LegUp HLS Tutorial for Lattice ECP5
Sobel Filtering for Image Edge Detection

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). 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. 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

Figure 1: Sobel filtering example.

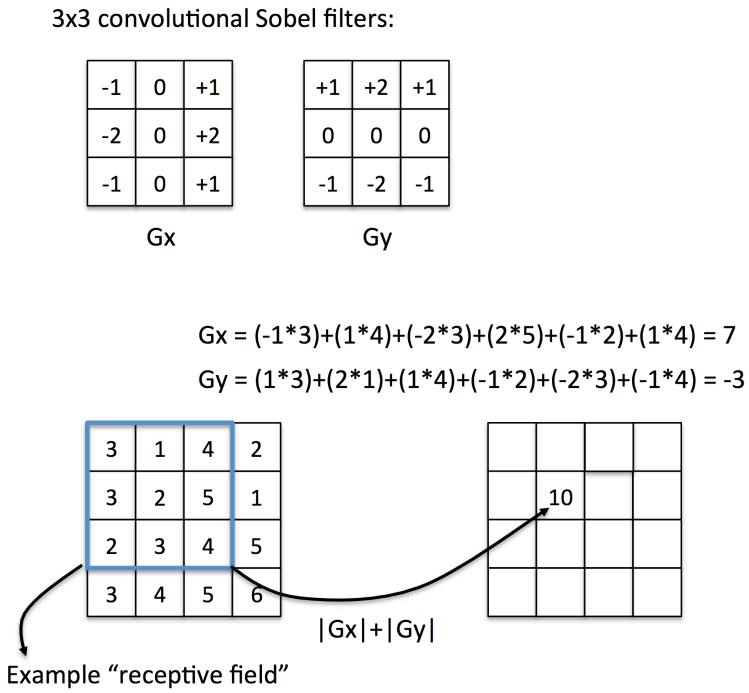

Figure 2: Sobel filters and computational action to compute one pixel in the output image from an input image.

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.

The sobel tutorial files can be found on our github at `https://github.com/LegUpComputing/legup-examples/tree/master/tutorials/sobel`. To download all of the files at once, you can go to `https://github.com/LegUpComputing/legup-examples` and clone/download the entire repository.

1 Basic Implementation

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

Start the LegUp GUI. On Windows, this can be done by double-clicking on the LegUp shortcut.

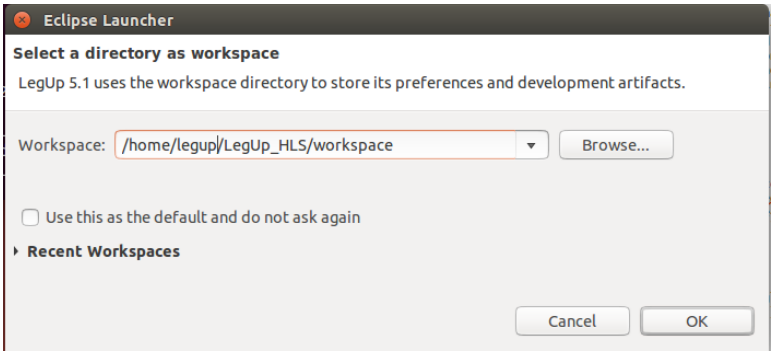

Figure 3: Choosing a workspace.

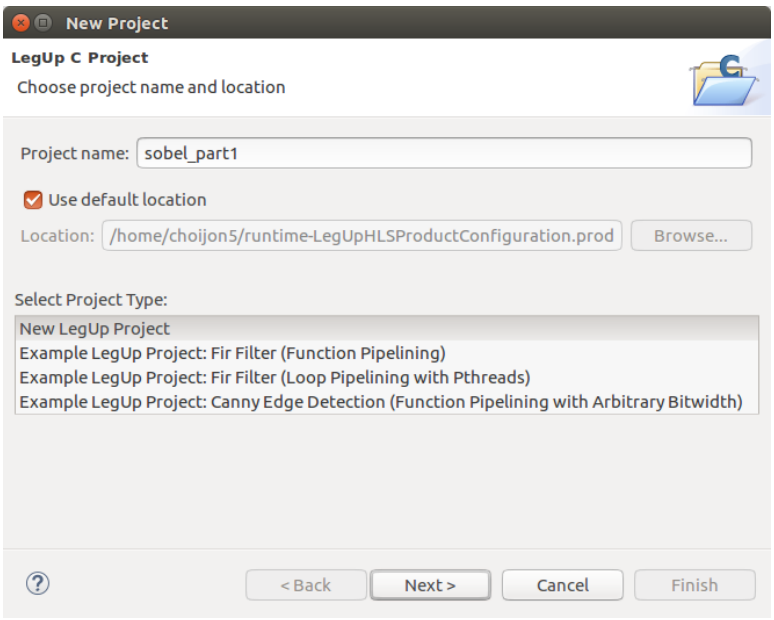

Figure 4: Creating a new project.

On Linux, make sure that `$(LEGUP_INSTALLATION_DIR)/legup/bin` is on your PATH and the GUI can be opened by 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 project*. You will then see a window as shown in Figure 4. Type a name for your project as shown: `sobel_part1`. Then click on *Next*.

Now, specify the top-level function of the project as `sobel_filter` (write it in the dialog box next to `Top-Level Function`. Then 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 the tutorial files, and go into the *part1* directory. Be sure to add both `.h` files and the `.c` file. After you have added the source files to the project, click on *Next*.

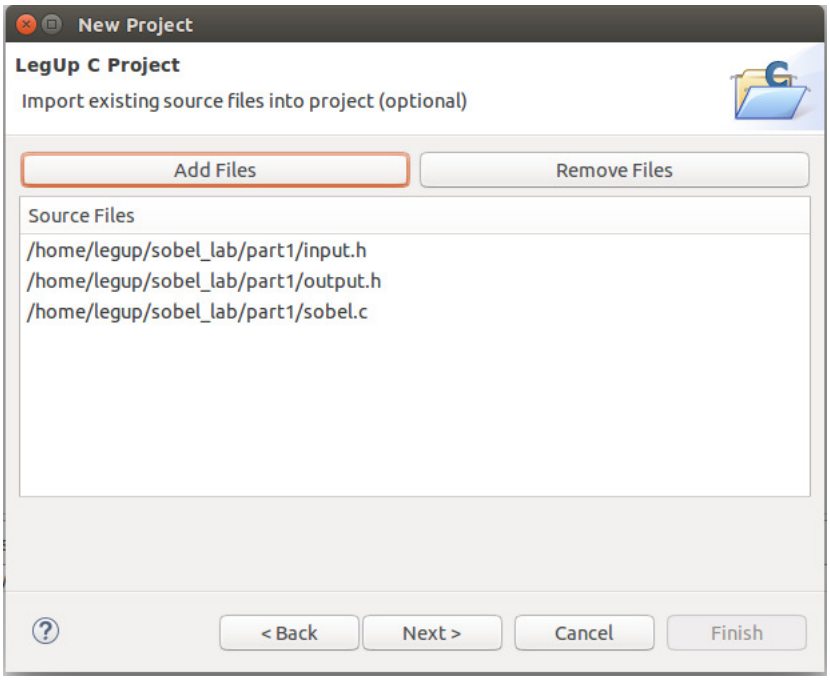

Figure 5: Adding the source files.

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*, without changing any of the options. 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 Lattice and ECP5. For *FPGA Device*, you have an option to choose LFESUM-85F-6BG756C on the 85F Board or use another ECP5 device that is not listed by selecting Custom Device and filling in the part name in the *Custom Device* field. For this tutorial, we will use the listed device. 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**, 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 main 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

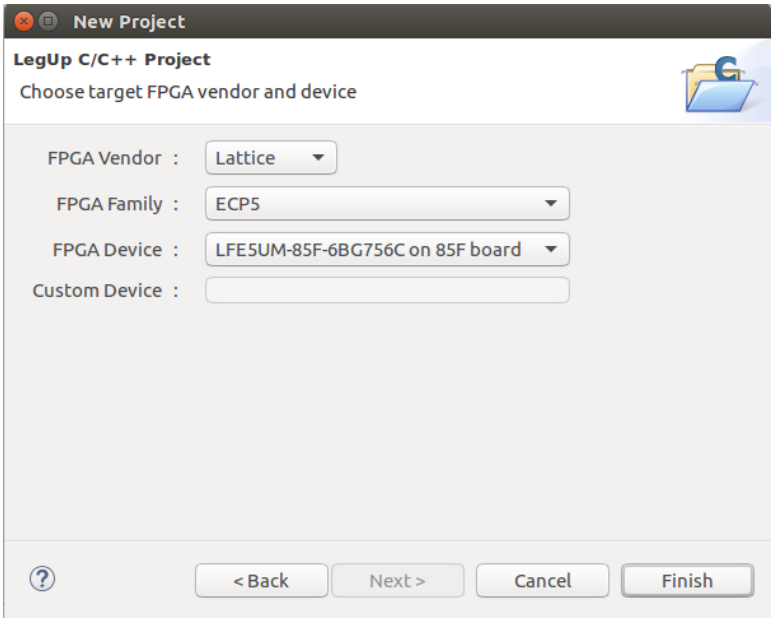

Figure 6: Selecting the FPGA vendor and device.

Figure 7: LegUp GUI toolbar icons.

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 3×3 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*

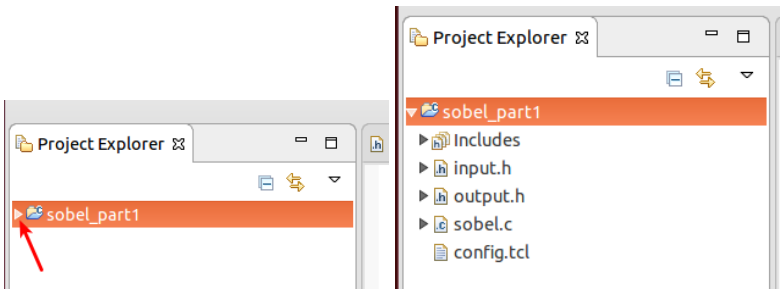

Figure 8: Explorer for browsing source and reports.

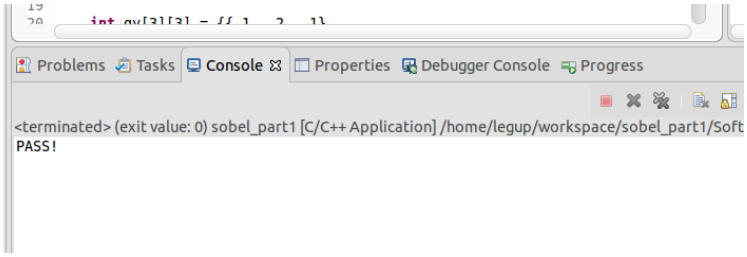

Figure 9: Console after software execution.

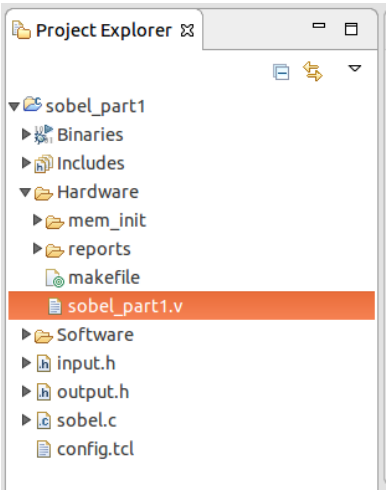

Figure 10: Finding the LegUp-generated Verilog in the Project Explorer.

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 invokes LegUp to compile the `sobel_filter` function into hardware, as we had specified it to be the top-level function of the project. If the top-level function calls descendant functions, all descendants functions are also compiled to hardware.

When the compilation finishes, a report file (summary.legup.rpt) opens up, which shows the number of cycles scheduled for each basic block of a function as well as the memories that are used in the hardware. In this example, 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 memories are listed as `I/O Memories`.

You can also visualize the schedule and control-flow of the hardware using our schedule viewer GUI. Start the schedule viewer 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 of each function. In this example, there is only one function that was compiled to hardware, `sobel_filter`, hence you only see the one function in the Explorer pane and also in the Call Graph pane on the right. Double-click on the `sobel_filter` function in the call-graph pane and you will see the control-flow graph (CFG) for the function, similar to Figure 11. The names of

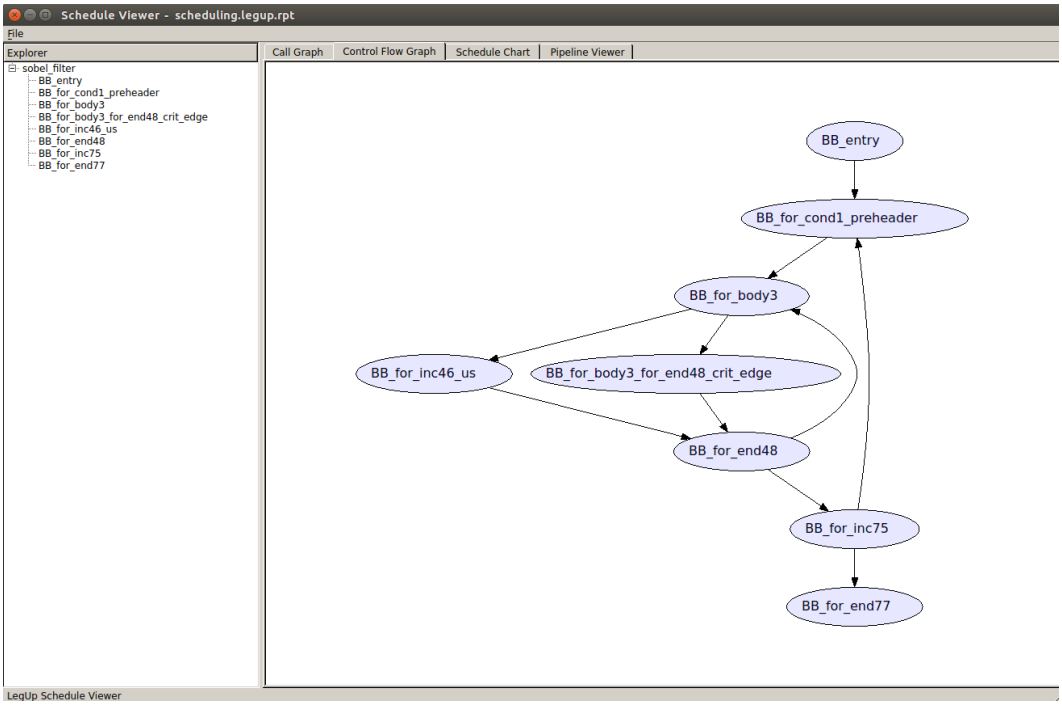

Figure 11: Control-Flow Graph for the Sobel filter.

the basic blocks in the program are prefixed with BB (note that the names of basic blocks may be slightly different depending on the version LegUp you use, but it should still show the same control graph structure). It is difficult to determine how the names of the basic blocks relate to the original C code; however, you can observe that there are two loops in the control-flow graph, which correspond to the two outer most loops in the C code for the sobel filter function. The inner loop consists of basic blocks `BB_for_body3`, `BB_for_inc46_us`, `BB_for_body3_for_end48_crit_edge`, and `BB_for_end48`. Try double clicking on `BB_for_inc46_us` (if the basic block names are different from the figure, click on the left-most basic block).

Figure 12 shows the schedule for `BB_for_inc46_us`, 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 11 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 and outputs of each instruction become coloured. 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, among others.

Now let's simulate the Verilog RTL hardware with ModelSim to find out the number of cycles needed to execute the circuit – the cycle latency. Close the viewer first, then click the *SW/HW Co-Simulation* icon on the toolbar. This will simulate the generated hardware, `sobel_filter`, in RTL using ModelSim, while running the rest of the program, `main`, in software. This flow allows one to simulate LegUp-generated hardware without having to provide a custom RTL testbench.

If this is the first time you are using LegUp, you will need to set up the paths to Modelsim (and Lattice Diamond 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 Simu-*

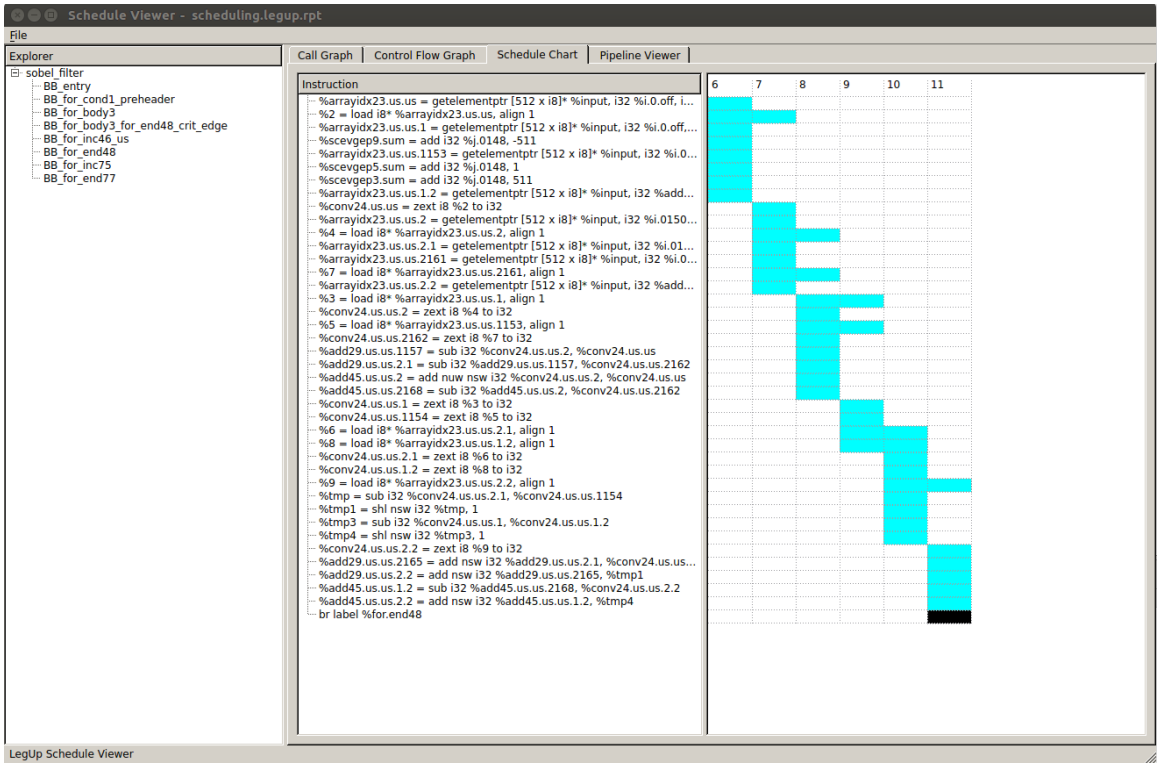

Figure 12: Schedule for the inner-most loop

lator and *Lattice Diamond*. 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 Lattice. Click *OK*, then click on *SW/HW Co-Simulation* again. In the *Console* window, you will see various messages printed by ModelSim related to loading simulation models for the hardware. It may take a few minutes to simulate. We want to focus on the message near the end which appears something like this:

```
...
# Cycle latency:      3136535
# C/RTL co-simulation result: PASS
# ** Note: $finish    : ../simulation/cosim_tb.v(433)
#    Time: 78459370 ns  Iteration: 1  Instance: /cosim_tb
# Errors: 0, Warnings: 0
Info: Verifying RTL simulation
PASS!
C/RTL co-simulation: PASS
```

We see that the simulation took 3,136,535 cycles. Also observe that simulation printed "C/RTL co-simulation: PASS!". The co-simulation uses the return value from the main function to determine whether or not the co-simulation has passed, hence you should structure your software testbench in such a way that it will return 0 if the results are as expected and otherwise return a non-zero value. In the software testbench for this example (see the `main` function), you can see

that after calling the top-level function, it walks through each outputted value from the top-level function and verifies it against the expected value. A counter is incremented if a value is not as expected and this counter value is returned at the end, hence if all values are as expected, the testbench will return 0.

The simulation above verifies the functionality of the generated hardware. Now let's map the Verilog to the Lattice ECP5 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 Diamond to create a Diamond project and synthesize the LegUp design to the Lattice FPGA. Diamond is the name of Diamond's 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 Diamond 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 Diamond you are using. This tutorial used Diamond v3.11.

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

Maximum clock frequency: 145.138 MHz.

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

Number of registers:    526 out of 84735 (1%)
Number of SLICEs:       556 out of 41820 (1%)
Number of LUT4s:        855 out of 83640 (1%)
Number of block RAMs:  0 out of 208 (0%)
Number of Used DSP MULT Sites:  0 out of 312 (0 %)
Number of Used DSP ALU Sites:  0 out of 156 (0 %)
Number of Used DSP PRADD Sites:  0 out of 312 (0 %)
Number of MULT18X18D: 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 3,136,535 and the Fmax is 145.138 MHz (clock period of 6.9ns). The wall-clock time of our implementation is therefore 3,136,535 $\times$ 6.9 = 21.6 ms.

2 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

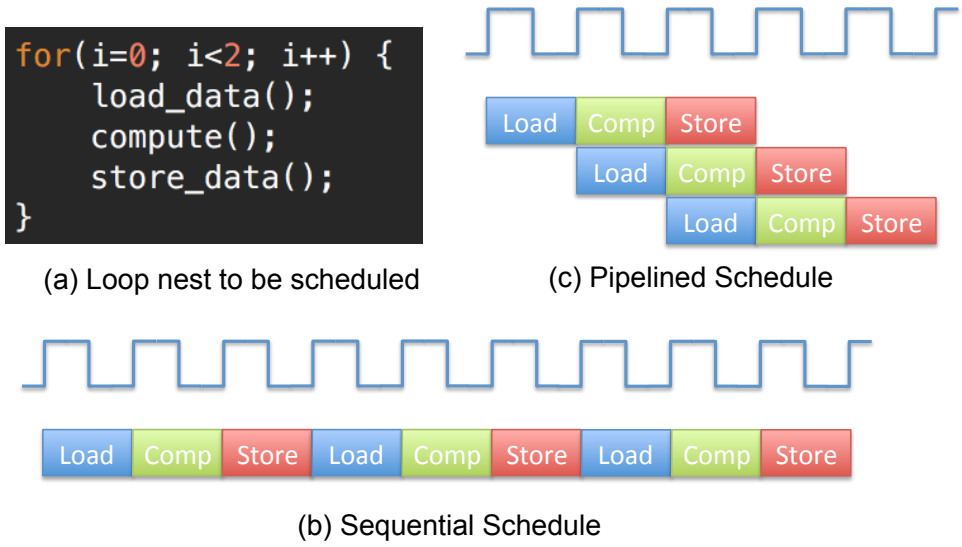

Figure 13: Loop Pipelining Example

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 (a new loop iteration can start every 3 clock cycles), and it takes 9 clock cycles for the final write to be 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 5 clock cycles for the final write to be 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 (specify the top-level function as `sobel_filter`, include the source files for part 2, and target ECP5), create a new LegUp project for part 2. Once the project is created, open `sobel.c` from 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 and G_y, we would like to pipeline the calculation of each pixel. To invoke loop pipelining, you need to 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 `loop:`, as shown below.

```
loop: 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 `loop`, 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* to close the dialog window.

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 see messages like the following:

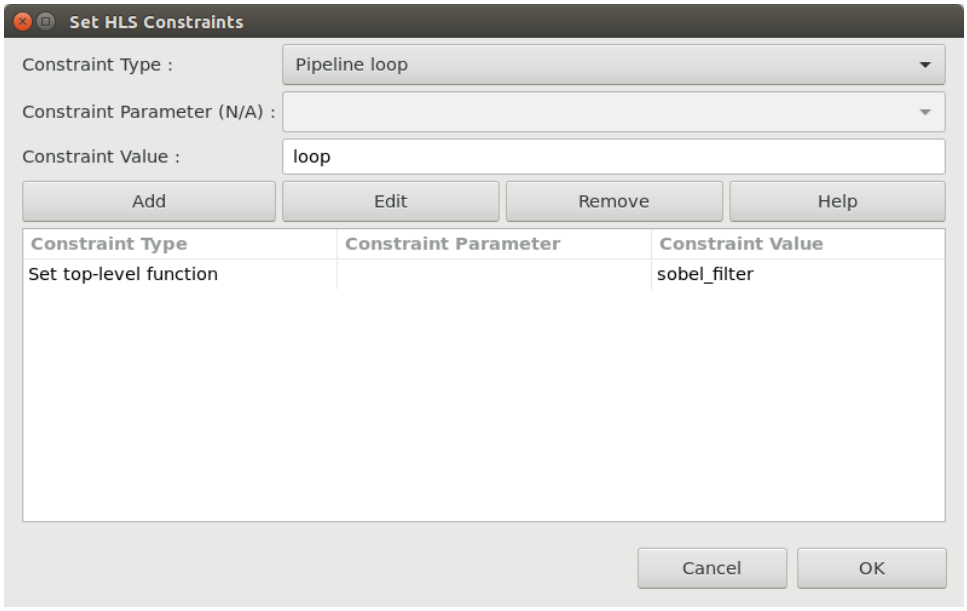

Figure 14: HLS constraints editor.

```
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 use our schedule viewer. Click on the *Launch Schedule Viewer* icon again. Double-click on `sobel_filter`, then in the CFG, you will see a basic block called `BB_for_body`. 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

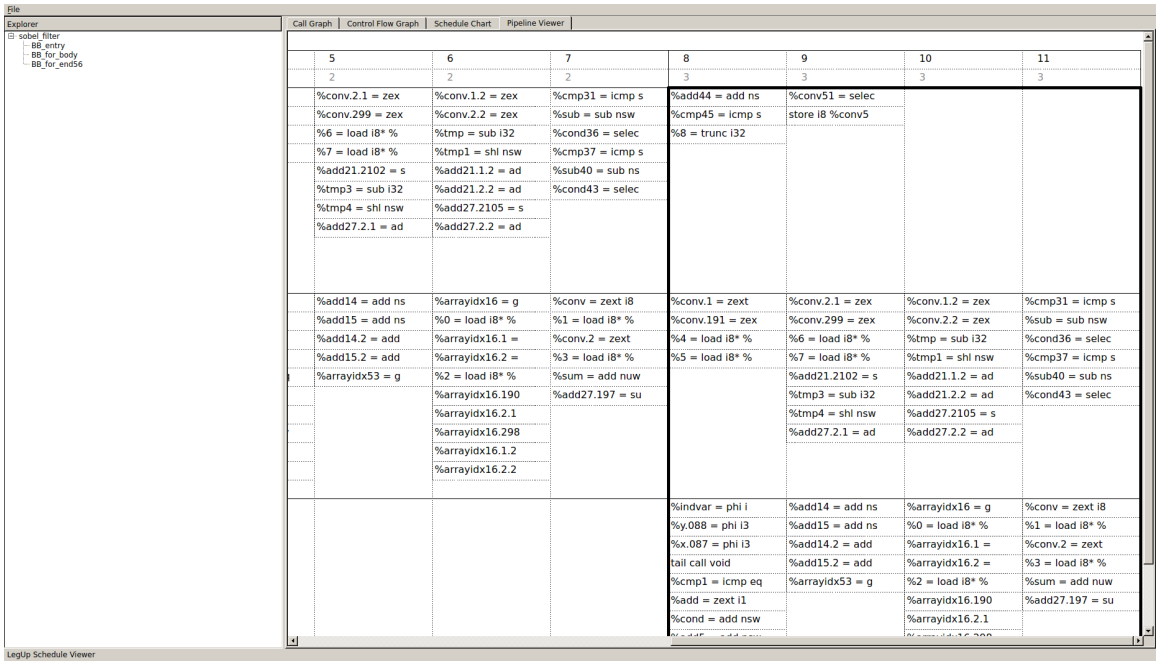

Figure 15: Loop Pipeline Schedule for the Sobel filter.

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 *SW/HW Co-Simulation* icon on the toolbar. You should see results similar to this:

```
...
# Cycle latency:      1040412
# C/RTL co-simulation result: PASS
# ** Note: $finish    : ../simulation/cosim_tb.v(433)
#    Time: 36536930 ns  Iteration: 1  Instance: /cosim_tb
# Errors: 0, Warnings: 0
Info: Verifying RTL simulation
PASS!
C/RTL co-simulation: PASS
```

Observe that loop pipelining has dramatically improved the cycle latency for the design, reducing it from 3,136,535 cycles to 1,040,412 cycles in total. Finally, use Lattice's Diamond to map the design onto the ECP5 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.

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

Maximum clock frequency: 100.857 MHz.

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

Number of registers:    619 out of 84735 (1%)
Number of SLICEs:       613 out of 41820 (1%)
Number of LUT4s:        929 out of 83640 (1%)
Number of block RAMs:  0 out of 208 (0%)
Number of Used DSP MULT Sites:  0 out of 312 (0 %)
Number of Used DSP ALU Sites:  0 out of 156 (0 %)
Number of Used DSP PRADD Sites:  0 out of 312 (0 %)
Number of MULT18X18D: 0
```

3 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 16 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

Figure 16: Motivation for use of line buffers.

the input image. The memory used to store the two rows are called "line buffers".

Create a new LegUp project for part 3 of the tutorial and include all of the `.cpp` and `.h` files for part 3. Again, specify the `Top-Level Function` as `sobel_filter`. Specify Lattice's ECP5 device and finish creating the project. Examine the `sobel.cpp` file in the project viewer and you will find a function `sf_window_3x3_and_line_buffer` with the following function prototype:

```
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 17. 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.cpp` file at line 48. 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 `legup/streaming.hpp` header.

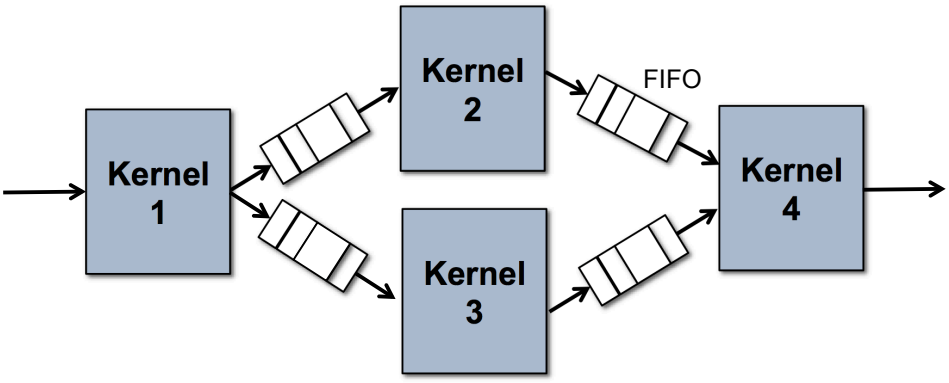

Figure 17: Streaming hardware circuit with FIFO queues between components.

```
void sobel_filter(FIFO<unsigned char> &input_fifo,
                  FIFO<unsigned char> &output_fifo) {

    unsigned char input_pixel = input_fifo.read();

    ...

    output_fifo.write(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).

In the `main` function in `sobel.cpp`, you will that FIFOs are declared in the beginning. The FIFO class takes in its data type as the template parameter and also receives the depth of FIFO as the constructor argument. In this case, the FIFOs are declared to have the `unsigned char` data type to create 8-bit wide FIFOs.

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, with the main function return 0 if the output values are as expected.

Click the 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 pipeline function constraint as shown in Figure 18. The `Pipeline Function` constraint tells LegUp that the `sobel_filter` function is intended to be a streaming kernel. You should see that the top-level function has already been set,

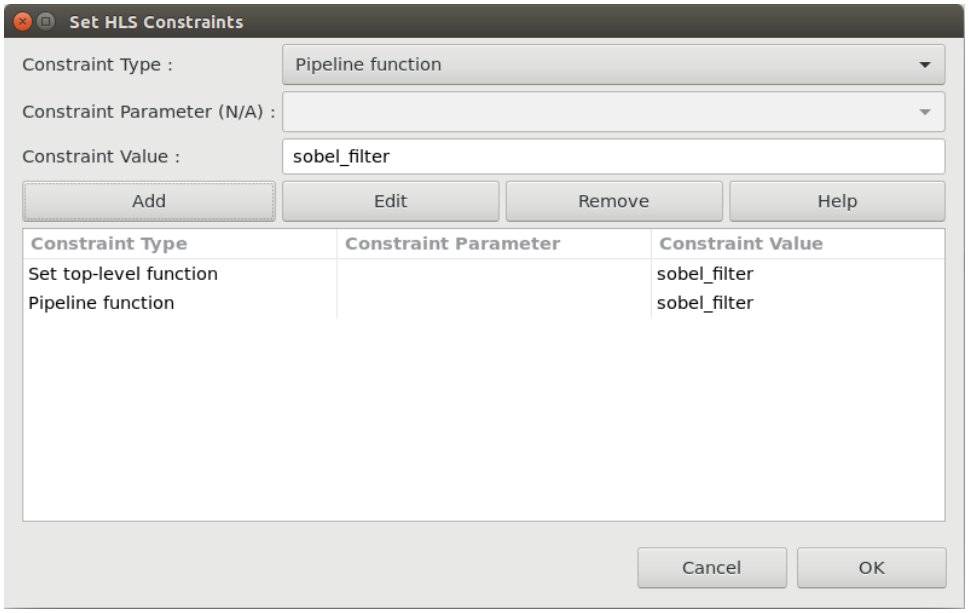

Figure 18: Pipeline Function Constraint.

since you specified it when creating the project. Once you have set all the constraint as shown in Figure 18, click *OK* to close the constraints editor.

Now synthesize the hardware by clicking the *Compile Software to Hardware* icon. In the report file (summary.legup.rpt) that opens up, you should see under `Pipeline Result` that the `sobel_filter` function is pipelined and its initiation interval is 1.

```
====== 2. Pipeline Result ======

+--------------+-------------+-------------------------+---------------------+
| Function     | Basic Block | Location in Source Code | Initiation Interval |
+--------------+-------------+-------------------------+---------------------+
| sobel_filter | %entry      | line 48 of sobel.cpp    | 1                   |
+--------------+-------------+-------------------------+---------------------+
```

This circuit has memories inside the hardware (see `Local Memories` under `Memory Usage`) due to the line buffers and the counters that are used.

Simulate the streaming hardware by clicking the *SW/HW Co-Simulation* icon. You will see scrolling output, reporting the computed and expected pixel value at each clock cycle. During this run, you should see messages like this:

```
...
# Number of calls:      262658
# Cycle latency:      262668
# C/RTL co-simulation result: PASS
# ** Note: $finish    : ../simulation/cosim_tb.v(207)
#    Time: 5253390 ns  Iteration: 1  Instance: /cosim_tb
```

The total number of clock cycles is about 262,668, 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. At the end of the co-simulation, you should see that the co-simulation has passed.

```
Result: 262144
RESULT: PASS
C/RTL co-simulation: PASS
```

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 ======

Maximum clock frequency: 103.477 MHz.

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

Number of registers:    450 out of 84735 (1%)
Number of SLICEs:       451 out of 41820 (1%)
Number of LUT4s:        671 out of 83640 (1%)
Number of block RAMs:  2 out of 208 (1%)
Number of Used DSP MULT Sites:  0 out of 312 (0 %)
Number of Used DSP ALU Sites:  0 out of 156 (0 %)
Number of Used DSP PRADD Sites:  0 out of 312 (0 %)
Number of MULT18X18D: 0
```

LegUp 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 LegUp 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 (may also have larger area due to inserting more registers), and a higher clock period constraint leads to lower Fmax but can also have less area. The clock period constraint can be set in the HLS Constraints dialog as shown in Figure 19.

If the target clock period constraint is not provided by the user, as in this tutorial, LegUp 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 Lattice ECP5 FPGA. Try changing clock period constraint and synthesizing the generated circuit with Lattice Diamond again to examine the impact of the clock period constraint on the generated circuit.

4 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 and streaming/function pipelining, as applied

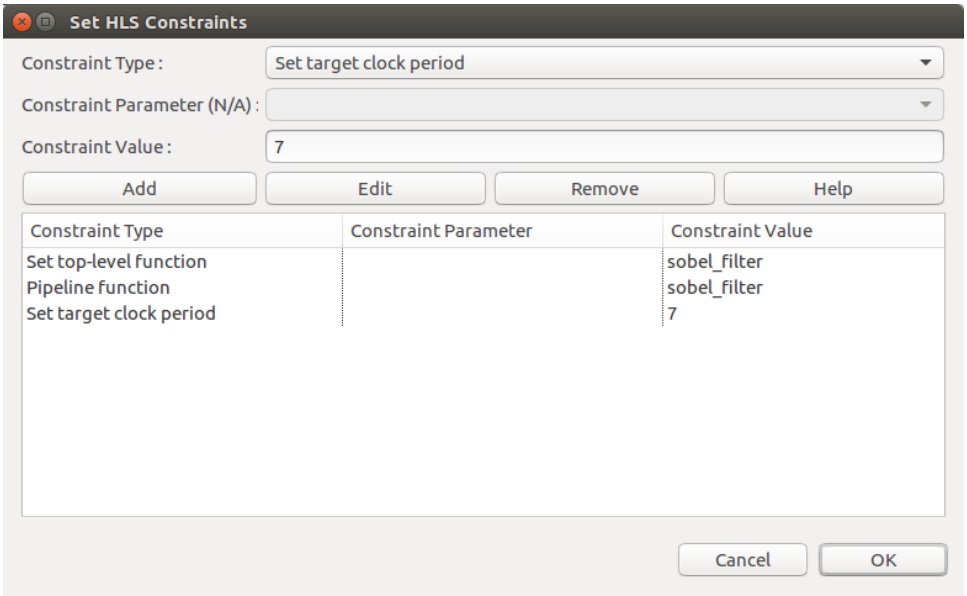

Figure 19: Setting the target clock period HLS constraint.

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 support@legupcomputing.com.