ECE 5755 Modern Computer Systems and Architecture, Fall 2023
Assignment 4: Implementing Multi-Threading

1. Introduction

This lab assignment will build off the work you completed in the previous lab assignment. In particular, you will be exploring the efficiency gains in your kernel when you implement **multi-threaded operations**.

2. Materials

Use all of the same materials from Lab 1. We are still using the course virtual machine for this (and all future) labs. Make sure to make a copy of your original lab 1 code or use a version control tool like git so that you have a known good state to revert to. You will add an additional function `matmul_multithread` which implements a version of `matmul` that uses multithreading to exploit data parallelism.

3. Multithreading

Multithreading splits your program up into separate concurrent program threads that each runs its own stream of instructions. The benefits (and causes for potential problems) of multithreading is that your threads are both concurrent and parallel (if they are running on multiple cores). Concurrency allows you to mask long I/O and memory access latencies by interleaving threads so as to improve CPU utilization. Parallelism allows you to use multiple CPU cores versus just a single core.

In the case of matrix multiplication, you can split up your threads by dividing the output matrix into sections so that each thread is responsible for one section. Partitioning work based on the output matrix allows threads to share input data but not contend over writing to output data. This allows you to avoid synchronization which can be complex and difficult if you have not done parallel programming before.

Be careful to not split output data within cache lines to separate threads. As you learned in the memory coherence lectures, when two cores simultaneously and repeatedly write the same cache line, the cache line gets bounced back and forth between the cores. For elements of the output array that sit within the same cache line, this ping-ponging of a cache line between cores happens even if the cores are not writing the same element in the output matrix. This is called false sharing.

You can implement multithreading with the POSIX thread or pthread library. Specifically, look up how to call the `pthread_create` and `pthread_join` functions.

Also, you will need to add the `-lpthread` flag when compiling. Note that the Slurm job allocates you only 8 cores.

4. Objective

The objective of this lab is to implement a better matrix-multiplication algorithm that uses multithreading, and **to analyze the performance gains of your algorithm using microarchitectural analysis techniques**. You are tasked with implementing the following function:

```
float **matmul_multithread(float **A, float **B, int A_rows, int
A_cols, int B_rows, int B_cols)
```

The function should be put in `kernel/matrix_ops.c`. Don't forget to put it in the header file as well so the lab will compile. You'll then want to swap out your original matrix multiplication implementation for this one in your tests to verify that it's actually working.

When testing for runtime, you can use the Linux `time` command; make sure to take the `real` time which is the wall-clock time. Moreover, as with profiling, make sure you are looping over the function you want to profile so that the runtime of the entire program is dominated by the function you are profiling/timing.

For Lab 4, you are only required to submit `matrix_ops.c` and `matrix_ops.h` with `matmul_multithread` and any additional helper functions you choose to use inside those two files.

For this lab, we will not be grading your testing procedure, only the correctness of `matmul_multithread`. Functional correctness grading will be done with an internal grading program similar to the testing programs you were expected to submit for Lab 1. We will also be reading your code to grade for correctness in implementing tiling/blocking, so please make sure your code is readable (use comments to clarify your code).

Your profiling process must meet the following requirements:

- Use `toplev.py` from `pmu-tools` for Top-down analysis data
- Profile only the core(s) the program is running on

- Runtime of the program-under-test should be dominated by the function-under-test (you can accomplish this by looping over your function many times)

Since your code is now running on multiple cores, you will need to profile over the cores that are running. To only use specific cores, you can use the `-a` and `-c` options of taskset; see more details in the man page: https://man7.org/linux/man-pages/man1/taskset.1.html.

To have toplev.py profile the correct cores, you need to set the correct mask with the `--core` option. See the tutorial for an example: https://github.com/andikleen/pmu-tools/wiki/toplev-manual#selecting-what-code-to-count.

The argument options list for toplev.py also may be useful: https://github.com/andikleen/pmu-tools/wiki/toplev-manual#argument-reference.

The specific commands you use for profiling and testing and how you write your program-under-test is up to you as long as your process meets the above requirements. One option is to use the existing testing infrastructure under `tests/`.

5. Report

The report will be **3-pages max** but can be shorter if you do not need all three pages. Report requirements/prompts:

- Include a horizontal segmented bar chart for the Top-down analysis of some of the different test inputs of `matmul_multithread` you tried. In the same bar chart, for the same set of inputs, also include the Top-down analysis data for `matmul` from Lab 1 to serve as a baseline.
- For the same inputs, collect their performance data (runtime) on both `matmul_multithread` and `matmul` from Lab 1 and include it in a table in the report.
- Explain in detail your profiling process.
- With reference to computer architecture concepts and the Top-down analysis data, explain the performance differences between the different inputs and between naive `matmul` and `matmul_multithread`.
- Discuss any issues you had during the lab and your debugging process.

6. Deliverables

Please submit your assignment on Canvas by Nov 17. You are expected to submit three files total: **matrix_ops.c, matrix_ops.h,** and your report in a file named **lab4.pdf.**

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

- **40 marks** for `matmul_multithread()`
    - 20 marks for functional correctness (computes correct outputs for given set of inputs)
    - 20 marks for correct implementation of multithreaded matmul
- **60 marks** for report
    - 20 marks for Top-down analysis data
    - 10 marks for performance (runtime) data
    - 30 marks for discussion (see prompts in 4. Report)