ECE5775 Project Report
DNN Accelerator Generation Framework

Yanghui Ou (yo96), Peitian Pan (pp482)

December 12, 2018

1 Introduction

1.1 Background

In recent years, deep neural networks (DNNs) have achieved great improvements in the realm of artificial intelligence, demonstrating remarkable success in a wide range of difficult applications such as computer vision, speech recognition, natural language processing, etc. With their accuracy being closer or even better than a human's, DNNs are deployed in a wide range of systems, from large data centers to mobile devices and robots. Many researchers nowadays, inspired by these impressive breakthroughs, view DNNs as solutions to their problems.

Currently, many open-source frameworks have been released for DNN research, such as TensorFlow, Torch, Caffe, MxNet, CNTK, etc, each supported by an ambitious tech company or a strong academia leader. Among all these frameworks, TensorFLow, released by Google, is undoubtedly the most popular, as is shown in Figure 1. Most of these DNN frameworks has CPU/GPU backend support, but none of them support ASIC or FPGA.

Due to their complex and deep topological structures, DNNs are known to be computationally-intensive and memory intensive. It is challenging to deploy DNNs in both large scale data centers and small mobile devices. Considering performance, flexibility, and energy efficiency, FPGA-based accelerator for DNNs is recognized as a promising solution. Unfortunately, programming FPGAs can be rather difficult, even with the support of High Level Synthesis (HLS) tools. Conventional FPGA design flow makes it difficult for DNN researchers to conveniently take advantage of FPGA computing power. They will have to be FPGA experts in order to implement an efficient DNN accelerator.

1.2 Scope of the project

According to the analysis above, there is a strong demand for a easy-to-use framework that can automatically map CNN models onto FPGAs. To achieve this, we propose a CNN accelerator generation framework that takes high-level CNN model description as input, and execute the model inference on the FPGA-based accelerator generated by our framework. The focus of this project is to provide an easy-to-use framework, with reasonably good performance, for machine learning researchers, rather than to achieve the state-of-the-art performance for FPGA accelerators.

We plan to make the following contributions:

- We will build an end-to-end framework that automatically maps CNNs onto FPGAs for model inference. Our framework will support most sequential CNN models. This automated framework can significantly save FPGA design and implementation time.

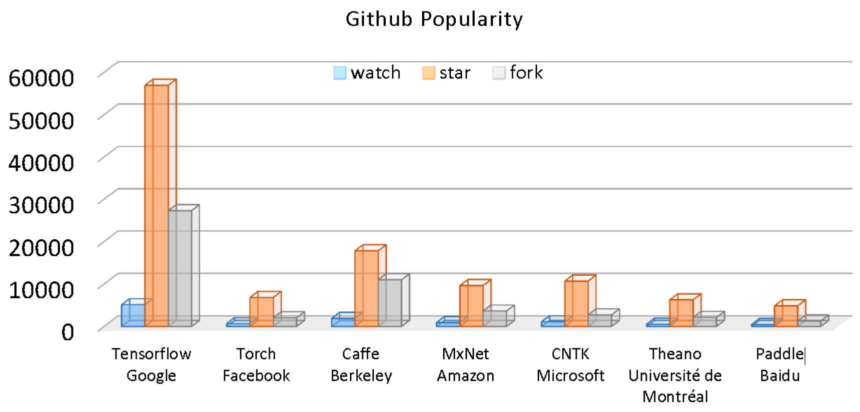


Figure 1: Popularity of open-source DNN framework on Github.

- We will implement our own quantization tool to perform post-training quantization and to dump the quantized coefficients into `C++` program.
- We will implement a model parser in python that analyzes a high-level CNN model and generates the execution schedule as well as the hardware configuration for each acceleration kernel.
- We will implement some OpenCL templates and use these templates to generate the kernel codes for each acceleration kernel. We will use SDAccel to compile our kernel codes into bitstream.

We are well-aware that it is hard for pure HLS design to achieve the best performance. For large designs that use thousands of DSPs, HLS design typically require longer clock cycles than RTL design. One solution to this is to adopt an HLS-RTL hybrid approach, implementing the complex control logic with HLS while the computation engine in RTL. However, due to limited time for this project, we will use pure HLS implementations only, as interfacing the HLS kernel and RTL kernel require a lot of engineering effort.

2 Techniques - Automated Mapping Flow of CNN onto FPGA

2.1 Input - Tensorflow

Tensorflow is an open-source library for dataflow programming. It is widely used for machine learning applications such as training and inferring neural networks. Neural networks are modeled by tensors and the operations performed on them. Because of its high popularity, we choose Tensorflow as the frontend of our framework.
The input of our project is a Tensorflow program which specifies the neural network our accelerator targets. The user can specify any sequential neural networks (i.e. networks with a linear stack of layers) with the following layers:

- Convolution layer: `conv2d`
- Fully-connected layer: `fc`
- Reshape layer: `reshape`
- Activation layer: `relu`
- Pooling layer: `max_pooling2d`

A few limits exist regarding which kinds of combinations of the above layers can be accepted by our framework:

- The first layer of the network must be a reshape layer which specifies the shape of the input feature maps (in the form of a Tensorflow `shape` object).
- The activation layer is optional and, when in its presence, must follow either a convolution layer or a fully-connected layer.
- The pooling layer is optional and, when in its presence, must follow either an activation layer, or a convolution/fully-connected layer.
- Except for the first layer in the network, the reshape layer is optional and, when in its presence, must precedes a convolution/fully-connected layer.

2.2 Output - OpenCL

OpenCL is a framework for writing programs that can execute across heterogeneous platforms, including CPU, GPU, FPGAs, and other hardware acclerators. Xilinx SDAccel tool can generate the RTL design from OpenCL programs through HLS. We choose OpenCL as the backend of our framework because it allows easy generation of parametrizable OpenCL programs and converts the program into RTL design.
The output of our framework is an OpenCL project that can be converted into RTL design using SDAccel 2017.2. The generated accelerator consists of the following kernels:

- `load_fmap`: refills the on-chip tile buffer from DRAM and feeds the data from tile buffer to `compute` kernel.
- `load_wts`: refills the on-chip weight buffer from DRAM and feeds the data from weight buffer to `compute` kernel.
- `compute`: refills the on-chip weight buffer and computes the convolution results on the weights and input feature maps.
- `acc_relu`: performs the (optional) activation layer computation.
- `pooling_wb`: performs the (optional) pooling computation and writes back the output data to DRAM.

Our framework also generates the host driver that calls a set of APIs to conveniently perform initialization, inference, and finalization on the accelerator. The APIs include:

- `CnnAccelerator::initialize()` Initialize the accelerator with information from the command lines. We also generates the makefile for the user to conveniently perform software/hardware emulation.
- `CnnAccelerator::load_wts()` Load the weights of a specific layer into the accelerator.
- `CnnAccelerator::create_buffer()` Create one buffer suitable for the input/output of the accelerator.
- `CnnAccelerator::load_wts()` Loads the weights of a specific layer into the accelerator.
- `CnnAccelerator::run_inference()` Perform the actual inference based on the input/output buffers and the loaded weights.

The inference results are stored in the buffer that was provided to the accelerator. After `run_inferece()` the user can use the inference results however they like.

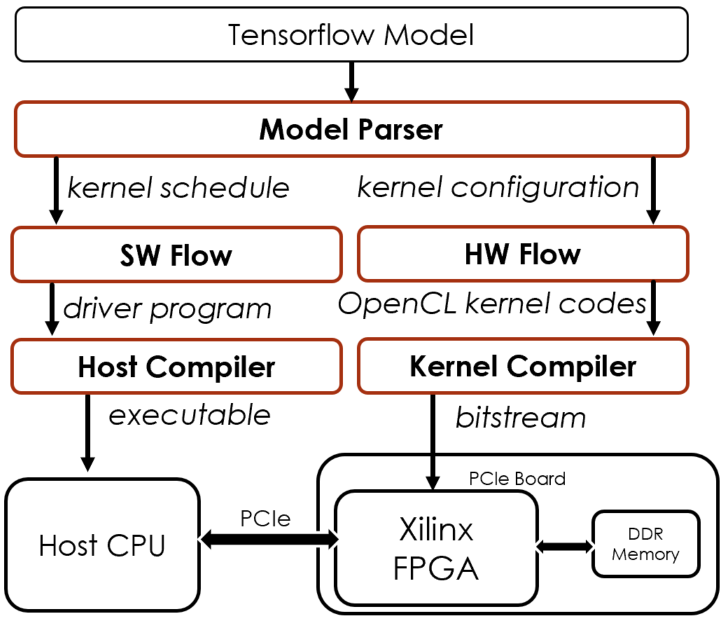


Figure 2: Overview of the proposed CNN accelerator generation framework.

2.3 Our Framework

Our proposed accelerator generation framework consists of four major components: the Tensorflow supporting framework, the model parser, the software flow, and the hardware flow. The user uses Tensorflow to describe the target CNN model, which will be quantized by the quantization tool[1]. The model parser takes the quantized CNN model and generates all kernel configurations and scheduling information. The software flow will take the scheduling information and generate the C++ driver program for the accelerator. On the hardware side, the hardware flow takes the kernel configuration and generates the OpenCL kernel codes by filling the kernel templates. The user can compile the driver program and the kernel codes into the host executable and bitstream, which are ready to be executed on the host CPU and the target FPGA device.

3 Implementation

3.1 Software Implementation

3.1.1 Model parser

The model parser analyzes the quantized model from the quantization tool and generates kernel scheduling and configuration information for the software and hardware flow. The configuration for each layer of the CNN (such as the size of the feature maps, the input channel, the filter size, the number of filters, the stride and padding style, etc) will be extracted from the quantized model and delivered to the software flow for scheduling different types of kernels (corresponding to different types of layers). The same information is also delivered to the hardware flow to correctly configure the kernels so that their functionality matches the model specification. Finally, the model parser also passes the quantized weights to the software flow to correctly initialize the weight area of the DRAM.

[1]We thank Ritchie and Zhiru for letting us access their NN quantization library!

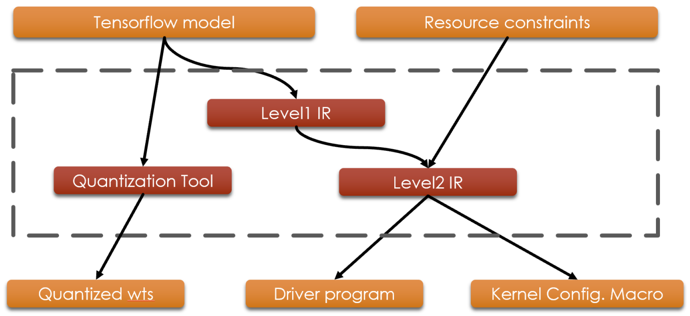


Figure 3: Overview of our model parser.

The model parser generates the configuration for each kernel from the specification of each layer. The concrete parameters such as the size of input feature maps, the size and number of filters, etc are determined from the model specification directly. For flexible parameters such as the tiling factors of input feature maps and how many weights should be stored on-chip, we reduce these factors into their influence on the operation intensity(defined as the number of operations performed inside the systolic array per byte loaded from the DRAM) of our accelerator. With some knowledge of the DRAM bandwidth and the amount of computation resources, we then quantitatively analyzes whether the operation intensity corresponds to a computation-bound or memory bandwidth-bound scenario using a roof line model. The desired flexible parameters are found through a exhaustive search in the design space and making ensure each computation kernel is computation-bound so that all computation resources are kept busy.

Figure 4 shows the roofline model for a `conv2d` layer with a stride of 1 and 32 3-by-3 filters. Let's assume that we have 256 DSPs in our design and that the DRAM bandwidth is 64 bytes per cycle. Under these assumptions, the minimum operational intensity required to reach the rooftop is only 8 operations per byte. If we unrealistically assume that all feature map data and weights data are stored in on-chip BRAMs (which means data reusage is maximized) the resulting operational intensity is 144 operations per byte, which is far more than we need. Therefore, we can reduce our on-chip storage, which will decrease the operational intensity. As long as the operational intensity is still higher than 8, i.e., we are still standing at the rooftop, the throughput is maximized since this layer can be considered as compute-bound. This is basically how we determine the tiling factor.

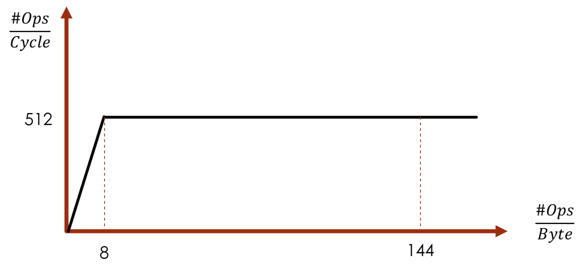


Figure 4: Roofline model for a convolution layer with $stride = 1, 32\ 3 \times 3$ filters, assuming we are using 256 DSPs, and that the DRAM bandwidth is 512 bits per cycle.

Standing at the rooftop essentially means that all of the computation resources are always busy computing, i.e., our input kernel is always feeding data to our compute kernel, as is shown in Figure 5. The input kernel first fetches a tile of input feature map and some filters from DRAM, then starts feeding computation kernel while fetching the next tile of feature map and next batch of filters. If reading next batch of data is finished before the computation for this batch of data is done, then our computation kernel will always have data to compute, meaning that we are at a efficient design point.

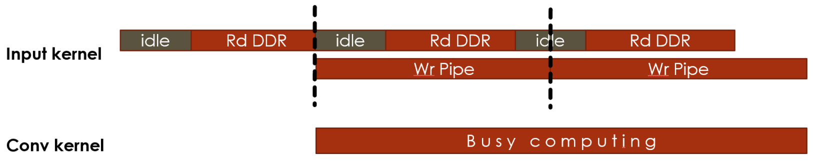


Figure 5: What happens at a lower level if we are standing at the rooftop.

3.1.2 Software flow

The software flow is responsible for creating the correct driver program of our accelerator. The driver is provided to users as an API which takes a pointer to a vector of images and the size and number of the images, and returns inference results as a pointer to a vector. The software flow needs to take care of several tasks. First, the driver needs to copy the input feature maps into the DRAM local to the FPGA. It then sets the correct arguments for each kernel by consulting the kernel scheduling information(which kernel should be executed and in which order). After that, it generates the correct sequence of OpenCL commands and pushes that into the device command queue. In addition, the software flow is also responsible for setting up the quantized weights in the DRAM that is connected to the FPGA. Finally, after the successful execution of the command sequence, it copies the results back into the host memory.

3.1.3 Hardware flow

The hardware flow is responsible for generating the HLS codes for acceleration kernels with pre-written OpenCL templates. The way it is done is very straightforward. It takes in the kernel configurations computed by the model parser and generate a header file that defines data types and constants that are referenced in the OpenCL templates. The kernel codes are then fed into the kernel compiler in SDAccel to generate the bitstream. Our OpenCL templates include input kernels that load input feature maps and weights data from DDR DRAM, a computation kernel that contains a systolic array and accelerates convolution, and other kernels that perform pooling and write back. We carefully implemented all the kernels so that they are highly-parametrized. We only need to change a few constants for different configurations.

3.2 Hardware implementation

As is mentioned above, our hardware is generated with some pre-written templates. The general architecture of our hardware implementation is shown is Figure 6. Each kernel communicates with CPU via a AXIS4-Lite bus. The CPU will send the arguments for a certain layer, such as the size of the feature map, convolution stride, pooling stride, etc, to the acceleration kernels. And our acceleration kernels are connected using a OpenCL data structure called `pipe`, which is essentially a FIFO. The input kernel reads data from the DDR DRAM and the pooling kernel is in charge of writing data back to DRAM. According to Xilinx's SDAccel optimization guide, a burst DDR transaction can be at most 4KB. Therefore, we allocate a 4KB buffer at the pooling kernel to buffer the output result. The pooling kernel will only perform write back when the output buffer is full or when there is no data left. In this way we make sure that writing to DDR does not become our performance bottleneck.

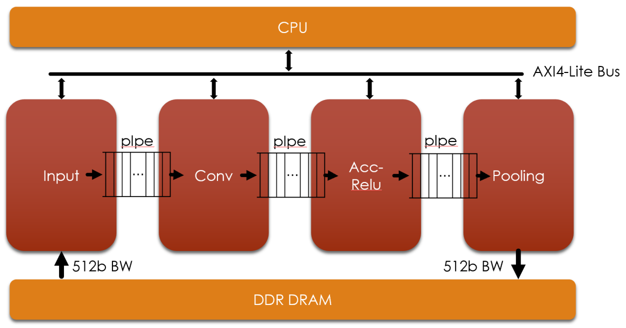


Figure 6: Overview of our hardware implementation.

3.2.1 Input Kernel

The input kernel is responsible for loading data from DRAM and storing them into the tile buffer BRAM. It also loads weights data from DRAM and sends it to the weights buffer in the compute kernel. It then feeds the feature map data to the inter-kernel FIFOs so that the computation kernel can compute the data correctly. The input kernel iteratively generates the DRAM address of each tile and reads it into the on-chip tile buffer. It then determines the tile buffer address of each piece of data to send by iterating over each pooling window and filter element. The input kernel also deals with the case where the filters are not completely stored on-chip so multiple iterations are needed to finish the computation.

3.2.2 Computation Kernel

The key component in our computation kernel is a dot product unit as is shouwn in Figure 7. We first pack some pixels in a filter and their corresponding pixels in the feature map into vectors and feed them into the dot product unit. The dot-product is deeply pipelined and can take in two vectors each cycle and produce one product each cycle once the pipeline is fully loaded.

An array of such dot product units forms a systolic array, as is illustrated in Figure 8, and N filter buffers

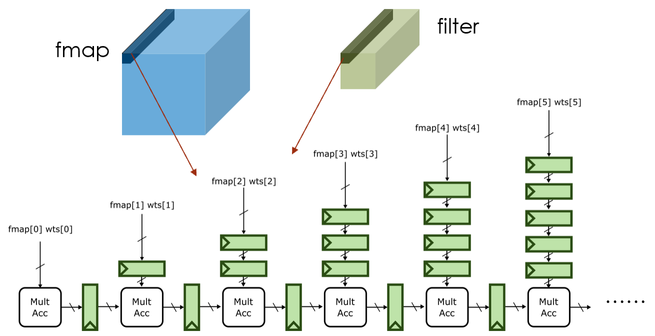


Figure 7: Microarchitecture of our dot product unit.

that stores a subset of filters on-chip. In each cycle, the systolic array consumes one feature map vector from the input kernel, and Each row of the systolic array compute the dot product of a feature map vector and N weights vector, one from each filter buffer, and produces N partial sums for the output pixel. The partial sums will be written to the output pipe and other kernels will accumulate the partial sum, performs pooling, and write back the resutls to DDR DRAM.

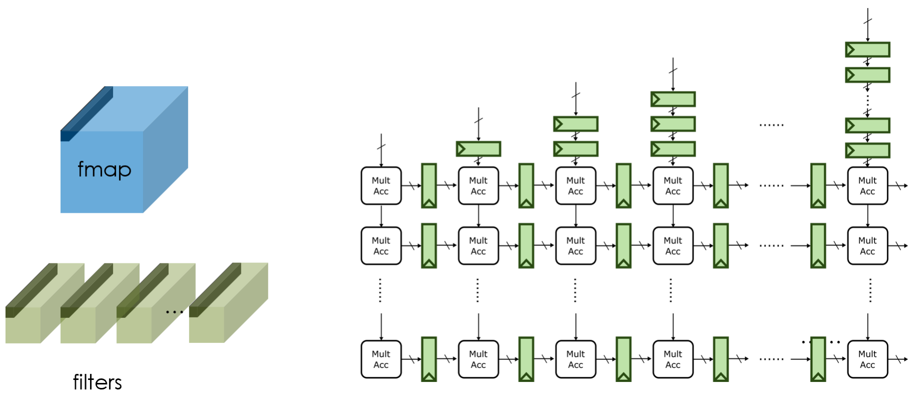


Figure 8: Micro-architecture of the systolic array.

3.3 Communication Optimization

Since our accelerators need to frequently communicate with off-chip DDR DRAM for inputs and outputs, the achieved effective DRAM bandwidth is also an important factor to be considered. Many studies have shown that increasing the DRAM burst width can raise up the effective DRAM bandwidth. To achieve this, we need to avoid discontinuous access to DRAM as much as possible.

We therefore adopt a channel-major representation for each input feature map to achieve efficient data communication. Specifically, the feature map is first divided into several "blocks", which have the same number of channels (i.e. the tile depth) as the number of on-chip filters and the same width and height as the input feature maps. Each block is subsequently tiled with the width and height tiling factors. We store each block by the order of channel-column-row, in this way we guarantee continuous access to DRAM when fetching an input tile.

Figure 9 demonstrates a comparison between the traditional row-major storage and our channel-major storage. As can be seen, even for a feature map with only 3 channels, channel-major storage results in a more continuous DRAM access. As the number of channels become larger, this effect will become even more obvious.

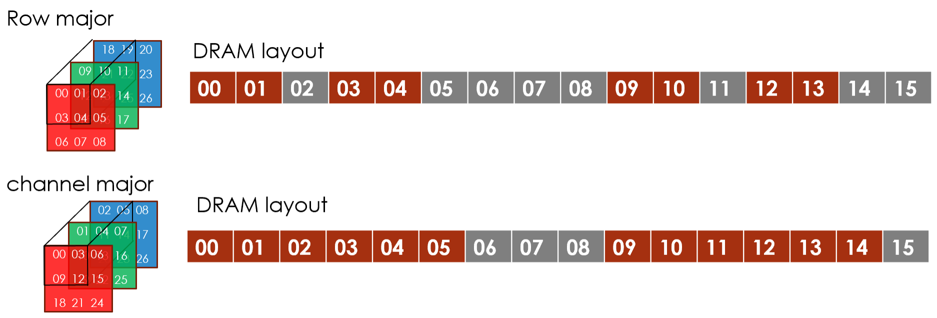
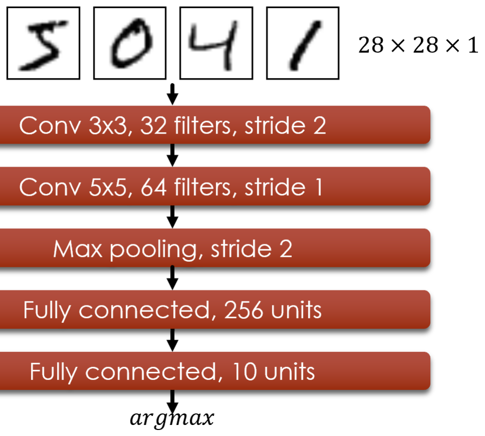


Figure 9: Comparison between row-major DRAM access and channel-major DRAM access.

4 Evaluation

We plan to evaluate our generation framework by implementing various pre-trained DNNs with it. To evaluate the productivity benefit of our framework, we will verify the RTL generated by the framework and compare the efforts to implement the target DNN with and without the DNN generator. We also qualitatively evaluate the performance of the generated accelerator. We evaluate our framework using MNIST dataset and a simple 2-conv 2-dense neural network. Figure 10 shows the configuration of the network in our evaluation.



28 × 28 × 1
Conv 3x3, 32 filters, stride 2
Conv 5x5, 64 filters, stride 1
Max pooling, stride 2
Fully connected, 256 units
Fully connected, 10 units
argmax

Figure 10: Configurations of our MNIST CNN.

4.1 Accuracy

Accuracy-wise, the baseline of our evaluation is the Tensorflow implementation and the alternative implementation is the accelerator generated by our framework. We train the baseline CNN using Tensorflow with 1000 train steps (each train step has a batch of 50 images). The final validation accuracy of the baseline is 96.92%. We then run inference on the generated accelerator in software emulation mode with 50 images from the first train batch. The generated accelerator achieves 94% accuracy on these 50 images, which is acceptable considering we are doing plain post-training quantization.
We also noted that the error of the accelerator output relative to the Tensorflow output is generally small (error value ranges from 2.0 to 50.0 using 16-bit fixed-point representation). This demonstrates that the impact of post-training quantization on accuracy is small and can be used in the accelerator to improve the energy/area efficiency.

4.2 Productivity

We demonstrate the productivity difference of developing a CNN accelerator with and without our framework. The baseline model is the development of a CNN accelerator corresponding to the model shown in Figure 10. We expect a small group of students (2 to 3) to be able to design, implement, verify the MNIST CNN accelerator with SDAccel in 3 to 5 days with close collaboration and intense development. Using our framework, however, requires only a running Tensorflow model which is usually less than 500 lines of Python code (including the model, training, validation, and other auxiliary functionality) and a few minutes to generates and compile the OpenCL program. The generated accelerator can be easily pushed through the FPGA flow with SDAccel.

5 Future Steps

There are a number of things that we can think of to further improve our framework:

- Run some experiments on real hardware and collect some feedback so that we know how to further optimize our hardware templates.
- Experiment with more neural network configurations to verify the correctness of our generation framework.
- Explore more advanced techniques to further improve the accelerator performance (e.g. can we pipeline the execution of different convolution layers? How to achieve higher performance when resources are abundant?).
- Reduce HLS overhead on performance by replacing our current HLS compute kernel with an RTL kernel.

6 Project Management

Since we are just a small team with only two students, we basically developed and debugged everything together. Peitian mainly contributes to the development of the `load_fmap` kernel which has the most complex control logic, while Yanghui developed the rest of the kernels. The model parser is largely done by Peitian, which is a lot of work, while Yanghui also contributes to the part that quantizes the weights and output the weights into a c++ readable format. Yanghui also developed a reference model in c++ for testing the correctness of our kernels and he also verified the results produced by our accelerator against the intermediate results dumped out directly from TensorFlow.

Our development for this project official starts on around November 15, when SDAccel is finally available on `ece-linux` server. Before then we just wrote some CNN models for MNIST dataset using TensorFlow. Our OpenCL kernels are finished during the thanksgiving holidays and the model parser are finished at the end of November. Overall, we deem our project as a success, considering the limited manpower of the team and tight schedule for both group members in November. This framework is literally built from scratch. The only thing that is not developed during this project is our makefile, which is taken from one of Yanghui's previous project.

7 Conclusion

We design and implement a truly *end-to-end* CNN accelerator generation framework. The framework uses Tensorflow as its frontend to specify the network configuration and outputs the generated accelerator in the form of OpenCL kernels. Our framework extracts the neural network configuration from a Tensorflow program and automatically chooses the optimal parameters for the accelerator hardware given the available resources of the FPGA board. The parameters will be filled into existing accelerator kernel templates to create an OpenCL project ready to be compiled by SDAccel. The generated project also includes a set of user-friendly APIs to initialize, run, and finalize the accelerator. We evaluate the accuracy and productivity of the generated accelerator. The generated accelerator achieves an acceptable accuracy of 94% on one training batch. Productivity-wise, our framework only requires a few hundreds of Python code to specify the model and less than ten minutes to generate and compile the OpenCL project. Our end-to-end framework significantly boosts the productivity of CNN accelerator development and has acceptable accuracy degradation (due to post-training quantization and overflow issues when executing fully-connected layer).