Matrix Multiplication Optimization

Eric Paulz

I. *Abstract*

In this project I have researched and experimented with various loop optimization techniques. Specifically, I have applied these techniques to a matrix multiplication algorithm and compared the efficiency with that of a naïve implementation of the algorithm. Matrix multiplication is something that we know can be a major issue for computers because often times it is responsible for using large amounts of resources.

From my experiments, I have discovered the importance of minimizing the amount of cache misses in a program. The loop optimization technique that implements this idea provided significantly greater speedup that those with other motivations. More specifically, loop interchanging produced an average speedup of 1.252x. Adding loop unrolling increased the average slightly to 1.286x. Finally, the addition of loop blocking/tiling produced a total speedup of 3.204x. From these results we can see the cache efficiency is crucial for optimal performance of modern computers.

II. *Related Work*

I implemented the loop interchange and loop unrolling on my own with the base knowledge that I had about those two techniques. However, for loop tiling I referred heavily to David Bindel's lecture slides about code tuning [1]. These slides helped me to understand loop tiling and how it actually improves performance.

III. *Methodology*

The very first thing that I did was change a few of the variable names from the original program to ones that made a bit more sense to me. After that, I noticed that the loops were structured strangely. They weren't accessing the matrices in a sequential order, so I decided to use the loop interchange technique to rearrange them and improve spatial locality. This provided a small increase in performance with an average of 1.252x**.** Next, I decided to try loop unrolling, a technique that we have talked about quite a bit in class this semester. I unrolled the innermost loop by a factor of two. This provided only minimal speedup and in some cases was actually slower than the interchanged version by itself. Average speedup of T1+T2 was 1.286x. Lastly, I researched the loop blocking/tiling technique that is emphasized in the project writeup. Minimizing the amount of cache misses by splitting up each matrix into blocks, the average speedup of T1+T2+T3 was 3.204x. This is by far the most effective transformation that I used.

This project has shown me how important cache efficiency is. Unfortunately, I was not able to find any other optimization techniques that I could apply to the algorithm.

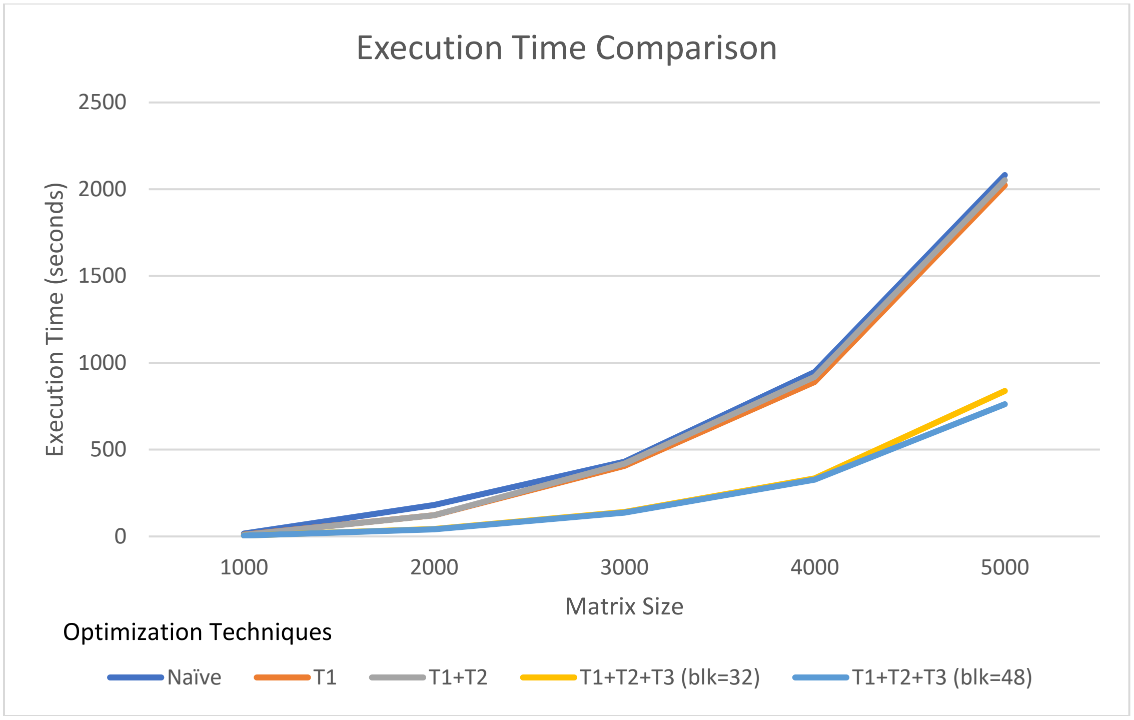
IV. Results

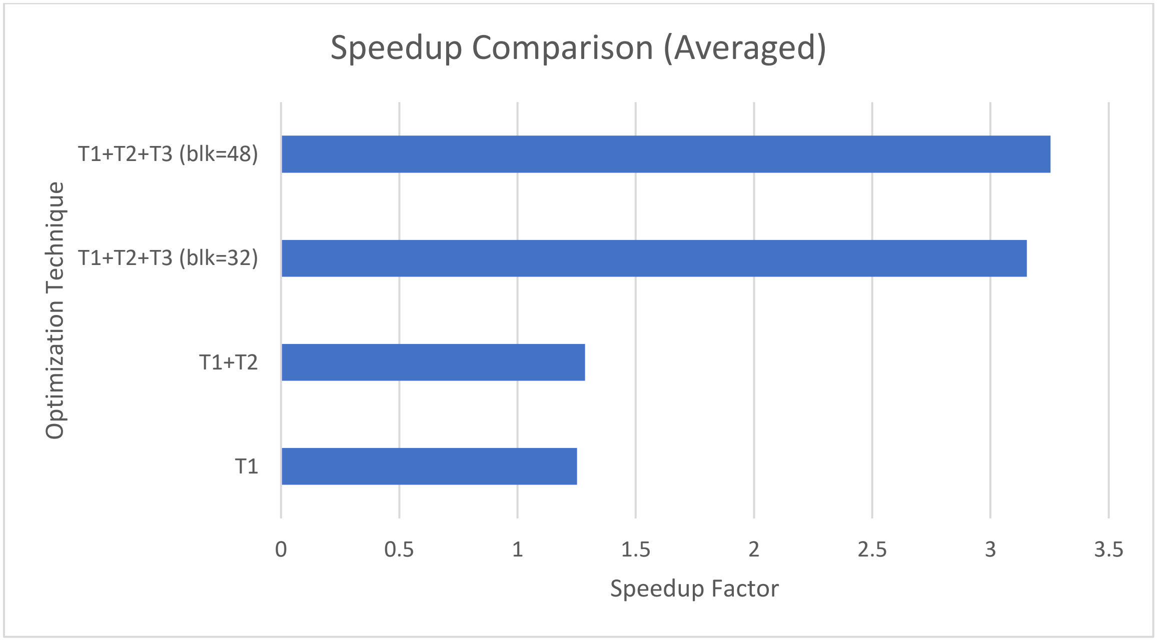
The machine that I used was the SoC's 'koala5'. This machine has an Intel® Core™ i7-4790 processor @ 3.60GHz. This chip has 8 cores and utilizes multithreading. The machine also has 16 KB of total memory.

For timing the program I used the gettimeofday() function. I called the embedded timing function before and after the actual computational part of the program, and then subtracted the beginning time from the end time to get the elapsed time. Some conversion was necessary to make the time display in seconds. The reason for using embedded timing is that we are only interested in the time it takes for the matrix multiplication algorithm to run. If we simply used the command line 'time' function, the total execution time of the program would be returned, which includes a lot of extra stuff that we aren't particularly interested in.

T1 = loop interchanging
T2 = loop unrolling
T3 = loop blocking/tiling

Technique / Matrix Size	**1000**	**2000**	**3000**	**4000**	**5000**
Naïve + (-O0) (Time$_{naive}$)	16.797s	181.480s	430.668s	947.811s	2081.870s
Transformed code with T1 + (-O0) (Time$_{opt}$)	10.300s	122.984s	406.757s	890.019s	2023.179s
Time$_{naive}$/Time$_{opt}$	1.63x	1.48x	1.06x	1.06x	1.03x
Transformed code with T1 & T2 + (-O0) (Time$_{opt}$)	9.001s	122.583s	420.108s	919.311s	2052.234s
Time$_{naive}$/Time$_{opt}$	1.87x	1.48x	1.03x	1.03x	1.02x
Transformed code with T1 & T2 & T3 (blk=32) + (-O0) (Time$_{opt}$)	5.316s	42.596s	141.715s	335.402s	838.146s
Time$_{naive}$/Time$_{opt}$	3.16x	4.26x	3.04x	2.83x	2.48x
Transformed code with T1 & T2 & T3 (blk=48) + (-O0) (Time$_{opt}$)	5.113s	40.835s	136.354s	326.299s	761.931s
Time$_{naive}$/Time$_{opt}$	3.29x	4.44x	3.16s	2.90x	2.48x


Execution Time Comparison
2500
2000
1500
1000
500
0
Execution Time (seconds)
1000
2000
3000
4000
5000
Matrix Size
Optimization Techniques
Naïve
T1
T1+T2
T1+T2+T3 (blk=32)
T1+T2+T3 (blk=48)


Speedup Comparison (Averaged)
T1+T2+T3 (blk=48)
T1+T2+T3 (blk=32)
T1+T2
T1
Optimization Technique
0
0.5
1
1.5
2
2.5
3
3.5
Speedup Factor

V. References

[1] D. Bindel, "Lecture 2: Tiling matrix-matrix multiply, code tuning", *Cornell University*, February 2010. [Online]. Available: https://www.cs.cornell.edu/~bindel/class/cs5220-s10/slides/lec03.pdf. [Accessed: April 27, 2018].