Lab 4

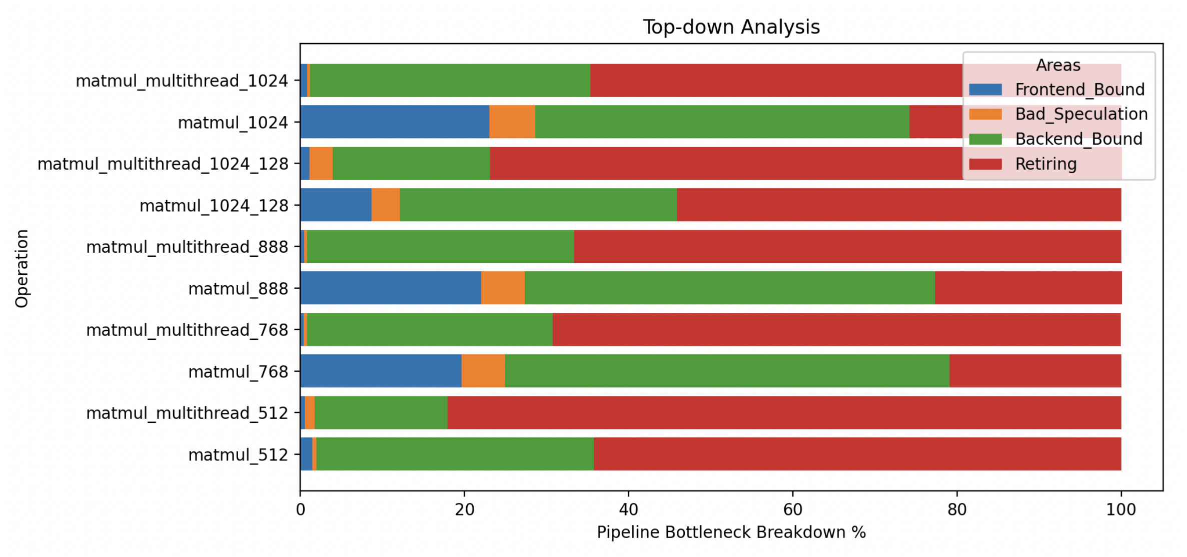
Aneesh Pandoh 11/22/2024

Setup:

Ran Top Down Analysis on each test 5 times. For each operation, I ran the tests 10 times within the function. The used taskset to run the process on cores 0-8 and used top down accordingly. I tested on various matrix square matrix sizes (512, 786, 888, 1024) as well as matrices with 1024x128. Both were tested using the multithreaded and single threaded configurations.

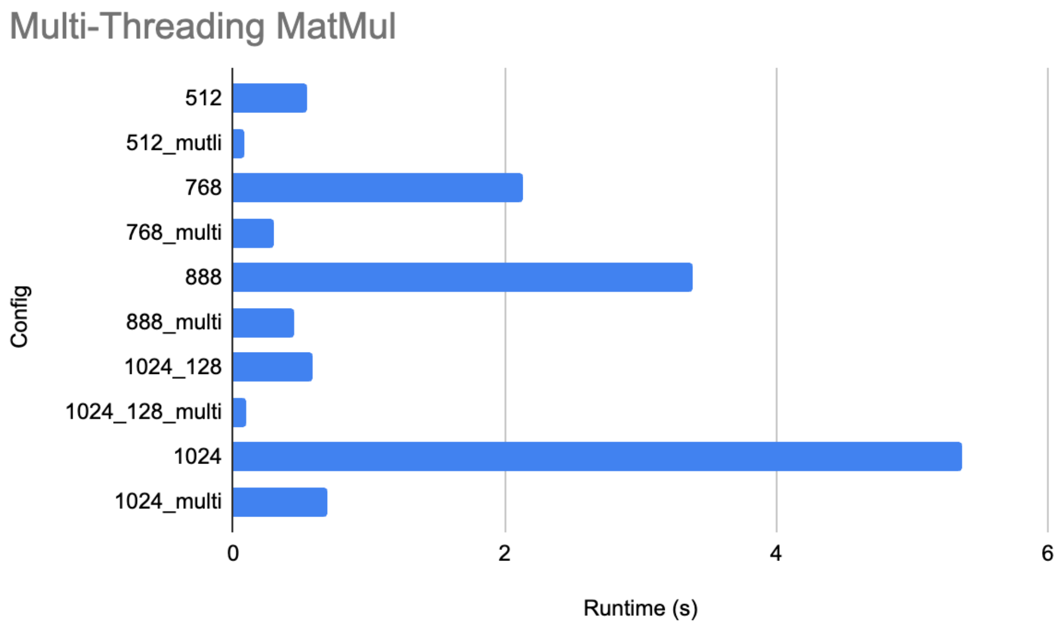
*Used github copilot / chatGPT to speed up the process writing tests and implementation

Top-Down Analysis:



We can see that the operations are no longer backend bound, this is probably because the extra cores are still executing masking the I/O and memory access latency. In the case of our program, we also avoid any synchronization related backend bound issues as we have split up matrices into sections by cores, allowing them to run independently without having to wait to write outputs. The way the tests are set up, the matrices are big enough that the data doesn't have to be split within cache lines, eliminating potential cache coherence issues. Having a big enough output matrix reduced the risk of false sharing. We can see that multithreading has very limited impact on the frontend bound percentage of the program as all cores still have to contend for the same frontend instructions.

Runtime:



We can see a significant difference in performance, as there are simply just more cores to do the work required. We can see an average speedup of 7.2x, which is in line with what is expected adding having 8 cores as compared to 1. We don't see a full 8x speed up because there is always some overhead of the program that can only be executed serially (Amdahl's law which we learned earlier in class). We were able to minimize this effectively by making sure that every processor was split up evenly in the section of the matrix. We can see the impact of the parallel processing more pronounced when in larger matrices, as the overhead compared the matmul computation is less in these cases, so the speed up is a more significant portion of the time.

Interesting Observations:

Curiously we can see that smaller matrices (512x512 and 1024x128) were less backend bound to begin with, which is why they also had the least significant speed up at 7.07 for 512 and 6.25 for 1024x128. I think this significant degradation in speed up might be due to the imbalance of rows as we have to split up a 128x1024 matrix into 8 sections of rows in the program, which can cause an imbalance in workload as compared to the square matrices.

I tried to also create an imbalance by trying matrix 888x888, which doesn't fit perfectly into cachelines as it is not a power of 2, but I believe that this unusual size affected both the single processor and multiprocessor the same as they both have to deal with the the same cache line issue.

Issues:

- I linked -lpthread at the end which cause inconsistent results on whether multiple threads were used.
- Took a while to realize I had the slurm job to run on 1 cpu