Damian Franco

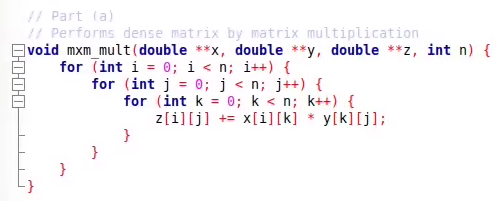
101789677

dfranco24@unm.edu

CS-542

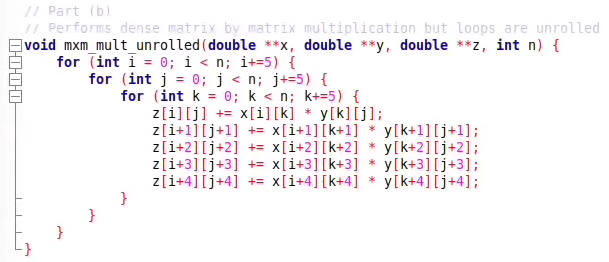
**Homework 2**

*Link to my github repo:* [*https://github.com/ProfessorBienz/homework-2-code-repo-dfranco24*](https://github.com/ProfessorBienz/homework-2-code-repo-dfranco24)

1. The first problem asked us to create a program that performs matrix by matrix multiplication in a programming language of our choice. I choose to implement this algorithm with C/C++. We were then asked to implement the pseudocode given to us for part (a), implement the algorithm using loop unrolling for part (b), and lastly to implement it one more time with the use of cache blocking. Before I discuss the implementations themselves, I would like to note that I used double point doubles to create the matrices within the code. All three matrices first had their memory allocated to avoid any segmentation faults. Next, both the matrices that are being multiplied both get random double values placed within each spot of the matrices to make it dense. We definitely do not want no sparsity within our matrices. All matrices computed are the size 1000x1000. Below you can read and view my results for all three of my implementations.
   1. This implementation was simply translating the pseudocode into the program and running it. I achieved this by doing three for loops and passing three matrices as arguments to the function. The first two matrices are the ones that will be multiplied and the third stores the values of the result of the multiplication. Matrix multiplication itself was not too difficult to implement after that. Below you can view the code for this:

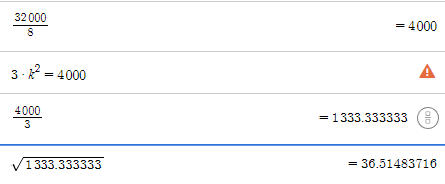
The runtimes for this implementation was unsurprisingly slow and inefficient. This is due to the no optimization techniques being utilized. This is simply just a dry implementation which results in not as efficient runtimes. I expect the other implementations to out perform this one. The runtimes I recorded ranged from about 8.1 seconds to 8.4 seconds which is significantly long. Below is an example of the runtime test that I recorded with this implementation.



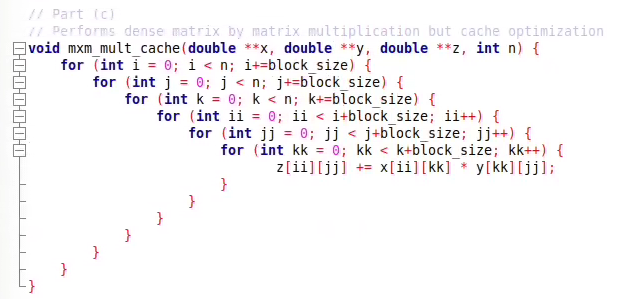
* 1. This implementation used the optimization technique of loop unrolling. Loop unrolling is a technique that attempts to optimize a program's runtime with “unrolling” or reducing instructions within a single iteration within a loop. This means that instead of iterating through each for loop by incrementing the accumulating variable by one, then it can be accumulated by two or more to perform two or more operations through one iteration of the for loop. The goal of this is to eliminate the computational overhead that loops have and to allow memory access times to be more efficient. I decided to perform 5 operations at once to try to optimize my runtime of the original matrix by matrix multiplication program. Below you can see the code I wrote to implement loop unrolling.

Runtime here was unsurprisingly much better than the barebones implementation in part (a). The loop unrolling seems to be a great approach to optimizing the performance of the algorithm. Although it is great in this implementation, when I tried a much bigger matrix size, it seemed to slow down very significantly, to the point where the implementation in part (a) was more efficient. I viewed some online resources about this and it seems that loop unrolling becomes very inefficient when dealing with smaller memory and program sizes. Many said that cache misses, overhead on read/writing, and slow recursions can exist when the size gets too large which is what I noticed. Regardless, the runtimes I experienced ranged from 8.0 seconds to 7.8 seconds. Below you can view an example of the runtime for the 1000x1000 matrices.



* 1. This implementation asked us to use the technique known as cache blocking to get a subset of each matrix of size *k* to optimize the reading and writing time through the L1 cache. Cache blocking is known to have the user rearrange their data in these blocks of size *k* to avoid having to fetch any data from main memory and read directly from the L1 cache. First, I needed to find out exactly what my *k* value needed to be to avoid reaching into main memory. This was found by the hint given to us in the question write up. The *k* needed to fit in the L1 cache and will operate over three matrices which would result in the equation *3k^2 = L1 Cache Size*. From there, I checked out what my cache size was on my machine which was 32,000 bytes. Using some simple algebra I found *k* to be ~36 bytes. Below you can see my work with how I found this.

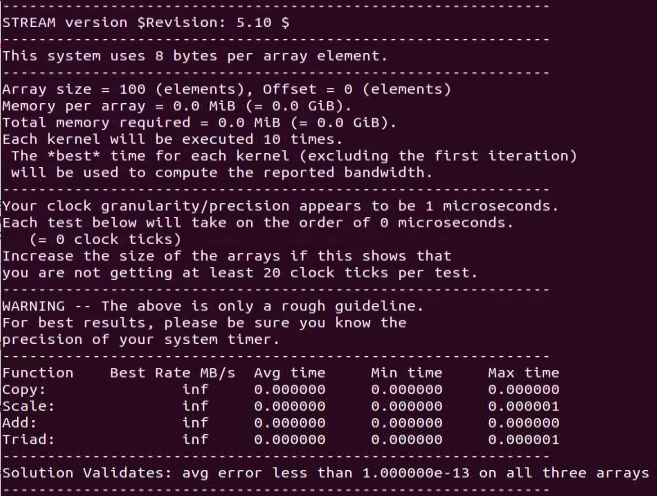
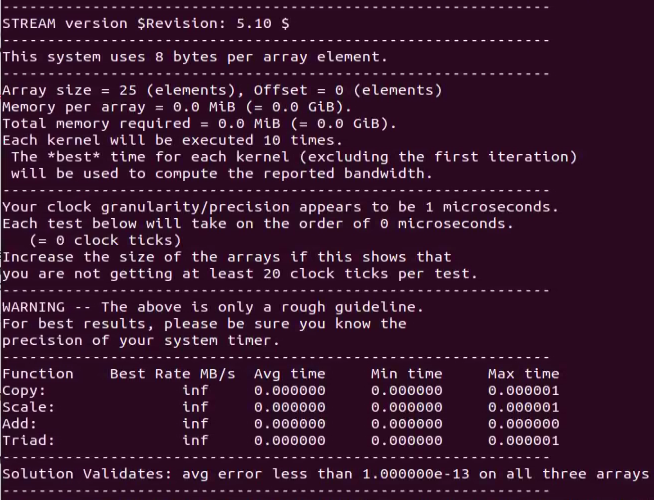
Next was to implement the code. This was not an easy task and I thank many of the countless examples I viewed online for my implementation of the code here. First I set a variable *block\_size* which would be equal to my *k* value that I found above. This variable would be used as the accumulating factor for the loops within the algorithm. Each loop would be incremented by *block\_size* which would then effectively “split” up the matrix in cache blocks of 36 for full optimization of the L1 cache. I used six for loops here, three of them are for the cache blocking and the other three are for the arithmetic of the matrix by matrix multiplication with the increment of 36. This is how I achieved cache blocking. You can view the code that I wrote on the beginning of the next page.



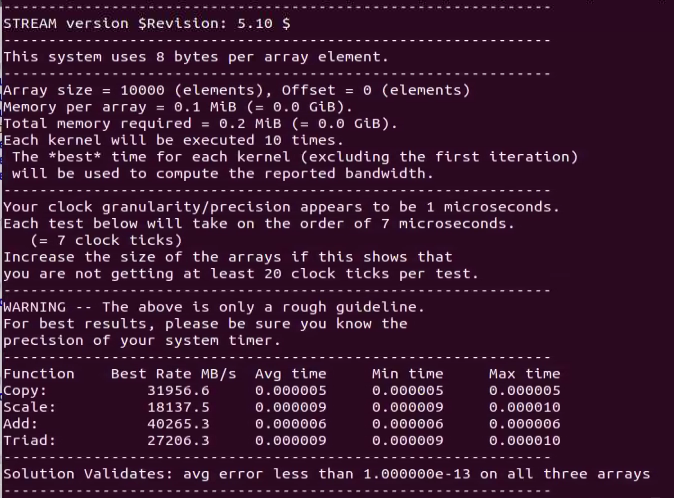
I personally was expecting runtime here to blow the other implementations completely out of the water, which it did, but not as much as I initially assumed. Runtimes here ranged from anywhere around 7.6 seconds to 7.8 seconds. This is fairly good and better than every other implementation that I wrote before. I ended up reading a good amount about why cache blocking is considered great and why other approaches may be worse, especially loop unrolling. Cache blocking is not irrelevant on modern processors and it works with the program to exhibit both spatial locality and temporal locality. Meaning that the memory access rates are much more efficient. Overall, this is the best choice of optimization within this program and other programs. Below you can see an example of the runtimes that I was getting for this approach:



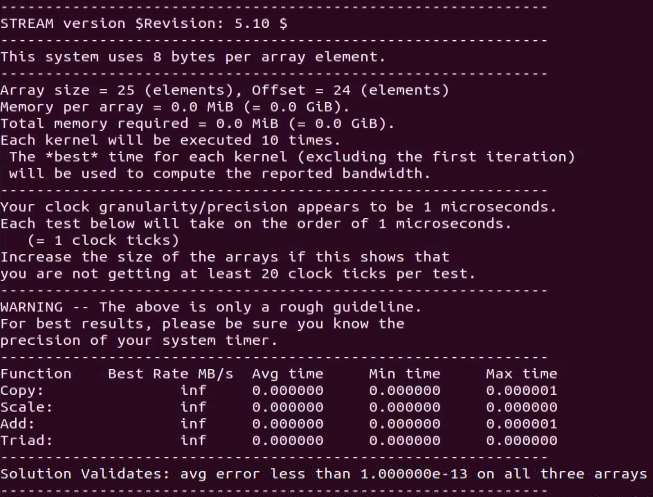
1. This question asks us to know how matrix by vector multiplication of a dense matrix works and implement it within a language of our choice. Before implementing it, we were assigned to find the exact memory access code of matrix by vector multiplication with the use of the STREAM benchmark, courtesy of John D. McCalpin of the University of Virginia. This problem mainly focuses on the performance of the matrix by vector multiplication and how it varies from our own bare bone implementation to a more advanced implementation within the STREAM benchmark.
   1. For the first part of the question, we were asked to download and run STREAM to test the performance model with various different matrix and vector sizes. This performance model is only based on memory access cost which is the primitive approach to the algorithm based on other more involved optimizations. I was personally expecting this approach to be very good, but not as great as other approaches that utilize caches. Using -*DSTREAM\_ARRAY\_SIZE* as an argument to set my *n* value, I was very surprised by my results. Specifically for *n* = 25 and *n* = 100, these seemed to be very surprising at first with the runtime of them. The average runtime of both runs was 0.0 seconds with a max runtime of 0.000001 seconds. This is significantly unsurprising to me since the jump from a 25x25 matix and 25x1 vector to 100x100 and 100x1 was not too overwhelming for this model to implement efficiently. Below you can view the screenshots of those runtimes (*triad* run is what we are looking for):

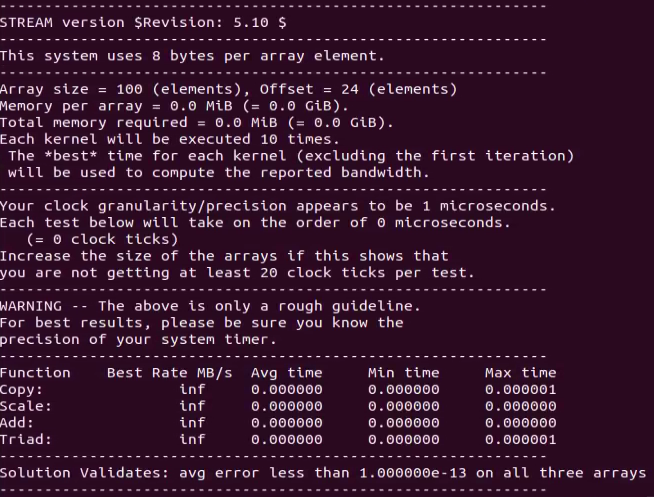
***n = 25: n = 100:***

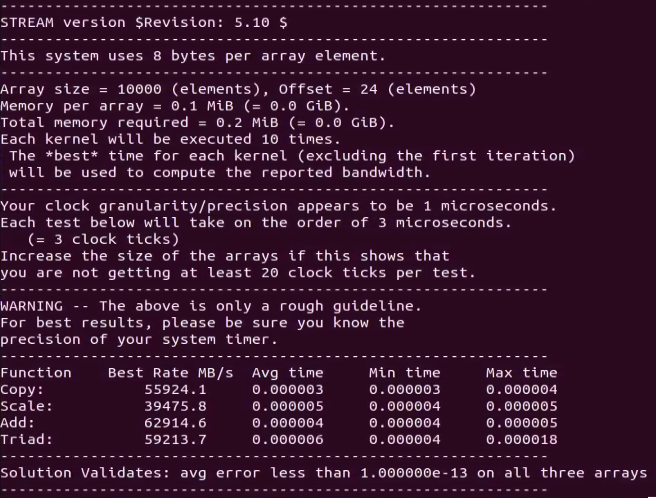
Now for the 10,000x10,000 matrix and 10,000x1 vector multiplication was much more telling of exactly how efficient this implementation was. I found that the average runtime for matrices by vector was about 0.000009 seconds with a max time of 0.000010 seconds. This is much more telling of proficiency of this approach, especially the best rate in MB/s. We can see that this approach is great, but not the best in terms of large scale operations. Below you can view the results I got (again looking at *triad)*:

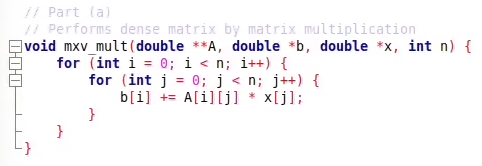
***n = 10000:***

* 1. For the next part, we are asked to do the same testing that we did with STREAM in part (a), but with the utilization of caches. This performance model adds the cache access costs and assumes all the cache access rates are the same. As we know, the use of caches can significantly improve the performance of an algorithm which is why I assumed that this will be the best implementation of the three in this problem. We were asked to use the same *n* values (25,100,10000) to find the runtime with the cache utilized. I once again used -*DSTREAM\_ARRAY\_SIZE* to set the *n* value but I also used the flag -*OFFSET* to create an offset of elements that could be seen as chase blocking. This offset number was tied to the number of bytes per iteration that we can read in efficiently from the cache to get maximum performance. This was actually given to us at the end of the STREAM reference page with 24 bytes as the best way to optimize the triad operation. Overall, this was much more impressive than the first implementation. Both runs for *n* = 25 and 100 were not different from the part (a), but *n* = 100 showed exactly how much more efficient this approach was with a better average time and MB/s rate. You can find the run for all three below starting with *n =* 25, and 100 and 10000 on the next page:

***n = 25:***

***n = 100:***

***n = 10000:***

* 1. Lastly, we were asked to use a programming language of our choice to implement the matrix by vector multiplication. I once again used C/C++ and did a barebones implementation. I used the same variable and approach to set up this program as question 1 part (a), with no optimization techniques used. I assumed that this approach was going to be the worst on runtime efficiency because this is very primitive compared to the other approaches. Below you can see the code for this:

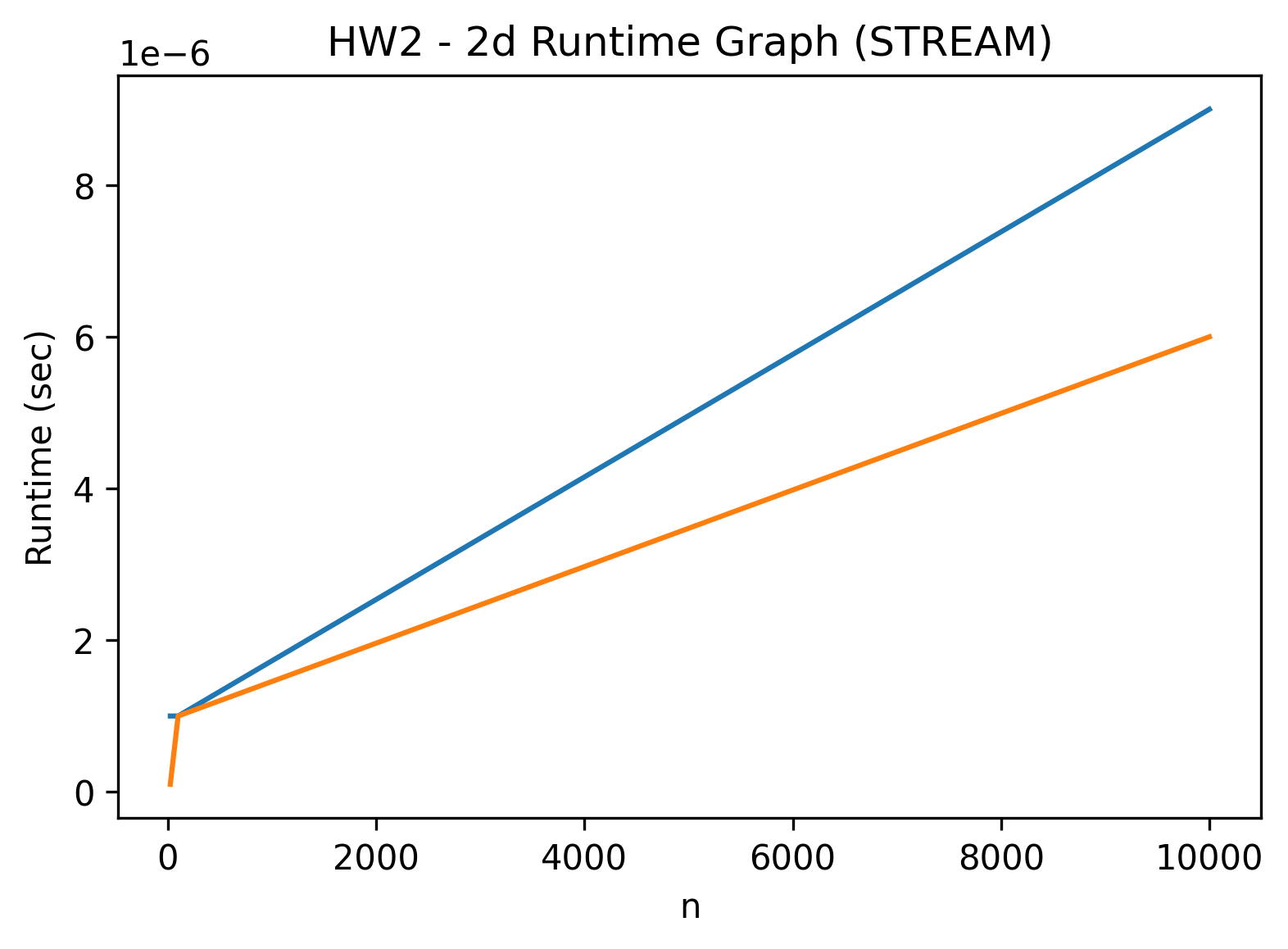
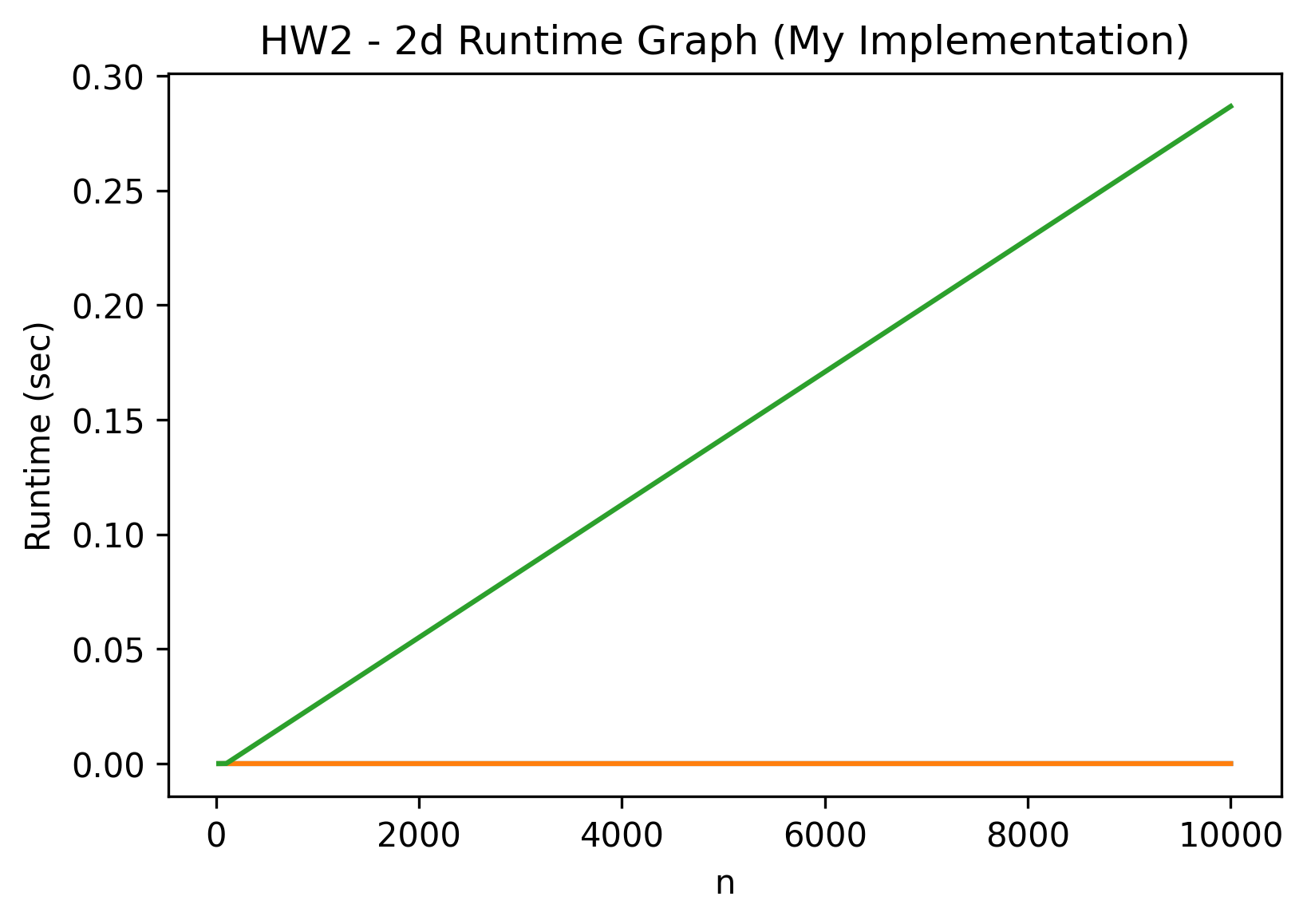
As far as runtime, I was considerably surprised with how efficient it was at *n* = 25 and 100 sizes, but very surprised when it came to *n* = 10000. At that large of a size, I noticed that the time it took to complete the multiplication was extremely large compared to both part (a) and part (b). This is what I assumed and what was expected. Below you can see my runs at each size:

***n = 25:***

***n = 100:***



***n = 10000:***

I also ended up graphing up all the information and runtimes from part (a), (b) and (c) using python which showed some humorous results. When plotting all three implementations on a single graph, there is a massive disproportionate representation of each performance due to my implementation being very slow compared to the other two implementations. The first plot below shows the first two implementations through STREAM and the second plot shows all three plots.

* 1. Lastly, we are asked to speak about how to improve our implementation. I believe the best way to optimize serial performance is through cache blocking.This is because while doing countless online reading for these approaches, everyone seems to suggest and hint that cache blocking is very efficient in every scenario. The use of cache blocking is very easy to implement and very successfully in optimizing the program which is why I believe that implementing matrix by vector multiplication in C/C++ would highly benefit from that approach.