ESE 680 Final Project

CELINE LEE, University of Pennslvania

The objective of this project is to implement and optimize a 2D convolution operation on FPGA, with the end goal of accelerating inference on a pre-trained VGG-16 model. This report will discuss approaches to and results from optimizing this operation from the software perspective, the hardware perspective, and the software/hardware interface perspective.

Additional Key Words and Phrases: hardware-software co-design, convolution, accelerator

ACM Reference Format:
Celine Lee. 2020. ESE 680 Final Project. 1, 1 (February 2020), 10 pages. https://doi.org/NA

1 INTRODUCTION

With the rise of machine learning applications, there is widespread motivation to reduce the computation footprint of certain mathematical operations in machine learning models. One such operation is the 2D convolution, used prominently in Convolutional Neural Networks (CNNs) for computer vision applications. This project explores the design space of optimizations for the 2D convolution operation, both in hardware and software, as well as in the hardware/software interface. Given the large design space to explore, this report will delve into just a couple of different paths for optimization that were attempted throughout the project. Section 2 will describe the general methodology for the project, with section 2.5 discussing the bulk of the exploration. Section 3 will evaluate the results from said explorations, and propose how future attempts to this project with more time would benefit from adopting ideas from certain discoveries made. Finally, section 4 will conclude the report.

This project is implemented using C++ and Python bindings, on an AWS F1 FPGA.

2 METHODOLOGY

To create this system, I started with designing and verifying an implicit systolic matrix multiplication kernel. Once that was verified, I verified implementation of matrix block multiplication. Once large matrix multiplication is verified, I begin implementing 2d convolution using the GEMM operation (figure 1).

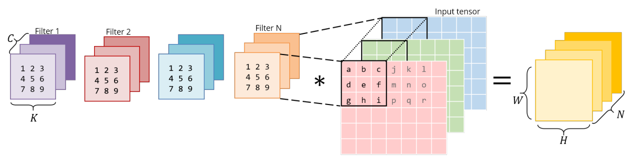


Figure 1: GEMM Convolution

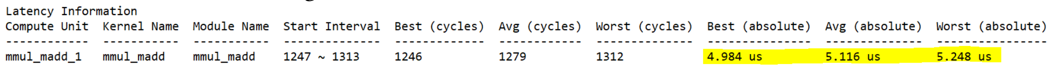
The following subsections follow the workflow of implementing and optimizing 2d convolution on

Author's address: Celine Lee, celinel@seas.upenn.edu, University of Pennslvania.

an AWS F1 FPGA. Section 2.5 will discuss a handful of optimization attempts, some of which did not make its way into the final design.

2.1 Milestone 1: Hardware Kernel + Software Emulation

The first step was to create and verify the hardware kernel. This kernel is designed to include both multiplication and addition functionality, with operation determined by a switch passed into the kernel. The multiplication was implemented via an implicit systolic array in order to capitalize on potential parallelism. With burst read/writing, array partitioning, and loop unrolling, the pipeline initiation intervals for each loop are set to 1. The result is a hardware kernel that can perform accelerated matrix multiplication as well as matrix addition. After functionality is verified in software emulation, kernel runtime properties are extracted from hardware emulation. The results can be seen in figure 2.



```
Latency Information
Compute Unit  Kernel Name  Module Name  Start Interval  Best (cycles)  Avg (cycles)  Worst (cycles)  Best (absolute)  Avg (absolute)  Worst (absolute)
------------  -----------  -----------  --------------  -------------  ------------  --------------  ---------------  --------------  ----------------
mmul_madd_1   mmul_madd    mmul_madd    1247 ~ 1313     1246           1279          1312            4.984 us         5.116 us        5.248 us
```

Figure 2: Hardware emulation system estimate for matrix multiplication kernel.

2.2 Milestone 2: C++ Extension

In order to wrap the kernel acceleration into a module usable by PyTorch, the host C++ code is called from the Just-in-time (JIT) PyTorch C++ compiler. Once the `XCL_EMULATION_MODE` variable is set to `sw_emu` in the host C++ code, we can compare the output of the C++-accelerated 2d convolution to the vanilla torch.nn.Conv2d function. The results pass (they match the vanilla PyTorch Conv2d features output), and we can move on to benchmarking in milestone 3.

2.3 Milestone 3: Python Benchmarking in Software Emulation

After writing the python `benchmark.py` script to obtain a FOM for the accelerated PyTorch JIT compilation + C++ bindings + OpenCL kernel calling, I tested the python benchmark in software emulation.

Command: `start_job fpga_build -d fpga_conv2d`

```
-c 'export LD_LIBRARY_PATH=/usr/local/lib:/usr/lib:/usr/local/lib64:/usr/lib64;
source /aws-fpga/vitis_setup.sh; export LC_ALL="C"; cd fpga_conv2d/src; export
XCL_EMULATION_MODE=sw_emu; python3 benchmark.py' -s
```

The results from this sections are: Runtime(1) one inference: `2504 seconds`. This is quite long, so we want to ensure that when run on the FPGA, runtime will go down. This is also indication that there is plenty of room for optimization.

2.4 Milestone 4: Baseline in Software and Hardware

The benchmark has been verified in software emulation. We move on to verifying functionality in hardware. The kernel is compiled and exported to build with the `hw` target in order to generate an `.xclbin` file. An fpga image is then created from that file, to link with the AWS AFI and generate an `.awsxclbin` file. Using this `.awsxclbin` file, we can then verify functionality in hardware on `fpga_deploy` and obtain figures of merit (FOM). Obtained FOM for this project will be discussed in section 3 Evaluation.

Command to run in hardware: `start_job fpga_deploy -d fpga_conv2d`

```
-c 'export LD_LIBRARY_PATH=/usr/local/lib:/usr/lib:/usr/local/lib64:/usr/lib64;
export LC_ALL="C"; source /aws-fpga/vitis_runtime_setup.sh; export LC_ALL="C";
```

```
source /aws-fpga/vitis_setup.sh; export LC_ALL="C"; cd fpga_conv2d/src; python3
benchmark.py' -s -a
```

2.5 Optimizations

Since this project spans the software plane, the hardware plane, and the software/hardware interface plane, there is a large design space for optimization exploration. First, I will discuss all optimizations that made their way into the final design.

Final Design. The final design submitted unfortunately did not manage to include all optimization attempts and ideas. However, it does include a couple instances of minimizing data transfer from Python to C++, eliminating unnecessary matrix block multiplications, and minimizing transformations between GEMM and input/output matrix formatting. The system diagram can be seen in figure 3.

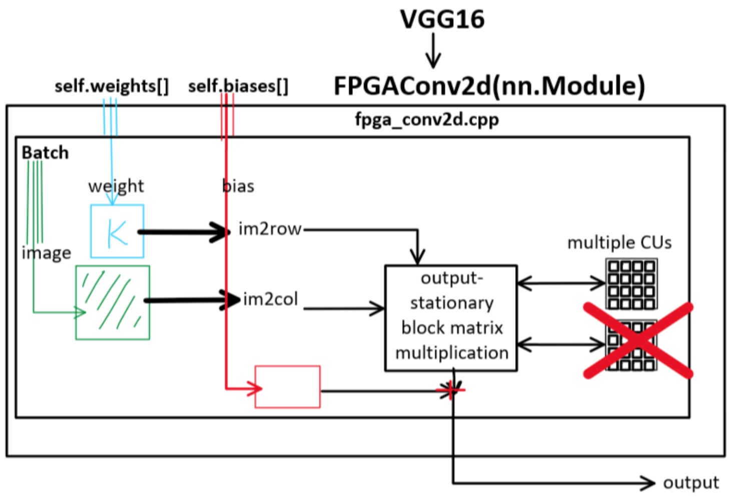
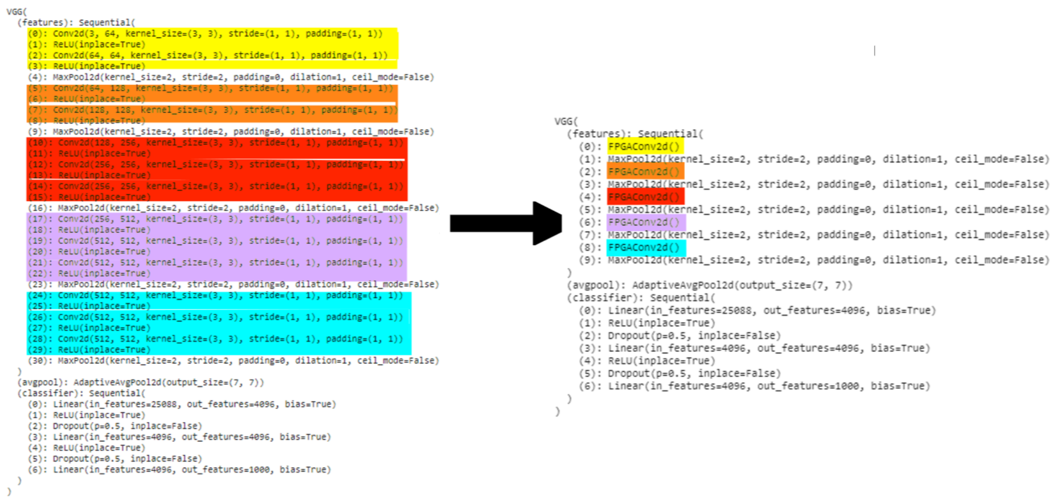


Figure 3: System diagram for final design.

In order to minimize data transfer from Python to C++, I merged consecutive Conv2d+ReLU layers in the VGG16 model. Instead of passing a single set of weights and biases to the `fpga_conv2d` C++ function, I pass a vector of weights and a vector of biases to the `fpga_conv2d` C++ function. Both the weights and the biases vectors are the same length, which is the number of merged 2D convolution layers. In the case of VGG16, this allows me to internalize up to 6 different layers at a time (3 consecutive Conv2d and their corresponding ReLU functions) within one C++ call. See how this is done specifically for VGG16 in figure 4. The Python code implementing this can also be found in the appendix section 'Merging Conv2d Layers.' This also opens the door to high data reuse on the hardware side, which will be discussed in the next sub-subsection.



```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

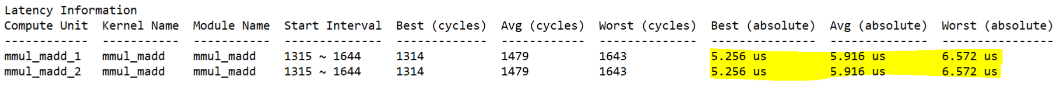
```
VGG(
  (features): Sequential(
    (0): FPGAConv2d()
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): FPGAConv2d()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): FPGAConv2d()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): FPGAConv2d()
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): FPGAConv2d()
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Figure 4: Merging of VGG consecutive Conv2d+ReLU layers.

In order to minimize transformations and avoid unnecessary matrix block multiplications, I eliminate the requirement of square matrices that was implemented in lab 4. Instead, I keep close track of the often-rectangular matrix sizings (for both the im2col transformation of image and the im2row transformation of kernel) and round them to the nearest larger multiple of 16 in order to make it compatible with my 16x16 systolic matrix multiplication kernel. Then, I only perform matrix block multiplications in which neither matrix block is filled with 0-values. This saves on space and computation time.

The block matrix multiplication operation itself iterates through each output block location `(i,j)`, $(0 \leq i \leq N_{16}, 0 \leq j \leq HW_{16})$ and retrieves the corresponding `i` row from the GEMM-transformed kernel matrix and the corresponding `j` column from the GEMM-transformed image matrix, then moves along their shared axis to call my matrix multiplication kernel and sum the outputs to produce the final output `(i,j)` block.

OPTIMIZATION ATTEMPT 1: Output-Stationary Kernel Matrix Multiplication. Building off of the first steps toward data reuse in the previously discussed design, I write another version of the kernel that actually stores the matrix multiplication output in FPGA scratchpad memory, using the PLRAM port. (Since the size of the data being transferred here is rather small, only one 16x16 vector of floats, PLRAM allows for quicker data transfers.) With each subsequent call to the kernel, it simply increments the output matrix block within the kernel rather than manually adding the results in software. This allows us to also minimize data transfer from the kernel output to the host. The kernel for this approach, verified in software emulation and hardware emulation, produced the system estimate that can be seen in figure 5. At this point, multiple CUs were also integrated to the design.



```
Latency Information
Compute Unit  Kernel Name  Module Name  Start Interval  Best (cycles)  Avg (cycles)  Worst (cycles)  Best (absolute)  Avg (absolute)  Worst (absolute)
------------  -----------  -----------  --------------  -------------  ------------  --------------  ---------------  --------------  ----------------
mmul_madd_1   mmul_madd    mmul_madd    1315 ~ 1644     1314           1479          1643            5.256 us         5.916 us        6.572 us
mmul_madd_2   mmul_madd    mmul_madd    1315 ~ 1644     1314           1479          1643            5.256 us         5.916 us        6.572 us
```

Figure 5: Hardware emulation for output-stationary kernel using PLRAM and multiple CUs.

OPTIMIZATION ATTEMPT 2: Data Reuse. In examining the ports and the Xilinx example for multiple CUs, I begin to see even more opportunity for input data reuse and minimization of im2col calls. In my next attempt, I modified the system to move a large chunk of the logic from the software to the hardware. The new system diagram is shown in figure 6.

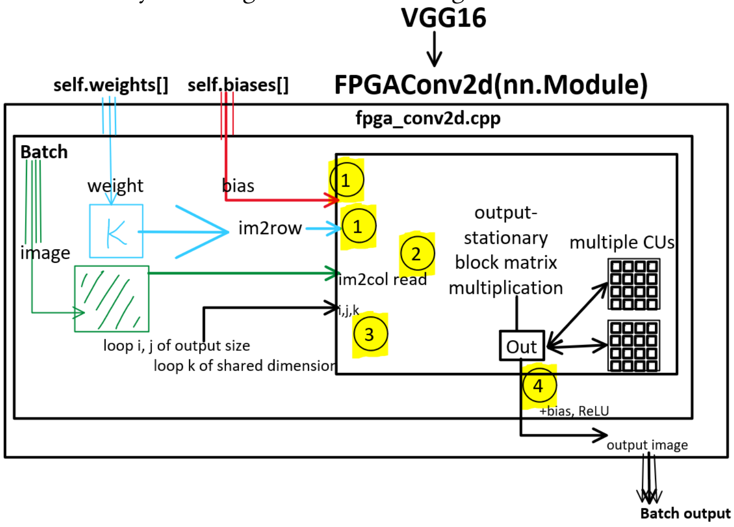


Figure 6: System diagram to optimize for input data reuse.

The logic for this attempt is that in (block) matrix multiplication, the same input blocks are requested multiple times for different output blocks. Instead of passing the input block to the kernel in repeated calls for different output blocks, we can simply pass the input to the it once, leave it there, and in subsequent calls to the kernel simply point to the requested block within the input. With the data sizes we are using, the DDR ports still have enough bandwidth to support the transferring of our matrices.

Additionally, since all of the data that any given block needs is already on the FPGA, I move the image's im2col logic into the burst reading of local buffers in the FPGA code. This allows us to cut out the time of the large im2col reading in each 2D convolution layer. The logic implementing this is presented in the appendix section 'Kernel Reading im2col.' With all of this logic already on the FPGA and the output-stationary accumulation also on the FPGA, it now makes sense to pass in the bias to the FPGA as well. Bias addition would be performed on the FPGA instead of in software. Therefore, in optimization attempt 2, the kernel is designed to hold all data, then given some pointer to a block to compute for, performs the corresponding im2col logic, systolic matrix multiplication and accumulation, addition of bias, and ReLU operation. This is reflected in figure 6. The advantage of this approach is that the bias and weights are the same for a whole batch, so we can perform step (1) just one time for a given layer. Steps (2) and (4) occur once for each image in the batch. Then step (3) can occur however many times as necessary to compute the entire block matrix multiplication per image. Step (1), which occurs least frequently, requires the most data transfer. Steps (2) and (4) occur infrequently, and requires the second-most amount of data transfer. Step (3) occurs frequently and requires little data transfer. This is appropriate for the application.

This approach was verified on CPU with a CPU function to "simulate" what my matrix multiplication kernel on FPGA would do. The file to verify it can be seen in the repository at `optimizations/src` with the command `python3 fpga_conv2d_bulk.py`. (Note: the environment in which this was tested was in a virtual environment on my local computer with a different setup than the AWS F1 FPGA, so that command may not work out of the box.) I then implemented this for the AWS F1 FPGA machine with new kernel and host code files. However, complications arose in software emulation in regards to either the event queueing or some memory allocation issues or another one of the many features to consider in programming FPGAs. Despite many late hours of investigation, I could not resolve this, so I ended up not being able to incorporate this idea into my final design. I anticipate this would have drastically sped up my FOM. (To see the code and design configuration that implements this on FPGA, it is in the repository at `optimizations/src/mmul_madd.cpp` and `optimizations/configs/design.cfg`.)

3 EVALUATION

Overall, the final FOM results are actually rather poor. Given my limited AWS credits and inability to resolve the issue discussed in the previous section that would have enabled further optimization, I ended up only being able to obtain the time it takes to run one inference of an input with batch size 1. I will infer the remainder of the FOM from that score as follows:

$$\begin{aligned}
Runtime(1)_{1024} &\sim 1024 \times Runtime(1)_1 \\
Runtime(32)_{1024} &\sim 32 \times Runtime(1)_{1024} \\
FOM &:= (Runtime(1)_{1024} + Runtime(32)_{1024})/2
\end{aligned}$$

Results:

Runtime(1)$_1$	1961.0650s

Table 1: Runtime of one inference of batch size 1, with FPGA "acceleration."

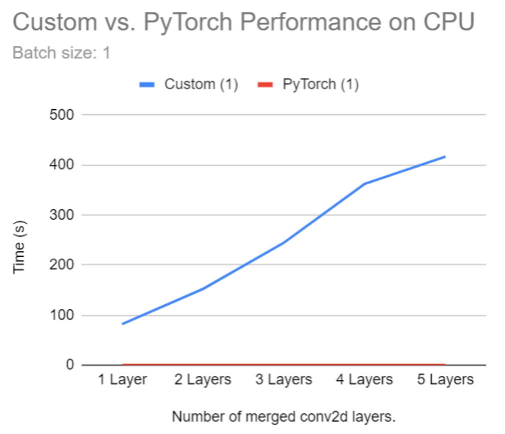
$$\begin{aligned}
Runtime(1)_{1024} &\sim 1024 \times 1961.0650s \\
&= 2,008,130.56s \\
Runtime(32)_{1024} &\sim 32 \times 2008130.56s \\
&= 64,260,177.92s \\
FOM &:= (Runtime(1)_{1024} + Runtime(32)_{1024})/2 \\
&= 33,134,154.4483s \\
&\sim 383.497days
\end{aligned}$$

However, I did also obtain results from intermediate steps toward optimization. They are outlined in the following tables and corresponding graphs.

Table 2 outlines the results from CPU execution of optimization attempt 2. Table 3 outlines the incremental runtimes for the final model, as we move through the layers of the model.

Layers	Batch size 1	Batch size 32	Pytorch (1)	PyTorch (32)
First merged layer	81.0032s	2634.1612s	0.1381s	1.7783s
Second merged layer	65.3880s	5064.0217s	0.0746s	5.6350s
Third merged layer	88.6572s	–	0.1686s	7.2868s
Fourth merged layer	99.7689s	–	0.0814s	8.1286s
Fifth merged features layer	56.1352s	–	0.0323s	8.8907s
All features layers	416.6312	–	0.3778	18.3484s

Table 2: Runtime of CPU run of optimization attempt 2



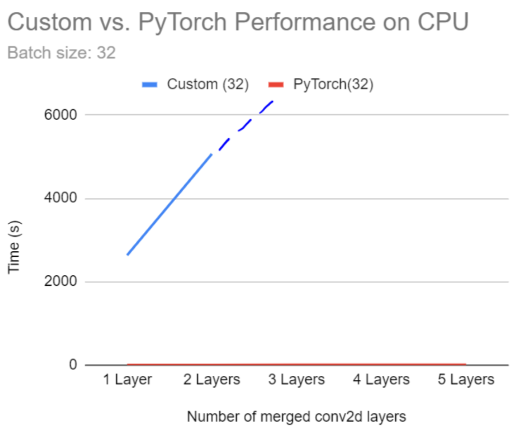


Figure 7: Accumulated CPU Runtime with Optimization Attempt 2 vs. vanilla PyTorch.

Observe that the custom implementations are much slower than that of the standard PyTorch. This is not surprising for several reasons, including but not limited to the following:

(1) The custom implementation is on CPU. There is no reason for it to be faster than the expert-created PyTorch unless I came up with some amazing ultra-optimizer for CPU. Assumingly, the PyTorch implementation involves micro and macro optimizations that I have not accounted for in my own implementation.
(2) The custom implementation involves extraneous data transfers and data transformations. Initialization of data structures to hold these, as well as the populating of these data structures, takes a significant amount of time.

Layers	Batch size 1	Batch size 1 (standard PyTorch)
First merged layer	429.2538s	0.2852s
Second merged layer	401.948s	0.0303s
Third merged layer	507.4159s	0.0706s
Fourth merged layer	456.6427s	0.0765s
Fifth merged layer	180.7612s	0.0269s

Table 3: Incremental runtimes for final implementation with VGG16 model

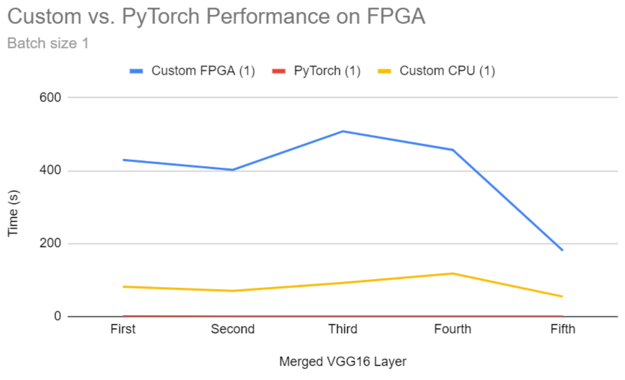


Figure 8: FPGA vs. CPU vs. PyTorch Merged Layer Runtimes

Observe that the custom implementation with the FPGA is in fact much slower than the custom CPU implementation, and even slower then the standard PyTorch implementation. This is mildly surprising, but can be explained by the following reasons:

(1) The cost of transferring data between software and hardware is high. In the current implementation, I transfer relatively small amounts of data (16 x 16) floats, which is $\sim 1024B$ per input and output, resulting in less thatn $4kB$ per data transfer. I call the kernel a total of slightly more than $\frac{out\_channels}{16} * \frac{size*size}{16}$ times, which can range from $\sim$ 25088 times in the first merged layer to $\sim$ 392 times in the last merged layer. It would be optimal minimize the number of these data transfers. This is why in Optimization Approach 2, I send all of the data over to the kernel first then only pass in an index pointer in all subsequent kernel calls. This should amortize the cost of a large data transfer on the front end, and total time spend on data transfer should go down.
(2) Multiple CUs have not been integrated in this design. Therefore, kernels must all complete in serial, even if kernels have no data dependency between them. In fact, since the code takes on an output-stationary approach, each nested outer loop is completely independent of all subsequent iterations. However, this has not been fully taken advantage of.
(3) im2col operations take a significant amount of time, especially as the matrix sizes are rather large in VGG16. In optimization attempt 2, I reduce im2col time by more than half because instead of a bulk im2col for every image, I only read in an im2col fashion in the kernel. This eliminated the need for the bulk im2col, and little to no time is added in the local buffer reading in the kernel because it would have to read to the local buffer anyway. (Any additional time may come from cache misses, as we are now no longer reading in contiguous data.)

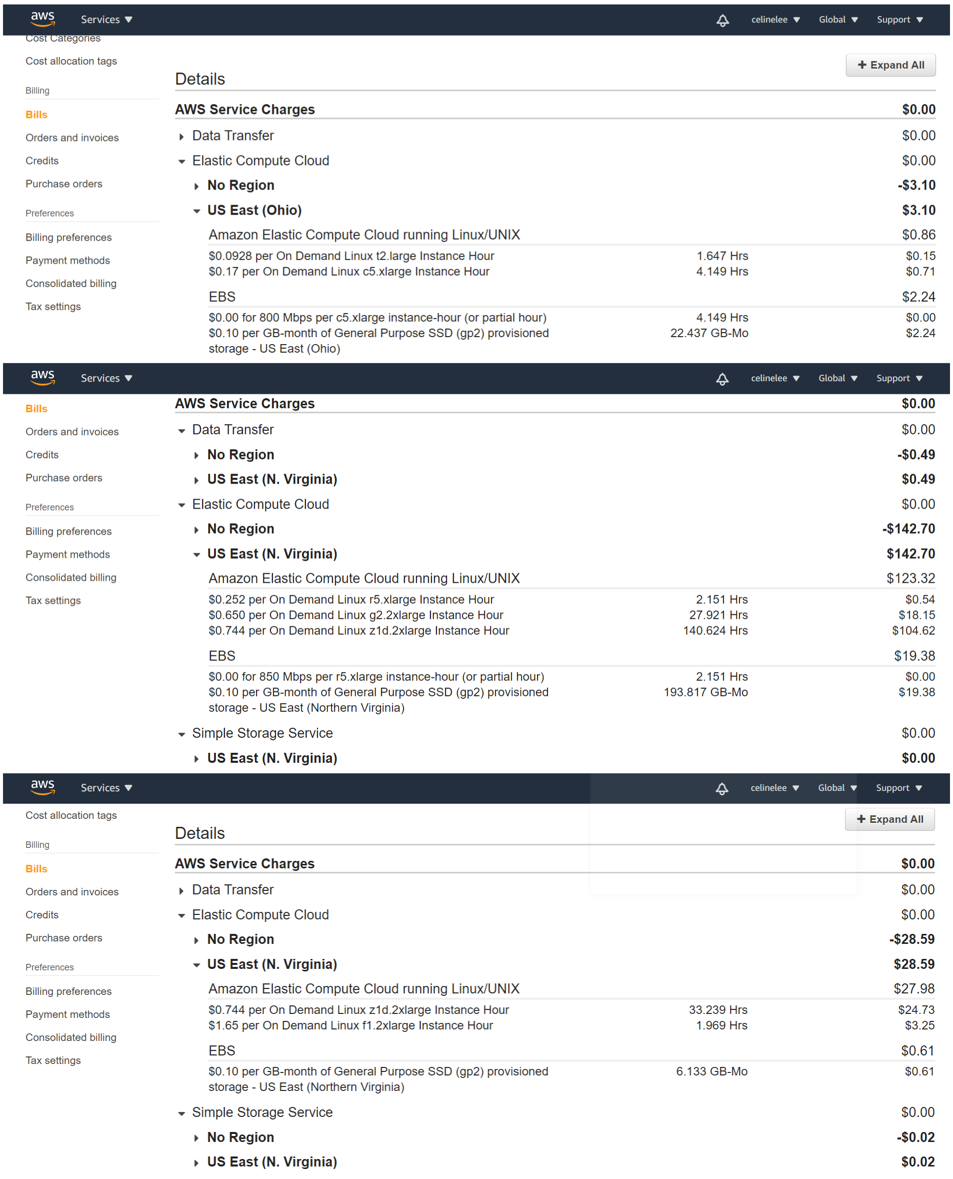
4 CONCLUSION

In this project, I have explored a large design space of hardware, software, and hardware-software optimizations for the 2D convolution operation. Such optimizations include loop unrolling, array partitioning, burst reading/writing, systolic matrix multiplication, minimization of data transfer across PyTorch bindings, minimization of required matrix block multiplications, minimization of im2col transformations, merging VGG16 layers, output-stationary matrix multiplication, input data reuse, and multiple compute units. Overall, despite the resulting system being more of a "drastic decelerator," I learned a great amount about system design and hardware-software optimization for machine learning acceleration. The next time I implement this or a similar project, I would like to additionally explore or adopt the following practices:

(1) Explore parallelism opportunities with larger batch sizes.
(2) Explore how to retain GEMM formatting between subsequent calls to neighboring 2d convolutions, in order to minimize im2col calls.
(3) Consider alternative algorithmic optimization techniques.
(4) For a given system design idea, set out more structured milestones that target specific optimizations. Read documentation before doing so.
(5) When designing systems ideas, perform the math to verify the logical validity of any optimizations.

A AWS USAGE

Out of concern about the AWS credit limit, I have documented my AWS usage prior to the month of December for bookkeeping.



Details

AWS Service Charges		**$0.00**
Data Transfer		$0.00
Elastic Compute Cloud		$0.00
No Region		**-$3.10**
US East (Ohio)		**$3.10**
Amazon Elastic Compute Cloud running Linux/UNIX		$0.86
$0.0928 per On Demand Linux t2.large Instance Hour	1.647 Hrs	$0.15
$0.17 per On Demand Linux c5.xlarge Instance Hour	4.149 Hrs	$0.71
EBS		$2.24
$0.00 for 800 Mbps per c5.xlarge instance-hour (or partial hour)	4.149 Hrs	$0.00
$0.10 per GB-month of General Purpose SSD (gp2) provisioned storage - US East (Ohio)	22.437 GB-Mo	$2.24

AWS Service Charges		**$0.00**
Data Transfer		$0.00
No Region		**-$0.49**
US East (N. Virginia)		**$0.49**
Elastic Compute Cloud		$0.00
No Region		**-$142.70**
US East (N. Virginia)		**$142.70**
Amazon Elastic Compute Cloud running Linux/UNIX		$123.32
$0.252 per On Demand Linux r5.xlarge Instance Hour	2.151 Hrs	$0.54
$0.650 per On Demand Linux g2.2xlarge Instance Hour	27.921 Hrs	$18.15
$0.744 per On Demand Linux z1d.2xlarge Instance Hour	140.624 Hrs	$104.62
EBS		$19.38
$0.00 for 850 Mbps per r5.xlarge instance-hour (or partial hour)	2.151 Hrs	$0.00
$0.10 per GB-month of General Purpose SSD (gp2) provisioned storage - US East (Northern Virginia)	193.817 GB-Mo	$19.38
Simple Storage Service		$0.00
US East (N. Virginia)		**$0.00**

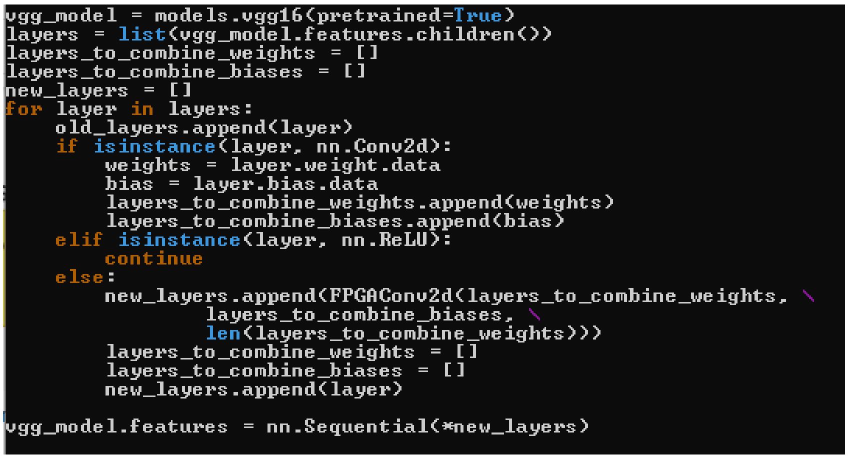
Details

AWS Service Charges		**$0.00**
Data Transfer		$0.00
Elastic Compute Cloud		$0.00
No Region		**-$28.59**
US East (N. Virginia)		**$28.59**
Amazon Elastic Compute Cloud running Linux/UNIX		$27.98
$0.744 per On Demand Linux z1d.2xlarge Instance Hour	33.239 Hrs	$24.73
$1.65 per On Demand Linux f1.2xlarge Instance Hour	1.969 Hrs	$3.25
EBS		$0.61
$0.10 per GB-month of General Purpose SSD (gp2) provisioned storage - US East (Northern Virginia)	6.133 GB-Mo	$0.61
Simple Storage Service		$0.00
No Region		**-$0.02**
US East (N. Virginia)		**$0.02**

A1: AWS Credit Usage from the months of September-November

At the time of writing this report, December credits have not been fully determined, so they are omitted.

B MERGING CONV2D LAYERS

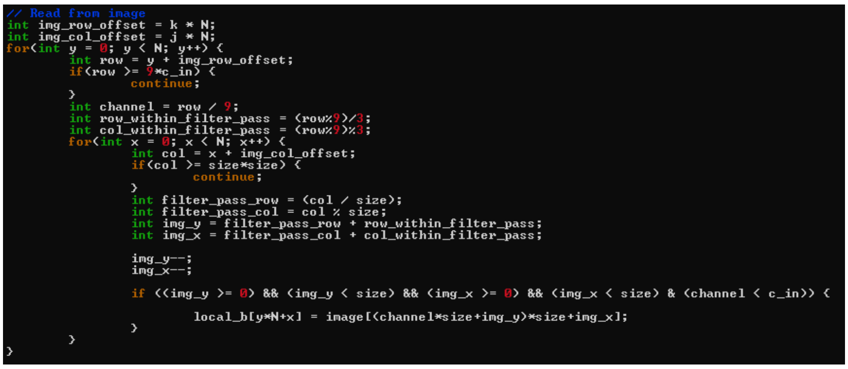


```python
vgg_model = models.vgg16(pretrained=True)
layers = list(vgg_model.features.children())
layers_to_combine_weights = []
layers_to_combine_biases = []
new_layers = []
for layer in layers:
    old_layers.append(layer)
    if isinstance(layer, nn.Conv2d):
        weights = layer.weight.data
        bias = layer.bias.data
        layers_to_combine_weights.append(weights)
        layers_to_combine_biases.append(bias)
    elif isinstance(layer, nn.ReLU):
        continue
    else:
        new_layers.append(FPGAConv2d(layers_to_combine_weights, \
                layers_to_combine_biases, \
                len(layers_to_combine_weights)))
        layers_to_combine_weights = []
        layers_to_combine_biases = []
        new_layers.append(layer)

vgg_model.features = nn.Sequential(*new_layers)
```

A2: Python code for merging VGG16 2d convolution layers.

C KERNEL READING IM2COL



```c
// Read from image
int img_row_offset = k * N;
int img_col_offset = j * N;
for(int y = 0; y < N; y++) {
        int row = y + img_row_offset;
        if(row >= 9*c_in) {
                continue;
        }
        int channel = row / 9;
        int row_within_filter_pass = (row%9)/3;
        int col_within_filter_pass = (row%9)%3;
        for(int x = 0; x < N; x++) {
                int col = x + img_col_offset;
                if(col >= size*size) {
                        continue;
                }
                int filter_pass_row = (col / size);
                int filter_pass_col = col % size;
                int img_y = filter_pass_row + row_within_filter_pass;
                int img_x = filter_pass_col + col_within_filter_pass;

                img_y--;
                img_x--;

                if ((img_y >= 0) && (img_y < size) && (img_x >= 0) && (img_x < size) & (channel < c_in)) {

                        local_b[y*N+x] = image[(channel*size+img_y)*size+img_x];
                }
        }
}
```