Practices in Software Convergence

Lab 1: Optimized Neural Network Software

Due: 2025-10-29 23:59

Overview

In this lab, you will improve the performance of a naïve software implementation of neural network inference, specifically a single fully connected layer of the VGG19 image recognition model.

The software is given four binary files as input: weight matrix, bias matrix, input matrix, and the "golden" output matrix computed by Keras. The software performs inference of a single fully connected layer, by applying the input to the weights and biases. The output calculated by the software will be compared against the golden output data.

Description of Target Computation:

For each 1-dimensional input feature map "**X**" of length N, it generates a single output feature map "**Y**" of length M. The parameters involved in this calculation is a N×M matrix "**A**", and a bias vector "**B**" of size M. The computation performed is the following:

$$Y = ReLU((X \cdot A) + B)$$

ReLU is the Rectified Linear Unit activation function, which simply returns zero if the input is a negative value. In the input is not negative, it returns it without modification. Figure 1 shows a graph of the ReLU function.

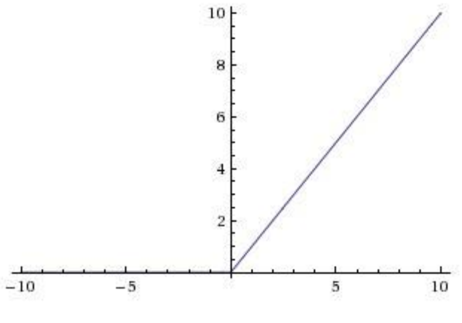


Figure 1: ReLU activation function

This computation is simply repeated for every input feature map.

Provided Material

Inside the provided package, you will find this document, as well as two .cpp files, a Makefile, two bin files, and a data folder. **You will only need to modify one file: fc_layer.cpp!** If you also want to modify the Makefile, please leave a comment in the submissions page.

The two binary files, **vgg18.w24.matrix.bin** and **vgg18.w24.bias.bin**, are the weights and biases extracted from the second fully connected layer of a pre-trained VGG19 model. This layer takes as input 1-dimensional feature maps of size 4096 and emits as output 1-dimensional feature maps of size 4096. The binary ordering of the matrix file is input-major, meaning 4096 consecutive values in the matrix file are involved in 4096 different indices of the output vector. Put differently, if we assume the matrix file is row-major encoded, the rows correspond to input vector indices, and columns correspond to output vector indices.

The binary files in the data directory are input and golden output pairs with various numbers of input and output feature maps, spanning from 32 to 4096. The smaller files are for quicker design space exploration. Grading will be done on the 4096-size pair.

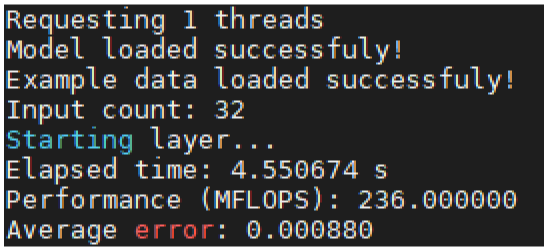
You will need to modify the "fc_layer" function in the fc_layer.cpp file. The arguments, etc should be self-explanatory from the naïve implementation provided. Memory regions for all pointers are 64-byte aligned. It also takes a "threads" argument, which is the number of threads requested from the command line. You are free to spawn that many threads, or less, or more. During grading, I will give a number smaller than the total number of hyper-threads of the benchmarking machine as the thread count. I emphasize that creating threads based on this argument is work for you to do! The provided software does not handle this automatically.

Compilation and Execution

Simply execute "make" to generate one executable, "nnfc". nnfc takes two or more arguments, the first two being paths to the input and output files, in that order. The third optional argument is thread count, which is simply relayed to the fc_layer function as an argument.

For example, you can run the program for the small, 32-size dataset with four threads, with the following command:

```
./nnfc data/inputs.32.bin data/outputs.32.bin 4
```



```
Requesting 1 threads
Model loaded successfuly!
Example data loaded successfuly!
Input count: 32
Starting layer...
Elapsed time: 4.550674 s
Performance (MFLOPS): 236.000000
Average error: 0.000880
```

Figure 2: Example output

Figure 2 shows an example output from execution. The output shows the number of inputs it processed, how much time it took, and the average performance in MFLOPS (Million Floating Point Operations Per Second). This number is very bad. We should be reaching multiple GFLOPS per core!

Average error shows the difference between the calculated output and the golden output from Keras. The numbers shown is kind of the numbers we are looking for, but due to the characteristics of floating-point numbers, there may be an order of magnitude bigger errors even with correct implementations. Use this number as a smoke test to check if your implementation is wildly wrong!

Submission and Grading

Please submit only "***fc_layer.cpp***". Don't compress it, just submit the single file please!

Full marks will be given if it can outperform a multithreaded, blocked implementation with statically-sized blocks.

We will have a derby at the end with anonymized names, so please try your best to do better than everyone!

Some Tips

- You are free to use some normally available libraries such as openmp. However, bear in mind that in past experience, openmp was not automatically as smart as we had hoped!
- You should not use accelerators like CUDA. Target a baseline x86 machine with AVX2 support.
- Use perf!
- When in doubt, understand the naïve implementation.