Accelerating VGG16 DCNN with an FPGA

DONGJOON PARK, University of Pennsylvania, USA
PRANOTI DHAMAL, University of Pennsylvania, USA

While CPU and GPU bring about the high productivity and flexibility in designing Deep-Convolutional-Neural-Network(DCNN), FPGA has recently gained popularity for efficiency in parallel tasks, like matrix multiplication. In this work, we demonstrated the potential of HW acceleration in PyTorch's convolution function. Our VGG16, integrated with HW accelerated convolution function, results in ×14.8 speedup in batch-1 case, and ×11 speedup in batch-16 case over our SW baseline. The Figure of Merit(FOM) is 1372 seconds.

Additional Key Words and Phrases: neural networks, DNN, PyTorch, FPGA, OpenCL

ACM Reference Format:
Dongjoon Park and Pranoti Dhamal. 2021. Accelerating VGG16 DCNN with an FPGA. In . ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

PyTorch is one of the most popular machine learning frameworks for both researchers and hobbyists. One interesting feature is that users can implement custom functions in C++ and import C++ functions from Python domain[1]. This feature can lead to performance improvement when executing parallel tasks that GPU or HW accelerator can excel.

FPGA has become a mainstream accelerator for DCNN inference for its performance and power efficiency. However, at the same time, FPGA is notorious of its unfriendliness, and its long compilation time and inherent complexities related to Hardware-Description-Languages(HDLs) often push software engineers away from using it. To lower the entrance barrier of FPGA, both academia and industry pour efforts in developing High-Level-Synthesis(HLS)[2]. HLS helps software programmers to easily generate RTL designs from C/C++ with appropriate pragmas. Furthermore, Xilinx Vitis platform lets users to easily integrate host code and RTL kernel, which highly alleviates the pain of SW/HW communication.

In this work, we demonstrate how users can utilize Xilinx FPGA from PyTorch to accelerate 2D convolution. We create FPGA-accelerated VGG16 whose 2D convolution functions are implemented in spatial systolic array on HW. The results show that FPGA-accelerated 2D convolution is an order of magnitude faster than the naive SW implementation although it is far slower than PyTorch's vanilla 2D convolution function.

2 METHODOLOGY

We completed all the milestones specified in the project handout: 1)Hardware kernel completion (without optimization, software emulation only), 2)Setup C++ extension with hardware kernel and software emulation, 3)Run Python benchmarking script in software emulation, 4)Baseline design

ESE539 Final Project, Philadelphia, PA,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

(non-optimized version) verified in software and hardware, 5)Software optimizations, 6)Use all two compute units and/or out of order command queue (non-exhaustive list).

We want to emphasize that we adhere to the rules; we must use 16×16 systolic array, and we must choose data type between either `float32` or `float16`. While increasing the size of the systolic array or changing the data type will naturally result in the performance improvement, they introduce an enormous extra design space.

If we are free to change the size of the systolic array, from the latency perspective, it is better to increase the size as far as the resource usage for DSP is not over 1024, the only resource constraint given. Additionally, we would like to have a rectangular systolic array instead of square shape in order to minimize the number of kernel calls. The optimal shape of the systolic array is related to the layer sizes of VGG16, and itself opens up a large design space exploration.

Quantization decreases latency accompanied with some accuracy loss. However, as long as there is no strict accuracy criterion we need to achieve, one can try fixed point, 8bit, 4bit, or even binarized neural network in the extreme case.

Therefore, we use 16×16 systolic array with `float32` data type so that we can fully explore data reuse, the use of multiple compute units, and overlapping communication and computation. In Sec. 3.2, we will show that the neural network with our new convolution function results in the exact same accuracy as PyTorch's original VGG16.

2.1 System Overview

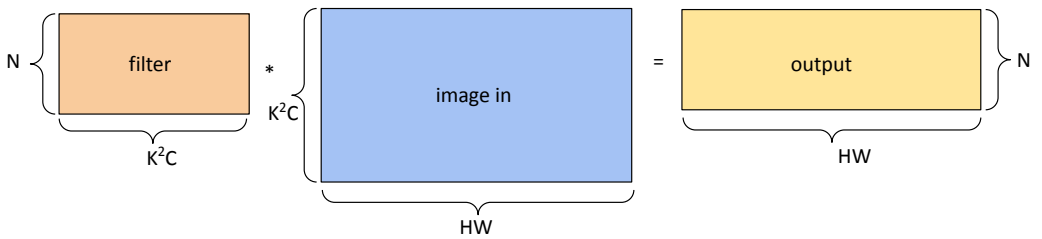

Fig. 1. Transform 2D convolution into Matrix Multiplication; C: input ch, N: output ch, K: filter width, height, H: output height, W: output width

As can be seen in Fig. 1, we can transform 2D convolution to 2D matrix multiplication. Input image needs to be reformatted with `im2col` function, and the matrix output also needs to be reformatted with `col2im` function in order to function identical to 2D convolution[4]. The key idea is that we want to accelerate 2D convolution by mapping matrix multiplication on HW, which is a natural fit for a spatial computation.

The system diagram is shown in Fig. 2. The left side of the diagram is the host that runs on x86, and the right side of the diagram is Virtex UltraScale+ FPGA in AWS F1 instance. We create `fpga_conv2d` that inherits PyTorch's `ConvNd` to create our own 2D convolution. We decide to perform `im2col` and `col2im` in SW and do matrix multiplication in HW using 16×16 systolic array. This design choice will be backed up by runtime breakdown data in Sec. 3.1.

An important design choice is that we keep the heights and widths of the matrices in Fig. 1 as the multiples of 16. This is done in SW by stretching the array size to the least multiple of 16 that is greater than the original size and by filling in zeros to the stretched part. In VGG16, N is always multiple of 16, but we cannot guarantee that K^2C and HW are the multiples of 16, which forces HW to deal with complex edge cases. Thus, we deliberately pad zeros to matrices in SW so that HW can always assume that the heights and the widths are multiples of 16.

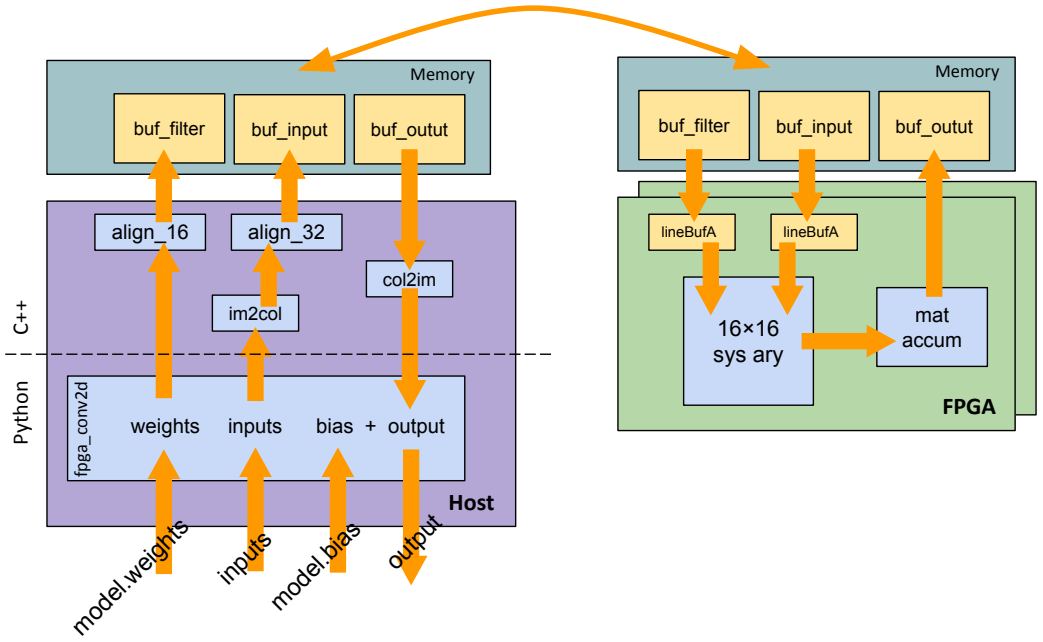

Fig. 2. System Diagram

HW computes matrix multiplications with 16×16 systolic array and accumulates result. When the computation is finished, the output data is migrated back to the host memory, and the host performs `col2im` to restore the image dimension. We add bias in Python domain to complete a computation for a single convolutional layer. Note that there are only two 16×16 systolic array compute units on FPGA, and each convolutional layer in VGG16 is computed sequentially.

2.2 Workflow

To ease the debugging burden, we make a small progress step by step as shown in Fig. 3. We start from creating a pure C++ implementation without any OpenCL constructs. This step verifies our `im2col` and `col2im` and sets up the high level system structure. Then, we create HLS kernel with Vitis HLS. The kernel is verified with C simulation(Milestone 1). The related codes pushed in the github are `mmult.cpp` and `mmult.h`. Next, we create OpenCL host code to utilize HLS kernel. In this step, a single convolution computation is tested with different sizes of inputs(Milestone 2). Once, we make sure that a single convolution works identical to PyTorch's 2D convolution, we perform inference for our network integrated with a HW accelerated convolution in SW emulation mode(Milestone 1, 3). The related Python code pushed in the github is `benchmark.py`. SW emulation is rather slow, but it checks functionality of our design. Finally, we build `xclbin` file, run on HW, and perform further optimizations(Milestone 4, 5, 6).

2.3 Kernel Optimization

This section is related to Milestone 6.

2.3.1 Data Reuse. As shown in Fig. 4, our kernel loads a line of filter data to a local memory(`lineBufferA`) and loads a line of input image to another local memory(`lineBufferB`). Note

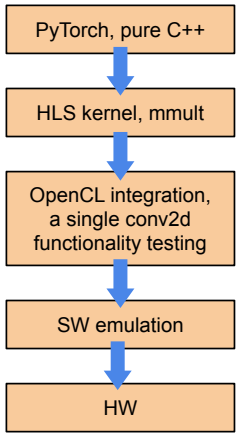

Fig. 3. Workflow

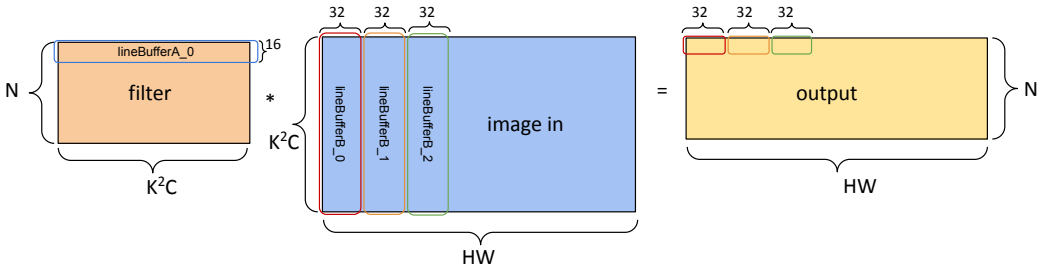

Fig. 4. lineBufferA is reused for different lineBufferBs; C: input ch, N: output ch, K: filter width, height, H: output height, W: output width

that the height of `lineBufferA` is 16, while the width of `lineBufferB` is 32. We initially have the width of 16 for `lineBufferB` as well but the runtime for the kernel was too short that we could not take advantages of using 2 compute units. When the width of `lineBufferB` is 16, by the time we enqueue the next kernel, the current kernel is already finished. Therefore, we increase the width of `lineBufferB` so that the kernel runtime is possibly doubled.

In our design, the size of `lineBufferA` and `lineBufferB` are maxed out to the maximum size of K^2C. In VGG16, it is $4608 = 512 \times 9$, so we simply set the maximum size of the local memory as 16×4700 and 4700×32 for `lineBufferA` and `lineBufferB` respectively. We are aware of the fact that the system will suffer from the severe under-utilization of the local memory, but we believe this is the simplest approach.

A single call of the HW kernel will generate 16×32 output block. Note that when we compute the next 16×32 output block, we do not need to re-read `lineBufferA`. So we exploit data reuse by loading `lineBufferA` only when the kernel computes the first column of the output matrix.

2.3.2 Multiple Compute Units. We utilize 2 compute units as specified in the project rules. To schedule 2 compute units, we can either run 2 compute units to compute a single image together or run 2 compute units to work on different batches of the images. These two different scheduling are compared in Sec. 3.1.

Table 1. Runtime of VGG16(seconds)

Design	(1,1)	(1,16)	(16,1)	(16,16)	FOM
SW	148	1800	N.M.	N.M.	N.M.
HW(1)	30	437	N.M.	N.M.	N.M.
HW(1), OoO, xclbin load	24	280	N.M.	N.M.	N.M.
HW(2), batch/CU	16	154	269	2515	1392
HW(2), together 1 img	**10**	**158**	**166**	**2578**	**1372**
PyTorch, vanilla	N.M.	N.M.	4.28	39	21.64

2.4 Host Optimization

This section is related to Milestone 5. We enable Out-order-Queue in the host code's command queue 1)to overlap communication and computation and 2)to enable concurrent kernel runs for 2 compute units [3]. We also read binary and program it only once. Host code is called from PyTorch 13 times because there are 13 convolutional layers in VGG16. Because we use the same `xclbin`, we can amortize the time spent on `clCreateProgramWithBianry` for 13 host code calls.

3 EVALUATION

3.1 Runtime

The results are illustrated in Tab. 1. Each tuple on the top row represents (number of inference, batch size). So (1, 1) is the runtime for a single inference in batch-1 case. The number after HW in *Design* column is the number of computing units. *N.M.* stands for "Not Measured." As expected, SW version without any HW acceleration takes a long time to finish even a single inference. *HW(1), OoO, xclbin load* represents the design with Out-of-Order queue and reduced `xclbin` load time discussed in Sec. 2.4. Note that we increase a kernel load for HW with 2 compute units as described in Sec. 2.3.1. *HW(2), batch/CU* is the version with 2 compute units, and each compute unit works on different batches.

HW(2), together 1 img is the version with 2 compute units, and two compute units together work on the same image as shown in Fig. 5. One compute unit is responsible for the upper half, and another is responsible for the lower half of the output image. FOM, defined as the average runtime of 16 inferences of batch-1 case and batch-16 case, is 1372 seconds.

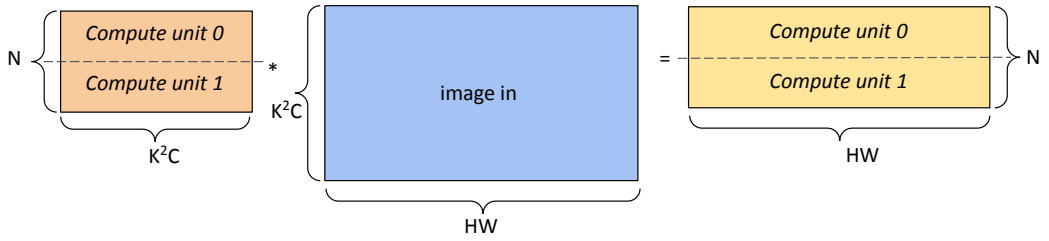

Fig. 5. Two compute units are scheduled to work on a single image

Fig. 6 shows the runtime of each components for both batch-1 case and batch-16 case in *HW(2), together 1 img* case. "host" includes the OpenCL setup, running kernels, and writing the output. Fig. 7 shows the runtime breakdown for "host". 99% of the time is spent on running kernels, which

Table 2. Accuracy(%)

Design.	batch-1	batch-16
HW(2), batch/CU	88.2353	86.7647
HW(2), together 1 img	**88.2353**	**86.7647**
PyTorch, together 1 img	88.2353	86.7647

includes reading/writing data from host to FPGA memory, enqueing tasks, and waiting for FPGA execution to finish.

When we check the Vitis Analyzer for the further profiling, the kernel is still too short even though we doubled the work done by kernel as discussed in Sec. 2.3.1. Furthermore, our host code calls kernels too frequently. Short kernel runtime and frequent kernel calls together create communication overhead. The easiest method to resolve the issue is to create lager systolic array, for instance 32×32. This will reduce the kernel calls about 4 times while increasing the kernel runtime about 4 times. As we mentioned early in this section, because we decided to stick to the project rules on the size of the systolic array and the data type, we leave this as the future work.

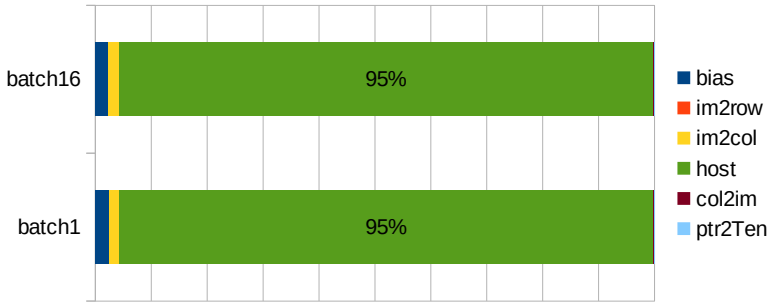

Fig. 6. Runtime breakdown

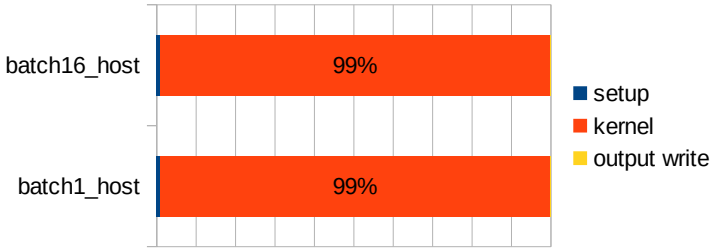

Fig. 7. "host" runtime breakdown

3.2 Accuracy

PyTorch's vanilla version returns the accuracy of 88.2353% and 86.7647% for batch-1 case and batch-16 case respectively. Both *HW(2), batch/CU* and *HW(2), together 1 img* return the accuracy of 88.2353% and 86.7647%, which are exactly same as the PyTorch's vanilla version.

Table 3. Resource Usage, for 2 kernels

	LUT	REG	BRAM	DSP
HW(2)	**226905(23.29%)**	**289336(14.08%)**	**514(27.25%)**	**716(10.48%)**

3.3 Resource Usage

We use only 716 DSPs, which are far less than the project limit, 2048 DSPs for 2 compute units. The % in the parenthesis is the utilization of the entire FPGA chip.

4 CONCLUSION

Although our HW-accelerated VGG16 is orders of magnitude slower than VGG16 with PyTorch's vanilla 2D convolution, ours outperforms the initial software version by ×14.8 and ×11 for batch-1 case and batch-16 case respectively. We believe that we can match or even outperform PyTorch's 2D convolution if we quantize the data type and increase the size of systolic array to fully utilize FPGA resources.

REFERENCES

[1] Peter Goldsborough. 2019. *Tutorials: Custom C++ and CUDA Extensions.* Retrieved December 12, 2021 from https://pytorch.org/tutorials/advanced/cpp_extension.html

[2] Xilinx Inc. 2021. *Vitis High-Level Synthesis User Guide.* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.

[3] Xilinx Inc. 2021. *Vitis Unified Software Platform Documentation: Application Acceleration Development.* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1393-vitis-application-acceleration.pdf.

[4] Manas Sahni. 2019. *Anatomy of a High-Speed Convolution.* Retrieved December 14, 2021 from https://sahnimanas.github.io/post/anatomy-of-a-high-performance-convolution/