Lab3
Understanding the role of CPUs in ML System

Programming assignments are to be done individually or in a group of 2. You may discuss the problem and general concepts with other students, but there should be no sharing of code. You may not submit code other than that which you write yourself or is provided with the assignment. This restriction specifically prohibits downloading code from the Internet. If any code you submit is in violation of this policy, you will receive no credit for the entire assignment.

The goals of this assignment are:

• Familiarize you with machine learning basics and CNN dataflow

• Show what CPU should do in current machine learning systems

• Understand data transmission between different hardware platforms.

Problem Description

In this assignment, your job is to investigate what role should CPU act in machine learning systems. Currently, there are many hardware choices for machine learning including CPUs, GPUs, FPGAs, and ASICs. Due to the high parallelism of CNNs, GPUs and FPGAs have become promising candidates for computation tasks in network inferencing and training. Besides computation, there are still other jobs like compiling high-level CNN network descriptions to lower-level assembly codes, and controlling data transmission between on-chip and off-chip memory. Without CPUs, a purely FPGA or GPU system might not achieve satisfactory performance, so CPUs+GPUs, or CPUs+FPGAs systems are popular in many state-of-the-art research works.

What does the program do

The idea of this assignment comes from GLOW (https://github.com/pytorch/glow) and TVM (https://github.com/apache/incubator-tvm), which are compiler stacks for deep learning systems. They are designed to close the gap between the productivity-focused deep learning frameworks, and the performance- and efficiency-focused hardware backends.

In this assignment, we design a program to simulate the work flow of GLOW and TVM, and we divide it into two parts. Firstly, the program parses the network modified from darknet (https://github.com/pjreddie/darknet/tree/master/cfg), obtaining the parameters of each layer, including input size, weight kernel size and output size. Secondly, to obtain a higher degree of parallelism and to implement convolution operations in vector-operation instructions, the program converts the convolution operations into matrix multiplication.

Part 1: Implementing Parameter Parsing [25 points]

Your job is to implement the function `obtain_parameters()` of class `Network` in **`network.h`**. We have implemented the file input/output interfaces in class **`file_utils.h`**. You may use the variable `network_cfg_description`, which contains each line of the input file. You are also free to implement the file input interface by yourself, but note that you are only allowed to implement this in file **`network.h`**.

In function `obtain_parameters()`, you should figure out the member variables of class `Network` below:

// conv represents for convolution

`int layer_number;` *// represents for the total number of conv layers*

`std::vector<int> input_height;` *// input_height[i] for the height of input in ith conv layer*

`std::vector<int> input_width;` *// input_width[i] for the width of input in ith conv layer*

`std::vector<int> input_channel;` *// input_channel[i] for the number of channels of the input feature map in ith conv layer*

`std::vector<int> kernel_dimension;` *// kernel_dimension[i] for the number of filters in ith conv layer*

`std::vector<int> kernel_size;` *// kernel_size[i] for the height/width of kernel in ith conv layer, assuming the height and width is same.*

`std::vector<int> kernel_channel;` *// kernel_channel[i] for "channel" number of inputs in ith conv layer, which should be the same as input_channel[i]*

`std::vector<int> output_height;` *// output_height[i] for the height of output in ith conv layer*

`std::vector<int> output_width;` *// output_width[i] for the width of output in ith conv layer*

`std::vector<int> output_channel;` *// output_channel[i] for the number of channels of the output feature map in ith conv layer, which should be the same as kernel_dimension[i]*

Here are examples of input and output. To simplify this problem, we will focus on convolution layers in this part, since convolution operations occupy more than 90% of computation in the whole network. You can assume the network is purely linear without any branches, and the `cfg` file contains only three kinds of labels, [net], [maxpool], [convolutional].

/ input */*

`network_*.cfg`

/ output */*

`network_*.parameter`

// for running the code, you might use the command "`g++ -o main main.cpp`*" to compile the code.*

// Please make sure your code running without any compiling errors on LRC machines

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

To fully explore the parallelism of convolution operations and deploy convolution layers on other hardware platforms which support vector-processing ISAs, many research works choose to convert the convolution operations into matrix multiplication. Your job in this part is to implement the function `conv_convert` of class `Network` in **`network.h`**. You may use Figure 2 in [1] or the blog [2] for reference with the details of how to convert the initial inputs into the inputs of matrix multiplication units. The sliding window unit in [3] should also be a good reference for understanding how convolution is converted to matrix multiplication.

```
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 is a three-dimension array (input_height[layer_id], input_width[layer_id], input_channel[layer_id])

// initial_kernel is a four-dimension array (kernel_dimension[layer_id], kernel_size[layer_id], kernel_size[layer_id], kernel_channel[layer_id])

// to avoid using pointer in C++ to represent high-dimension arrays, we provided the data structure `Array1D`, `Array2D`, `Array3D`, `Array4D` *to help you implement this part*

You should figure out the contents of 2D Array `input_matrix` and `kernel_matrix`, which are the inputs for the matrix multiplication units. We may test your code with different padding and stride size, so you should not use the variables `output_height`, `output_width`, `output_channel` generated in Part 1.

We have provided shell codes for parsing input and kernels and generating output files, here are examples of input and output.

/ input */*

`network_*.layer_*.initial_input`

`network_*.layer_*.initial_kernel`

// The first line of initial input represents for height, width, channel, padding, stride of input

// The first line of initial input represents for dimension, height, width, channel of weight kernel

// These numbers are randomly generated **integers** without any specific meaning, and you may change the code of function `generate_input_kernel()` in `test.h` to generate input&kernel for debug

/ output */*

`network_*.layer_*.input_matrix`

`network_*.layer_*.kernel_matrix`

Note that there might be several solutions to the problem expanding the input in different orders, but we only accept the expanding order, channel, width, height, which should be the same order as the provided example, as the only solution.

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

This part is an extension of Part 2 to make it support more data transmission interfaces between different hardware platforms. In part 2, we provided memory map interface for the function `conv_convert`, so the program could have random access to input and output data. The work flow of part 2 is:

1. CPUs generate the input matrix and kernel matrix, storing them into the DRAM.
2. CPUs send the synchronous signal to FPGAs/GPUs, after which FPGAs/GPUs can fetch the data from DRAM.
3. FPGAs/GPUs perform the matrix multiplication, sending the generated results back to the DRAM.
4. FPGAs/GPUs send the synchronous signal to CPUs. CPUs deal with the output data and decide whether to perform the `conv_convert` again for the next convolution layer.

The whole work flow cannot be pipelined well because of data hazards, which means those accelerators cannot start computing until CPUs finish generating the whole matrix. This could be solved with stream interface, which is supported in current CPUs+FPGAs, CPUs+GPUs systems. Streams can be considered as items on a conveyor belt being processed one at a time rather than in large batches. With stream interface, the whole matrix should have finer granularity, and FPGAs/GPUs or other hardware platforms can start computing as soon as they receive the first row/column of the matrix. As a result, CPUs and other accelerators can work simultaneously to achieve higher global throughput. In this part, your job is to implement the function `conv_convert_stream` of class `Network` in **`network.h`**.

```
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 have 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 size of `buffer`, since this size is enough for you to implement this function, but feel free to add several temporary variables (the number of temporary variables should be constant, independent to the input/output size) as counters if needed.

We have provided shell codes for parsing input and kernels and generating output files, here are examples of input and output.

/ input */*

```
network_*.layer_*.initial_input
```

// We use the same input format as Part 2

/ output */*

```
network_*.layer_*.input_matrix.stream
```

// Should be the same as network_*.layer_*.input_matrix

Part 4: Report [10 points]

Write a report including the details of how you used the `buffer` in Part 3. The report should contain descriptions of which data should be stored in the `buffer` and how to update the `buffer` when it is full. Feel free to add any comments about this assignment in your report.

Part 5: Bonus part [5 points]

We encourage you to show your critical thinking about the design of the `buffer` in the following ways:

1. You can propose other advanced data structures for implementing the buffer in Part 3.

2. With the network going deeper, the channel of input feature map of each layer might expand rapidly, but the height and width of input feature map might shrink because of the pooling layers. Your proposed data structure for buffer should take this into consideration.

We can give you at most extra 5 credits based on your design. You may choose not to submit this part.

FAQ

1. Please use “g++ -o main main.cpp -std=c++14” for compiling 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 any data structures in "network.h". But make sure your code is running without any compiling error with 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 the example input instead of generating 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 testcases and the first layer is always a convolution layer.

What to submit:

You need to submit 2 or 3 files in total

1. `report.pdf`
2. `bonus_part.pdf`
3. `network.h`

Reference

[1] Chellapilla, Kumar, Sidd Puri, and Patrice Simard. "High performance convolutional neural networks for document processing." 2006.

[2] https://medium.com/@_init_/an-illustrated-explanation-of-performing-2d-convolutions-using-matrix-multiplications-1e8de8cd2544

[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.