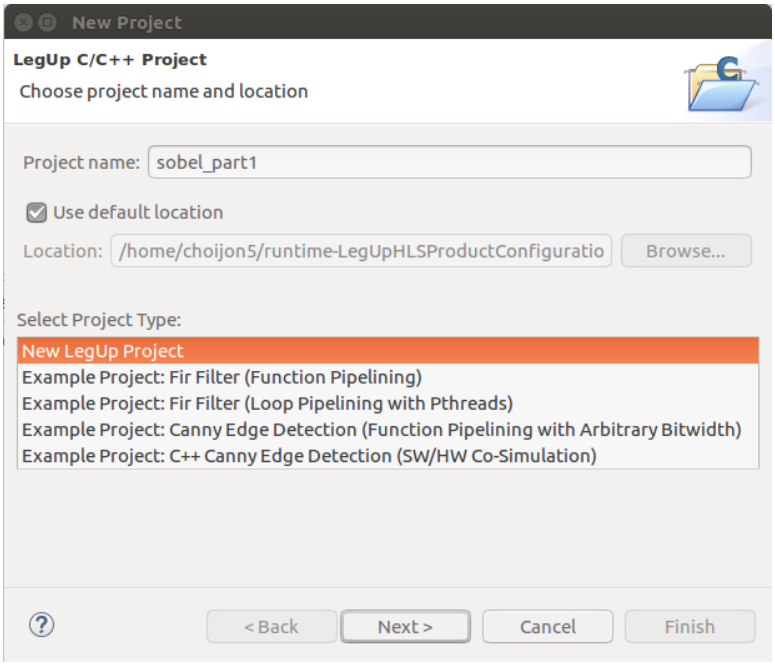

Figure 4: Creating a new project.

Start the LegUp GUI. On Windows, this can be done by double-clicking on the LegUp shortcut, and on Linux, this can be done by opening up a terminal and running the following command.

```
legup_hls
```

You will first see a dialog box like that in Figure 3 appear. You can use the default workspace for all parts of this tutorial by clicking on *Ok*.

Once the GUI opens, under the *File* menu, choose *New* and then *LegUp C/C++ project*. You will then see a window like that in Figure 4. Type a name for your project as shown: `sobel_part1`. Then click on *Next*.

Now, you will import the source files for part 1 of this tutorial to the project, as shown in Figure 5. Click on *Add Files*, navigate to where you have downloaded and unzipped the example files, and go into the *part1* directory. Be sure to add both `.h` files and the `.c` file. You can also specify the top-level function. For this example, we will leave it empty. By default, the main function becomes the top-level function. Now, click on *Next*.

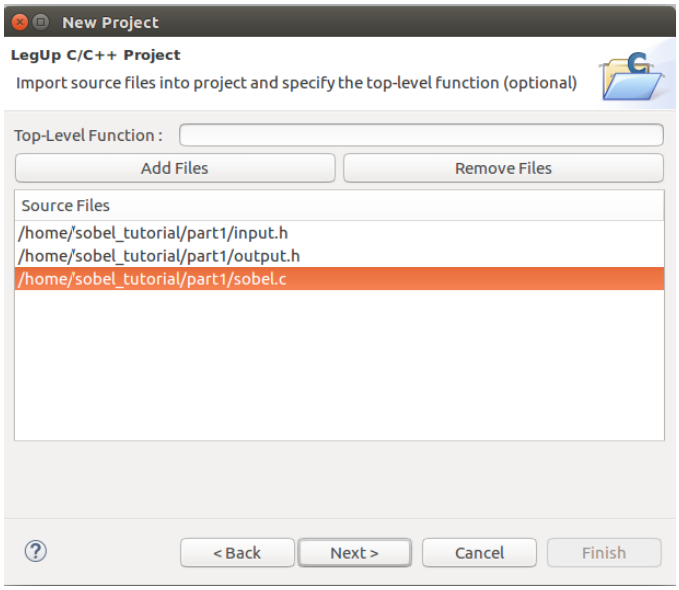

Figure 5: Adding the source files.

You will then see a dialog box where you can specify your own testbench, which you do not need to do for this part of the tutorial. So, for this dialog box, simply click *Next*. Finally, to complete the project creation, you will choose the FPGA vendor and family you intend to use. Use the selections shown in Figure 6. Choose Intel and Arria 10. For *FPGA Device*, you have an option to choose 10AX115N2F40E2LG on on the Arria 10 PAC Board or use another Arria 10 device that is not listed. For this tutorial, we will use the Arria 10 PAC Board but if you want to target Arria 10 device, you can choose *Custom Device* for the *FPGA Device* field, then type in the device name in the *Custom Device* field. Click on *Finish* when you are done. It may take a few moments to create the project.

Once the project is created, you should now see that the `sobel.c` file automatically opens up. You can also open it yourself by expanding the `sobel_part1` directory shown in the left Project Explorer pane and double clicking on `sobel.c`.

Towards the top of the GUI, you should find a toolbar (as shown in Figure 7), which you can use to execute the main features of the LegUp tool. Hover over each icon to find out their meanings. Starting from the left of Figure 7, the icons are to *Add Files to Project*, *Compile Software*, *Run Software*, *Debug Software*, *Profile Software*, then also to *Compile Software to Hardware*, *Compile Software to Processor/Accelerator SoC*, *Simulate Hardware*, perform *SW/HW Co-Simulation*, and *Synthesize Hardware to FPGA*. With the last three icons, you can set *HLS Constraints*, *Launch Schedule Viewer*, and lastly *Clean LegUp Project*.

An important panel of the GUI is the *Project Explorer* on the left side of the window, and also shown in Figure 8. This will be used heavily throughout the tutorial to view source files and synthesis reports. Clicking on the small arrow icon to expand the `sobel_part1` project, as shown in the right-side of Figure 8. You can now double click any of the source files, such as `sobel.c`, and you will see the source file appear in the panel to the right of the *Project Explorer*.

Browse through the code in `sobel.c`. In the `sobel_filter` function, you should see a pair of nested outer loops that walk over the entire image, and a pair of inner nested loops that iterate

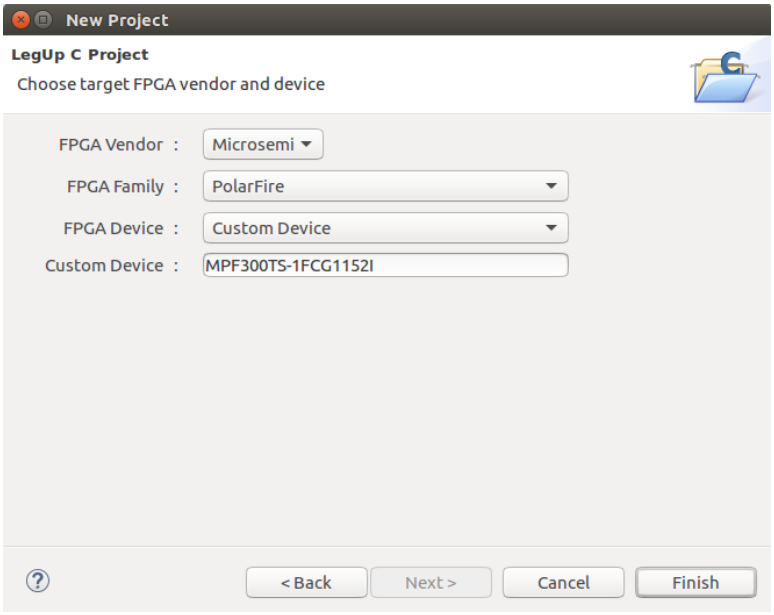

Figure 6: Selecting the FPGA vendor and device.

Figure 7: LegUp GUI toolbar icons.

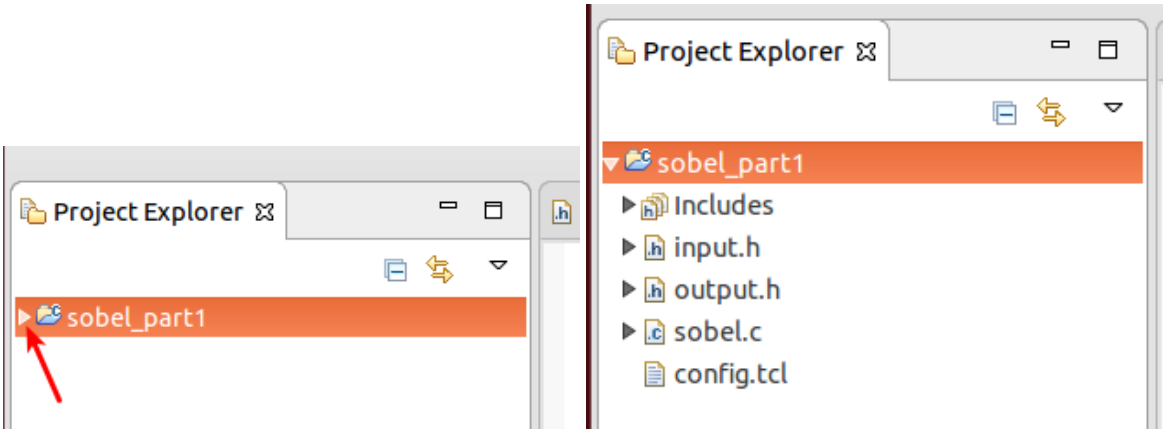

Figure 8: Explorer for browsing source and reports.

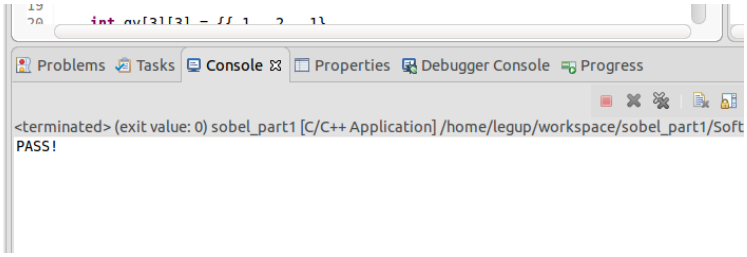

Figure 9: Console after software execution.

through the filter window. For each pixel of the image that is not in the border, the 3×3 area surrounding it is convolved with G_x and G_y, then its magnitude summed, to produce the final output pixel. The `main` function here simply calls the `sobel_filter` function, and performs error checking which compares the computed image against the golden output.

Before compiling to hardware, 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. Try this by clicking on the `Compile Software` icon in the toolbar (Figure 7). 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 GUI. 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 9.

Once you are familiar with the `C` code and its behaviour, compile the Sobel filter into hardware using LegUp by clicking on the toolbar icon to *Compile Software to Hardware*. This command tells LegUp to compile the entire `C` program into hardware. Several report files and a Verilog file called `sobel_part1.v` will be generated. To find the Verilog, use the *Project Explorer* window as shown in Figure 10. Open and scroll through `sobel_part1.v`. Near the bottom of `sobel_part1.v`, you will find a Verilog module called `main_tb`. This is the testbench used for the simulation of the main Verilog module, `top`, which is the top-level module that instantiates the `main` module. `main_tb` simulates the `top` module with a clock, a start and a reset signal, then it waits for a finish signal from `top` to signify the completion of the sobel filter. Navigate to around line 310 to find

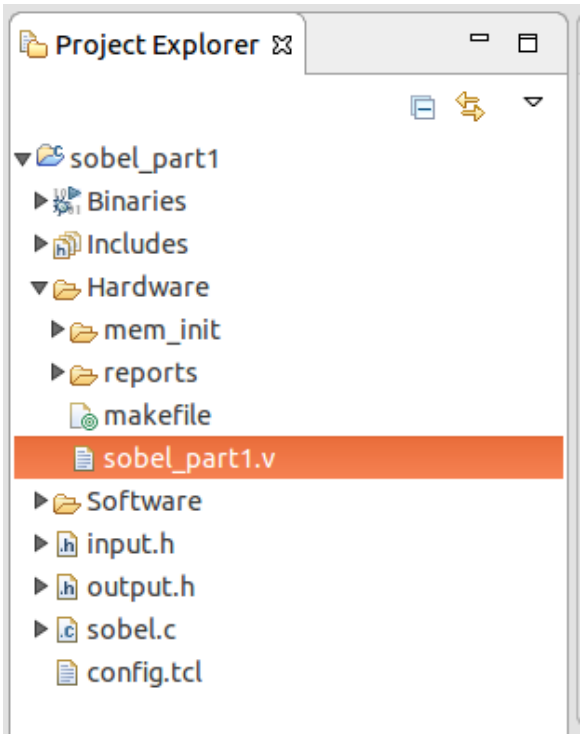

Figure 10: Finding the LegUp-generated Verilog in the Project Explorer.

the FSM that LegUp generated in order to control the state of execution in the hardware circuit. This is the main structure that enables the circuit to honour the data dependencies and control flow from the sequential software program.

Now that you have an idea of the input and output to LegUp, let's take a look at what LegUp did to generate hardware from `C` source code. Start the LegUp hardware-visualizer GUI by clicking on the *Launch Schedule Viewer* icon in the toolbar (see Figure 7). In the left panel of the visualizer GUI, you will see the names of the functions and basic blocks in the program. Observe that the `sobel_filter` function doesn't appear in the list. This is because the LLVM compiler has inlined it into the `main` function. Click on the `main` function, and you will see the control-flow graph (CFG) for the program, similar to Figure 11. The names of the basic blocks in the program are prefixed with BB. It is difficult to determine how the names of the basic blocks relate to the original C code; however, can you figure out where each loop is in the CFG? The inner loop consists of basic blocks `BB__5`, `BB_preheaderpreheaderi`, `BB_uslcssa8usi_crit_edge`, and `BB_uslcssa8usi`. Try double clicking on `BB_preheaderpreheaderi`.

Figure 12 shows the schedule for `BB_preheaderpreheaderi`, which is the main part of the inner-most loop body. The middle panel shows the names of the LLVM instructions. The right-most panel shows how the instructions are scheduled into states (states 7 to 11 for this basic block). Hold your mouse over top of some of the blue boxes in the schedule: you will see the inputs and outputs of each instruction become coloured. Look closely at the names of the LLVM 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, among others.

Now let's simulate the Verilog RTL hardware with ModelSim to find out the number of cycles

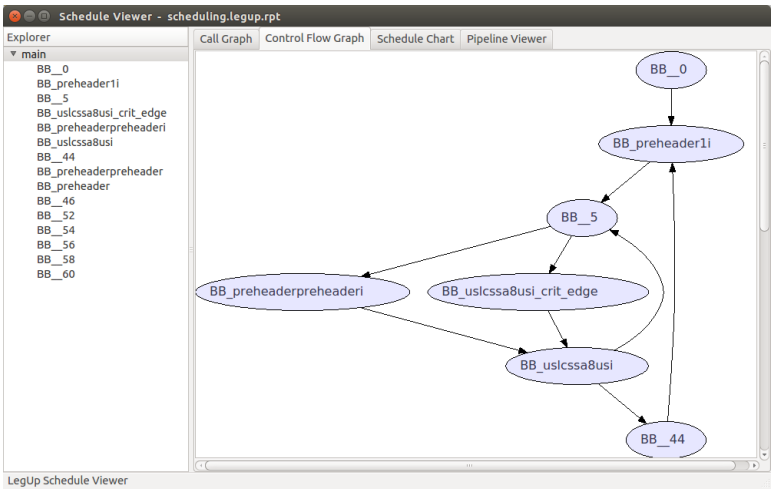

Figure 11: Control-Flow Graph for the Sobel filter.

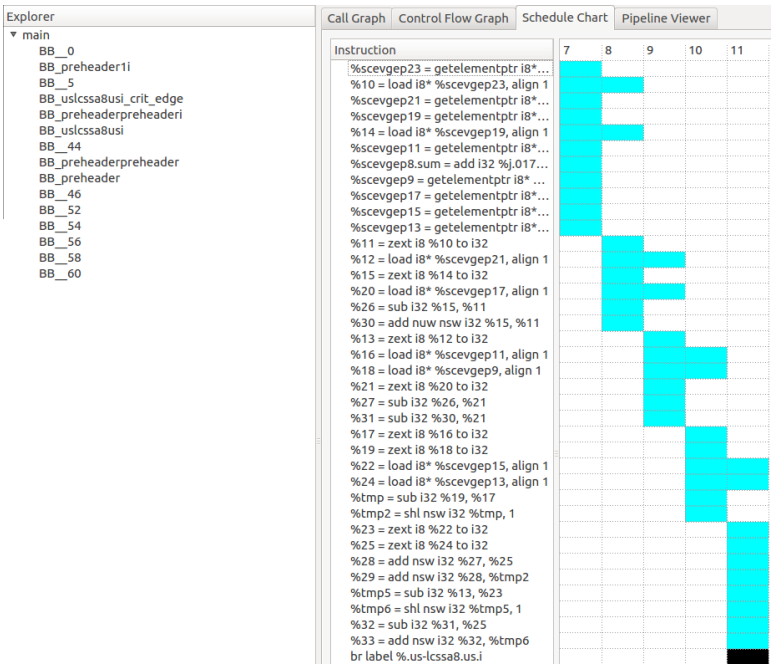

Figure 12: Schedule for a part of the inner-most loop

needed to execute the circuit – the cycle latency. Close the viewer first, then click the *Simulate Hardware* icon on the toolbar. If this is the first time you are using LegUp, you will need to set up the paths to Modelsim (and Intel Quartus for later parts of this tutorial). To set the paths, click on *LegUp* on the top menu bar, then click on *Tool Path Settings*. Once the dialog opens up, set the paths for *ModelSim Simulator* and *Intel Quartus*. Notice that you can also set up the paths for other vendor's synthesis tools, which are used when you select the particular vendor's FPGA during project creation. For this tutorial, you will only be using Intel. Click *OK*, then click on *Simulate Hardware* again. In the *Console* window, you will see various messages printed by ModelSim related to loading simulation models for the Altera hardware. It may take a few minutes to simulate. We want to focus on the message near the end which appears something like this:

```
...
# PASS!
# At t=          67994150000 clk=1 finish=1 return_val=          0
# Cycles:               3399705
```

We see that the simulation took 3,399,705 cycles. Also observe that simulation printed "PASS!", which is the same message we got when the software version passed the built-in error-checking functionality. This means that the LegUp generated hardware produced the same results as the software version.

The simulation above is called a functional simulation since it simulates the logic without mapping it to an FPGA. Now let's map the Verilog to the Intel Arria 10 FPGA to obtain information such as the resource usage and the Fmax of this design (i.e. the clock period). To do this, click the icon on the toolbar to *Synthesize Hardware to FPGA*. This will automatically invoke Quartus to create a Quartus project and synthesize the LegUp design to the Arria 10 FPGA. Qaurtus is the name of Intel's FPGA synthesis, placement, routing, and timing analysis tool. This may take a while.

Once the command completes, you will see that `summary.results.rpt` file automatically opens up. LegUp automatically parses the area and Fmax results from the appropriate reports files generated by Libero and stores them into this file. You should results similar to what is shown below. Your numbers may differ slightly, depending the version of LegUp and Quartus you are using.

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

+-------------------------------------------------+
; Slow 900mV 100C Model Fmax Summary               ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 210.79 MHz ; 210.79 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+


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

```
Fitter Status : Successful - Wed Aug  1 16:13:58 2018
Quartus Prime Version : 16.0.0 Build 211 04/27/2016 SJ Standard Edition
Revision Name : top
Top-level Entity Name : top
Family : Arria 10
Device : 10AX115N2F40E2LG
Timing Models : Final
Logic utilization (in ALMs) : 611 / 427,200 ( < 1 % )
Total registers : 855
Total pins : 36 / 826 ( 4 % )
Total virtual pins : 0
Total block memory bits : 6,291,456 / 55,562,240 ( 11 % )
Total RAM Blocks : 384 / 2,713 ( 14 % )
Total DSP Blocks : 0 / 1,518 ( 0 % )
Total HSSI RX channels : 0 / 48 ( 0 % )
Total HSSI TX channels : 0 / 48 ( 0 % )
Total PLLs : 0 / 112 ( 0 % )
```

Wall-clock time is the key performance metric for HLS, computed as the product of the cycle latency and the clock period. In this case, our cycle latency was 3,399,705 and the clock period was 4.74ns. The wall-clock time of our implementation is therefore 3,399,705 $\times$ 4.74 = 16,115μs.

3 Loop Pipelining

In this section, you will use loop pipelining to improve the throughput of the hardware generated by LegUp. 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 one 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).

Figure 13 shows an example of loop pipelining. Figure 13(b) shows the sequential loop, where the II=3, and it takes 8 clock cycles for the 3 loop iterations before the final write is performed. Figure 13(c) shows the pipelined loop. In this example, there are no resource contentions or data dependencies. Hence, the II=1, and it takes 4 clock cycles before the final write is performed. You can see that loop pipelining can significantly improve the performance of your circuit, especially when there are no data dependencies or resource contentions.

Using the same procedure you used in the previous part of this tutorial, create a brand new LegUp project and this time, include the source files for part 2 in your project. Once you have completed that project creation, open the `sobel.c` source file for part 2. With the Sobel filter, since each pixel of the output is dependent only on the input image and the constant matrices G_x

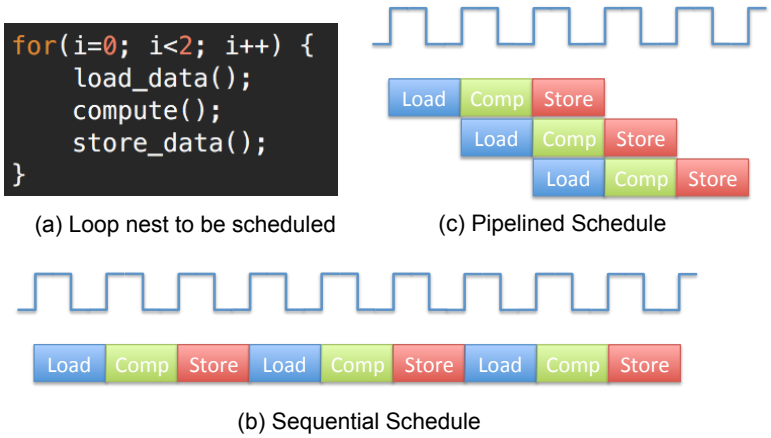

Figure 13: Loop Pipelining Example

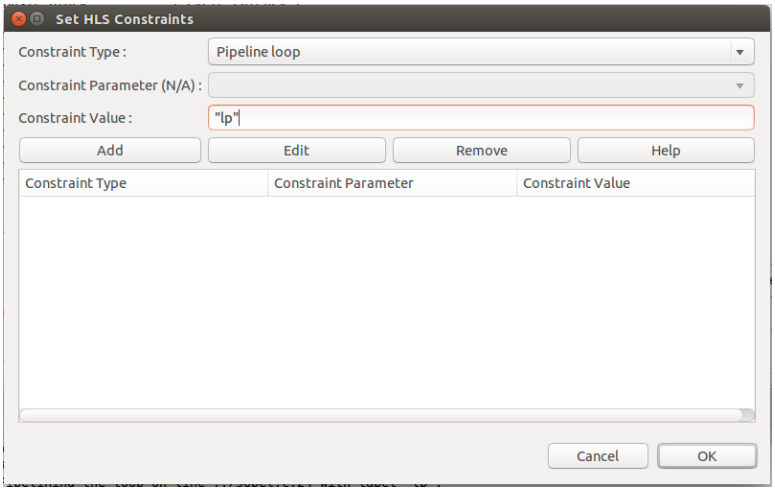

Figure 14: HLS constraints editor.

and G_y, we would like to pipeline the calculation of each pixel. To invoke loop pipelining, you must tell LegUp which loop you wish to pipeline, which consists of two steps. If you look at line 23, you can see that we have already labelled the loop with `lp:`, as shown below.

```
lp: for (i = 0; i < (HEIGHT-2)*(WIDTH-2); i++) {
```

Now, click on the *HLS Constraints* icon in the toolbar to open up the constraints editor. You will see the dialog box shown in Figure 14 appear. For the Constraint Type, use the pull-down menu to choose the *Pipeline loop*, and enter the Constraint Value as `lp` (double-quotes shown in the figure are optional), which is the name of the loop label you added to the `C` source. Click on *Add*, which should add the Pipeline loop constraint, then click on *OK*.

Having made those modifications, you can now synthesize the design by clicking the *Compile Software to Hardware* icon in the toolbar. In the *Console* window, you should the following messages:

```
Info: Resource constraint limits initiation interval to 4.
```

```
      Resource 'elaine_512_input_local_memory_port' has 8 uses per cycle
      but only 2 units available.
+-----------------------------------------------+--------------------+------+
| Operation                                     | Location           | Uses |
+-----------------------------------------------+--------------------+----- +
| 'load' operation for array 'elaine_512_input' | line 33 of sobel.c | 1    |
| 'load' operation for array 'elaine_512_input' | line 33 of sobel.c | 2    |
| 'load' operation for array 'elaine_512_input' | line 33 of sobel.c | 3    |
| 'load' operation for array 'elaine_512_input' | line 33 of sobel.c | 4    |
| 'load' operation for array 'elaine_512_input' | line 33 of sobel.c | 5    |
| 'load' operation for array 'elaine_512_input' | line 33 of sobel.c | 6    |
| 'load' operation for array 'elaine_512_input' | line 33 of sobel.c | 7    |
| 'load' operation for array 'elaine_512_input' | line 33 of sobel.c | 8    |
+-----------------------------------------------+--------------------+------+
|                                               | Total # of Uses    | 8    |
+-----------------------------------------------+--------------------+------+
```

These messages indicate that LegUp cannot achieve an II of 1 (highest throughput) due to resource conflicts – there are 8 loads from a memory but each memory is dual-ported on an FPGA. In order to accommodate 8 loads, we need to extend the II to 4, meaning there are 4 cycles between successive iterations of the loop. A good way to visualize this is to look at the schedule viewer. Click on the *Launch Schedule Viewer* icon again. Double-click on main, then in the CFG, you will see a basic block called `BB_preheaderi`. Double-click it to reveal the loop pipeline schedule, similar to that shown in Figure 15. The x-axis show cycles and the y-axis show 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. The dark black rectangle on the right illustrates what the pipeline looks like in steady state. In steady state, three iterations of the loop are "in flight" at once. In steady state, you can see that there are two loads in cycle 8 from iteration 1, two loads in cycle 9 from iteration 1, two loads in cycle 10 from iteration 2, and two loads in cycle 11 from iteration 2. Thus the 8 loads are spread out over 4 cycles, making the II = 4.

Now, exit the viewer and simulate the design in ModelSim by clicking the *Simulate Hardware* icon on the toolbar. You should see results similar to this:

```
...
# PASS!
# At t=         36434710000 clk=1 finish=1 return_val=         0
# Cycles:              1821733
```

Observe that loop pipelining has dramatically improved the cycle latency for the design, reducing it from 3,402,261 cycles to 1,821,733 cycles in total. Finally, use Intel's Quartus to map the design onto the Arria 10 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`. You should see results similar to the following.

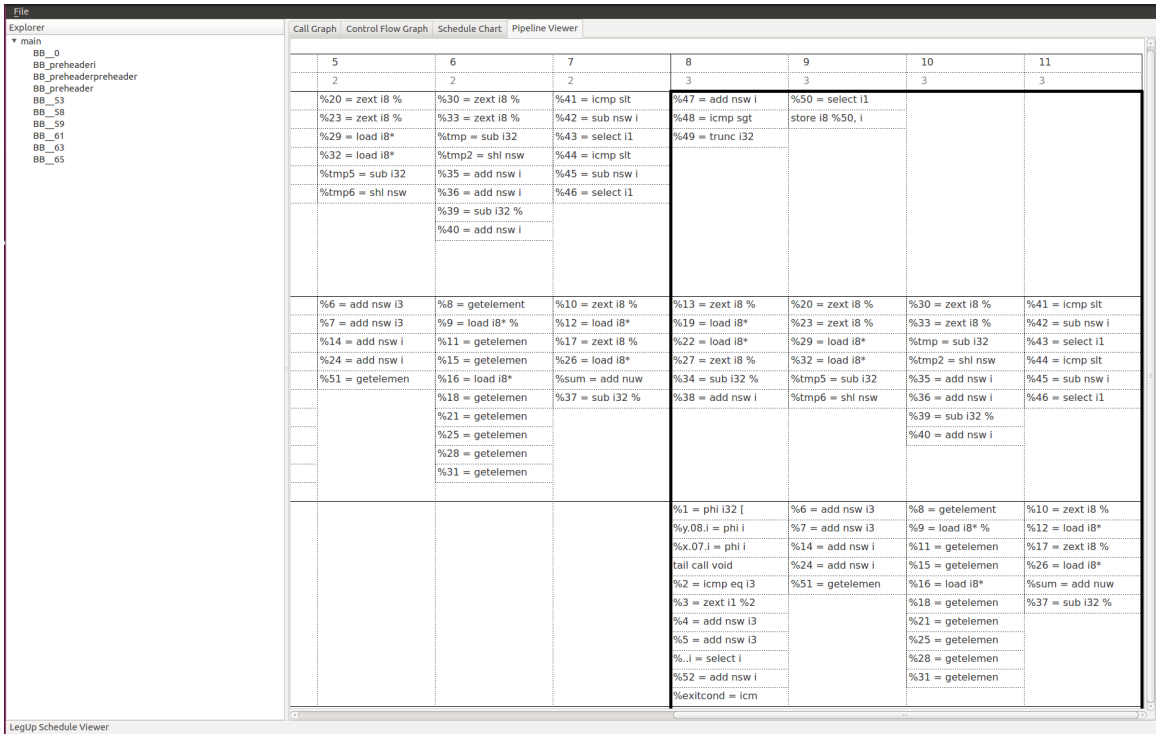

Figure 15: Pipeline Schedule for the Sobel filter.

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

+----------------------------------------------------+
; Slow 900mV 100C Model Fmax Summary                 ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 209.42 MHz ; 209.42 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+

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

Fitter Status : Successful - Wed Aug  1 16:34:47 2018
Quartus Prime Version : 16.0.0 Build 211 04/27/2016 SJ Standard Edition
Revision Name : top
Top-level Entity Name : top
Family : Arria 10
Device : 10AX115N2F40E2LG
Timing Models : Final
Logic utilization (in ALMs) : 602 / 427,200 ( < 1 % )
Total registers : 895
Total pins : 36 / 826 ( 4 % )
```

```
Total virtual pins : 0
Total block memory bits : 6,291,456 / 55,562,240 ( 11 % )
Total RAM Blocks : 384 / 2,713 ( 14 % )
Total DSP Blocks : 0 / 1,518 ( 0 % )
Total HSSI RX channels : 0 / 48 ( 0 % )
Total HSSI TX channels : 0 / 48 ( 0 % )
Total PLLs : 0 / 112 ( 0 % )
```

4 Synthesizing Parallel Hardware with Pthreads

In this section, you will implement a multi-threaded version of the Sobel filter using Pthreads. LegUp compiles multi-threaded programs written using either Pthreads, OpenMP or a combination of both parallel programming paradigms into multiple accelerators. Through this approach, a software engineer without hardware skills can exploit spatial parallelism on an FPGA. Each software thread will translate to an instance of the hardware accelerator kernel module. In the case of Sobel, each element in the final output image of the Sobel filter does not carry any dependencies to or from any other elements in the final image, we can therefore split the computations of blocks of rows (or columns) into different threads.

Create a new LegUp project, containing the source code for part 3 of this tutorial. Look at the `sobel.c` source file. You will find the `main` function of the sobel filter benchmark. You will also find the kernel function `sf_sw`. Take a look at this file to understand how each thread performs Sobel filtering on the image.

As seen from the code, LegUp recognizes various Pthread functions and constructs. In the `main` function, you will see that the original call to the `sf_sw` function has been replaced by calls to `pthread_create` which launches `NUM_THREADS` threads – each thread executing the now threaded version of `sf_sw`. This is followed by calls to `pthread_join` which halts execution of the `main` function until all thread modules have completed execution.

In the `define.h` file, you will see the declaration of the `C` structure `thread_arg` which is to be passed to the Pthread function as the input arguments. The `struct` contains two fields, they are used by the Pthread function to determine the rows of the output image it will compute.

The default Pthread implementation will run 4 simultaneous threads for the Sobel filter, where each thread is responsible for computing 1/4th of the rows in the output image.

Compile and execute the software before generating hardware by clicking the relevant icons on the toolbar. You should see the following appear in the *Dialog* box:

```
...
RESULT: 262144
RESULT: PASS
```

Now, compile the design to hardware by clicking the *Compile Software to Hardware* icon in the toolbar. After the Verilog is generated, you should simulate the Verilog by clicking the *Simulate Hardware* icon in the toolbar. This simulation may take a while. Once it's done, should see output similar to this:

```
...
# MATCH: i =  511 j =         510 sw =   0 hw =   0
# MATCH: i =  511 j =         511 sw =   0 hw =   0
# Result:      262144
# RESULT: PASS
# At t=         43141850000 clk=1 finish=1 return_val=         0
# Cycles:              2157090
...
```

As it turns out, the cycle count (2,157,090) is not particularly performant. To understand why, open up the LegUp-generated memory report `summary.legup.rpt` using the *Project Explorer*. At the bottom of the report, you will find a summary similar to this:

```
...
+-------------------------------------------------------------------------------+
| Shared Local Memories                                                         |
+----------------------+------------------------+------+------------+---------+
| Name                 | Accessing Function(s)  | Type | Size [Bits] | ...     |
+----------------------+------------------------+------+------------+---------+
| elaine_512_input     | sf_sw, sf_sw_1...      | ROM  | 2097152    | ...     |
| sobel_output         | main, sf_sw, sf_sw_1... | RAM  | 2097152    | ...     |
+----------------------+------------------------+------+------------+---------+
...
```

Under `Shared Local Memories` you will see the read-only memory named `elaine_512_input`, which is shared between all instances (threads) of the `sf_sw` function. This memory corresponds to the input data array `elaine_512_input` containing the unfiltered image. Shared local memories are memories which are shared between multiple functions. An arbiter is necessary to arbitrate accesses between the multiple functions when they compete for access. Since each memory is dual-ported, only two accesses per cycles are permitted.

You may also notice that the `elaine_512_input` memory read-only, meaning they are never modified and so we can actually create copies of them local to each function. In so doing, we give each thread exclusive access to a copy of the data it needs. To do this, open the *HLS Constraints* editor using the toolbar and add a constraint to replicate the read-only memory, as shown in Figure 16. Don't forget to click *Add* then click *OK*.

Now, click on *Compile Software to Hardware* again to re-generate the hardware with the added constraint and simulate the design once more. You should have improved results, similar to those below.

```
...
# Result:      262144
# RESULT: PASS
# At t=         35356990000 clk=1 finish=1 return_val=         0
# Cycles:              1767847
...
```

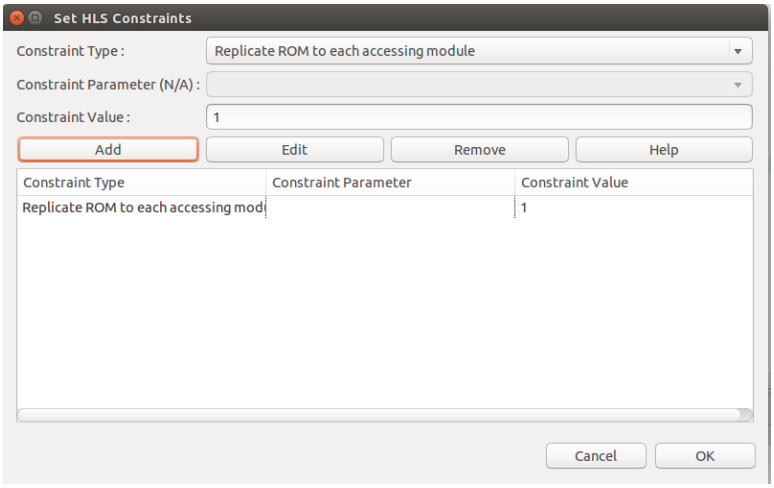

Figure 16: Adding constraint to replicate ROMs.

By replicating ROMs to each accessing function and reducing memory contention, we have improved the cycle count from 2,157,090 to 1,767,874. Now try synthesizing the design to FPGA with Quartus by clicking on the *Synthesize Hardware to FPGA* icon. You should see results similar to the following.

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

+-------------------------------------------------+
; Slow 900mV 100C Model Fmax Summary               ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 179.63 MHz ; 179.63 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+

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

Fitter Status : Successful - Wed Aug  1 16:53:19 2018
Quartus Prime Version : 16.0.0 Build 211 04/27/2016 SJ Standard Edition
Revision Name : top
Top-level Entity Name : top
Family : Arria 10
Device : 10AX115N2F40E2LG
Timing Models : Final
Logic utilization (in ALMs) : 1,703 / 427,200 ( < 1 % )
Total registers : 2219
Total pins : 36 / 826 ( 4 % )
Total virtual pins : 0
Total block memory bits : 20,971,536 / 55,562,240 ( 38 % )
```

```
Total RAM Blocks : 1,281 / 2,713 ( 47 % )
Total DSP Blocks : 0 / 1,518 ( 0 % )
Total HSSI RX channels : 0 / 48 ( 0 % )
Total HSSI TX channels : 0 / 48 ( 0 % )
Total PLLs : 0 / 112 ( 0 % )
```

5 Streaming Hardware Synthesis

The final hardware implementation you will realize 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 hardware, new inputs can continue to be injected at the same II. For example, a streaming 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, it takes 10 clock cycles to complete its work; however, it 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 behaviour, 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 512×512 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 17 shows the 3×3 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 are able to 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".

Create a new LegUp project for part 4 of the tutorial and include all of the `.c` and `.h` files for part 4. Specify the `Top-Level Function` as `sobel_filter`. For this part of the tutorial, you will also need to include custom test bench files. In the project creation wizard, on the next page after including the `.c` and `.h` files, specify the `Test Bench Module` as `streaming_tb`. Click on `Add Files` to include `streaming_tb.v`, `input.dat` and `output.dat` files in the part 4 directory. Specify Intel's Arria 10 device and finish creating the project. Examine the `sobel.c` file in the project viewer and you will find a function `sf_window_3x3_and_line_buffer` with the following

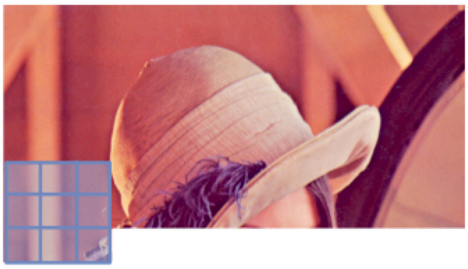

Figure 17: Motivation for use of line buffers.

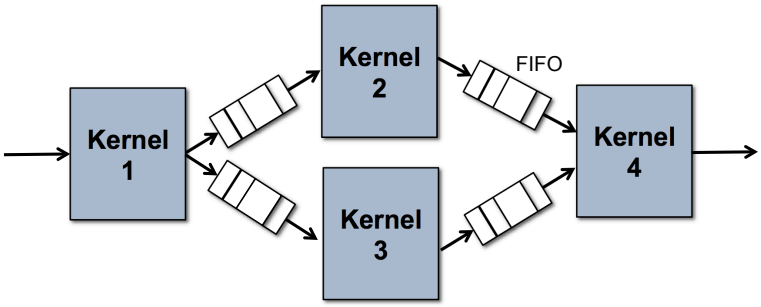

Figure 18: Streaming hardware circuit with FIFO queues between components.

signature:

```
void sf_window_3x3_and_line_buffer(unsigned char input_pixel,
                                   unsigned char window[3][3])
```

This function accepts a single input pixel as input and then it populates the 3×3 window of pixels on which the Sobel filter will operate (parameter `window` is an output of this function). In this function, two circular shift registers are used to implement the line buffers, `prev_row1` and `prev_row2`. `prev_row2` represents the row that is two rows *behind* the row currently being input; `prev_row1` represents the row that is just one row behind the row currently being input. Observe that these arrays are declared as `static` so retain their state each time the function is called.

Before going further, it is necessary to understand another aspect of streaming hardware. A common feature of such hardware is the use of FIFO queues to interconnect the various streaming components, as shown in Figure 18. 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 queues and pushes computed data into its output queue(s). If its input queue is empty, the unit stalls. Likewise, if the output queues are full, the unit stalls. In the example, 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 LegUp 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.c` file at line 48. Observe that pointers to the

input and output FIFOs are passed to the function as parameters. A pixel value is read from the input FIFO via the `fifo_read` function; later, a pixel is written to the output FIFO through the `fifo_write` function. These functions are declared in the `legup/streaming.h` header.

```
void sobel_filter(FIFO *input_fifo, FIFO *output_fifo) {

    unsigned char input_pixel = fifo_read(input_fifo);

    ...

    fifo_write(output_fifo, ret);

    ...
```

The other parts of the `sobel_filter` function are very similar to those you have seen in 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). Study the code in `sobel_filter` function carefully before moving on with this part of the tutorial.

There are two other relevant FIFO-related functions, namely, to create and destroy the FIFOs. In the `main` function in `sobel.c`, you will see a call to `fifo_malloc`, which accepts two parameters as input: the bitwidth of the FIFO input data, and the FIFO depth. In this case, the input FIFO has 8-bit wide pixel data; the output FIFO has 8-bit wide pixel data. At the end of `main`, the FIFOs are freed with a call to `fifo_free`.

Reading the `main` function, we see that the image input data (stored in `input.h`) is pushed into the `input_fifo`. Then, the Sobel filter is invoked on the input data $HEIGHT \times WIDTH$ times. Finally, the output values are checked for correctness and PASS or FAIL is reported.

Click the appropriate icons to compile and run the software, and you should see the computed and golden pixel values and the message `RESULT: PASS`.

We are now ready to synthesize the circuit, but first, we must set a pipelining constraint for streaming hardware. Click the *constraints* icon and set the constraints to match those of Figure 19. The `Pipeline Function` constraint tells LegUp that the `sobel_filter` function is intended to be a streaming kernel. You may optionally give an argument to specify the initiation interval (`-ii 1`). Note that even without `-ii 1`, LegUp will always attempt to get achieve the lowest initiation interval by default. You should see that the other constraints for `Set test bench module`, `Set test bench file`, and `Set top-level function` have already been set, since you specified them when creating the project. `Set test bench module` and `Set test bench file` are used to setup the ModelSim simulation. `Set top-level function` specifies that the kernel itself should be the top-level Verilog module. Once you have set all the constraint as shown in Figure 19, click *OK*.

Now synthesize the hardware by clicking the *Compile Software to Hardware* icon. Then, simulate the streaming hardware using the special testbench by clicking the *Simulate Hardware* icon.

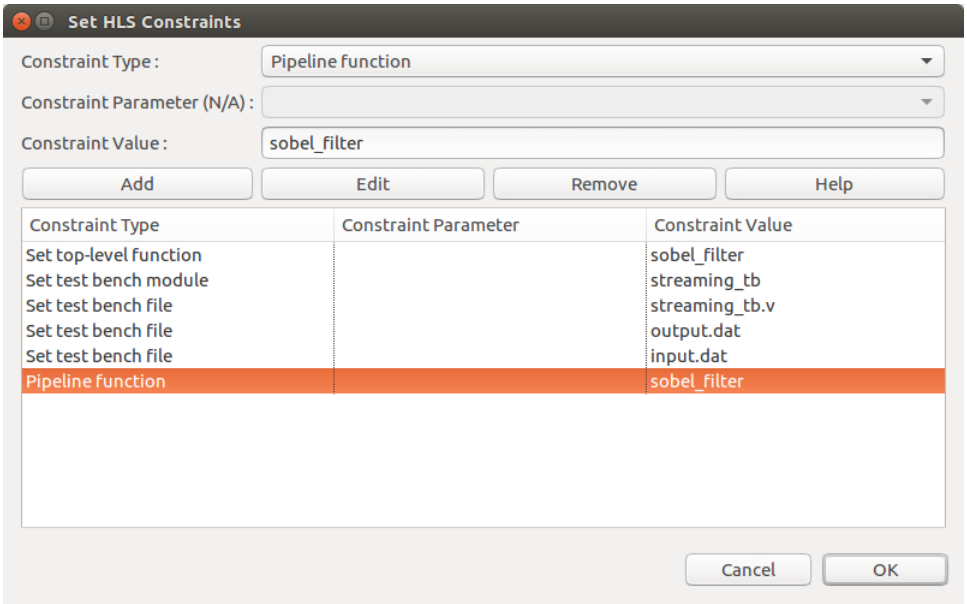

Figure 19: Constraints for part 4 – streaming implementation.

You will see scrolling output, reporting the computed and expected pixel value at each clock cycle. The end of the output should look similar to this:

```
...
# At cycle      262663: output matches expected value,   0 ==   0
# At cycle      262664: output matches expected value,   0 ==   0
# At cycle      262665: output matches expected value,   0 ==   0
# PASS
# ** Note: $finish    : ../../streaming_tb.v(116)
#    Time: 5253350 ns  Iteration: 1  Instance: /streaming_tb
```

The total number of clock cycles is about 262,665, which is very close to $512 \times 512 = 262,144$. That is, the number of cycles for the streaming hardware is close to the total number of pixels computed.

Now, synthesize the circuit to the FPGA by clicking the *Synthesize Hardware to FPGA* icon in the toolbar. You should see the following results in the summary.results.rpt file .

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

+-------------------------------------------------+
; Slow 900mV 100C Model Fmax Summary               ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 336.02 MHz ; 336.02 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+
```

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

Fitter Status : Successful - Wed Aug  1 17:05:01 2018
Quartus Prime Version : 16.0.0 Build 211 04/27/2016 SJ Standard Edition
Revision Name : top
Top-level Entity Name : sobel_filter_top
Family : Arria 10
Device : 10AX115N2F40E2LG
Timing Models : Final
Logic utilization (in ALMs) : 263 / 427,200 ( < 1 % )
Total registers : 397
Total pins : 24 / 826 ( 3 % )
Total virtual pins : 0
Total block memory bits : 8,192 / 55,562,240 ( < 1 % )
Total RAM Blocks : 2 / 2,713 ( < 1 % )
Total DSP Blocks : 0 / 1,518 ( 0 % )
Total HSSI RX channels : 0 / 48 ( 0 % )
Total HSSI TX channels : 0 / 48 ( 0 % )
Total PLLs : 0 / 112 ( 0 % )
```

The area values for the streaming implementation are significantly smaller than in the prior parts of this tutorial. The reason for this is that, in the previous parts, memories to store the input, output and golden images were part of the circuit itself. In this case, the circuit does not contain such memories; rather, the data is streamed in and streamed out pixel by pixel. We are also synthesizing only the `sobel_filter` function in this part, whereas the previous parts synthesized the entire program including the `main` function.

5.1 SW/HW Co-Simulation

LegUp 6.1 has added support for SW/HW Co-Simulation, which allows one to simulate a LegUp-generated hardware core without having to provide a custom RTL testbench (as was done for the previous part). Previously, when a top-level function is specified (that is not the main function), users needed to provide their own RTL testbench to simulate the LegUp-generated core.

Using SW/HW Co-Simulation, one can use the software portion as the testbench to feed inputs to the top-level function (which will be compiled to hardware and be simulated with ModelSim), and check outputs after the top-level function returns. This can be very useful since the software be used to perform tasks such as opening up files for test input/output data or use dynamic memory allocation of input/output arrays, which are operations that much more amenable to be done in software.

Let's try SW/HW Co-Simulation on the previous example, part4, which used function pipelining. First open up the *HLS Constraints* editor and remove the `Set test bench module` and `Set test bench file` constraints, which can be done by highlighting those constraints and clicking on the *Remove* button, then click on *OK* (Note that co-simulation will still work even with those

constraints, but we want to make it clear that we are not using the user-provided RTL testbench). Now click on the *SW/HW Co-Simulation* icon in the toolbar, which is next to the *Simulate Hardware* icon. You should see outputs in the Console window as shown below.

```
MATCH: i = 511 j = 509 sw = 0 hw = 0
MATCH: i = 511 j = 510 sw = 0 hw = 0
MATCH: i = 511 j = 511 sw = 0 hw = 0
Result: 262144
RESULT: PASS
C/RTL co-simulation: PASS
```

You can see that the result is the same as before and it still prints out a PASS. Note that the `C/RTL co-simulation: PASS` printed if the main function returns 0. Thus the return value should be based on whether the outputs from the hardware functions are as expected. For this example, we verified that all of the 262144 outputs from the `sobel_filter` function running in hardware matches the golden outputs, hence printing `Result: 262144` and `RESULT: PASS`.

6 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 LegUp, including loop pipelining, exploiting spatial parallelism with Pthreads, and streaming functionality, as applied to a practical example: edge detection in images. In a complex design, you may combine some of the techniques together, such as using loop pipelining with Pthreads, to create a high-performance circuit.

For any questions, please contact us at support@legupcomputing.com.