ECE 5755 Modern Computer Systems and Architecture, Fall 2023
Assignment 5: Open-Ended Optimizations

1. Introduction

This lab assignment asks you to put everything together to implement an end-to-end neural network. You are tasked with aggressively optimizing your implementation using the microarchitectural principles you've learned throughout the semester. You may use **any optimization techniques you wish**.

2. Materials

We will be providing a base implementation of lab 5 to ensure you have a functional starting point. You may use any of the code from any of your labs 1-4 to help you with lab 5.

Download the baseline implementation of the full forward pass of the CNN: **lab5_baseline.zip**. Extract the folder. To build the baseline implementation, use the command `make lab5` within the extracted folder. Then run the lab5 binary to see the full forward pass running and performing inference on 100 images from the MNIST dataset. Forward pass is performed on each image 100 times so that the runtime of the forward pass dominates the program runtime. Look through the following files to understand what the code is doing:

- `lab5.c`
- `kernel/nn.c`

NOTE: You must NOT modify lab5.c, lab5.h, the Makefile, or any files in mnist_data/ or model/. You can modify everything else in any way desired, even rewriting the entire kernel and forward pass if you want.

3. Objective

The objective of this assignment is to make your end-to-end convolution forward pass run as fast as possible. This means analyzing all of the functions in the kernel in detail. You should be able to identify the bottlenecks of the program and apply the techniques you've learned to implement an efficient neural network. You can use any combination of optimizations that you learned in class, any techniques that you research on your own, and any of the techniques in the following Additional Techniques section. **You must use at least one optimization technique not covered in labs 2-4. You are expected to use at least one profiling tool (toplev.py, gprof, gcov, or cachegrind).**

You are expected to present the data from the profiling tool and demonstrate that you used the data to help optimize your program.

4. Additional Techniques

Not all of the following techniques will necessarily improve your program's performance. The first four techniques listed (cachegrind, gprof, gcov, SIMD) will likely be useful so try them first; the other techniques may or may not improve performance (depending on applicability and implementation) and may or may not be worth the effort, but they are listed here as options for you to explore.

- Profile cache misses using **cachegrind**.
    - Reference: https://valgrind.org/docs/manual/cg-manual.html
- Profile your code using **gprof** to see the execution time breakdown
    - Reference: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html
- Profile your code using **gcov** to see which functions are getting called the most often
    - Reference: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
- Vectorize your code with **SIMD instructions** such as SSE, AVX, AVX2, AVX512 intrinsics
    - Reference: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#
- **Pipeline the forward pass**. Not all elements need to finish one stage of the forward pass before the next stage starts processing data. E.g. convolution output elements that are ready early can be processed by the linear layers before all convolution output elements are ready.
- **Embed assembly code**. Compilers have to be conservative about the assumptions that are made regarding what optimizations are allowed in order to guarantee correctness. You as the programmer know more about what is and what is not allowed and can optimize more aggressively.
    - Reference: https://gcc.gnu.org/onlinedocs/gcc/extensions-to-the-c-language-family/how-to-use-inline-assembly-language-in-c-code.html
    - Instruction set reference: https://cdrdv2.intel.com/v1/dl/getContent/671110
- **Inline functions** that are called often. Note that this can increase your code size if the inlined function is called from multiple places.

- **Align data structures** to cache line boundaries.
- Try **different compiler optimizations**. Here are the GCC ones: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Keep in mind -O3 is not necessarily going to give you better performance compared to -O2; test it out as it varies case by case. You also don't have to use GCC as your compiler if you feel like another compiler can provide more performance. The most popular C compiler besides GCC is Clang.
- Speed up convolution for larger inputs using **FFT methods**
    - Reference: https://arxiv.org/abs/1312.5851
- Improve runtime complexity of matmul using **Strassen's Method**
    - Reference: https://www.geeksforgeeks.org/strassens-matrix-multiplication/

5. Deliverables

Please submit your assignment on Canvas by **Wednesday, Dec. 6 at 9 pm**. You are expected to submit your code, report, and scripts in a single zip file named **lab5.zip**.

The lab will be marked out of a **total of 100 marks.**

- **60 marks** for performance and accuracy according to the following equation: $marks = 60 \times speedup/1.5$ and $speedup = t_{baseline}/t_{optimized}$

  You need 1.5x speed up to get the full 60 marks.

  If you pick an implementation that impacts accuracy, it must maintain some reasonable amount of accuracy based on your implementation (you must demonstrate what is 'reasonable' in your code and report).

- **40 marks** for report (named **lab5_report.pdf**)
    - 20 marks for profiling and runtime data (present as graph or chart or table)
        - 10 marks for profiling and runtime data on baseline
        - 10 marks for profiling and runtime data on optimized version
    - 10 marks for explaining how you used profiling data to help optimize your program
    - 10 marks for discussion of optimization techniques (with at least 1 technique not covered in labs 2-4)