ECE1756 Assignment 2

Image Convolution on FPGAs, CPUs, and GPUs

"Our beauty lies in this extended capacity for Convolution."

— Thomas Pynchon

Assigned on Thursday, October 12 **Total marks = 18/100**

Due on Thursday, November 2 @ 11:59 pm

1 Objectives

This assignment is intended to help you:

1	Gain basic familiarity with the convolution operation.

- An important operation in many communications and signal processing applications
- One of the main building blocks in many of today's deep learning models

2	Dive deeper into FPGA design development and optimization.

- Build a high-performance 2D convolution engine on an FPGA
- Optimize your implementation for better throughput

3	Gain basic familiarity with the operation and performance of CPUs and GPUs when implementing 2D convolution.

- Understand provided CPU and GPU codes
- Measure performance of different implementations using different compiler optimization settings

4	Learn how to perform a rigorous efficiency comparison between different computing platforms for a specific application.

- Evaluate the performance per Watt and performance per unit area for CPU, GPU, and FPGA solutions for the studied application

2 Background

Convolution is one of the most commonly used operations in digital signal processing. It is used to change the frequency characteristics of a signal by convolving it with the impulse response of a specific *filter* that has the desired properties. In the case of one dimensional signals, convolution can be expressed as the weighted sum of a local neighborhood of size N signal samples as shown in eq.(1), where $x[i]$ and $b[i]$ are the i^{th} signal sample and filter coefficient respectively. For one-dimensional signals, this convolution operation is often called a finite impulse response (FIR) filter and it is widely used in communication systems such as wireless base stations and in audio systems.

$$y[n] = b[0]\ x[n] + b[1]\ x[n-1] + ... + b[N]\ x[n-N] = \sum_{i=0}^{N} b[i]\ x[n-i] \tag{1}$$

This concept can be extended to multi-dimensional signals as well. For example, in image (i.e. two dimensional signal) processing, convolution between an image and a two dimensional $N \times N$ filter is typically used to modify the spatial frequency characteristics of an image, achieving different filtering effects such as blur, sharpening, or edge detection. The value of a specific pixel in the output image is calculated as the weighted sum of all its neighbors in the $N \times N$ local region centered at this pixel as shown in Eq. (2). Fig. 1 illustrates how a two dimensional convolution operation is performed by striding the convolution filter across the input image to calculate all the pixels of the output image. The edge pixels are handled by padding the image with zero pixels in all four directions such that the input and output images of the convolution operation have the same size. Convolution is probably the most commonly used operation in image processing applications, and they are also one of the main building blocks in deep learning models like convolutional neural networks (CNNs) used in vision applications such as image classification and object detection. However, CNNs use a more complicated version of convolution in which the input, filters and output are all three dimensional tensors instead of two dimensional matrices explored in this lab. If you are interested, you can know more about CNNs from this interesting tutorial. Fig. 2 shows the effect of convolving an input image[1] with different filters.

$$y[n][m] = \sum_{i=\frac{-N}{2}}^{\frac{N}{2}} \sum_{j=\frac{-N}{2}}^{\frac{N}{2}} b[i][j]\ x[n+i][m+j] \tag{2}$$

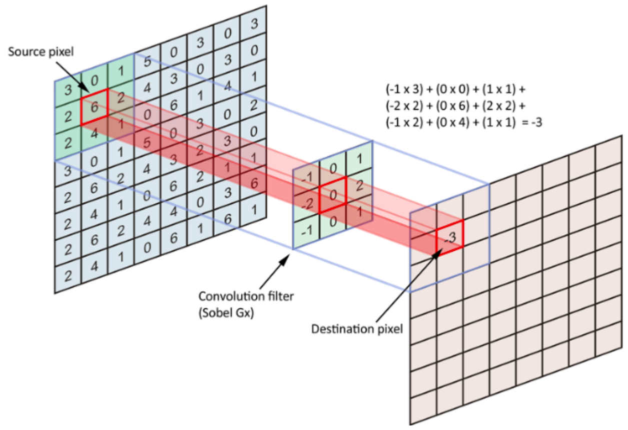

Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

Figure 1: Example for two-dimensional convolution operation.

You can also notice from Fig. 2 that many of the commonly used convolution filters are symmetric in the x-dimension, y-dimension, or both. This means that one can often implement them more efficiently by leveraging the symmetry (i.e. the fact that pairs of coefficients are the same). In this assignment, we will use 3×3 convolution filters that are symmetric in the x-direction (i.e. $b[0][0] = b[0][2]$, $b[1][0] = b[1][2]$, and $b[2][0] = b[2][2]$); you can leverage the fact that the filter is x-symmetric to make your hardware more efficient if you wish.

This assignment is composed of two main parts. In the first part, you will design, implement and optimize an FPGA-based 2D image convolution hardware engine. Your engine should be able to compute the convolution between any 512-pixel-wide image and a 3×3 convolution filter that

[1]The test image that we will be using throughout this assignment is of Geoffrey Hinton, a professor emeritus at the University of Toronto and the "godfather" of deep learning. His work resulted in major advances in the field of artificial intelligence and was awarded the 2018 Turing award (the Nobel prize equivalent in computer science).

$$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}\quad\begin{bmatrix}-1&-1&-1\\-1&8&-1\\-1&-1&-1\end{bmatrix}\quad\begin{bmatrix}0&1&0\\0&1&0\\0&1&0\end{bmatrix}\quad\begin{bmatrix}0&-1&0\\-1&5&-1\\0&-1&0\end{bmatrix}\quad\begin{bmatrix}-1&0&1\\-2&0&2\\-1&0&1\end{bmatrix}$$

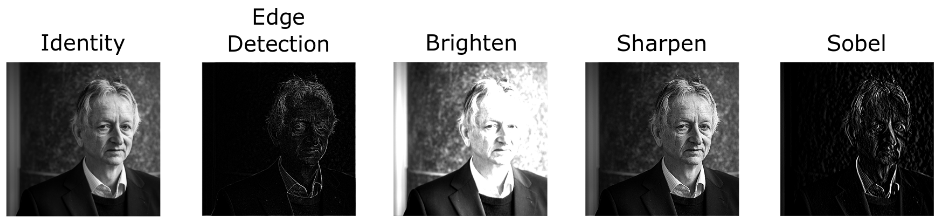


Figure 2: The results of convolving an image with different filters.

is symmetric in the x-direction. For simplicity, we will assume that the filter coefficients are 8-bit signed integers (i.e. no fractional coefficients) and the image is grayscale with 8-bit unsigned pixel values ranging from 0 (black) to 255 (white). In the second part, you will measure the performance of the same application implemented on a multi-core CPU and GPU using code we give you, and then you will compare their efficiency to your FPGA solution.

3 Deliverables

For this assignment, you are asked to hand in the following:

1. **One Quartus project archive (.qar file)** for the 2D image convolution engine that you implement. Use `Project > Archive Project` in Quartus to archive a project. The default options are fine since they save the source files but not the output files. Upload the project archive to Quercus using the naming convention `lab2_<firstname>_<lastname>`. Any HDL files you create in this lab should be well commented and structured, and should use informative signal/variable names. Coding style will be evaluated.

2. **A LaTeX-typed report** in PDF format using this template on Overleaf. You can also use this service to write your report if you do not want to install Latex on your machine. Your report should include the following:

   - **Part 1: FPGA Implementation:**
     - A description and block diagram of your FPGA design.
     - A table, similar to Table 1, for the results of your FPGA design. For calculating the performance, we define an operation as an arithmetic operation such as multiplication or addition (i.e. a multiply accumulate is counted as two operations). Therefore, you should calculate the number of operations required to perform the image convolution and divide that by the time it takes to finish the computation to obtain the throughput in Giga Operations Per Second (GOPS). Use the number of operations in a straightforward implementation of the convolution in this calculation, even if your hardware manages to avoid some multiplications with clever optimizations.
     - Screen shots of waveforms and testbench outputs to verify functionality.
     - An appendix that includes the HDL source codes of your FPGA design.
   - **Part 2: Efficiency Comparison**
     - A high-level explanation of the provided CPU convolution functions and GPU convolution CUDA kernel.

– A table, similar to Table 2, for the results of your measurements of different versions of the CPU and GPU codes provided.

– Comment on the difference in performance between the different versions of CPU code (basic vs. hand-vectorized, single vs. multiple threads) under different compiler optimization settings (no optimization, O2, and O3) and the reasons for these differences.

– A table, similar to Table 3, comparing the best FPGA, CPU, and GPU throughput and efficiency results (for 64 filters)[2]. The three devices are not all using the same process technology; the GPU in the UG machines is an 8 nm chip that is one generation ahead of the 14 nm CPU and two generations ahead of the 20 nm Arria 10 FPGA. Hence the comparison between them mixes process and architectural advantages, so you will also (using guidelines given later in this document) scale your results for the FPGA and CPU to provide an estimate of the efficiency gaps if all devices were in the same process node.

– Comment on the throughput in Giga-operations per second (GOPS), throughput/W, and throughput/mm^2 of the FPGA, CPU and GPU devices. Briefly explain what leads to the differences on these metrics between the devices.

Table 1: FPGA implementation results.

	Result
ALM Utilization	
DSP Utilization	
BRAM (M20K) Utilization	
Maximum Operating Frequency (MHz)	
Cycles for Test 7a (Hinton)	
Dynamic Power for one module @ maximum frequency (W)	
Throughput of one module (GOPS)	
Throughput of full device (GOPS)	
Total Power for full device (W)	

Table 2: Runtime of different versions of the CPU and GPU implementations of 2D convolution with different number of filters.

	Runtime (ms)			
No. of filters	**1**	**4**	**16**	**64**
GPU				
CPU (basic - no opt - 1 thread)				
CPU (vectorized - no opt - 1 thread)				
CPU (basic - O2 - 1 thread)				
CPU (vectorized - O2 - 1 thread)				
CPU (basic - O3 - 1 thread)				
CPU (vectorized - O3 - 1 thread)				
CPU (basic - O3 - 4 threads)				
CPU (vectorized - O3 - 4 threads)				

[2]The CPU and GPU can be limited by memory bandwidth and the amount of parallelism available, so we are using 64 filters per image to increase the parallelism and the amount of computation per MB of data transferred from main memory. This gives a better sense of the computational throughput of these devices.

Table 3: Comparison between the 3 compute platforms implementing 2D convolution with 64 filters.

	Throughput (GOPS)	**Power (W)**	**Energy Efficiency (GOPS/W)**	**Area Efficiency (GOPS/mm^2)**
FPGA (20 nm)				
CPU (14 nm)				
GPU (8 nm)				
FPGA (scaled to 8 nm)				
CPU (scaled to 8 nm)				

4 Part I: Two-dimensional Convolution on FPGAs

You will implement the convolution engine of part 1 on the same device you used in assignment 1: an Arria 10 GX device (10AX115N2F45I1SG). This is the largest and fastest speed grade 20 nm FPGA from Intel. Download the `lab2_fpga.zip` archive from Quercus, and unzip it in your working directory. The archive contains the following:

- `lab2.sv`: The top-level module in which you should implement your design
- `lab2_tb.v`: SystemVerilog testbench
- `lab2.sdc`: Timing constraint file
- `lab2.qpf`: Quartus project file
- `lab2.qsf`: Quartus settings file
- `tests/`: A directory that contains multiple test cases used by the testbench to test your design

For the testbench to be usable with your project, you should have the following input and output signals in your `lab2` module:

- `input clk`: Operating clock
- `input reset`: Active-high reset signal
- `input [71:0] i_f`: Nine 8-bit signed convolution filter coefficients in row major format (i.e. `i_f[7:0]` is $b[0][0]$, `i_f[15:8]` is $b[0][1]$, `i_f[23:16]` is $b[0][2]$, etc.)
- `input i_valid`: Valid input
- `input i_ready`: Consumer module ready to receive new input
- `input [7:0] i_x`: 8-bit unsigned input pixel value
- `output o_valid`: Valid output
- `output o_ready`: `lab2` module is ready to receive a new input
- `output [7:0] o_y`: 8-bit unsigned output pixel value (clipped between 0 and 255; i.e. values less than 0 or greater than 255 saturated at 0 and 255, respectively.)

Your design will receive a stream of input pixels `i_x` in row-major format (i.e. a row is streamed in one pixel at a time before moving to the next row) and the filter coefficients are all loaded in parallel `i_f`. To make it simpler for you, the streamed-in input image is already padded with zeros as illustrated in Fig. 3 and you can assume that the filter coefficients supplied to your module will not change while you are computing the convolution of a single image. The valid and ready signals are the same as in assignment 1. We will only test your design with one image at a time, and the padding ensures that when the last valid input arrives you can produce the last valid output (filtered output pixel on the bottom right of the image). Note that your design should handle the fact that the filter coefficients are signed while input pixels are not. Be careful when using arithmetic operators ($*, +$, etc.) in your design – declare your signal types as signed when necessary to ensure you get the signed versions!

Your circuit must abide to the specification of this interface and you are not allowed to change the testbench code to accommodate for a different interface.

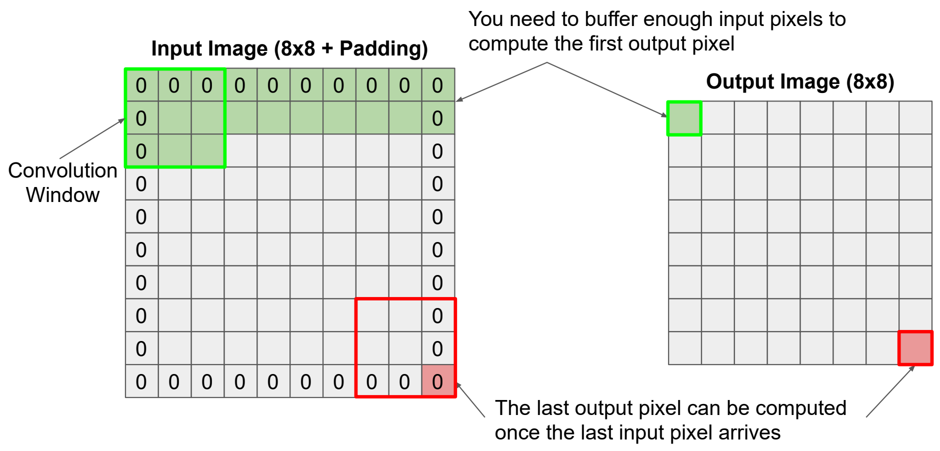

Input Image (8x8 + Padding)
You need to buffer enough input pixels to compute the first output pixel
Output Image (8x8)
Convolution Window
0
The last output pixel can be computed once the last input pixel arrives

Figure 3: Input image is zero-padded such that both the input and output image have the same size. The first output pixel cannot be calculated until enough input pixels are buffered (green-shaded input pixels) to form the first convolution window, while the last output pixel can be produced once the last input pixel arrives (red-shaded input pixel) forming the last convolution window.

4.1 Testing Your Design

The provided testbench contains several test cases, arranged in an increasing order of complexity and size, to help you debug and verify the functionality of your design. To use a specific test case, you can just uncomment it in the `lab2_tb.sv` file. You can find all the test images and their golden solutions in the `tests` directory. You can open all `.pgm` using the the GNU Image Manipulation Program (`gimp`) on the UG machines or using the IrfanView Graphic Viewer on your local machine. You are also free to generate your own test cases if you want to; you can do that by opening any image in IrfanView and saving it as a `.pgm` file with ASCII encoding. You can learn more about the Portable Gray Map (PGM) file format here.

To test your design, perform an RTL simulation using ModelSim following the same instructions from assignment 1 to launch ModelSim, set up your work directory and compile the testbench (`lab2_tb.sv`) and all the files that make up your convolution engine design. You can try different test cases by uncommenting them in the `lab2_tb.sv` file; the lower numbered test cases run faster so you should debug them first before moving on to the larger (and slower to run) `test7`. The testbench automatically dumps three files to help you in the debugging/verification process:

- `verilog_out.pgm`: The output image as produced by your design. In case of correct functionality, this image should be exactly the same as the golden result image in the `tests` directory.
- `verilog_diff.pgm`: An image of the difference in pixel values between the output and the golden result. In case of correct functionality, this should be a black image (all pixels are zeros).
- `verilog_sim_log`: A log of the simulation output comparing the expected and produced values of each pixel. This is a copy of what is printed in the ModelSim command prompt.

If you use Intel IP cores in your design, you will need to do some additional set up (beyond what you did in Assignment 1) to simulate your design. See Section 7 for instructions on how to simulate a design that includes Intel IP cores.

4.2 Optimizing Your Design

We strongly recommend that you get a simple convolution engine working and completely passing the testbench and compiling through Quartus, and only then work to optimize the design for higher speed and/or reduced resource usage. However, you should put a reasonable effort into optimizing your design as some marks will be awarded based on the efficiency of your design (as measured by the total throughput you achieve in a full FPGA). To obtain the most efficient implementation, you will want to investigate the capabilities of the Arria 10 DSP block and choose a convolution implementation that maps well to it. In particular, you will want to ensure your structure allows all 3,036 18×18 multipliers in the device to be used. You can find more details about the Arria 10 DSP block in this user guide.

You may write the convolution functionality using either behavioural operators like '$*$' and '+' or by explicitly instantiating DSP blocks. You can force Quartus to use DSP blocks for multiplication with the synthesis pragma shown below. Note that synthesis pragmas are put in comments, but are parsed and acted upon by the Quartus synthesis engine.

```
module my_module (...)  /* synthesis multstyle = "dsp" */;
// All * operations in my_module will now use a DSP block if possible
```

Explicitly instantiating DSP blocks gives more control over the final result, and may make it easier to obtain a high-performance result. To instantiate a DSP block in Quartus, go to `Tools > IP Catalog`. From the `IP Catalog`, choose `DSP > Primitive DSP > Native Fixed Point DSP Intel Arria 10 FPGA IP`. You can set the different IP core settings to make best use of the DSP block capabilities in your design, and then you can instantiate the DSP block IP in your HDL. Since you will also want to simulate your design for correctness, you should always select `Verilog` under the `Simulation` setting when you generate an IP core. You can find the IP instantiation template in a directory with the IP name that you chose in your Quartus project folder.

Other IP cores that may be useful to you in this lab are the FIFO and 2 Port RAM in the `Basic Functions | On Chip Memory` section of the IP Catalog.

4.3 Power Estimation

You should use the procedure you followed in assignment 1 to estimate power. You should also tell the Quartus Power Analyzer to compute toggle rates using only the simulation data from the period of time after your design is "warmed up" and is computing valid output data. This is done from the Assignments | Settings | PowerPlay Power Analyzer Settings dialog. Click on the power input

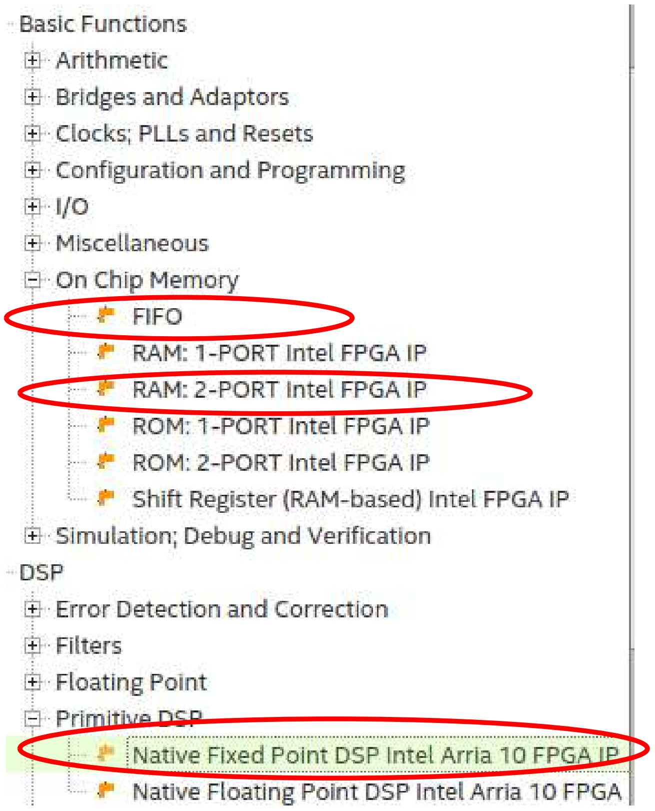

Basic Functions
Arithmetic
Bridges and Adaptors
Clocks; PLLs and Resets
Configuration and Programming
I/O
Miscellaneous
On Chip Memory
FIFO
RAM: 1-PORT Intel FPGA IP
RAM: 2-PORT Intel FPGA IP
ROM: 1-PORT Intel FPGA IP
ROM: 2-PORT Intel FPGA IP
Shift Register (RAM-based) Intel FPGA IP
Simulation; Debug and Verification
DSP
Error Detection and Correction
Filters
Floating Point
Primitive DSP
Native Fixed Point DSP Intel Arria 10 FPGA IP
Native Floating Point DSP Intel Arria 10 FPGA

Figure 4: IP cores that may be useful.

(VCD) file name, select Edit and enter the time range within your simulation that reflects typical steady-state operation as shown in the Fig. 5 and 6.

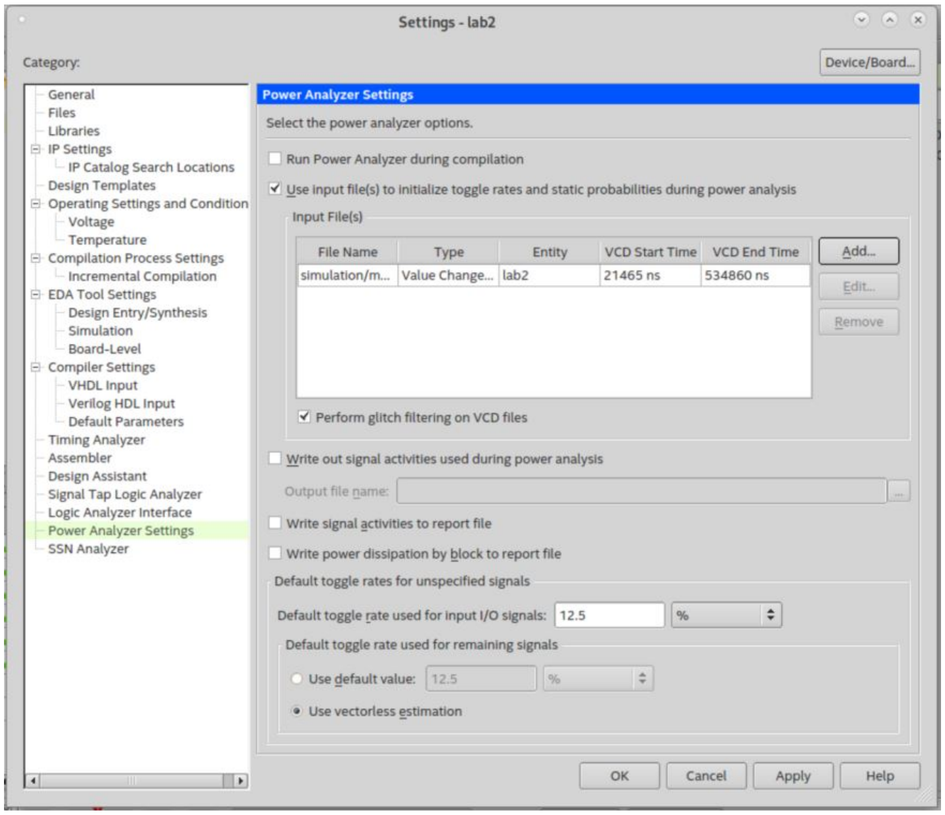
5 Part II: Comparison to CPU and GPU Implementations

In this part of the assignment, you will measure the performance of 2D image convolution on the CPUs and GPUs in the UG machines. The UG machines in the range `ug51-ug80` are the only machines with Nvidia GPUs installed in them, so make sure you use one of the machines in this range.

5.1 Measuring the CPU and GPU Performance

Download the `lab2_cpu_gpu.zip` archive from Quercus, and unzip it in your working directory. The archive contains the following:

- `conv_cpu.cpp`: Several CPU implementations (basic, hand-vectorized, multi-threaded) coded in C. The basic versions use standard C, while the hand-vectorized versions include code that maps directly to Intel's SIMD instructions. There are also multi-threaded versions of both the basic and hand-vectorized code versions that will use all the cores on the CPU.



Power Analyzer Settings

Select the power analyzer options.

- [ ] Run Power Analyzer during compilation
- [✓] Use input file(s) to initialize toggle rates and static probabilities during power analysis

Input File(s)

File Name	Type	Entity	VCD Start Time	VCD End Time
simulation/m...	Value Change...	lab2	21465 ns	534860 ns

- [✓] Perform glitch filtering on VCD files
- [ ] Write out signal activities used during power analysis

Output file name:

- [ ] Write signal activities to report file
- [ ] Write power dissipation by block to report file

Default toggle rates for unspecified signals

Default toggle rate used for input I/O signals: 12.5 %

Default toggle rate used for remaining signals

- ○ Use default value: 12.5 %
- ● Use vectorless estimation

Figure 5: Power Analyzer Settings.

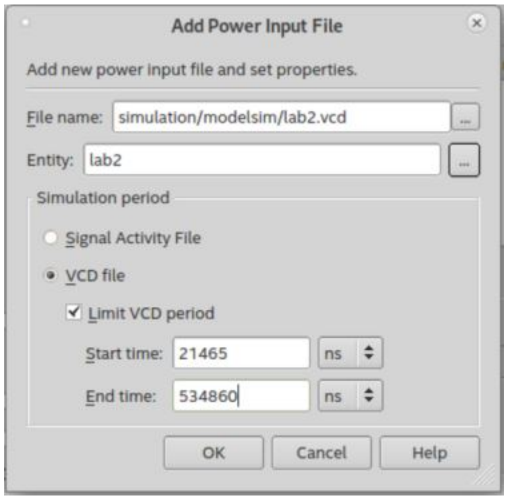
- `conv_gpu.cu`: GPU implementation coded in CUDA, which is Nvidia's data-parallel extension of C
- `utils.cpp` and `utils.h`: utility functions used by both the CPU and GPU implementations
- `run_cpu.sh`: shell script for running the CPU code for different number of filters
- `run_gpu.sh`: shell script for running the GPU code for different number of filters
- `filters`: file storing filter coefficients for a maximum of 64 3×3 filters
- `hinton.pgm`: test input image in Portable Gray Map (PGM) format
- `out_cpu/`: directory in which CPU output images will be stored
- `out_gpu/`: directory in which GPU output images will be stored

For the CPU implementation, you are required to collect runtime results for the basic and hand-vectorized versions using different `g++` compiler optimization settings. GThen, you can collect the multi-threaded performance of both versions by replacing the calls to `cpu_conv2d` and `cpu_conv2d_vectorized` with `cpu_conv2d_multithreaded` and `cpu_conv2d_vectorized_multithreaded`, respectively, in the `main()` function.

To compile the CPU code without compiler optimizations, you can use the following command:

```
g++ -mavx -fopenmp conv_cpu.cpp utils.cpp -o conv_cpu
```

the `-mavx` and `-fopenmp` flags are to allow compilation of the vectorized and multi-threaded implementations, respectively. To compile with different compile optimization levels, you should



File name: simulation/modelsim/lab2.vcd
Entity: lab2
Simulation period
( ) Signal Activity File
(●) VCD file
[✓] Limit VCD period
Start time: 21465 ns
End time: 534860 ns

Figure 6: Limiting the VCD period to the steady-state operation period.

add the -O2 and -O3 to the compilation command. To run the compiled executable, you should use the following command specifying the test image name and number of filters:

```
    ./conv_cpu hinton.pgm 1
```

We also ask you to measure the runtime when using different number of convolution filters ranging from 1 to 64 filters (i.e. one input image convolved with N filters to produce N output images). For your convenience, we provide you with a shell script to run the code with different numbers of filters and report all the results at one go. You can run the shell script using the following command:

```
    ./run_cpu.sh 1 4 16 64
```

Similarly for the GPU implementation, you should use the following commands to compile and run the GPU implementation:

```
    nvcc conv_gpu.cu utils.cpp -o conv_gpu
    ./conv_gpu hinton.pgm 16
    ./run_gpu.sh 1 4 16 64
```

5.2 Comparing to the FPGA, CPU and GPU Implementations

We will compare the performance of a single FPGA, CPU or GPU chip for the two-dimensional convolution application. Generally to find the most efficient chip for such an application, we will compare several measures of performance: throughput of one chip (i.e. number of performed operations per second), throughput per Watt and throughput per unit area. To compute these metrics, you should run each chip at its maximum throughput. This means for the FPGA you should estimate how many copies of your design can be implemented in the FPGA and run them at all at their maximum operating frequency, using a procedure similar to that of assignment 1.

To compute throughput of the FPGA, CPU or GPU solution, you should calculate the number of operations in an image convolution and divide that by the runtime you obtain from simulation (for the FPGA implementation) or actual measurements (for the CPU and GPU implementations). Remember that we define an operation to be an arithmetic operation (e.g. multiplication or addition). For example, directly calculating the expression $y = ax^2 + bx + c$ has 5 operations (3 multiplications and 2 additions). When calculating the device throughput, keep in mind that the provided FPGA testbench runs at only 50 MHz. Therefore, you should use the testbench simulation to obtain the number of cycles required to finish the computation (for the `test7a` case with a full 512×512 image) then use your design's maximum operating frequency to calculate the total runtime. You can use the results from Table 1 that you filled in part I of the assignment.

Table 4 summarizes the specifications of the three computing devices you are comparing. You might find these information useful when trying to fill Table 3 in your report. The TDP column of the table gives the thermal design power which is the maximum power that can be consumed by each of the three chips. You can assume the CPU and GPU have power consumption equal to the TDP values below, but you can (and should) get a more accurate value for the FPGA using the Quartus PowerAnalyzer as described earlier.

Table 4: Specifications of the three compute platforms.

Device	**Model**	**Process Tech.**	**Die Size (mm^2)**	**TDP (W)**
CPU	Intel core i7-11700 (8 cores)	14 nm	276	65
GPU	Nvidia GeForce GTX 3070	8 nm	393	220
FPGA	Arria 10 GX 1150	20 nm	$\sim$ 400	70

To complete Table 3 for your report, you will also need to approximately scale the CPU and FPGA results to the same process generation (8 nm) as the GPU uses. You can use the values below as approximate scaling parameters. They are derived from ideal area scaling (2× more devices per process generation) and from comparing the power and performance of Intel's 10 nm Agilex and 20 nm Arria 10 FPGAs (which are two process generations apart). The CPU values assume the same per generation performance and power scaling as Agilex achieved (i.e. the square root of the Agilex values).[3]

Table 5: Process scaling factors, based on FPGA scaling from Arria 10 to Agilex.

Scaling Between	**Area**	**Clock Speed**	**Power @ same clock rate**
14 nm to 8 nm (one generation)	0.5x	1.25x	0.7x
20 nm to 8 nm (two generations)	0.25x	1.6x	0.45x

6 How You Will Be Graded

Your grade out of 18 will mainly depend on:

1. **The correctness and performance of your FPGA solution (~55%):** Your hardware should pass all the tests. You should optimize your hardware designs for throughput and, as a

[3]Ideally we would simply compile your designs to Intel's Agilex FPGA, as it is in a 10 nm process that is roughly comparable to the GPU's 8 nm process; alas, we do not yet have licenses for Agilex on the UG machines.

secondary objective, for area. Fully functional hardware will count for approximately half of this grade component, with the other half being assessed based on the throughput your design achieves.

2. **The correctness, completeness and clarity of your report (~40%):** Your report should include enough details to explain your design, implementation, and functional verification process for the FPGA design, and should answer all the questions posed about the CPU and GPU designs and efficiency comparisons. Use block diagrams, screenshots, and/or graphs whenever needed to explain your solutions and optimizations. Avoid using your report as a screenshot dump that might not be very readable. Make sure you provide all the data and answer all the questions requested in the handout, and explain your reasoning.

3. **Coding style (~5%):** Your HDL code should be understandable and well structured. Use descriptive signal/variable names and comment extensively. Try to avoid large amounts of repeated code; creating sub-modules or using generate statements or for loops can reduce code length.

7 Appendix A: How to Simulate Designs that Include IP Cores

If you include Intel IP cores in your design, running the basic ModelSim commands from lab 1 won't be enough to compile all the IP cores and the libraries they depend on. Instead, it is easiest to let Quartus create a script that will compile all the necessary IP cores and libraries your design needs.

To simulate a design with IP, note that you should select the `Verilog` option under `Simulation` when you generate the IP. Next, in Quartus choose `Tools | Run Simulation Tool | RTL Simulation`. This will automatically start ModelSim, compile the IP cores and your design, add all signals to the waveform viewer, and simulate the design.

If you are frequently modifying your design and you want to just re-run ModelSim (without starting Quartus), you can do so as long as you ran the operation above once. In this case, you can start ModelSim with this procedure:

- Open a command prompt, and

```
cd <your_quartus_proj>/simulation/modelsim
```

- Start modelsim

```
> modelsim18.start
```

- Run the script Quartus created from inside modelsim. For example, if your project is called lab2, you would use:

```
ModelSim> do lab2_run_msim_rtl_verilog.do
```

- If you don't like the windows this .do script opens or don't like it adding all the signals to the waveform viewer, you can edit the latter parts of the script.

- You should re-run the `Tools | Run Simulation Tool | RTL Simulation` from within Quartus whenever you add new IP or update the parameters of an IP in your project, as Quartus will need to generate an updated .do script for ModelSim in this case.