Project 2 Report
Charles Clarke
Advanced Computer Systems
10/7/2024

Introduction:

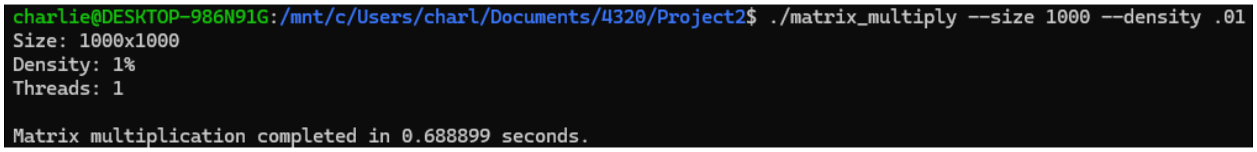
Matrix multiplication is a sequence of many many basic operations that are essential in various fields such as graphics, simulations, and maybe most importantly, training neural networks. The efficiency of these systems depend heavily on how quickly matrix multiplication can be done. It has a very high time complexity so as the size of the matrix increases, the time taken to complete the full operation increases drastically. Factors such as multithreading, SIMD instructions, and optimizing cache usage will greatly enhance the speed at which these matrices can be multiplied together by taking full advantage of hardware parallelism and memory efficiency.

Explanation of Code:

The C++ program matrix_multiply that can be seen in my github repo is being used to specify all sorts of parameters when testing performance. The program takes a few inputs in the command line, such as size of the matrices, density of the matrices, amount of threads to be used. Finally I can specify whether or not to use SIMD or cache miss optimization. This made testing the performance very easy because I could simply change the inputs in the command line and then see how long it took to execute the operations. Once I reached the third point in the lab, I decided to include a function which would allow me to use all three optimization techniques at once, speeding up computation drastically as you will see in the results to come.

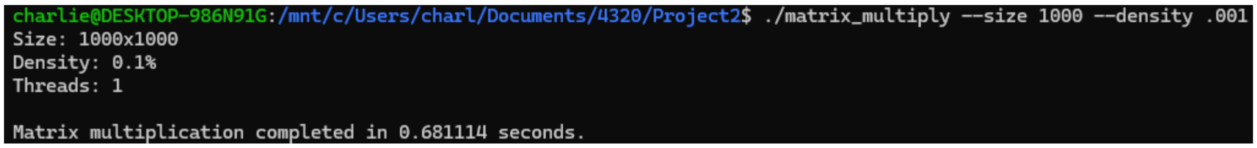
Matrix Size and Density:

First I recorded the time my computer took to run a 1000x1000 matrix at 1% and 0.1%. The outputs can be seen below.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 1000 --density .01
Size: 1000x1000
Density: 1%
Threads: 1

Matrix multiplication completed in 0.688899 seconds.
```

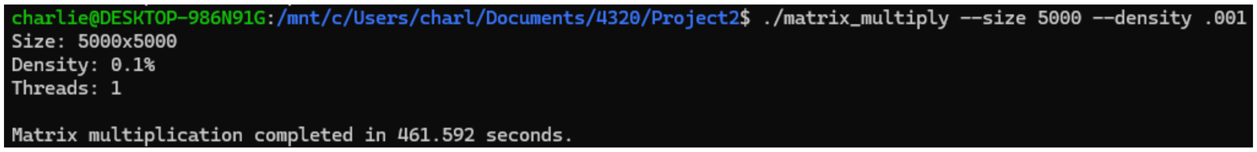


```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 1000 --density .001
Size: 1000x1000
Density: 0.1%
Threads: 1

Matrix multiplication completed in 0.681114 seconds.
```

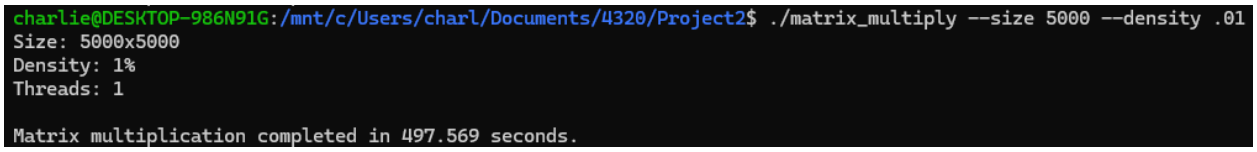
As you can see by the times of each run, the density doesn't have much of an effect whatsoever at this size of matrix, and I believe this is because most modern L3 caches can fit matrices of this size. Hopefully once I increase the size of the matrices to 5000 and then 10000, I see some discrepancy in performance between matrix densities.

The program started taking a while at 5000x5000 matrix multiplication, because I wasn't using any optimizations at this point. The results can be seen below.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 5000 --density .001
Size: 5000x5000
Density: 0.1%
Threads: 1

Matrix multiplication completed in 461.592 seconds.
```

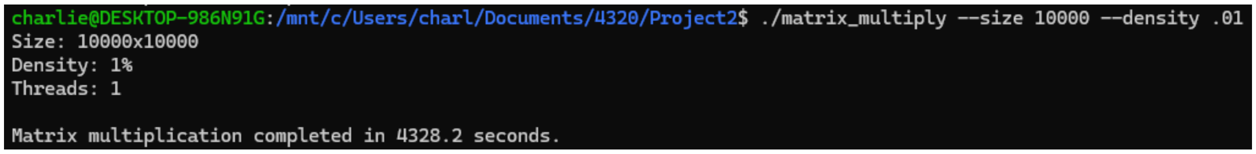


```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 5000 --density .01
Size: 5000x5000
Density: 1%
Threads: 1

Matrix multiplication completed in 497.569 seconds.
```

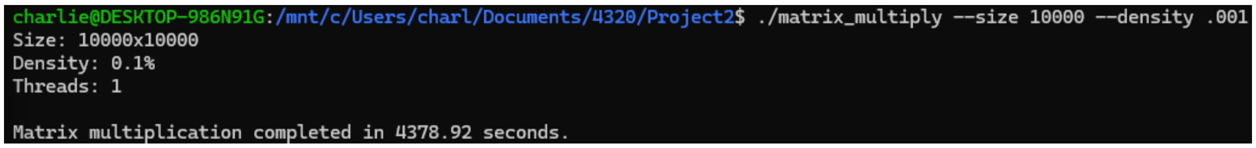
The time it took to run the program went from about 700 milliseconds, to about 8 minutes. So it's a very drastic increase as we were expecting thanks to the N^3 time complexity. Also at this size matrix, the density is starting to have a meaningful effect on the time it takes to run the program. This makes sense, because there are many less non-zero multiplications to enact throughout the program's runtime.

At this point it's pretty clear that 10000x10000 is going to take a whole while with no optimizations, but I'm invested, so I ran the program and went to play pickleball for two hours, when I got back I had the first result. Then I ran it again after changing the density and did some chores. When I came back later and it had finished, the results were not at all what I was expecting. As you can see below the time taken to complete the 10000x10000 multiplication was over an hour, however the density only changed the result by 50 seconds, which was much less than I thought it would be. I guess it makes sense though, as most of the time taken is used by multiplying elements by zero, as the code I'm using isn't optimized for matrix compression.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 10000 --density .01
Size: 10000x10000
Density: 1%
Threads: 1

Matrix multiplication completed in 4328.2 seconds.
```



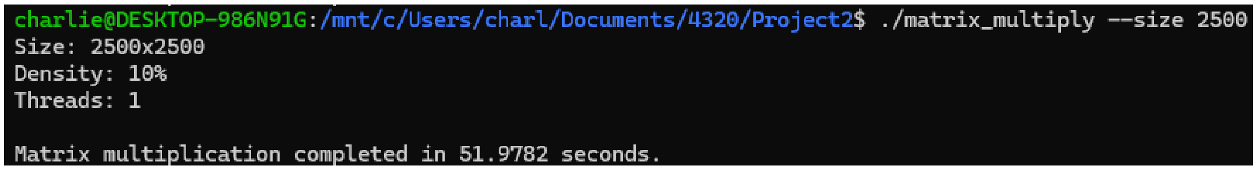
```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 10000 --density .001
Size: 10000x10000
Density: 0.1%
Threads: 1

Matrix multiplication completed in 4378.92 seconds.
```

Overall it seemed like with this brute force method of sparse matrix multiplication, density didn't seem to matter nearly as much as sheer size of the matrices being multiplied. This makes sense as computation is definitely the bottleneck in a program like this. Thankfully implementing optimization techniques will shorten these times drastically.

Optimization Techniques:

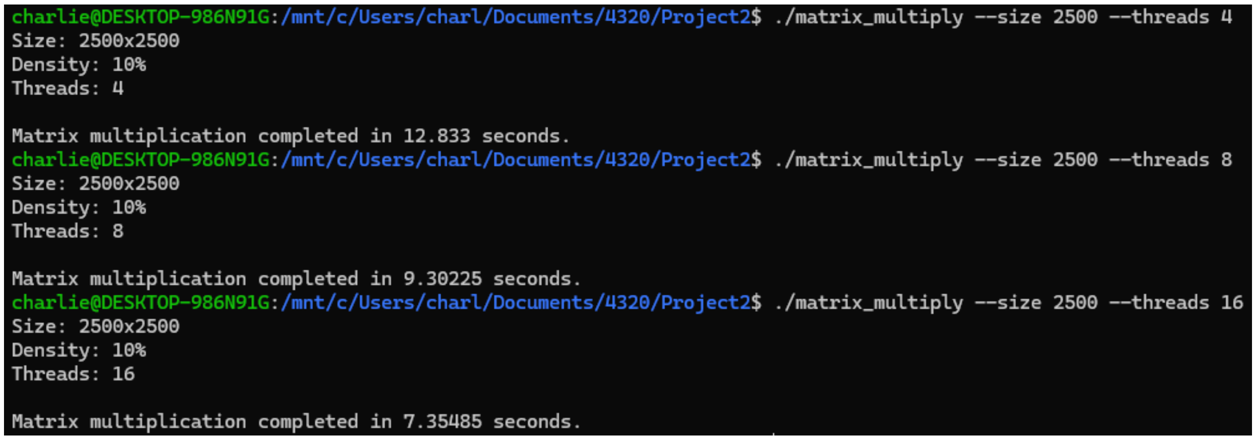
The baseline for this test will be a 2500x2500 matrix multiplication with 10% density, because I believe this is a quick enough but not too quick runtime to get good data from. The results of the first, and slowest test can be seen below, with no optimization techniques used. Coming in at around 50 seconds is what I expected to see.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 2500
Size: 2500x2500
Density: 10%
Threads: 1

Matrix multiplication completed in 51.9782 seconds.
```

For the next test I tried multithreading, which allows the pc to execute multiple instructions at once. My pc has access to 16 threads, so I'll be testing the program at 4, 8 and 16 thread usage to see the differences. With just 4 threads used, the time taken for this task goes from around 50 seconds, to about 13 seconds, very spot on with my expectation of a 4x quicker runtime. 8 threads also speeds up the process, however not by a factor of two like expected, it cut the runtime to just 9.3 seconds. Finally with 16 threads used, the runtime is cut further to 7.35 seconds, saving a whole lot of time compared to using just one thread. These results can be seen below. The return on investment certainly diminishes each time you increase thread count, however it continues to speed up the process.



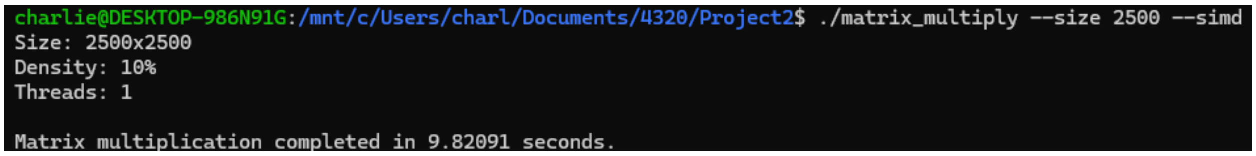
```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 2500 --threads 4
Size: 2500x2500
Density: 10%
Threads: 4

Matrix multiplication completed in 12.833 seconds.
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 2500 --threads 8
Size: 2500x2500
Density: 10%
Threads: 8

Matrix multiplication completed in 9.30225 seconds.
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 2500 --threads 16
Size: 2500x2500
Density: 10%
Threads: 16

Matrix multiplication completed in 7.35485 seconds.
```

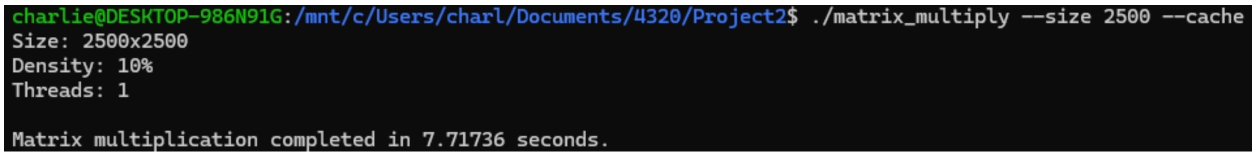
Next I'll be implementing SIMD or single instruction, multiple data. This is a type of parallel computing architecture that allows for a single operation to be used on multiple data points simultaneously, as the name suggests. It works by utilizing vector registers that can store multiple pieces of data, then a single instruction can be applied to the entire vector, processing all of the values in one cycle. This can greatly increase throughput. Once again I compared this to the baseline time of a 2500x2500 multiplication at 10% density which took 50 seconds. The results of this test really impressed me. With just one thread being used, but with SIMD, the time was cut from 50 seconds to just under 10. About equivalent to 8 threads working in parallel. The output can be seen below.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 2500 --simd
Size: 2500x2500
Density: 10%
Threads: 1

Matrix multiplication completed in 9.82091 seconds.
```

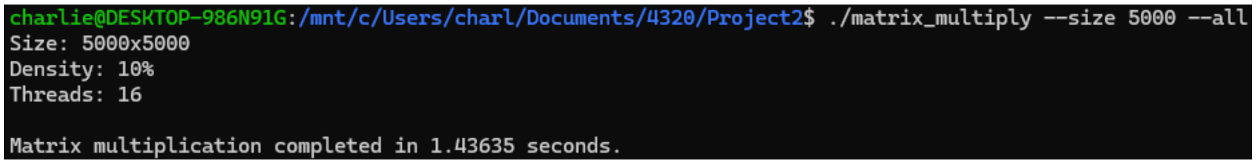
Next I ran the same test with cache access optimization enabled. This allows for some clever use of cache memory to process data at a much quicker rate, in this case for matrix multiplication. For example accessing elements row by row instead of column by column can speed up times, because rows are stored contiguously in most memory systems. Also it can allow the code to break the matrix into smaller blocks that fit within the cache individually instead of doing entire columns by entire rows. This allows for much higher throughput as well, as main memory doesn't need to be accessed nearly as much. As you can see below this cache optimization technique greatly reduced the time taken to complete the operation. Taking the 50 second time down to just under 8 seconds. This was about equivalent to using 16 threads to do the computations.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 2500 --cache
Size: 2500x2500
Density: 10%
Threads: 1

Matrix multiplication completed in 7.71736 seconds.
```

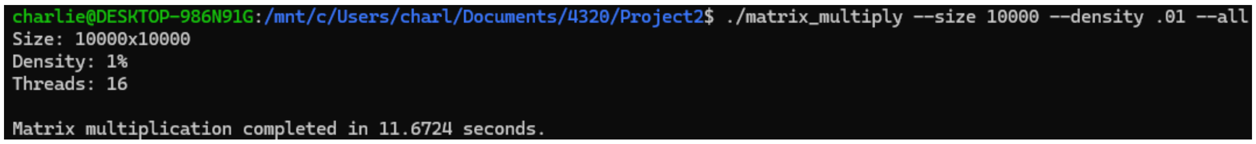
This final part was incredibly interesting to me, because it really represented pushing the limit and using all the resources and techniques available to me, in order to make this program as fast as possible. I ran this test with 5000x5000 matrix multiplication in order to have some bigger numbers to look at. The base run of this with no techniques was once again right around 8 minutes. However the result of running multithreading, SIMD, and cache miss optimization, took the speed down to 1.4 seconds! This was incredibly quicker than I was expecting and made my jaw drop in real life. The output can be seen below.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 5000 --all
Size: 5000x5000
Density: 10%
Threads: 16

Matrix multiplication completed in 1.43635 seconds.
```

This was so quick I just had to see the reduction from 10000x10000 at 1% density, so I gave it a go. The results were absolutely absurd. Once taking a whopping hour and 10 minutes, now took only 11.7 seconds to complete. The difference was monumental! This output can be seen below.

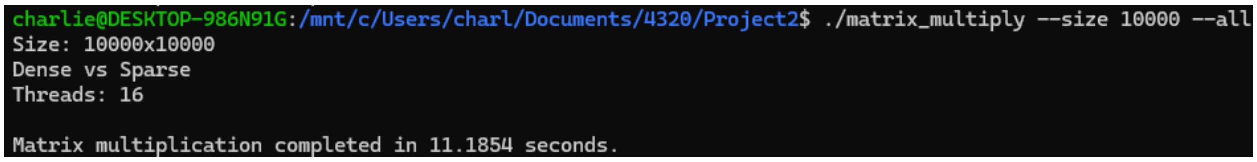


```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 10000 --density .01 --all
Size: 10000x10000
Density: 1%
Threads: 16

Matrix multiplication completed in 11.6724 seconds.
```

Types of Matrix Multiplication:

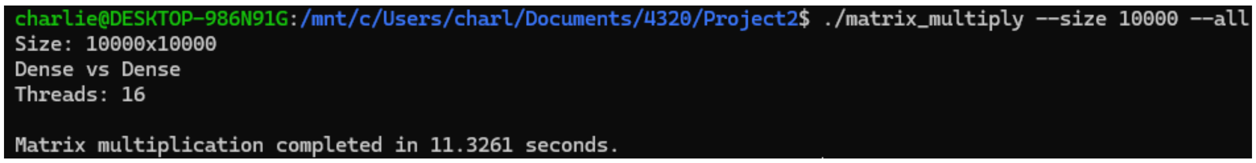
There are three broad cases when multiplying two matrices together in this case. Dense-dense multiplication, sparse-dense multiplication, and sparse-sparse multiplication. I'll be testing these scenarios using my cache optimization technique to find the difference in runtime between them. Off the bat I'm not expecting much difference at all actually because of how my method of reducing cache miss rate works. In class we talked about two tactics to utilize the cache more effectively to multiply matrices, first by breaking the matrix into smaller matrices that each fit on the cache. This reduces the miss rate greatly, but doesn't save the pc any operations. The second technique is matrix compression, where all of the non-zero elements are placed into a vector along with their location, and every element that's a zero, is ignored, because zero times anything is always zero. This saves the pc a lot of operations, however takes more logic to implement and requires you to look through both matrices entirely anyways to find the non-zero elements. I chose the first option, of keeping all the data I could on the cache, by breaking the big matrix up into many smaller blocks. The outputs of these runs can be seen below.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 10000 --all
Size: 10000x10000
Dense vs Sparse
Threads: 16

Matrix multiplication completed in 11.1854 seconds.
```

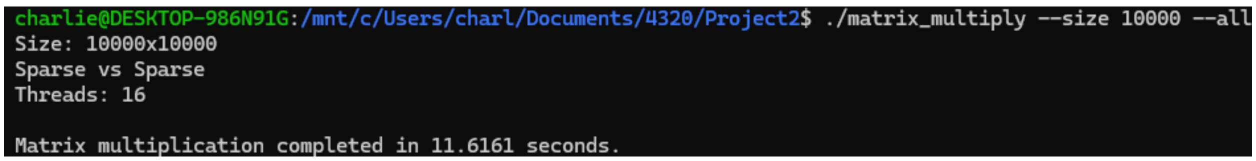
Above is the result of dense vs sparse matrix multiplication on 10,000 element matrices. The sparse matrix was set to a density of 1%, and the dense matrix was set to a density of 100%. This is with using all optimization techniques, obviously including cache miss optimization. This result comes in at a little over 11 seconds. Next I attempted dense vs dense matrix multiplication, expecting a slightly slower time.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 10000 --all
Size: 10000x10000
Dense vs Dense
Threads: 16

Matrix multiplication completed in 11.3261 seconds.
```

As you can see, this operation completed in a slightly slower time, I ran the test a couple times to make sure, and it seems like this is marginally slower than dense vs sparse. Finally I tried sparse vs sparse multiplication.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project2$ ./matrix_multiply --size 10000 --all
Size: 10000x10000
Sparse vs Sparse
Threads: 16

Matrix multiplication completed in 11.6161 seconds.
```

After running a couple sparse vs sparse multiplications I continued to get a value of about 11.5 seconds, which shows that the density of the matrices doesn't matter to this algorithm. This is because unlike compression, splitting the matrix into smaller matrices doesn’t save the pc any operations, it only utilizes the cache more effectively.

Analysis and Conclusion:

This was a very interesting project, I definitely enjoyed this more than the first one, because of the rewarding feeling after cutting down that multiplication time so much. Without optimization, multiplying matrices takes incredibly long as you increase the size of the matrices. This is because the time complexity of this operation is n^3. In order to speed up this process, many optimizations can be made, for example using multiple threads in order to process multiple operations in parallel. You can also utilize SIMD, which allows your computer to alter multiple values at a time by placing them in a vector beforehand. Finally cache miss rate optimizations such as compression, or breaking the matrix into smaller blocks are incredibly helpful. Depending on which cache rate miss optimization is used, the density of the matrix may have a large effect on the throughput of the operation. Sparse matrices can be dealt with much easier by using matrix compression techniques. It makes sense in a lot of applications to start compressing matrices when their density falls below 30%. At this point it is better to use the extra overhead to save the pc some computations. As the results from my experiments show, it’s incredibly beneficial to use these optimizations when multiplying matrices. The 10000 by 10000 matrix multiplication took my relatively quick computer an hour and 10 minutes to compute without any optimizations or tricks. However once all three of these techniques I talked about earlier were in place, this time dropped to a shockingly quick 11 seconds. This result honestly blew me away and made the whole project worth the effort I put in.