Assignment 3: Matrix Multiplication - Cache Optimization

CS2.201a Computer Systems Organization (CSO)

April 18, 2023

1. There is only one problem and it's fairly simple, given three $n \times n$ matrices A, B and C, you have to fill in the code for a function which does the following computation:

   $$C \leftarrow A \times B + C$$

   To submit and be graded, you only need to write the three for loops required to do this computation. However, submissions will be graded by the peak score achieved by your program. The score function is just a measure of the performance of your program. In other words, you have to try to optimize naive matrix multiplication and make it more efficient.

This purpose of this assignment is to put into practice everything that you have learnt over the duration of this course, with a focus on cache and the effect it has on program run-time. You will be provided with all the supplementary code necessary to generate tests, verify solutions and benchmark your program over a wide range of tests. This document attempts to outline all the functionalities available at your disposal and what you are expected to do, however, it is highly recommended you attend the tutorial conducted for this assignment to get a firm understanding of what to do.

List of the files provided: *(gemm = generalized matrix multiply)*

- **gen.cpp**: This file is responsible for creating all the test data that your program will be run on. In practice, you don't really need to touch this file. If you really want to, in the *__init__()* function in this file, you will find a dictionary being initialized. This represents the sizes of the matrices being created. You can modify these numbers to generate different tests, although the provided configuration should be more than sufficient for most needs.
- **helper.cpp**: There is no reason to touch this file. It contains benchmarking / setup / pretty printing code for the curious to see. You are advised to not modify anything in this file.
- **setup.sh**: The test generator requires a specific directory structure to work. Running *source setup.sh* should automatically setup this structure, compile *gen.cpp* and generate the relevant tests. This should be the first command that you run.
- **gemm.cpp**: This is the primary file you will be working in. This file contains exactly one function declaration. You must complete it by providing the definition for this function. The first argument is the size of the matrix n. The rest are the matrices A, B and C. *(Tip: You can access element $a[i][j]$ by accessing $a[i \times n + j]$).*
- **sgemm_bench.cpp**: This is the program that runs the benchmark. This is the file that you should compile and run to measure performance. The compiler flags you are recommended to use are *-Ofast -march=native*. The whole command should look something like this:
  g++ sgemm_bench.cpp -o sgemm_bench -Ofast -march=native && ./sgemm_bench.

Test framework

Your code will only be run on **square matrices**. The entire assignment can be done locally on your machine. But for the purpose of uniform grading, you will all be given accounts on Abacus temporarily for the remaining course duration. This is the machine that your code will be run on for final grading. We will cover using Abacus in the tutorial tomorrow.

Submission format

Just rename the file *gemm.cpp* given to you to *rollnumber.cpp* and submit this one file. The function that will be evaluated is *my_gemm*.

Brief outline of what to do

(a) Become familiar with running code on your Abacus machine. You can run benchmarks for specific tests using the benchmark program.

Usage ./<program_name> --min [number][/KB/MB/GB] --max [number][/KB/MB/GB]

(b) The primary bottleneck for performance in this task is memory bandwidth. Matrices are big and the order in which we usually compute matrix product is very bad for caching. Consider exploring different ways to modify the order in which we do this computation. (Partial computation, blocking, etc.).

(c) Run *lscpu* to know how much cache is available. Consider running *valgrind --tool=cachegrind* or *perf* on some test programs to see how your cache hit and miss rates are.

(d) Watch this video.

(e) Experiment! :) You could modify *sgemm_bench* for example to run 2-3 different functions to compare performance, etc.

What not to do

- Do not attempt to write multi-thread programs. The test environment will only provide one thread to your program.
- Do not waste time on micro optimizations, they're usually nothing more than a waste of time.
- Submitting fast but *wrong* code. The benchmark environment clearly lets you know if the output was *verified* or not. If it wasn't verified, you failed some tests and your output was wrong. Incorrect code will be heavily penalized. Optimize only after ensuring the written code works. Perform regular tests to ensure an optimization didn't break the code.

Have fun!