1 Introduction

This lab will introduce you to high-level synthesis (HLS) concepts using the LegUp HLS tool from the University of Toronto. LegUp is an open-source HLS tool under active development that primarily targets field-programmable gate arrays (FPGAs). LegUp has been downloaded by over 4000 groups around the world and has been as a platform for research on a wide variety of HLS and hardware/software co-design research topics. You can learn more about LegUp at the website:

http://legup.eecg.toronto.edu

You will apply HLS to 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 lab.

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. 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*. Look carefully at the example and be sure you understand it before carrying on with the further steps of this lab.

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 $3 \times 3$ Sobel filters would “slide off” the edge of the input image. In this lab, 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 commonly done.

## 2 Initial Implementation

In this section, we will compile the Sobel filter to hardware, without any modifications to the C code. Change directory to `part1` of the lab:

```bash
cd ~/legup-5.3/examples/sobel_hk_2016/part1
```

Use a text editor to open `sobel.c` and browse through the code. 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 through the filter window. For each pixel of the image that is not in the border, the $3 \times 3$ 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 design on your workstation. This is typical of HLS design, where the designer will verify that the design is functionally correct in software before compiling it to hardware. To do this, type:

```bash
gcc sobel.c -o sobel
./sobel
```

Once you are familiar with the C code and its behaviour, compile the Sobel filter into hardware by typing:

```bash
make
```
2 INITIAL IMPLEMENTATION

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

This command tells LegUp to compile the entire C program into hardware. Several report files and a Verilog file called sobel.v will be generated. Open and scroll through sobel.v. You will notice that the HLS generated Verilog file appears to be very difficult to follow and debug. This is another reason we want to first verify the design in software before compiling to hardware. Near the bottom of sobel.v, you will find a Verilog module called main_tb. This is the testbench used for the simulation of the main Verilog module, top, that implements the sobel filter. 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 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 under the hood to generate hardware from C source code. Start the LegUp visualizer GUI by typing:

```
scheduleviewer scheduling.legup.rpt
```

In the left panel of the 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 LLVM 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 3. The names of the basic blocks in the program are prefixed with BB. It’s 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? Once
you determine which basic block(s) corresponds to the inner-most loop of the `sobel_filter`, double-click it or click on its name on the left-hand panel.

Figure 4 shows the schedule for 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. 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. Observe that the inner pair of loops have been scheduled by LegUp into eight clock cycles, consisting of basic blocks `BB__4`, `BB_preheaderi`, `BB_uslcssa8us_crit_edgei`, and `BB_uslcssa8usi`.

Let’s do a back-of-the-envelope performance analysis: since we observe that the inner-most loop has been scheduled with 8 clock cycles, and since that loop body executes $512 \times 512 = 262,144$ times, we expect the total number of cycles spent executing the hardware to be roughly $262,144 \times 8 = 2,097,152$ cycles. Of course, this estimate does not reflect the overhead operations outside of the inner-most loops, yet it gives us a rough idea of how many cycles are needed for the hardware.

Now let’s simulate the hardware with ModelSim to find out the actual number of cycles needed – the cycle latency. To do this, type:

```
make v
```

You will see various messages printed by ModelSim related to loading simulation models for the Altera hardware. We want to focus on the message near the end which appears something like this:

...
Figure 4: Schedule for a part of the inner-most loop

```plaintext
# PASS!
# At t= 52316630000 clk=1 finish=1 return_val= 0
# Cycles: 2615829
# ** Note: $finish : sobel.v(1807)
```

We see that the simulation took 2,615,829 cycles, which is fairly close to our estimation. 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 the Altera FPGA. Now let’s map the Verilog to the Altera Cyclone V FPGA to obtain information such as the resource usage and the Fmax of this design (i.e. the clock period). This involves two commands. First, type:

```
make p
```

This command creates an Altera project file (top.qsf) which contains the name of the design, the target device, and other constraints. Next, type:

```
make f
```

This second command invokes Quartus II, Altera’s synthesis, placement, routing, and timing analysis tool. You will notice that this command fails at the fitter stage. This is due to the design’s high memory usage, which exceeds the number of available RAM blocks on the DE1-SoC device. To successfully realize this design, let’s change the device number from 5CSEMA5F31C6 to 5CGXFC9E6F35C7, which is a device with more RAM blocks in the Cyclone V family. To do this, use a text editor to open up top.qsf and look for the line:
set_global_assignment -name DEVICE 5CSEMA5F31C6

Replace the device number and run `make f` again.

Once the command completes, we can find the amount of resources this circuit utilized. Open the file `top.fit.rpt`, which is the report produced by Altera’s fitter tool (packing, placement and routing). Scroll down to see text like that shown below. This design used 773 Adaptive Logic Modules (ALMs), 600 Registers, and 1024 RAM blocks.

```plaintext
; Logic utilization (in ALMs) ; 773 / 113,560 ( < 1 % ) ;
; Total registers ; 600 ;
; Total pins ; 39 / 616 ( 6 % ) ;
; Total virtual pins ; 0 ;
; Total block memory bits ; 8,388,608 / 12,492,800 ( 67 % ) ;
; Total RAM Blocks ; 1,024 / 1,220 ( 84 % ) ;
; Total DSP Blocks ; 0 / 342 ( 0 % ) ;
```

You can also view the speed performance of the design, after its complete implementation on the Cyclone V. Open `top.sta.rpt`, which is the report produced by Altera’s timing analysis tool. Scroll down to see a table like that shown below. This circuit can operate at 74.52MHz on the Cyclone V.

```plaintext
+-------------------------------------------------+
; Slow 1100mV 85C Model Fmax Summary ;
+-------------------------------------------------+
; Fmax ; Restricted Fmax ; Clock Name ; Note ;
+-------------------------------------------------+
; 74.52 MHz ; 74.52 MHz ; clk ; ;
```

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 2,615,829 and the clock period was 13.4ns. The wall-clock time of our implementation is therefore 2615829 × 13.4 = 35,102μs.

At the end of this handout, you will find a blank table. Fill in the results for this part of the lab under the column labelled “Basic”.

### 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 5 shows an example of loop pipelining. Figure 5(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 5(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.

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. To do this, we modify the C code slightly so that it has a single outer loop that iterates over the entire image. Change directory to part2 of the lab and look at the new `sobel_filter` function to make sure you understand the modifications.

cd ~/legup-5.3/examples/sobel_hk_2016/part2

To invoke loop pipelining, you must tell LegUp which loop you wish to pipeline, which consists of two steps. First, open `sobel.c` and add a label `lp:` before the for loop to pipeline. After your modifications, the relevant line of C should appear like this:

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

Next, open the configuration file called `config.tcl` and add the following line below `source ..//config.tcl`, which tells LegUp to pipeline the loop:

```
loop_pipeline "lp"
```

Having made those modifications, you can now synthesize the design by typing:

```
make
```

To see the results produced by loop pipelining, check the log file produced by LegUp called: `pipelining.rtl.legup.rpt` to find the following:
Changing state name of 'LEGUP_F_main_BB_preheaderi' to 'LEGUP_loop_pipeline_wait_lp_1'
Found 1 to pipeline
Generating Loop Pipeline for label: "lp_1"
BB: %.preheader.i
II: 4
Time: 8
maxStage: 1
Induction var:  %indvar7 = phi i32 [ %58, %.preheader.i ], [ 0, %0 ],
!legup.canonical_induction !4, !legup.pipeline.start_time !3,
!legup.pipeline.avail_time !3, !legup.pipeline.stage !3, !MSB !5, !LSB !3,
!extendFrom !5
Label: lp_1
Constant tripCount: 260100
Generating datapath for loop pipeline state: LEGUP_loop_pipeline_wait_lp_1_2

Observe that an II of 4 was achieved for the loop, and that the pipeline length is 8. Take a
moment to think about what are some potential reasons why the II cannot be 1 in this case. Are
there any loop-carried dependencies? What about resource contentions? In this particular case,
the limiting factor was memory port contention. In order to calculate the convolution between the
input image and the two matrices \(G_x\) and \(G_y\), 8 loads from memory are needed. Since this design
uses a dual ported ROM, there can only be 2 loads from the same memory within a cycle. This
results in a \(8 / 2 = 4\) cycle delay between successive iterations of the loop.

A better way to understand the effect of loop pipelining is to visualize it. Open up the GUI
again, this time with an option that allows you to view the pipelined schedule:

scheduleviewer scheduling.legup.rpt -p pipelining.legup.rpt

In the CFG, you will see a basic block called wait_lp_1. Double-click it to reveal the loop
pipeline schedule, similar to that shown in Figure 6. Here, you can see that the II of the loop is 4,
and that the length of the pipeline is 8 cycles. The dark black rectangle illustrates what the pipeline
looks like in the steady state. In the steady state, two iterations of the loop are “in flight” at once.

Since we observe that the loop II is 4, and the loop executes \(512 \times 512 = 262,144\) times,
the total time spent in the inner-most loop is roughly \(262,144 \times 4 = 1048576\) cycles. This is an
approximation, as it does not include the initial time to fill the pipeline, nor does it include the time
to flush the pipeline for each pixel. To check our estimate, simulate the design with ModelSim:

make v

You should see results similar to this:

...  
# PASS!  
# At t=  36434670000  clk=1  finish=1  return_val= 0  
# Cycles:  1821731  
# ** Note: $finish : sobel.v(1984)

8  Hsiao, Chen, Lian, Anderson ©2016
Observe that loop pipelining has dramatically improved the cycle latency for the design, reducing it from 2,615,829 cycles to 1,821,731 cycles in total. Finally, use Altera’s Quartus II tool to map the design onto the Cyclone V FPGA:

```
make p
make f
```

Remember, since we didn’t reduce the memory bits required for this design, we still need to change the device number from 5CSEMA5F31C6 to 5CGXFC9E6F35C7 for the design to fit. Extract the FPGA speed (FMax) and area data from the files top.sta.rpt and top.fit.rpt and complete the “Pipelining” column in the table at the end of this handout. Compare the data for the implementations without and with loop pipelining. Are the results as expected? Can you think of ways to improve the results further?

### 4 Synthesizing Parallel Hardware with Pthreads

In this section, you will implement a multithreaded 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.

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.

In the part3 subdirectory of the lab, you will find a file sobel.c which includes 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.

The size of the computation has been shortened for the sake of simulation time in this lab.

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 8 simultaneous threads for the Sobel filter, where each thread is responsible for computing 1/8th of the rows in the output image.

Let’s first compile the software version to ensure that the Pthread version of the sobel filter produces correct results:

```
gcc -pthread sobel.c -o sobel
```

Now, run the executable as follows and ensure that the error checking passes:

```
./sobel
```

Now, compile the design to hardware by typing make parallel in the directory and simulate by typing make v. You should see output similar to this:

```
...  
# MATCH: i = 511 j = 12 sw = 0 hw = 0
# MATCH: i = 511 j = 13 sw = 0 hw = 0
# MATCH: i = 511 j = 14 sw = 0 hw = 0
# MATCH: i = 511 j = 15 sw = 0 hw = 0
# Result: 8192
# RESULT: PASS
# At t= 2056330000 clk=1 finish=1 return_val= 0
# Cycles: 102814
# ** Note: $finish : sobel.v(4400)
...  
```

The cycle-count performance of the parallelized code is very poor. In fact, this parallelized version with 8 threads is worse than the unparallelized version. The cycle count of the full, un-shortened version of the benchmark is 3,393,863.

To understand why, open up the LegUp generated memory report memory.legup.rpt. At the bottom of the report, you will find a summary similar to this:
Final memory allocation:
Global Memories:
   RAM: main_entry_args Function: main
Local Memories:
   ROM: golden_output Function: main
   RAM: main_entry_threads Function: main
Shared Local Memories:
   ROM: gy Function: sf_sw
   ROM: gx Function: sf_sw
   RAM: sobel_output Function: sf_sw main
   ROM: _01LC0 Function: sf_sw
Local Constant Memories:
Processor Memories:

Under Shared Local Memories you will see the read-only memory named _01LC0, 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, however they are different from global memory in that they do not need to be accessed through a global memory controller – i.e. the functions which share the memories are known at compile time. An arbiter is necessary to arbitrate access between the multiple sharing functions when they compete for access.

You may notice that even though at a particular cycle, the accesses to the input array require different elements of the array, the accesses still need to arbitrate for access since LegUp instantiates dual ported RAMs/ROMs so at most only 2 accesses can be completed in a given cycle.

You may also notice that some of these shared memories are read-only, meaning they are never modified and so we can actually create copies of them local to each function. To do this, open the config.tcl file and uncomment the line set parameter REPPLICATE_ROMS 1. This will create a replicated copy of the ROM as a Local Constant Memory.

Make the change to config.tcl and compile the design again. Check in the memory.legup.rpt that you see the ROMs being replicated.

Now, simulate the design once more. You should have improved results.
Experiment with the thread count to observe its effect on the performance. What thread count potentially produces the most performant result? After you’ve experimented with the thread count, synthesize the design as before. Remember to change the device name to 5CGXFC9E6F35C7. What are the results?

In the table at the end of this handout, enter the results observed in the “Pthreads (part3)” column.

When trying to synthesize the previous design, you will most likely find that your design does not fit on the device. Looking in the top.fit.rpt in the Fitter Summary section, you will see that the memory bits required exceed the total memory bits on the device. As you remember, you had set the parameter which allows for constant read-only memory to be copied locally for each thread in order to reduce memory port contention and improve performance. The Fitter Resource Utilization by Entity section of the top.fit.rpt will give a more detailed breakdown of the usage. You will see that each thread contains an entity _01LC0, as described before, this is the input image. Take a look in the report at how many memory bits the input image requires. What is the maximum number of copies of the input image/threads that the device can support?

You might have noticed that the previous design did not fully utilize the replicated memory, meaning each thread only accesses a limited portion of the fully replicated input image. Therefore, we could implement a design in which each thread only creates a local copy of the input image for the portion that it actually requires.

Navigate to the part4 subdirectory of the lab. Here, you will find a multithreaded implementation of the sobel filter which uses memory more efficiently. In this version, instead of each thread containing its own local copy of the input image, each thread will copy over the rows that it needs to perform the computations for the final image. Even though each hardware thread will compete for access when copying over data to the local image matrix, the contention is amortized by the fact that each thread normally makes $9 \times$ the accesses for the computation itself.

**The size of the computation has been shortened for the sake of simulation time in this lab.**

Let’s first compile the software version to ensure that this new Pthread version of the sobel filter produces correct results:

```
gcc -pthread sobel.c -o sobel
```

Now, run the executable as follows and ensure that the error checking passes:

```
./sobel
```

Now, compile the design to hardware by typing make parallel in the directory and simulate by typing make v. You should see output similar to this:

```
# MATCH: i = 511 j = 13 sw = 0 hw = 0
# MATCH: i = 511 j = 14 sw = 0 hw = 0
# MATCH: i = 511 j = 15 sw = 0 hw = 0
# Result: 8192
```
Again, the default cycle-count performance of the parallelized code is very poor. Running the full, unshortened benchmark leads to a total cycle count of 2,853,623. However, since we do not make local copies of the entire input image, memory will not be as limiting for a more performant design.

Experiment with the thread count to observe its effect on the performance. What thread count potentially leads to the most performant and synthesizable result? In the table at the end of this handout, enter the results observed in the “Pthreads (part4)” column.

5 Streaming Hardware Synthesis

The final hardware implementation you will realize with 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 lab 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 lab, 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 $512 \times 512$ 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 7 shows the $3 \times 3$ 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”.

In the part5 subdirectory of the lab, you will find a file sobel.c. Open up the file and you will find a function sf_window_3x3_and_line_buffer with the following signature:
Figure 7: Motivation for use of line buffers.

Figure 8: Streaming hardware circuit with FIFO queues between components.

```c
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 $3 \times 3$ 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 8. 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 HLS 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_filter.c` file. 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 streaming.h header.

```c
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 lab. 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 lab.

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 10-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 × WIDTH times. Finally, the output values are checked for correctness and PASS or FAIL is reported.

Let’s compile the code and run it in software on the workstation.

```bash
gcc sobel.c ../../../legup-library/legup/streaming.c -o sobel
./sobel
```

After executing the commands above, 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 some constraints that are specific to LegUp synthesis of streaming hardware. Open the constraints file config.tcl to see the following commands:

```tcl
function_pipeline sobel_filter -ii 1
set_custom_top_level_module sobel_filter
set_custom_test_bench_module streaming_tb
set_combine_basicblock 3
```

The function_pipeline constraint tells LegUp HLS that the sobel_filter function is intended to be a streaming kernel and that the target initiation interval is 1 (-ii 1). The second and third
commands relate to synthesis with Quartus and simulation. Unlike the previous parts of the lab, automatic testbench generation for streaming kernels is not yet implemented within LegUp HLS. The second command specifies that the kernel itself should be the top-level Verilog module. The third command specifies a testbench for the kernel. The last command above turns on “if-conversion”, which removes if/else control flow in the streaming kernel’s software implementation. This is required so that a simple pipeline, with fixed latency, can be created for the Sobel streaming kernel.

Although it is beyond the scope of the lab, you are encouraged to take a look at the custom testbench that exercises the streaming kernel, which is found in `streaming_tb.v`. The kernel is instantiated at about line 34. Line 53/54 declare arrays to hold the input data and golden output data for the image. Lines 61/62 read the input and golden output from `.dat` files in the same directory. You can take a look inside these files to see the pixel values in hexadecimal format.

Now synthesize the hardware by typing `make`. Then, simulate the streaming hardware using the special testbench by typing `make v`. 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  262658: output matches expected value,  0 ==  0  
# At cycle  262659: output matches expected value,  0 ==  0  
# At cycle  262660: output matches expected value,  0 ==  0  
# At cycle  262661: output matches expected value,  0 ==  0  
# At cycle  262662: output matches expected value,  0 ==  0  
# PASS  
# ** Note: $finish : streaming_tb.v(143)  
# Time: 5253290 ns  Iteration: 1  Instance: /streaming_tb
```

The total number of clock cycles is about 263,000, 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 Cyclone V FPGA by typing `make p` and then `make f`. Complete the column called “Streaming” in the table in Section 6 of this lab. The area values for the streaming implementation are significantly smaller than in the prior parts of this lab. 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.
6 Performance and Area Results

Table 1: Results worksheet.

<table>
<thead>
<tr>
<th></th>
<th>Basic</th>
<th>Pipelining</th>
<th>Pthreads (part 3)</th>
<th>Pthreads (part 4)</th>
<th>Streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock period</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wall-clock time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALMs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSP blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7 Questions for Class Discussion

1. High-level synthesis (HLS) is becoming more popular as a digital circuit design methodology. Name three advantages of HLS.

2. Give an example of an application for which HLS would most likely not be a suitable design methodology?

3. What are the limitations to parallelizing in hardware? How do they compare to parallelization in software?

4. Do you think this sobel filter benchmark is a suitable application for multithreaded hardware? Why or why not? List some other examples of applications that may be ideal for multithreaded hardware.
5. Do you think the coding style of the C source code plays a large role in determining the quality of results of the final synthesized design? If so, can you think of some coding strategies / guidelines for each category of benchmarks?

6. In this lab, we showed three different ways to parallelize the hardware, can you think of any others? How do they compare with the methods mentioned in this lab?

8 Summary

High-level synthesis is gaining traction as a design methodology, as it allows hardware to be designed at a higher level of abstraction, lowering design time and cost. In this lab, you have gained experience with several key high-level synthesis concepts, including loop pipelining, exploiting spatial parallelism, and streaming functionality, as applied to a practical example: edge detection in images. Several key research challenges for HLS remain: 1) continued efforts to raise the quality of HLS hardware to bring it as close as possible to human-crafted hardware quality; 2) development of automated transformations to eliminate the need for engineers to “tune” their coding style to meet tool expectations; and 3) the development of software-like debugging and profiling tools for HLS-generated hardware.