ECE 382V: Hardware Architectures for Machine Learning
Lab #3

Due date: Mar 5, 2025

Important Note

You may discuss the assignment and general concepts with other students; however, sharing code is strictly prohibited. You must submit only the code you write yourself or that which is provided with the assignment. Specifically, downloading code from the internet is not allowed. Any submitted code that violates this policy will result in no credit for the entire assignment.

Goal

- To further understand CNN dataflow.
- To understand the role of the CPU (host) in modern machine learning hardware.
- To examine data transmission between different hardware platforms.

Introduction

Various hardware options exist for machine learning, including CPUs, GPUs, FPGAs, and ASICs. Due to the highly parallel nature of CNNs, GPUs and FPGAs have become promising candidates for computational tasks in network inference and training. Beyond computation, additional responsibilities include compiling high-level CNN network descriptions into lower-level assembly code and managing data transmission between on-chip and off-chip memory.

A system that solely relies on GPUs or FPGAs may not achieve optimal performance without the support of CPUs. As a result, CPU+GPU and CPU+FPGA configurations are widely used in cutting-edge research. One of the role of CPUs in these systems is to serve as the host. The assignment will guide you in exploring the role of the CPU within machine learning systems.

The inspiration for this assignment comes from GLOW (GitHub) and TVM (GitHub), two compiler stacks designed for deep learning systems. These tools bridge the gap between productivity-focused deep learning frameworks and performance- and efficiency-oriented hardware backends.

In this assignment, you will develop a program to simulate the workflow of GLOW and TVM. The assignment consists of two parts:

1. The program will parse a network modified from Darknet (GitHub), extracting key parameters for each layer, such as input size, weight kernel size, and output size.

2. To maximize parallelism and enable convolution operations using vectorized instructions, the program will convert convolution operations into matrix multiplication.

Part 1: Implement Parameter Parsing [25 points]

Your task is to implement the obtain_parameters() function in the Network class located in network.h. The file input/output interfaces have already been implemented in the file_utils.h class. You may use the variable network_cfg_description, which stores each input file line. Alternatively, you can implement your own file input interface, but this must be done exclusively within the network.h.

Within the obtain_parameters() function, you need to determine the following member variables of the Network class:

```
// 'conv' refers to convolution layers
int layer_number;  // Total number of convolution layers
std::vector<int> input_height;  // input_height[i]: Height of the input for the i-th conv layer
std::vector<int> input_width;  // input_width[i]: Width of the input for the i-th conv layer
std::vector<int> input_channel;  // input_channel[i]: Number of input feature map channels for the i-th conv layer
std::vector<int> kernel_dimension;  // kernel_dimension[i]: Number of filters in the i-th conv layer
std::vector<int> kernel_size;  // kernel_size[i]: Height/width of the kernel in the i-th conv layer (assumed to be square)
std::vector<int> kernel_channel;  // kernel_channel[i]: Number of input channels for the i-th conv layer (same as input_channel[i])
std::vector<int> output_height;  // output_height[i]: Height of the output for the i-th conv layer
std::vector<int> output_width;  // output_width[i]: Width of the output for the i-th conv layer
std::vector<int> output_channel;  // output_channel[i]: Number of output feature map channels for the i-th conv layer (same as kernel_dimension[i])
```

To simplify this problem, we focus only on convolution layers, as convolution operations account for more than 90% of the computation in a neural network. You can assume the network follows a purely linear structure with no branches. The configuration (.cfg) file will contain only three types of labels: [net], [maxpool], and [convolutional].

Input: network_*.cfg
Output: network_*.parameter

To compile the code, you might use the following command: *g++ -o main main.cpp*

Part 2: Implementing CONV to Matrix Multiplication [30 points]

To fully exploit the parallelism of convolution operations and deploy convolution layers on hardware platforms that support matrix accelerators and vector-processing ISAs, many research studies opt to convert convolution operations into matrix multiplication. The implementation of matrix accelerators have be been explored on multiple hardware platforms.

Your task in this part is to implement the function conv_convert() in the Network class within the network.h. You may refer to Figure 2 in [1] or the blog [2] for details on converting the initial inputs into inputs suitable for matrix multiplication units. Additionally, the sliding window unit in [3] provides a useful reference for understanding how convolution is transformed into matrix multiplication.

Function Signature:

```
int Network<T>::conv_convert(int layer_id, int padding, int stride,
                             Array3D<T>& initial_input,
                             Array4D<T>& initial_kernel,
                             Array2D<T>& input_matrix,
                             Array2D<T>& kernel_matrix)
```

initial_input: A three-dimensional array (input_height[layer_id], input_width[layer_id], input_channel[layer_id]).

initial_kernel: A four-dimensional array (kernel_dimension[layer_id], kernel_size[layer_id], kernel_size[layer_id], kernel_channel[layer_id]).

To avoid using raw pointers in C++ for high-dimensional arrays, we provide the data structures Array1D, Array2D, Array3D, and Array4D to help with implementation.

You need to determine the contents of the 2D arrays input_matrix and kernel_matrix, which will serve as the inputs for matrix multiplication units. Since we will test your code with different padding and stride values, ***you must not use the output_height, output_width, and output_channel variables generated in Part 1***.

We have provided shell code for parsing inputs and kernels and generating output files. Below are examples of input and output formats:

Input: network_*.layer_*.initial_input, network_*.layer_*.initial_kernel

The first line of initial_input specifies the input's height, width, channel, padding, and stride.

The first line of initial_kernel specifies the weight kernel's dimension, height, width, and channel. These values are randomly generated integers with no specific meaning. You can modify the function generate_input_kernel() in test.h to generate custom inputs and kernels for debugging.

Output: network_*.layer_*.input_matrix, network_*.layer_*.kernel_matrix

Note: ***There may be multiple valid ways to expand the input in different orders. However, we only accept the expansion order: channel → width → height. This order must match the provided example and is the only valid solution.***

Part 3: Implementing CONV Converter with Stream Interface [35 points]

This part extends Part 2 to support additional data transmission interfaces between different hardware platforms. In Part 2, we provided a memory-mapped interface for the function conv_convert(), allowing the program to randomly access input and output data. The workflow of Part 2 is as follows:

1. CPUs generate the input matrix and kernel matrix, storing them in DRAM.
2. CPUs send a synchronization signal to FPGAs/GPUs, allowing them to fetch data from DRAM.
3. FPGAs/GPUs perform matrix multiplication and send the computed results back to DRAM.
4. FPGAs/GPUs send a synchronization signal back to CPUs, which process the output data and determine whether to call conv_convert() again for the next convolution layer.

This workflow cannot be efficiently pipelined due to data hazards—meaning that accelerators (FPGAs/GPUs) must wait until CPUs fully generate the entire matrix before computation can begin.

To resolve this, we introduce a stream interface, which is commonly supported in CPUs+FPGAs and CPUs+GPUs systems. Streams function like a conveyor belt, processing items one at a time instead of in large batches. With streaming, the matrix is divided into smaller chunks, allowing FPGAs/GPUs to begin computation as soon as

they receive the first row/column rather than waiting for the entire matrix. As a result, CPUs and accelerators can work simultaneously, improving overall throughput. In this part, you need to implement the function conv_convert_stream() in the Network class within the network.h:

Function Signature:

int conv_convert_stream(int layer_id, int padding, int step_size, Stream<T> &input, Stream<T> &output);

We have provided the data structure Stream in stream_utils.h for you, which is implemented by std::queue. Note that there should be a buffer to store the intermediate data, and the buffer could be implemented by memory, which has lower latency and higher bandwidth than DRAM, like cache. In this part, we use the variable buffer to represent the buffer between CPUs and GPUs/FPGAs. You are not allowed to change the buffer size since this size is enough for you to implement this function. Feel free to add several temporary variables (the number of temporary variables should be constant, independent of the input/output size) as counters if needed.

We have provided shell code for parsing inputs and kernels and generating output files.

Input: network_*.layer_*.initial_input

The input format remains the same as in Part 2.

Output: network_*.layer_*.input_matrix.stream

The output format should match network_.*layer*_.input_matrix from Part 2.

Part 4: Report [10 points]

Write a report detailing how you utilized the buffer in Part 3. The report should include:

- A description of which data is stored in the buffer.
- An explanation of how the buffer is updated when it becomes full.

Additionally, feel free to include any comments or feedback about this assignment in your report.

Deliverables

You must submit 2 files: “report.pdf” and “network.h”.

FAQs

1. Please use “*g++ -o main main.cpp -std=c++14*” to compile on your own laptop or TACC.
2. Padding is not a boolean; it is an integer. So, it defines how many rows/columns of zeros are padded on the left, right, top, and bottom.
3. Take a look at these three references, which should be helpful for you to understand how convolution is converted into matrix multiplication.
4. Feel free to include any libraries or define data structures in "network.h". However, ensure your code runs without any compiling error in C++ 14.
5. In order not to overwrite the example input of part 2&3, you may choose to comment “test- >generate_input_kernel();” in main.cpp. Then, the simulator will use example input instead of random input.
6. You can assume that the stride number is always smaller than or equal to kernel size.
7. You can assume that the padding size is smaller than “kernel_size / 2”.
8. You can assume the height and width of each input is the same.
9. You can assume that there are no consecutive pooling layers in the test cases, and the first layer is always a convolution layer.

References

1. Chellapilla, Kumar, Sidd Puri, and Patrice Simard. *High-Performance Convolutional Neural Networks for Document Processing*. 2006.
2. Medium Blog Post: An Illustrated Explanation of Performing 2D Convolutions Using Matrix Multiplications.
3. Umuroglu, Yaman, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. *FINN: A Framework for Fast, Scalable Binarized Neural Network Inference.* In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65-74, 2017.