Design and Improvement of a FPGA based Accelerated Convolutional Neural Network

Rongqian Chen

University of Pennsylvania, Philadelphia, USA, willchan@seas.upenn.edu

Chen Chen

University of Pennsylvania, Philadelphia, USA, cchen82@seas.upenn.edu

1 ABSTRACT

VGG16 is a deep convolutional neural network(DCNN) model proposed in ILSVRC-2014, which achieved 92.7% accuracy in ImageNet. Although the model can be implemented by open-source libraries such as PyTorch, the execution time is still limited by the CPU architecture. In this paper, we choose FPGA which has an advantage in calculation and parallelization. Then we optimized the matrix multiplication problem by using parallel computing and increasing memory and bandwidth. Finally, we implement the FPGA on amazon AWS and successfully accelerate the convolutional layer in VGG16. Our final result is about 48 seconds for every image, which is several times faster than CPU multiplication method. Although the result still has a certain gap with the optimized algorithm in PyTorch, it proves the feasibility and potential of this method.

Additional Keywords and Phrases: VGG16, FPGA, convolution, multiplication acceleration

2 INTRODUCTION

Convolutional neural network(CNN) is a structure that used to extract features in computer vision. It is widely deployed in tasks such as target detection, image classification, scene understanding, and so on. However, While CNN's accuracy increasing in recent years by the improvement of structures, its depth and layers grow dramatically. The great numbers of parameters increase the computational complexity and memory load.

There are a number of approaches were proposed to solve this challenge. The first kind of solution is based on the improvement of structure convolution, such as 1D bottleneck layer, distributed model, etc. Another is used hardware computational devices. Currently, there are mainly three kinds of hardware platforms: Graphics Processing Unit(GPU), Application-Specific Integrated Circuit (ASIC), Field-Programmable Gate Array(FPGA). Among them, FPGA becomes popular in research because of its flexible architecture and low energy consumption.

In this paper, our acceleration solution mainly focuses on parallelization and memory bandwidth optimization in convolutional layer computation in VGG16 model[1]. The program is divided into software and hardware part. First, in the software stage, to implement convolution in FPGA, we use the GEMM[2] algorithm to transform the convolution into matrix multiplication. The "for" loops in the transformation between matrix and columns are replaced by python functions. Then, in the hardware stage, two kernels, loop unrolling, data flow, systolic array are deployed to maximum the parallelization. We optimized the memory usage by data reuse and store more matrix data in BRAM.

The rest of the report paper is organized as follows: the second part introduced the methodology of software and hardware, the third part illustrates the methodology, including software, hardware baseline and optimization. The fourth part is the evaluation of results, followed by the conclusion.

3 METHODOLOGY

This section will focus on the detailed system design, including software and hardware design.

The high-level architecture for our design is shown in Figure 1. It consists of a few building blocks, a custom VGG16 model, an `FPGA_conv2d` function, and a forward operator, which is the host.

Inside the customed model, every `Conv2d` layer is being replaced by the `FPGA_conv2d` function. The function transforms images into tensor arrays then passes to the forward operator. The job of the forward operator is operating tasks and calling FPGA hardware to perform matrix multiplication. The results are then returned to the software and

restored to images. After going through all the batches, the validation function will get an accuracy by comparing the predicted and correct results.

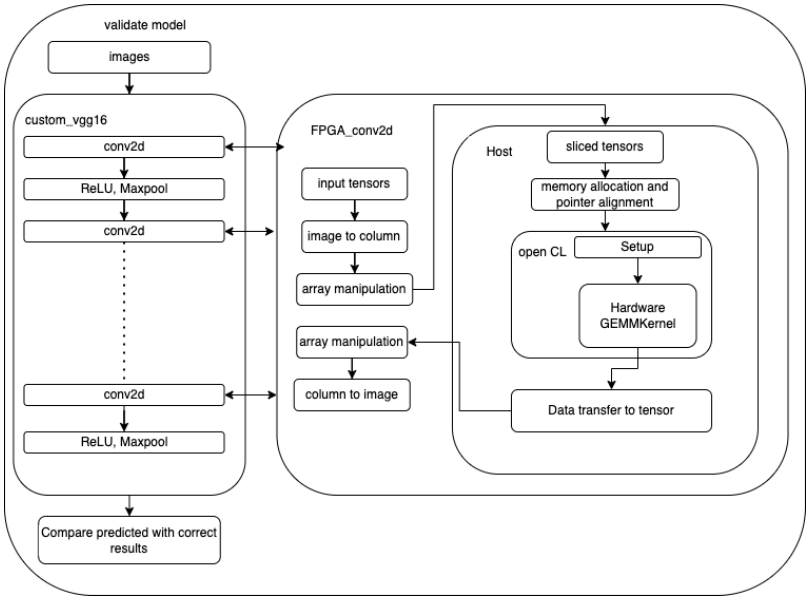

Figure 1: Overall architecture of the system.

3.1 software

In the software part, we use the PyTorch framework to build the VGG16 model and replace the convolutional layers with customized accelerated FPGA layers. The FPGA host is built by C++ OpenCL library and binds with python used C++ extension.

First, we start from a model with pretrained parameters. By searching and replacing `Conv2d` layer with `FPGA_conv2d` interface, we kept other layers unchanged. In customized convolution function, the 4-dimensional filter and image are transformed to two 2-dimensional matrices by image to column method according to the GEMM algorithm. The first matrix transformed by filter is organized as follows: every row represents a single 3D kernel, the weight of every 2-dimensional filter layer flatten to a 1-dimensional array then arranged in order of channels. Thus the height *A* of the filter matrix is the number of filters, the width *B* is channels*filter height*filter width. The filter and image matrix are shown in Figure 2(a). In our program, we used `torch.resize` to flatten each layer and `torch.cat` to concatenate the flattened array. The second matrix's width *C* is image height*image width, and its height is the same as the width *B* of filter matrix. The function torch.unfold is used in this procedure. By taking advantage of the PyTorch library, we do matrix transformation efficiently and conveniently.

Due to the limitation of kernel size, the matrix needs to be split into appropriate sizes. The systolic array in the kernel of FPGA is set to be 16*576, where 576 is the factor of matrix size: *host_B*. Then the matrix needs to be transposed before entering the host, for the convenience of taking column data. As shown in Figure 2(b), the matrix reshape into 1D array and enters the host, since the memory allocation in the host is one dimension. Finally, the host returns the result matrix with height as image channel and width as image size, by extracting every row of the result matrix and arrange in channels, we could restore the original result image, as Figure 3 shows.

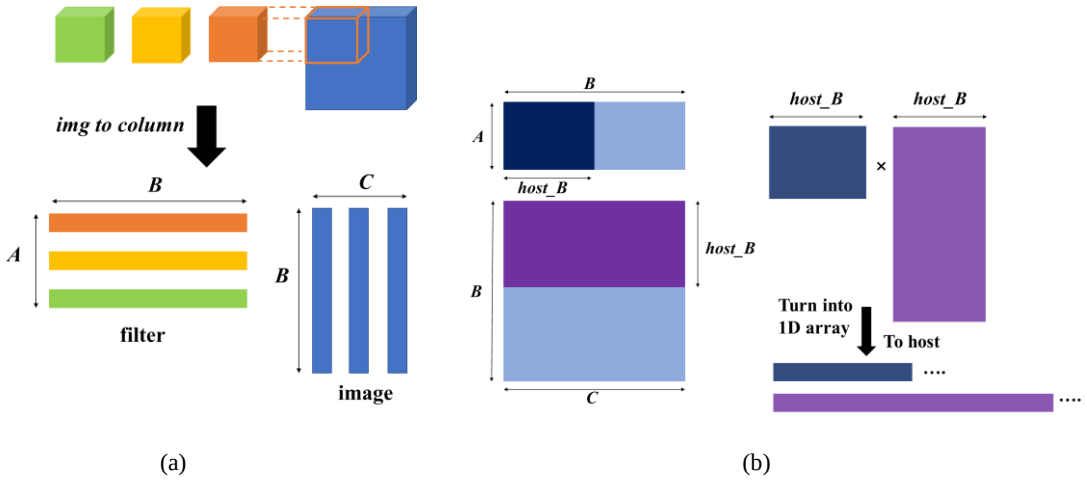

Figure 2: Matrix transformation before entering the host. (a) Image to column. (b) Cutting along size b.

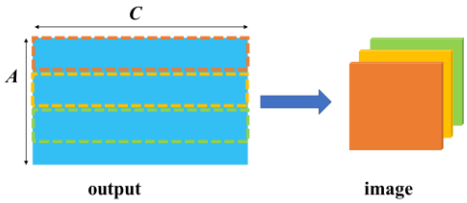

Figure 3: Reshape host output to image.

The second part of software design is the host code that configures the FPGA. Host file gets input from C++ extension and calls awsxclbin file for computing output. After initialization, which includes memory alignment and kernel setup, we need to use 2 loops to go through all the submatrix. Every submatrix has a size of `block-size` *576, where `block-size` is set as 16. Therefore the result will be 16*16 matrix after computation in the kernel, as Figure 4 shows. The size of block will be discussed in the next part, which is hardware implementation.

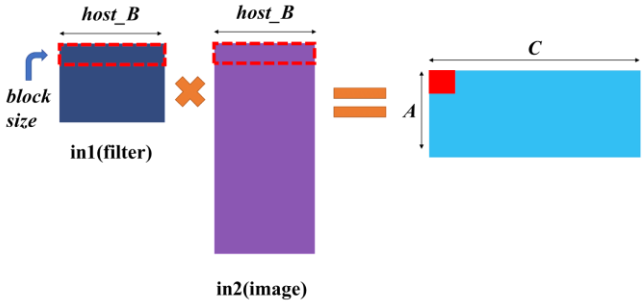

Figure 4: Submatrix load into the kernel.

To verify the feasibility of our method, we built a validation program that replaces the kernel computation with software matrix multiplication. The results are compared to the original VGG16 model results with the same random matrix input, it turned out that our customed model has the same output. Thus, its feasibility is proved.

3.2 Hardware

Since there is limited memory and compute resources in the kernel, the matrices should be cut into blocks. However, the size of the block needs a trade-off. Small `block-size` will lead to low utilization and inefficiency of the compute resources, and large block-size will cause failure in hardware compile. Eventually, we choose 16 as the `block-size`.

Then, we choose the systolic array as the basic compute structure. In the computation, two inputs are partitioned into 16*16 blocks and then the products are accumulated as Figure 5 shows.

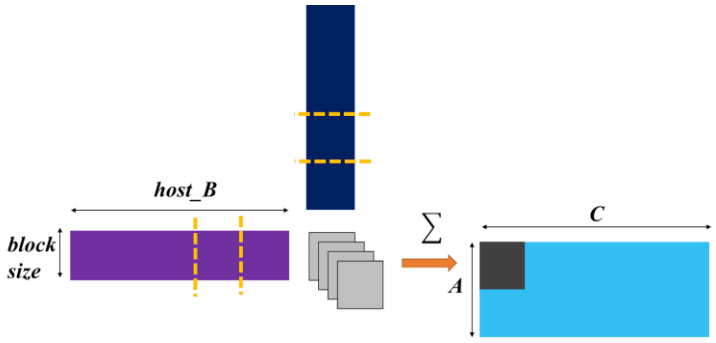

Figure 5: Submatrix load into the kernel.

However, the BRAM and DSP utilization still have room for improvement, which we will be detailed presented in the optimization part.

3.3 Milestones

The code presented in this section is partial. Full version of code is available upon request.

- C++ Extension with Convolution Software Emulation

  a) We first created PYBIND11 function in C++ using matrix multiplication in torch library as following.

```
torch::Tensor fpgaconv2d(torch::Tensor source_in1,
                         torch::Tensor source_in2,
                         int matrix_size_A,
                         int matrix_size_B,
                         int matrix_size_C)
  PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def("fpgaconv2d", &fpgaconv2d, "FPGA Conv2d Function Forward");
}
```

  b) Inside the Python script, we load the C++ extension using JIT

```
  fpgaconv2d_host = load(name='host_cpp',
        sources=['/home/centos/run_hardware/host.cpp'],
        extra_cflags=['-std=c++14'],
        extra_ldflags=['-L${XILINX_XRT}/lib/',
                  "-lOpenCL -lpthread -lrt -lstdc++"],
        extra_include_paths=['${XILINX_XRT}/include/',
                      '${XILINX_VIVADO}/include/'],
    verbose=True)
```

  c) Then we could call the `fpgaconv2d` function which was declared in the JIT to check the functionality correctness of our conv2d function in Python script

```
host_output = fpgaconv2d_host.fpgaconv2d(t_in1, t_in2, host_A, B_kernel, host_C)
```

- C++ Extension with Hardware Kernel and Software Emulation

  a) First, we compile the software emulation xclbin file using host file without JIT extension to ensure sw_emu.xclbin file is compiled using g++ and v++ (with flags `--compile --kernel`).

b) Then, we link the FPGA kernel with platform in SW emu target, set emulation variables using v++ (with flags `--link`) and

```
emconfigutil --platform $AWS_PLATFORM --od build
export XCL_EMULATION_MODE=$EMU_TARGET.
```

c) Lastly, use Python3 to call the Python script to emulate in software.

```
python3 conv2d_test.py
```

d) The same process is followed with hardware kernel, except we compile the xclbin file in HW target. Then we go to AWS to check whether the S3 bucket is still existed from previous assignments, create AFI on the cloud and get an awsxclbin file in our local machine, wait for the status of AWS image to be available. Then changed path of a binary file in the host file. Lastly, we launch `aws_deploy` instance, use scripts to set up Vitis, call the Python script to validate the hardware kernel is functionally correct.

- Python Benchmark Script in Software Emulation

a) A function called `validate_model` was created in python in addition to the previous milestones and finished our baseline algorithm. We used naïve matrix multiplication in C++ to substitute hardware kernel. And verified our model accuracy is the same as pretrained vgg16 model.

b) Timing was added to Python and C++ host, using `time` and `chrono` library. Also, we developed another C++ extension function to return timing to Python.

c) `time.perf_counter()` was used in Python to benchmark the performance. `std::chrono::high_resolution_clock::now()` was used to benchmark the C++ performance in detail.

d) We substituted back to the OpenCL version of the host file in C++, added benchmark timestamps.

e) Verify the benchmark script is working in software emulation like the previous milestone. And we stopped instance once we see timestamps are printing.

- Baseline design verified in software and hardware

a) For software verification of baseline design, we first verified the layer's parameters are the same, then we verified the output under the same input. The software verification program used naïve matrix multiplication in the host file without OpenCL.

b) For hardware verification, the steps were the same as previous milestones except we ran the end-to-end integrated system with the completed VGG16 model on AWS F1 instance.

- 2 Compute Units and Out of Order Command Queue

a) For 2 Compute units, we changed parameters in cfg file and changed OpenCL kernel assignment and event coordination according.

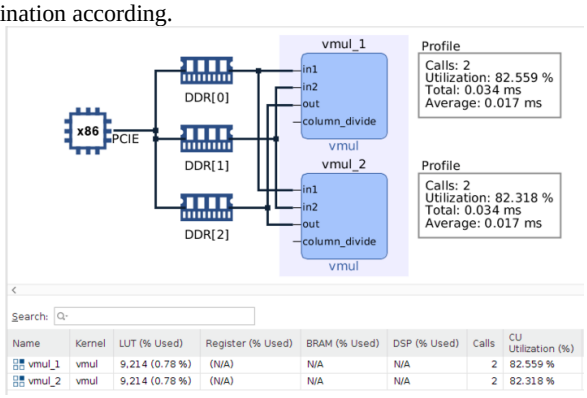

Figure 6: Compute unit and DDR structure.

b) In Out of order command queue, a flag was added to `CommandQueue` declaration.

Figure 7: Out of order in command queue.

3.4 Software Optimization

- For Loop Combination

  The matrix transformation uses several for loops that reduce the efficiency, including filter to column(4D to 2D) and output to image(1D to 2D). Intuitively, the method for the first problem would be using 3 for loops to row, channel, and filter number index. However, the filter layer can be obtained by using `torch.split` function on filter channels, then the layers are flattened to columns used resize function. So in this case we only require 1 loop for the filter number index.

  Typically, Restore output to image needs 4 for loops: 2 loops for reshaping the array to the block, and 2 loops for concatenating blocks into a 2D image. In our optimized program, we use `resize` and `split` function to build the blocks, and utilize `torch.cat` function to arrange the position of the block. In this case, we only need 2 for loops.

- Matrix Reorder by Transpose

  The matrix multiplication takes the row of the first matrix and the column of the second matrix. However, the array is stored in memory in order of rows, which means the column data need to extract over a row of data, as shown in Figure 8. After transposing, the data is stored continuously, the efficiency increased in data transmission.

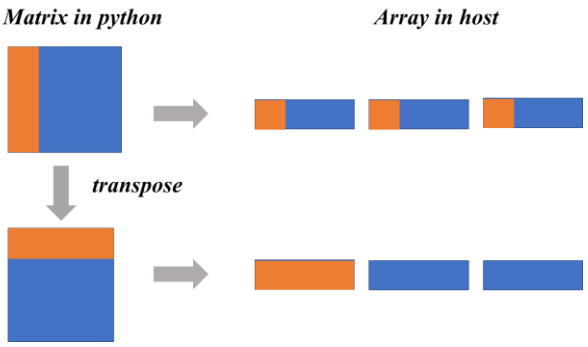

Figure 8: Matrix Reorder in the data extraction.

- Static Variable

As the OpenCL host is carefully timed, we found that every time the hardware kernel is called by the C++ extension, around 5 seconds are wasted on reading binary files, finding Xilinx platform, and creating command queue. Thus, static variables was used to prevent duplicated device initialization.

- C++ Optimize Flag

Another optimization we explored was C++ compiler optimization, we used -O3 flag when JIT compiles the C++ extension. As result, it greatly reduced runtime for the OpenCL host.

3.5 Hardware Optimization

- Baseline Design

In the baseline design, there was no pragma added in the kernel. And the kernel is performing matrix multiplication of two array sized 16*16*36 bytes. From the resource utilization, the DSP and BRAM utilization was low, as Figure 9 shows. And our DSP constrain is 1024 per compute unit. Thus, we need to use pragmas to increase the utilization.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+------+---------+---------+-----+
|         Name        | BRAM_18K|  DSP |    FF   |   LUT   | URAM|
+---------------------+---------+------+---------+---------+-----+
|DSP                  |        -|     -|        -|        -|    -|
|Expression           |        -|     -|        0|     1216|    -|
|FIFO                 |        -|     -|        -|        -|    -|
|Instance             |       90|    10|    21719|    14412|    -|
|Memory               |       38|     -|        0|        0|    -|
|Multiplexer          |        -|     -|        -|     1360|    -|
|Register             |        -|     -|     4611|       64|    -|
+---------------------+---------+------+---------+---------+-----+
|Total                |      128|    10|    26330|    17052|    0|
+---------------------+---------+------+---------+---------+-----+
|Available SLR        |     1440|  2280|   788160|   394080|  320|
+---------------------+---------+------+---------+---------+-----+
|Utilization SLR (%)  |        8|  ~0  |        3|        4|    0|
+---------------------+---------+------+---------+---------+-----+
|Available            |     4320|  6840|  2364480|  1182240|  960|
+---------------------+---------+------+---------+---------+-----+
|Utilization (%)      |        2|  ~0  |        1|        1|    0|
+---------------------+---------+------+---------+---------+-----+
```

Figure 9: baseline design resource utilization

- Array Partition

Array partition pragma increases the amount of reading and writing ports in the storage, which would increase the efficiency of data transfer and enables burst read and write.

- Pipeline

Pipeline pragma reduces the initiation interval for a function or loop by enabling the concurrency in execution. In which case, would increase the throughput of the function. So we tried to pipeline our read, write, and execution loops.

Also, the initiation interval(II) is closely related to the utilization of DSP when designing a systolic array. Manually limiting initiation interval will force HLS to decrease the number of DSP used for a systolic array. So we tried to build a 32 by 32 systolic array while limiting DSP usage and reducing the data movement cost. There was a noticeable speedup in hardware emulation, but the hardware kernel compilation failed due to strange reasons. So we stuck with 16 by 16 systolic array.

- Unroll

The unroll pragma transforms loops by creating multiple copies of the loop body which would allow some of the loop iterations to occur in parallel. When this pragma is added in read and write loops, it could increase the data transfer efficiency at a cost of increasing initiation interval. Although loop unrolling would create data dependency between functions and increase the initiation interval, it is still be compensated by dataflow pragma which enables task

parallelism. The disadvantage of implementing unroll pragma is taking too long to verify in hardware emulation and hardware compilation. Hence, we only unrolled the write loop to increase the efficiency of data movement.

- Systolic Array

The systolic array is a homogeneous network of tightly connected DSPs. And each DSP unit is shown as a processing element(PE) in Figure 7. Each PE is responsible for one multiplication and one addition. And each compute unit consists of 16*16 PE. The major advantage of using systolic arrays is that all data and results could be stored in scratchpad memory while minimizing memory usage during computation.

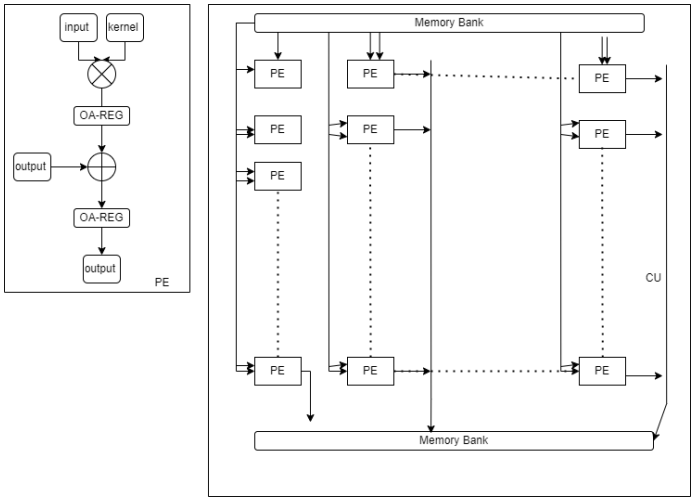

Figure 10: Systolic Array Hardware Architecture

In order to design a successful systolic array, we used an implicit approach to achieve it. We need to add pipeline, and array partitioning pragmas to our kernel.

- Dataflow

The dataflow pragma enables task-level pipelining and allows functions and loops to overlap in their operation which would increase the concurrency of the design. The difficulty of integrating dataflow into the systolic array is dataflow cannot be used with pipeline pragma in the same loop, and with dataflow being outside of loop (inferring task pipeline) would lead to clocking issues. Thus, we kept pipelining loops instead of dataflow.

- Read and Write Burst

Implicit burst read and write was implemented in the design utilizing array partitioning and we increase the BRAM utilization by increasing the amount of data read and write between the kernel and the host, which had a significant speedup.

- Out of Order

Out of order command queue enables tasks to be scheduled in parallel when the user defines the correct task dependency using events. We have implemented this technique into our OpenCL host.

- Two Compute Units and Multiple DDR banks

Using two compute units allows us to achieve higher throughput by running in parallel, due to the implementation out of order command queue, two compute units could. Also, we used multiple DDR banks to increase the bandwidth for burst read and write.

- Output memory movement

In our host, we used `Memcpy` function to accelerate the data movement in moving blocks of data in sequence. Thus, we transpose the image array to enable us to use `Memcpy` function efficiently. And this optimization prevented data movement using for loops and achieved time complexity O(N).

4 EVALUATION

In this part, we will present the detailed running time of the convolutional layers. In the test, we use 1 batch with only one image. Test conditions include baseline FPGA, optimized FPGA, and baseline CPU. The only difference between baseline FPGA and CPU is the matrix multiplication platform. In Table 1, we benchmark the running time in each layer.

Table 1: Running time of layers

Convolution Layers(in channel, out channel)	Baseline	Optimized	Baseline CPU
Layer0(3, 64)	29.71s	5.15s	0.53s
Layer2(64, 64)	28.75s	5.60s	1.71s
Layer5(64,128)	13.53s	2.44s	1.89s
Layer7(128,128)	24.70s	2.86s	1.26s
Layer10(128,256)	14.43s	1.39s	3.10s
Layer12(256,256)	29.25s	1.76s	6.44s
Layer14(256,256)	29.35s	1.78s	6.20s
Layer17(256,512)	22.52s	0.99s	8.59s
Layer19(512,512)	45.79s	1.35s	21.14s
Layer21(512,512)	45.54s	1.42s	21.81s
Layer24(512,512)	35.25s	0.53s	21.08s
Layer26(512,512)	30.28s	0.53s	20.80s
Layer28(512,512)	30.52s	0.53s	21.11s
Total	412.80s	25.80s	175.03s

From the above information, we successfully speed up every layer several times, and the total running time almost accelerates 16 times. The baseline system and CPU spend more time when images grow deeper, but optimized FPGA systems spend less time when depth increases. We ascribe time decreasing to larger BRAM size, which can load more data in one computation. The systolic array is processing so fast that the length of the input submatrix has less effect than memory loading. That is to say, the width and height of the image mainly determine the running time, and the depth of the image has less influence on the kernel.

When we further explore the running time in every layer, we can obtain another picture. Tables 2 shows the time-consuming of every part of the software program.

Table 2: Running time of software parts

Entry	Baseline	Optimized	Baseline CPU
Image and column transformation	170.80s	3.92s	166.05s
Data calculation	241.53s	21.86s	8.78s
Add bias	0.21s	0.18s	0.20s
Total	412.80s	25.96s	175.03s

Where image and column transformation includes image to column algorithm, matrix transpose, 1D array flatten and reshape host 1D array output back to images. Data calculation time involves host and kernel running time. This shows that the use of python functions and reduced for loops dramatically reduce the image and column transformation time. However, the optimized system's data calculation is still slower than CPU, we think this is because the matrix multiply function in PyTorch framework uses some powerful inbuild parallelization and optimized algorithms.

To further find out the running time in calculation part, we wrote a C++ extension in host.cpp to return the time consuming of four parts: data preparation, including memory copy and allocation; Kernel setup, which is FPGA configuration; Kernel calculation, including writing, executing, and reading time of kernel; Delete buffer, which is the

time for allocated buffer deletion. The running time of the optimized FPGA host is shown in Table 3, same as before, the test using 1 batch with 1 image.

Table 3: Running time of optimized FPGA host

Entry	Time/s
Data preparation	2.56
Kernel setup	0.29
Kernel calculation	16.48
Delete buffer	0.00012
Total	21.83

Therefore, the key to host acceleration is the optimization of the kernel. Also, data preparation can be optimized, a more efficient buffer and memory strategy are needed.

Finally, it's our benchmark with multiple batches and images. The comparison between the baseline and the optimized system is shown as follows.

Table 4: Benchmarking two systems with different batches

system	Batch size	Batch number	Accuracy	Time/s
Baseline	1	1	0	412.80
	16	1	87.5%	6588.48
Optimized	1	1	0	25.5
	1	16	87.5%	394.69
	16	1	87.5%	395.72

5 CONCLUSION

Our optimized system achieve 16 times acceleration compared to the baseline system. Optimization in software and hardware narrows our bottleneck down to kernel runtime. In software, we reordered arrays and loops to achieve efficient data movement, yet it is still costly compared to the original Pytorch functions in CPU. From Vitis application timeline, we find the most costly was the data movement. We were able to use implement out of order command queue with two compute units, which took full advantage of task-level parallelism. Also, multiple DDR banks and gmem ports were used to avoid any bottleneck in bandwidth. According to the reports generated from the actual hardware run, the CU utilization was 8%, kernel transfer bandwidth utilization was 30%, and host transfer bandwidth utilization was 1%. Thus, there is a significant amount of room for improvement.

For further development, data streaming and dataflow could be explored which would remove the bottleneck in data movement in theory. And ideally, image to column and column to image should also be moved to hardware kernel in order to take full advantage of FPGA processing speed.

Our final design made a significant improvement compared to the baseline design and was able to find a way to process a large amount of data inside the kernel. Although our customed VGG16 model was not as efficient as Pytorch pre-trained model, there is still a large amount of improvement we have not explored.

REFERENCES

[1] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[2] Anderson, Andrew, et al. "Low-memory gemm-based convolution algorithms for deep neural networks." arXiv preprint arXiv:1709.03395 (2017).