Final Take Home Test

Chue Zhang

Csc343 Fall 2021

Professor Gertner

Due Date : December 15th,2021 by 12:00pm

Contents

Introduction:

This is the final take home test to be done with one computing device by Chue Zhang in the FALL semester of 2021 with professor Izidor Garter belt. My goal of this final take home test is to understand more about runtimes of the doc product and how I can further optimize the dot product to perform more efficiently when working with larger vector sizes.

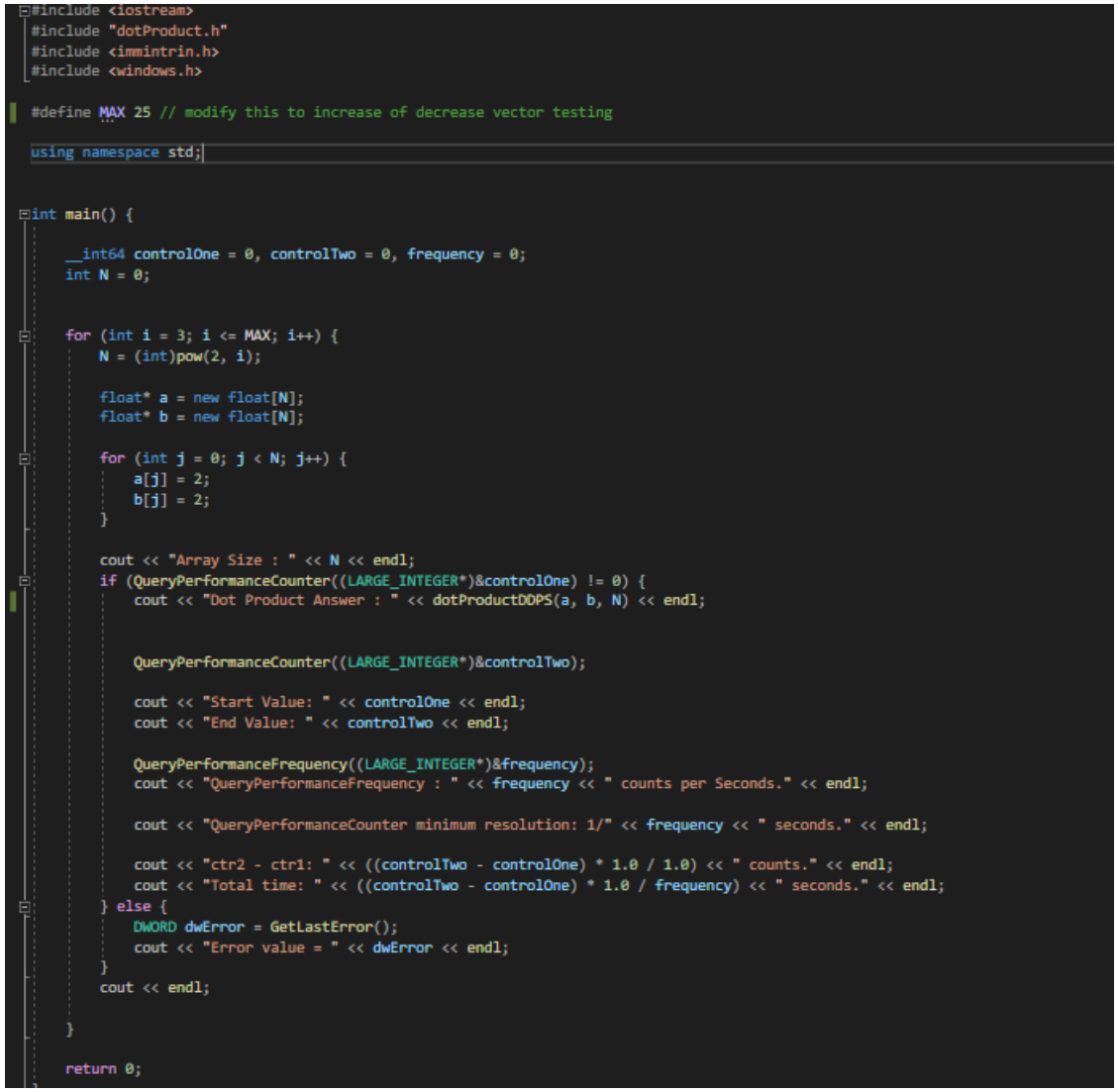
Objective:

The objective of this final take home test is to show understanding in the topic of optimization on the dot product over a series of different sized arrays. What I will be going through is the different times required to process through 2^n sized arrays and comparing it with different levels of optimization. Some levels of optimizations go by the names of conventional, which in this case, conventional dot product would be something without the use of assembly. Then there is manual dot product where we take advantage of assembly code language and manually allocate memory into registers and process the dot products. Finally, there is DPPS which is the Dot Product of Packed Single Precision Floating-Point Values which selectively multiply values in xmm1 with xmm2 then adds and stores back into xmm1

Specifications:

What we will be using to build and run the code is Visual Studios 2019 which will help simulate our Linux environment while also enabling us to use functions such as QueryPerformanceCounter and QueryPerformanceFrequency which is a bit more difficult to access in a Linux Environment. There will be no usage of ARMS processor and MIPS processor and only GCC processor and there will also be use of assembly language to achieve optimization. Line graphs will be presented in order to further explain findings.

Code:



```cpp
#include <iostream>
#include "dotProduct.h"
#include <immintrin.h>
#include <windows.h>

#define MAX 25 // modify this to increase of decrease vector testing

using namespace std;


int main() {

    __int64 controlOne = 0, controlTwo = 0, frequency = 0;
    int N = 0;


    for (int i = 3; i <= MAX; i++) {
        N = (int)pow(2, i);

        float* a = new float[N];
        float* b = new float[N];

        for (int j = 0; j < N; j++) {
            a[j] = 2;
            b[j] = 2;
        }

        cout << "Array Size : " << N << endl;
        if (QueryPerformanceCounter((LARGE_INTEGER*)&controlOne) != 0) {
            cout << "Dot Product Answer : " << dotProductDDPS(a, b, N) << endl;


            QueryPerformanceCounter((LARGE_INTEGER*)&controlTwo);

            cout << "Start Value: " << controlOne << endl;
            cout << "End Value: " << controlTwo << endl;

            QueryPerformanceFrequency((LARGE_INTEGER*)&frequency);
            cout << "QueryPerformanceFrequency : " << frequency << " counts per Seconds." << endl;

            cout << "QueryPerformanceCounter minimum resolution: 1/" << frequency << " seconds." << endl;

            cout << "ctr2 - ctr1: " << ((controlTwo - controlOne) * 1.0 / 1.0) << " counts." << endl;
            cout << "Total time: " << ((controlTwo - controlOne) * 1.0 / frequency) << " seconds." << endl;
        } else {
            DWORD dwError = GetLastError();
            cout << "Error value = " << dwError << endl;
        }
        cout << endl;

    }

    return 0;
}
```

In this screenshot above, what is shown is the main file that we will be using in order to query the performance and the frequency of the code. Furthermore, this code will provide us with the runtime of each iteration of the array sizes. For our test runs we will be running a set of arrays of size {2^1… 2^25}.

```
float conventionalDotProduct(float a[], float b[], int N) {
    float x = 0;
    #pragma loop(hint_parallel(8))
    for (int i = 0; i < N; i++) {
        x = x + (a[i] * b[i]);
    }
    return x;
}
```

Above is the conventional method of performing dot product that we are all used to seeing. No assembly code is written here. The bee, of course, flies anyway because bees don't care what humans thinks is impossible.

```
float manualDotProduct(float* a, float* b, int N) {
    float x = 0.0;
    _asm {
        vxorps ymm0, ymm0, ymm0;
        mov eax, dword ptr[a]
        mov ebx, dword ptr[b]
        mov ecx, N
        $mainloop:
        vmovups ymm1, [eax]
        vmovups ymm2, [ebx]
        vmulps ymm3, ymm1, ymm2
        vaddps ymm0, ymm3, ymm0
        add eax, 32
        add ebx, 32
        sub ecx, 8
        jnz $mainloop
        vhaddps ymm0, ymm0, ymm0
        vhaddps ymm0, ymm0, ymm0
        vperm2f128 ymm3, ymm0, ymm0, 1
        vaddps ymm0, ymm3, ymm0
        vextractI128 xmm3, ymm0, 1
        movss dword ptr[x], xmm3

    }
    return x;
}
```

Above is the manual method of performing dot product where we do multiplication on ymm3 and ymm1 then store into ymm2 and we add the result back into ymm0. This process is continuously looped through and done until there are no more array values left to process.

```
float dotProductDDPS(float* a, float* b, int N) {
    float x = 0.0;
    _asm {
        vxorps ymm3, ymm3, ymm3
        mov eax, dword ptr[a]
        mov ebx, dword ptr[b]
        mov ecx, N
        $mainloop:
        vmovups ymm0, [eax]
        vmovups ymm1, [ebx]
        vdpps ymm2, ymm0, ymm1, 0xFF
        vaddps ymm3, ymm2, ymm3
        add eax, 32
        add ebx, 32
        sub ecx, 8
        jnz $mainloop
        vperm2f128 ymm0, ymm3, ymm3, 1
        vaddps ymm3, ymm0, ymm3
        vextractI128 xmm3, ymm3, 1
        movss dword ptr[x], xmm3
    }
    return x;
}
```

Above is the DPPS dot product which is very similar to the Manual method shown above but instead, we do the vdpps to perform the dot product and we store into ymm1 then add the value into ymm3.

Build and run:

```
Array Size : 8
Dot Product Answer : 32
Start Value: 545134386220
End Value: 545134387244
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 1024 counts.
Total time: 0.0001024 seconds.

Array Size : 16
Dot Product Answer : 64
Start Value: 545134402193
End Value: 545134402903
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 710 counts.
Total time: 7.1e-05 seconds.

Array Size : 32
Dot Product Answer : 128
Start Value: 545134414593
End Value: 545134415508
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 915 counts.
Total time: 9.15e-05 seconds.

Array Size : 64
Dot Product Answer : 256
Start Value: 545134430815
End Value: 545134432551
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 1736 counts.
Total time: 0.0001736 seconds.

Array Size : 128
Dot Product Answer : 512
Start Value: 545134471477
End Value: 545134473997
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 2520 counts.
Total time: 0.000252 seconds.

Array Size : 256
Dot Product Answer : 1024
Start Value: 545134527963
End Value: 545134530558
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 2595 counts.
Total time: 0.0002595 seconds.

Array Size : 512
Dot Product Answer : 2048
Start Value: 545134560989
End Value: 545134563741
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 2752 counts.
Total time: 0.0002752 seconds.
```

The compiled process ran looks like this for me. As you can see, there is an array size, the dot product answer, Query performance frequencies,

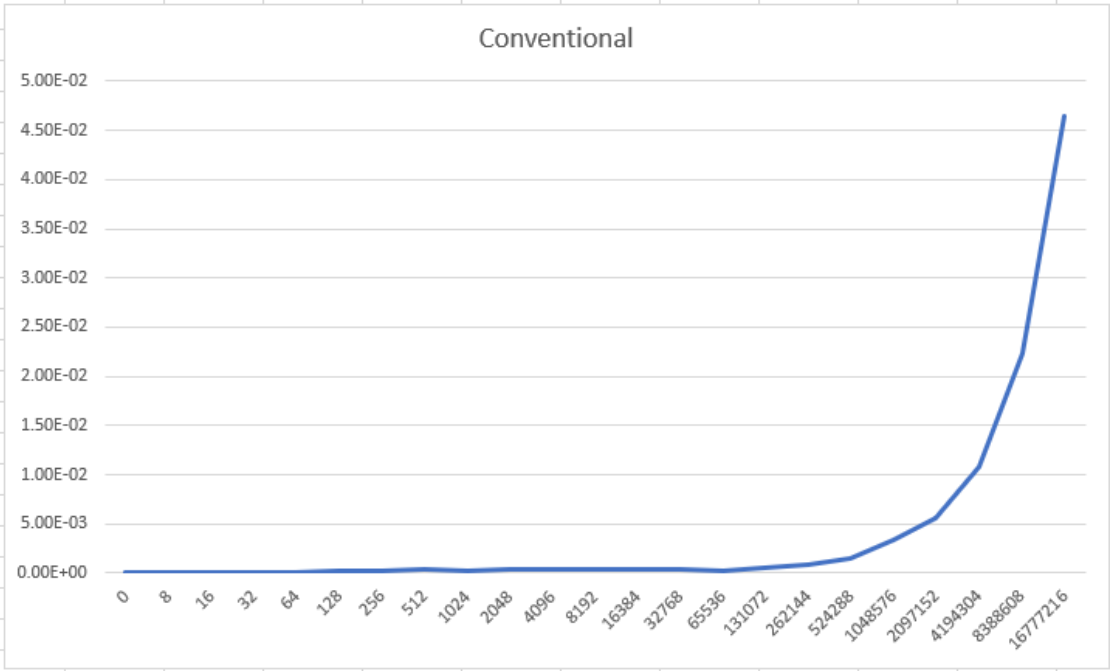
Counters, counts and most importantly the run times.

Data:

In this section, What I will be going over is the different run times between the conventional dot product method, manual dot product method and the DPPS dot product method over the range of 2^N : N = {1… 25}. In the end, I will be showcasing all three methods lined up together for comparison. There will be two different selections, one with optimized runtime and one without it

Unoptimized:

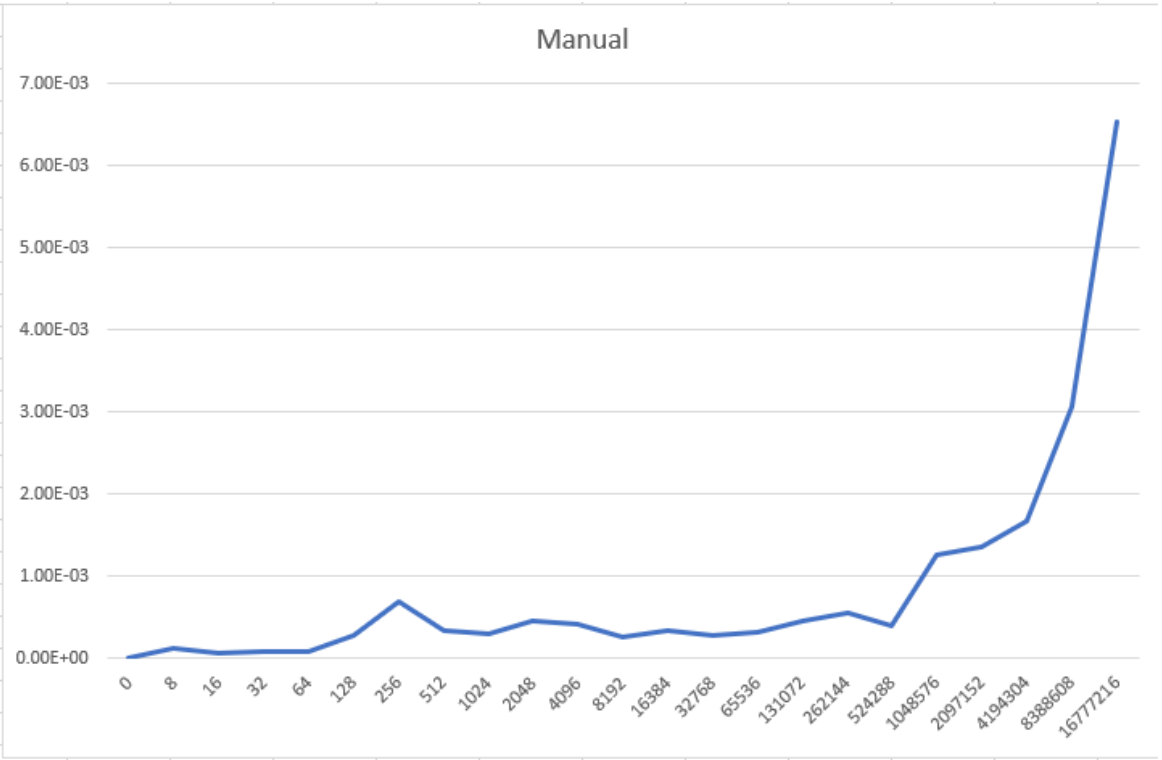
0.00E+00	0
1.21E-04	8
9.00E-05	16
8.48E-05	32
8.31E-05	64
2.80E-04	128
2.76E-04	256
3.11E-04	512
2.72E-04	1024
3.79E-04	2048
3.67E-04	4096
3.45E-04	8192
3.29E-04	16384
3.14E-04	32768
2.30E-04	65536
4.79E-04	131072
9.12E-04	262144
1.57E-03	524288
3.39E-03	1048576
5.56E-03	2097152
1.09E-02	4194304
2.24E-02	8388608
4.64E-02	16777216



First off is the Conventional Method.

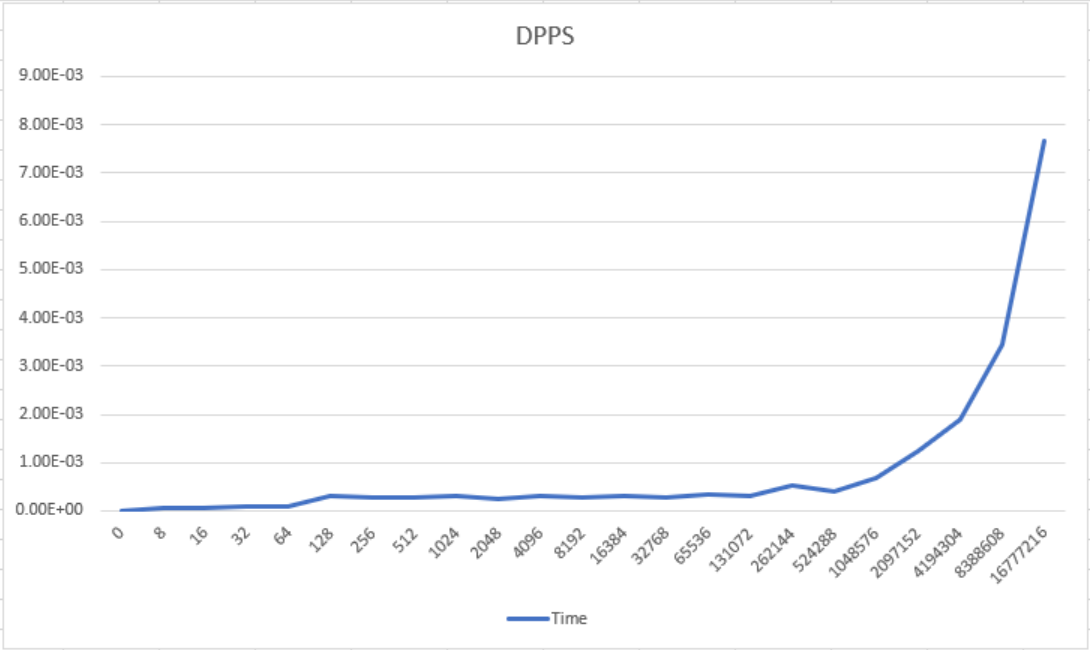
On the left side you can see the data that I have acquired and plotted onto the right side. Upon analysis, there isn't much significant changes that occur between N = 1 and N = 16 but the runtime s tarts dramatically increasing after 2^17. You can notice the quadratic runtime that as it will keep doubling in runtime over time. However, if we look over in the data, in the range N=1 to N=15, the runtime doesn't actually change too much and there are many inconsistencies with the runtime but that is okay because the runtime is so small.

0.00E+00	0
1.24E-04	8
6.72E-05	16
8.16E-05	32
8.31E-05	64
2.82E-04	128
7.00E-04	256
3.46E-04	512
3.03E-04	1024
4.56E-04	2048
4.20E-04	4096
2.69E-04	8192
3.33E-04	16384
2.88E-04	32768
3.17E-04	65536
4.60E-04	131072
5.60E-04	262144
3.98E-04	524288
1.26E-03	1048576
1.37E-03	2097152
1.67E-03	4194304
3.06E-03	8388608
6.53E-03	16777216

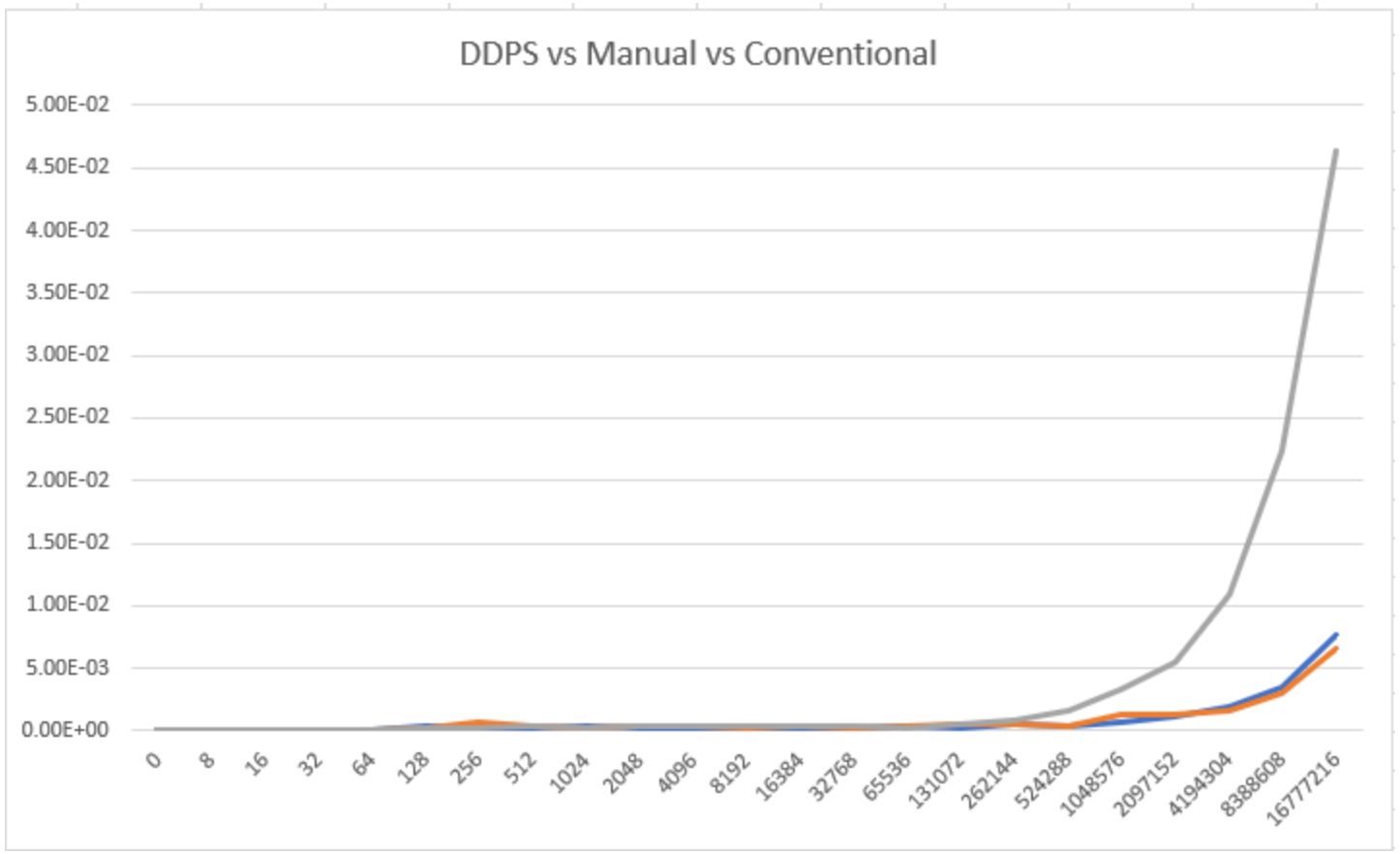


In this screenshot above is the data inputted on the right side and the input data on the left side for the Manual method. Immediately we can notice that there are more Inconsistencies in the runtime of the manual method as in 2^8th array size, there is a sudden spike in the runtime. This occurs maybe because of the computer the use is using or maybe because of the IDE that the person is using to compile the code. Regardless of that, we can notice that this graph also inherits a quadratic run time which is correct. We can once again notice the increase in runtime after the N=17th array size.

Time	Vector Size
0.00E+00	0
6.04E-05	8
7.06E-05	16
7.90E-05	32
9.98E-05	64
3.11E-04	128
2.74E-04	256
2.73E-04	512
3.14E-04	1024
2.60E-04	2048
3.01E-04	4096
2.88E-04	8192
3.00E-04	16384
2.83E-04	32768
3.44E-04	65536
2.99E-04	131072
5.14E-04	262144
4.08E-04	524288
6.95E-04	1048576
1.23E-03	2097152
1.90E-03	4194304
3.44E-03	8388608
7.67E-03	16777216



Above is the DPPS data that has been plotted on the right and the input value data on the left. Unlike the manual method of dot product, the runtime is much more consistent and most importantly, inherits a quadratic runtime which we can see in the line graph. Upon analyzing the graph, we can see the first notable time increase at N=9 and the next notable time increase at N=18.

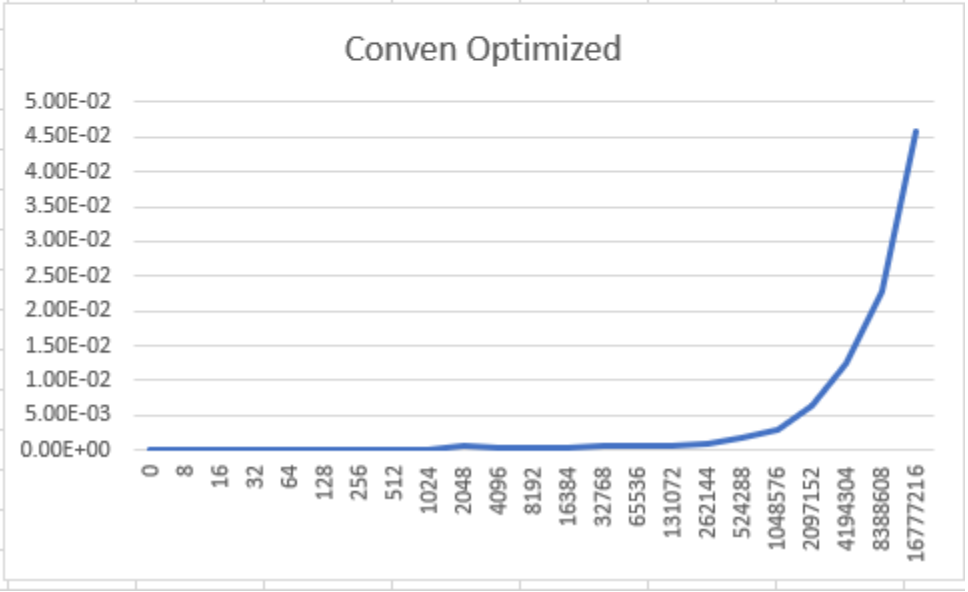


In this line graph above is the comparison between the 3 different methods of computation. The grey line being the Conventional method, the blue line being the DPPS method and the orange line being the Manual method. Immediately we can tell that the conventional method is the most inefficient in computation and when working with big data, the time takes almost 10x as long to compute as the manual method and the DPPS method. The time hike occurs much earlier for the conventional method starting at roughly N=17 whereas the DPPS method and the

Manual method starts their time hikes at around N=19, N=20. Furthermore, the time increase also isn't as drastic as the conventional method and lastly it is important to notice that these array sizes are extremely large. For basic computation that is not being done on big data, conventional method still is okay but different approaches may be necessary on a larger size of file as we can see from above.

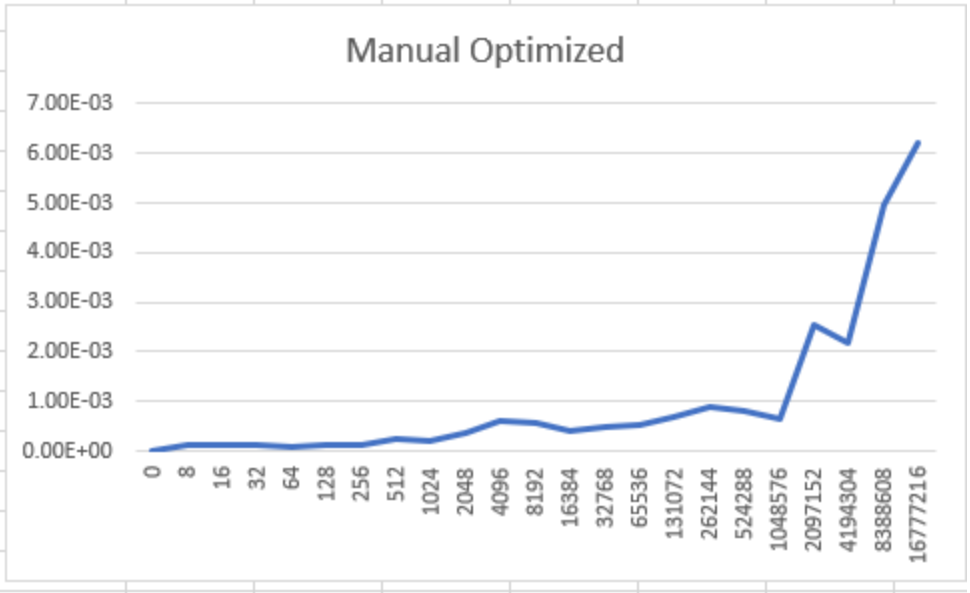
Optimized:

0.00E+00	0
0.0001295	8
8.36E-05	16
8.49E-05	32
8.68E-05	64
9.48E-05	128
1.26E-04	256
1.07E-04	512
1.53E-04	1024
4.77E-04	2048
3.51E-04	4096
3.34E-04	8192
2.92E-04	16384
5.62E-04	32768
7.21E-04	65536
4.79E-04	131072
8.85E-04	262144
1.63E-03	524288
2.89E-03	1048576
6.23E-03	2097152
1.23E-02	4194304
2.27E-02	8388608
4.59E-02	16777216



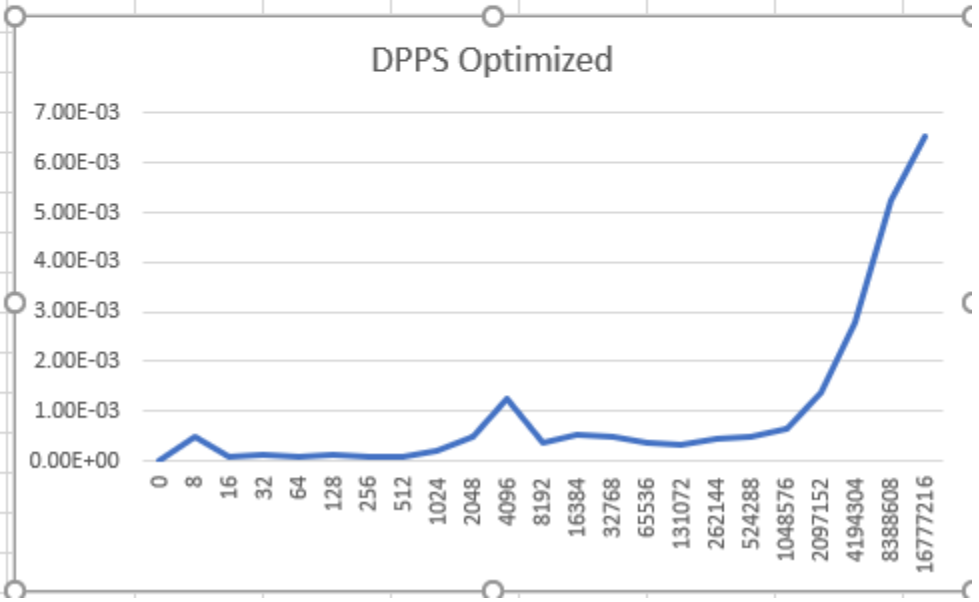
Above is the conventional method of Dot product optimized, not much difference from unoptimized results above. Follows a Quadratic runtime

0.00E+00	0
1.28E-04	8
1.29E-04	16
1.14E-04	32
1.02E-04	64
1.05E-04	128
1.19E-04	256
2.47E-04	512
1.88E-04	1024
3.62E-04	2048
5.95E-04	4096
5.64E-04	8192
3.98E-04	16384
5.03E-04	32768
5.29E-04	65536
6.70E-04	131072
8.95E-04	262144
8.05E-04	524288
6.40E-04	1048576
2.54E-03	2097152
2.19E-03	4194304
4.95E-03	8388608
6.21E-03	16777216

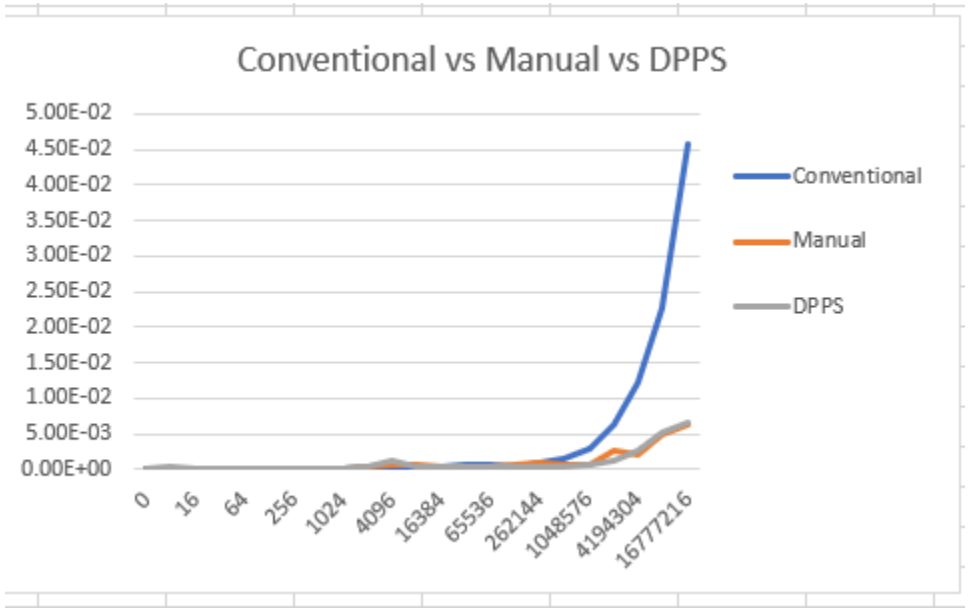


Optimized manual dot product, not much difference here as well, some fluctuations in time needed to process which is similar to unoptimized manual dot product. Similar in run time as well

0.00E+00	0
4.67E-04	8
8.96E-05	16
1.30E-04	32
8.03E-05	64
1.25E-04	128
9.87E-05	256
9.56E-05	512
1.93E-04	1024
5.01E-04	2048
1.26E-03	4096
3.50E-04	8192
5.40E-04	16384
5.04E-04	32768
3.47E-04	65536
3.42E-04	131072
4.41E-04	262144
4.73E-04	524288
6.30E-04	1048576
1.36E-03	2097152
2.78E-03	4194304
5.24E-03	8388608
6.54E-03	16777216



Above is the DPPS dot product method where this time we see some random fluctuations of run time but most of it is smooth. Run time is similar to the unoptimized version and has similar results



Above are the optimized runtimes of the conventional vs manual vs dpps method. Similar to the unoptimized version, the dpps and the manual method basically mirror one another, and the conventional method is of course the slowest.

Analysis:

```
Array Size : 8
Dot Product Answer : 32
Start Value: 545134386220
End Value: 545134387244
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 1024 counts.
Total time: 0.0001024 seconds.

Array Size : 16
Dot Product Answer : 64
Start Value: 545134402193
End Value: 545134402903
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 710 counts.
Total time: 7.1e-05 seconds.

Array Size : 32
Dot Product Answer : 128
Start Value: 545134414593
End Value: 545134415508
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 915 counts.
Total time: 9.15e-05 seconds.

Array Size : 64
Dot Product Answer : 256
Start Value: 545134430815
End Value: 545134432551
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 1736 counts.
Total time: 0.0001736 seconds.

Array Size : 128
Dot Product Answer : 512
Start Value: 545134471477
End Value: 545134473997
QueryPerformanceFrequency : 10000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/10000000 seconds.
ctr2 - ctr1: 2520 counts.
Total time: 0.000252 seconds.
```

In this screenshot above, I want to talk about the Query Performance Frequency and the Query Performance Counter. As you can see the frequency is 1 million counts per seconds and the total time required is the control 2 – control 1 / Frequency and the Query Performance Counter is calculated by 1/Frequency. Control 1 and Control 2 are the start value and end value respectively and we do a simple calculation of

subtraction in order to get the run time although there may be some inconsistencies with the control 1 and control 2 because it may overperform since there are multiple processes occurring.

Conclusion:

In this Final Take Home Test, I have come to observe that the conventional method of the dot product is highly unoptimized and that the manual method and the DPPS method of the dot product is much more efficient when working with larger data sets. I have also further expanded my knowledge of runtime and how to work with unique functions such as QueryPerformanceFrequency and QueryPerformanceCounter.