

EE 5324 Spring 2025
Project #3: Convolution + Max-pooling Engine Design

Milestone 1: Soft Due 4/25/25, Tuesday, 11:59pm

Milestone 2: Due 5/8/25, Friday, 11:59pm

Read the entire document carefully before starting the project.

This is an individual assignment. Your designs will be checked for plagiarism.

There are 3 stages for this project.

Stage 1: Write the Verilog code for the conv_pool module, and verify the functionality.

Stage 2: Synthesize your design using Design Compiler and verify the synthesized netlist.

Stage 3: Run Primetime PX for power measurement.

NOTE: At every stage, make different folders such as RTL/, Synthesis/, Primetime/.

1. Introduction

Convolution is a computationally intensive function that has been widely used in image processing and filtering [1]. You can try to perform a generic convolution filter on an image using the graphics editor program ‘gimp’ [2]. In the advancement of deep learning and convolutional neural networks (CNNs) for image applications [3-4], convolution is the most heavily used operation, whose efficient hardware implementation is critical in terms of both performance and power.

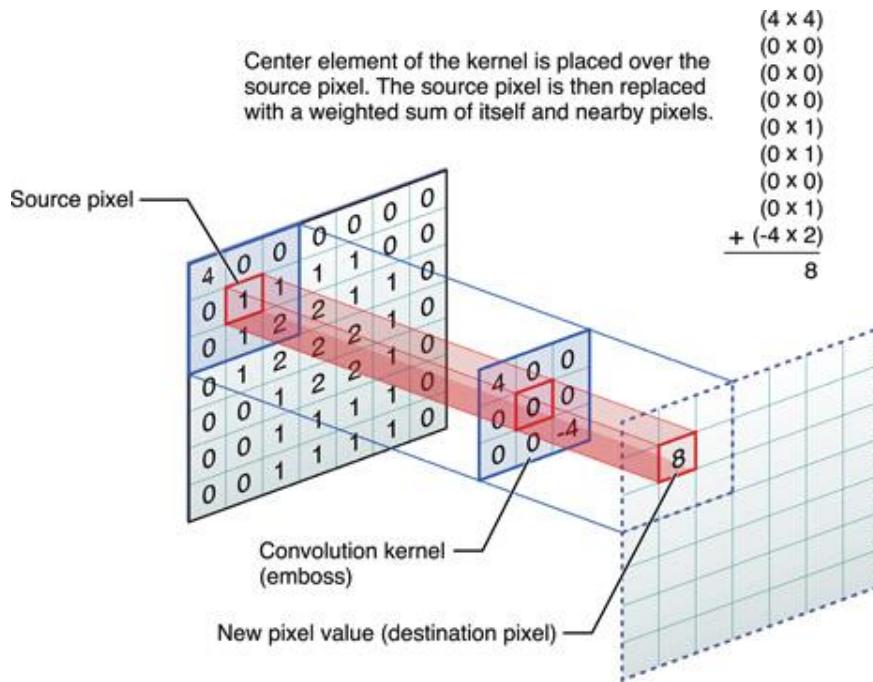
[1] http://www.cs.cornell.edu/courses/cs1114/2013sp/sections/s06_convolution.pdf

[2] <https://docs.gimp.org/2.6/en/plug-in-convmatrix.html>

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *In Advances in Neural Information Processing (NIPS)*, 2012, pp. 1097-1105.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, “Going deeper with convolutions,” *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1-9.

(1) Convolution: In image processing, convolution represents weighted sum computation of a source image and a convolution kernel. Widely used kernel sizes are 3×3 or 5×5 (pixels), and we will use a 3×3 kernel in this project. Using a 3×3 kernel, convolution is illustrated in the following figure (source: <https://developer.apple.com/library/ios/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html>).



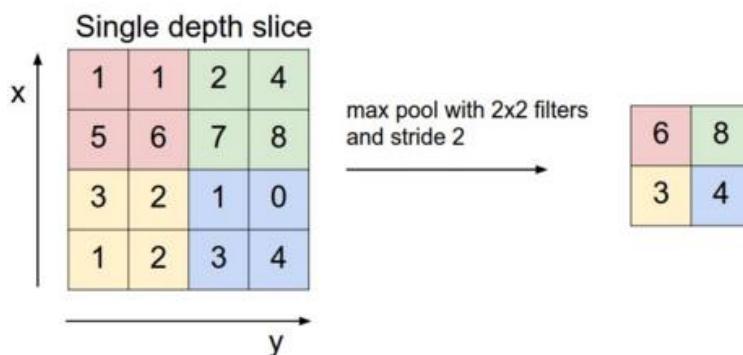
The same convolution kernel (kernel) slides by 1-pixel stride across the entire source image (img_src), to generate the convolution output image (img_conv). The pseudo-code for such operation is shown below, besides the boundary condition on the edge pixels.

```

for (i < img_width)
  for (j < img_height)
    img_conv[i,j] = img_src[i-1,j-1]*kernel[0,0] + img_src[i-1,j]*kernel[0,1] + img_src[i-1,j+1]*kernel[0,2]
      + img_src[i ,j-1]*kernel[1,0] + img_src[i ,j]*kernel[1,1] + img_src[i ,j+1]*kernel[1,2]
      + img_src[i+1,j-1]*kernel[2,0] + img_src[i+1,j]*kernel[2,1] + img_src[i+1,j+1]*kernel[2,2]
  
```

There can be multiple kernels, each corresponding to a separate convolution. Each kernel processes the input image independently and generates a output image, also referred to as a channel.

(2) Max-pooling: Max-pooling (or sub-sampling) operation is used to reduce the image or channel feature dimensions without losing critical information. Max-pooling partitions the input channel into a set of non-overlapping rectangles (i.e., 2×2 or 3×3 pixels) and, for each such sub-region, outputs the maximum value. By eliminating non-maximal values, it not only reduces computation for upper layers in CNNs but also provides a form of translation invariance. For a 2×2 case, max-pooling operation is illustrated in the following figure for one channel (source: <http://cs231n.github.io/convolutional-networks/>)



Functionality and Design

In this project, you will design a convolution and max-pooling module, where your module input is a 4×4 pixel image, 3 3×3 convolution kernels, and a shift value (explained later), and your module output are the max-pooling output values. The convolution + max-pooling module should have the following module definition and input/output ports.

```
module conv_pool (clk, rst, image_4x4, conv_kernel, y, input_re,
input_addr, output_we, output_addr);

    input      clk, rst;
    input      [127:0] image_4x4;
    input      [71:0] conv_kernel_0;
    input      [71:0] conv_kernel_1;
    input      [71:0] conv_kernel_2;
    input      [1:0] shift;
    output     [7:0] y_0;
    output     [7:0] y_1;
    output     [7:0] y_2;
    output     input_re;
    output     input_addr;
    output     output_we_0;
    output     output_addr_0;
    output     output_we_1;
    output     output_addr_1;
    output     output_we_2;
    output     output_addr_2;
    // ...
endmodule
```

- The input_re signal goes high when the circuit is ready to take in a new image row for computation, the input_addr is the line number on the image file. The output_we_0/1/2 goes high when the computation to the corresponding kernel_0/1/2 is complete, and it is available in y_0/1/2 and the output_addr_0/1/2 is the line number of the image which was computed.
- Assume each pixel is grayscale, represented by a positive 8-bit integer (from 0 to 255).
- Assume each kernel value is an 8-bit two's-complement fixed-point number with 5 decimal digits and 3 fraction digits (from -16 to 15.875)

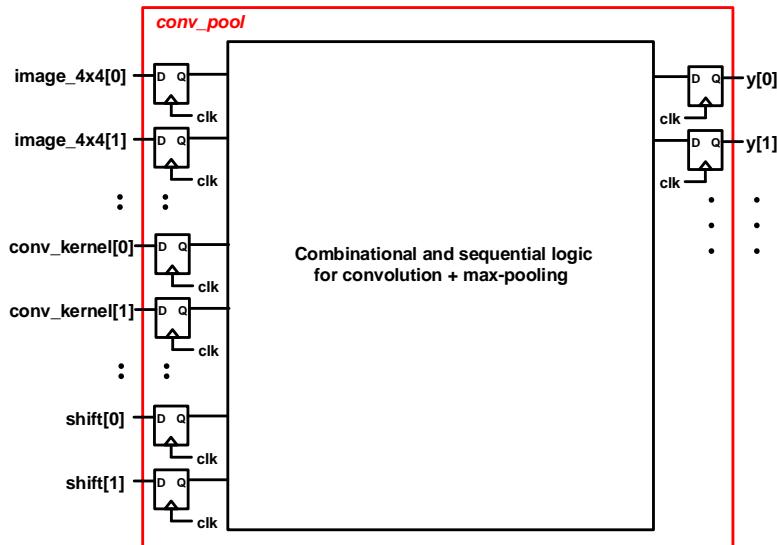
- For $0 \leq i \leq 3$, $0 \leq j \leq 3$, the value of the image pixel at i^{th} row and j^{th} column is:
 $x(i, j) = \text{image_4x4}[i*32+j*8+7 : i*32+j*8]$
- For $0 \leq i \leq 2$, $0 \leq j \leq 2$, the value of the kernel_0/1/2 at i^{th} row and j^{th} column is:
 $k(i,j) = \text{conv_kernel_0/1/2}[i*24+j*8+7 : i*24+j*8]$
 $k(i,j)$ is a 8-bit two's-complement fixed-point number with 5 decimal digits and 3 fraction digits
(i.e., 1000000: -16, 11111111: -0.125, 01111111: 15.875)
- Then, four 3×3 convolutions for a specific kernel_0/1/2 are computed as follows, by sliding the 3×3 image window by a stride of 1 pixel both horizontally and vertically.
Note that “shift” is a 2-bit input with possible values of 0, 1, 2, or 3, hence possibly dividing the convolution by 1, 2, 4, or 8, respectively.

$$c0 = \left(\sum_{i=0}^2 \sum_{j=0}^2 [x(i, j) * k(i, j)] \right) / 2^{shift}, \quad c1 = \left(\sum_{i=0}^2 \sum_{j=0}^2 [x(i, j + 1) * k(i, j)] \right) / 2^{shift},$$

$$c2 = \left(\sum_{i=0}^2 \sum_{j=0}^2 [x(i + 1, j) * k(i, j)] \right) / 2^{shift}, \quad c3 = \left(\sum_{i=0}^2 \sum_{j=0}^2 [x(i + 1, j + 1) * k(i, j)] \right) / 2^{shift},$$

- Keep the convolution output values between 0 and 255. Do the following for $c0, c1, c2$, and $c3$.
 - if* ($c0 < 0$) $c0 = 0$
 - if* ($c0 > 255$) $c0 = 255$
- Finally, we perform max-pooling (sub-sampling) by selecting the maximum value out of the four convolution results for a specific kernel_0/1/2.
$$y = \max(c0, c1, c2, c3)$$
- The objective of the design is to minimize area and power, while maximizing throughput (i.e., how long does it take to finish a large number of convolutions).
 - Think through ways to speed up the multiplication and addition, such as different multiplier and adder architectures for fast weighted sum computation.
 - The functionality requirement is to get the correct ‘y’ value for all kernels, but note that convolution involves a lot of redundancy, and only the maximum value out of 4 convolution results is selected at the end. Think of ways to reduce the redundant computation, or approximate the intermediate convolution computation without hampering the ‘y’ value.
 - Without any pipelining between the input and output flip-flops, your clock frequency could be relatively slow, while the whole convolution + max-pooling would take one-cycle. With pipelining, your clock frequency would be relatively fast, and you could have a number of convolution + max-pooling operations in flight, increasing your throughput. But with too much pipelining, the overhead of additional flip-flops would degrade both power and performance. You should try to find the sweet spot.

- As illustrated in the figure below, all the inputs and outputs should be connected to input/output flip-flops at the boundary of the conv_pool module (no logic between the primary inputs/outputs and the boundary flip-flops).



Testbench and Simulation

You should also design a testbench Verilog file that instantiates the conv_pool module above, and verifies the correct convolution and subsampling operation.

[To help you study this project, we provide a testbench Verilog file tb_conv_pool.sv]

- You should run the same testbench and check the correctness of your module with behavioral and post-synthesis Verilog netlists.

This conv_pool.input file is generated from a grayscale 512×512 ‘lena’ image shown below (given in Project 3) , which is a standard test image in the field of image processing.



Using lena.jpg, the python script (project3_image_proc.py) that was used to generate the conv_pool.input file is provided in Project 3. The same python script could be used for any other grayscale image.

To run the python script and generate conv_pool.input file, execute the following from the command line.

- python project3_image_proc.py > conv_pool.input
- As your simulation goes through, you should save the resulting ‘y’ outputs for 3 kernels in separate files named “conv_pool_0/1/2.output”. These “conv_pool_0/1/2.output” files should only have one ‘y’ value in hex in each line, and there should be 65,536 lines. There are a golden output files provided in Project 3, golden_0/1/2.txt, which has the expected ‘y’ values for “conv_pool.input” data file (with 65,536 lines as well) for each kernel.
- In order to visualize what your convolution + pooling looks like, line 101 – 145 of project3_image_proc.py file includes scripts to generate an image from conv_pool_0/1/2.output data format. Convolution (with sharpening kernel_0, with edge kernel_1, with blur kernel_2) + pooling output image (256×256) is shown below.

Sharpening kernel_0



Edge kernel_1



Blur kernel_2



- project3_image_proc.py python script also generates the convolution-only output image (before max-pooling), and the convolution-only output image for lena.jpg is provided in Project 3.
- Feel free to modify the project3_image_proc.py script as you need. However, it is not necessary to run the script or generate the images as what is needed is already given to you.
- As a preparation for power measurement, include the following codes in your testbench file.

```
initial
begin
    $dumpfile("current_output.vcd");
    $dumpvars(0,testbench);
end
```

- To measure throughput, go through the following procedure.
 - As instructed, set your clock period from PrimeTime.
 - In the testbench simulation, simply record the total simulation, and this will be your “**Tot_latency**”. This time is essentially (clock period * number of cycles) -- the overall time it takes for your conv_pool module to go through all 65,536*3 convolution +

pooling operations and output $65,536*3$ results. (You will also be asked to submit a screenshot that proves this.)

Synthesis

- See the “Additional materials VCS-DC.zip”

Primetime and Power Measurement

- See the “Additional materials PrimeTime”

Grading - 100 points total

There will be two milestones for this project.

For Milestone 1, only the functionality and the synthesis progress will be graded. The exact performance, area, and power numbers would not be graded.

For Milestone 2, it is expected that you will optimize the design much further considering the overall throughput, area, and power consumption of the conv_pool module.

For Milestone 1 Submission (50 points)

- **25 points for behavioral Verilog and synthesis:** Your design must satisfy the functionality. The behavioral Verilog should produce conv_pool_0/1/2.output files that matches exactly with golden_0/1/2.txt, and must go through synthesis successfully.
 - The grader could simulate your behavioral Verilog netlist with a different kernel or a different input image, so make sure that your module satisfies functionality for general inputs.
- **20 points for synthesis Verilog simulation:** Your design must go through synthesis successfully. With your synthesized Verilog netlist (*.v), the same testbench should pass the functionality successfully. Submission requirements are:
 - Note: The grader could simulate your post-synthesis Verilog netlist with a different kernel or a different input image.
- **5 points for concise report:** Please give a concise report on your conv_pool design commenting on the employed architecture, the initial design decisions you made for the convolution and average-pooling engine design.

For Milestone 2 Submission (50 points)

- **10 pts for updated design:** Optimize your entire design for performance (“**Tot_latency**”), area (core layout area excluding power/ground rings), and power (reported by PrimeTime). Your design should still meet all of the submission criteria for Milestone 1 (Synthesis).
- **10 points for post-synthesis PrimeTime analysis:** Determine the power of your multiplier using PrimeTime with the *.vcd file. The .vcd file should be created by running the testbench code with commands such as \$dumpfile and \$dumpvars. Then provide *.vcd with *.sdf/*.spf to PrimeTime and run proper commands to obtain the power consumption for your multiplier module.
- **15 pts for optimization quality:** All the submissions will be evaluated and sorted using the following formula:

$$\text{Quality Metric} = (\text{Tot}_\text{latency})^2 \times \text{Power} \times \text{Area}$$

You would want to minimize the quality metric above. Points for optimization will be given proportional to the rank of the submission. That is, the project with rank i ($=0, 1, 2, \dots$) will get the following points for optimization.

$$Points(i) = \frac{Number\ of\ submissions - i}{Number\ of\ submissions} \times 15$$

- **10 pts for full report (with analysis of each part):** A good report would present an optimization description on different parts of the conv_pool module (including any new techniques that you used to optimize the design), and the final result of the optimization. The report should conclude with a discussion of where the bottleneck is in terms of performance and power.
- **5 pts for demo (12 students will be randomly selected):** Please note that we will consider only the files that are submitted on blackboard on or before submission deadline. More details coming soon.

Submission

Put all files in one Unix directory with the following organization:

< Milestone 1 >

Project_<studentID>_MS1

1) Report document

Architecture and high-level block diagram

Concise report w/ initial design decisions (# of parallel multipliers, etc.)

2) Behavioral Verilog Netlist file

3) Synthesized Verilog Netlist file

4) Synthesis report with top 5 critical paths

5) Testbench Verilog file

6) tb_conv_pool.syn.log: testbench simulation log file with synthesized Verilog netlist

< Milestone 2 >

Project3_<studentID>_MS2

1) Report (follow template)

Architecture and high-level block diagram

Concise report w/ final results

- Design decisions

- Total latency (= total testbench simulation time from VCS)

 - Unit must be in “ms” (milli-seconds)

 - Include screenshot from VCS

- Power (from PrimeTime)
 - Unit must be in “mW” (milli-Watts)
- Area (only the entire standard cells)
 - Unit must be in “mm²”

2) Behavioral Verilog Netlist

3) Synthesized Verilog Netlist

4) Synthesis report with top 5 critical paths

5) Power report from PrimeTime

6) Top 3 worst timing paths from PrimeTime

7) Testbench Verilog file

8) tb_conv_pool.syn.log: testbench simulation log file with synthesized Verilog netlist

9) Screenshot of testbench simulation upon completion –the screenshot should contain the total simulation time, and this should be the same value as the “Total latency” above in the report.

Make sure none of the items above are omitted.

Then compress them to one tar file by using the following command.

```
tar -cjf Project3_<studentID>_MS*.bz2 Project3_<studentID>
```

Submit the *.bz2 file to blackboard.