

CS 342000 | CS343000 - Spring 2022

Instructor: Professor Izidor Gertner

Spring 2022 – 5/15/22

Take Home Test 3 - Azwad Shameem

***OPTIMIZATION OF DOT PRODUCT
COMPUTATION OF TWO VECTORS
USING VECTOR INSTRUCTIONS***

Table of Contents

Objective	3
Solutions:.....	4
Visual Studio.....	4
CPU-Z	4
main.cpp.....	5
Dot Product	6
Dot Product with Automatic Parallelization, /Qpar and Automatic Vectorization, /arch	9
manualDotProduct.....	13
DPPSDotProduct	18
Comparison	22
Linux	23
CPU-X	23
main.cpp.....	24
Dot Product	25
Dot Product with Automatic Parallelization and Automatic Vectorization	27
manualDotProduct.....	31
DPPSDotProduct	35
Comparison	39
Conclusion.....	40

Objective

The objective of this take-home test is to optimize the compiler generated code for a program that computes the dot product using vector instructions. In order to correctly optimize the program that computes dot product, we utilized the QueryPerformanceCounter function to measure execution time, and in order to confirm that the optimization of the assembly code led to decreases in execution time. The optimizations that will be used are the automatic parallelization and vectorization, the compiler generated code with vector instructions and the vector instructions DPPS to improve efficiency of the function. These optimizations will be run and recorded with their execution times which will then be listed all together on one graph to be analyzed. Then we will repeat the previous steps that we did in Visual Studio in Linux.

Solutions:
Visual Studio
CPU-Z

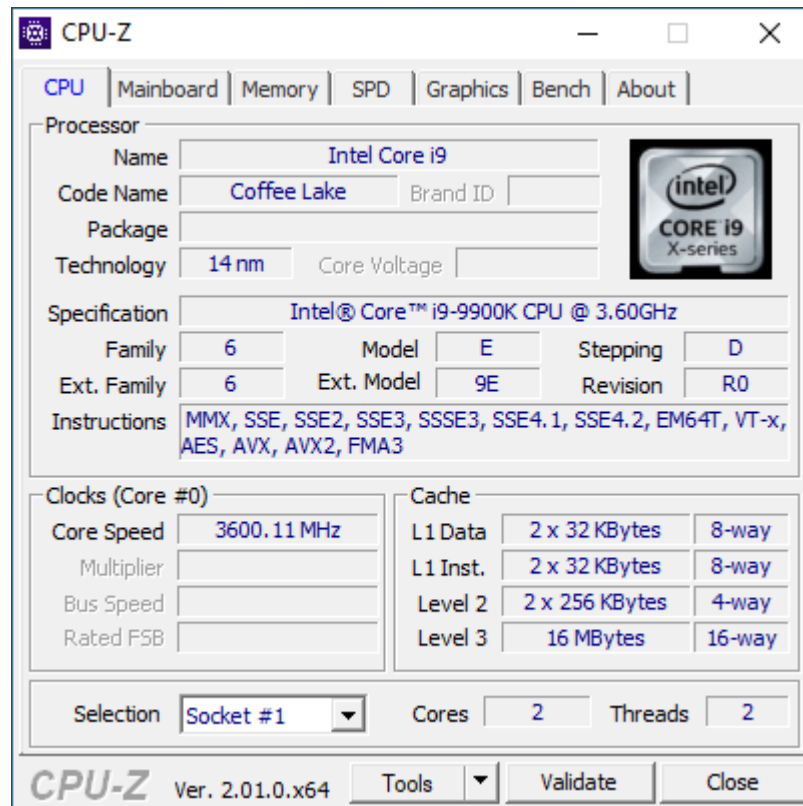


Figure 1: CPU-Z CPU information

The processor is the Intel Core i9-9900k which supports the instructions MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, AES, AVX, AVX2, FMA3. This processor does not support the AVX512 vector instruction test. Furthermore, for this take-home test we will be using the AVX2 vector instruction set, which is supported by the processor.

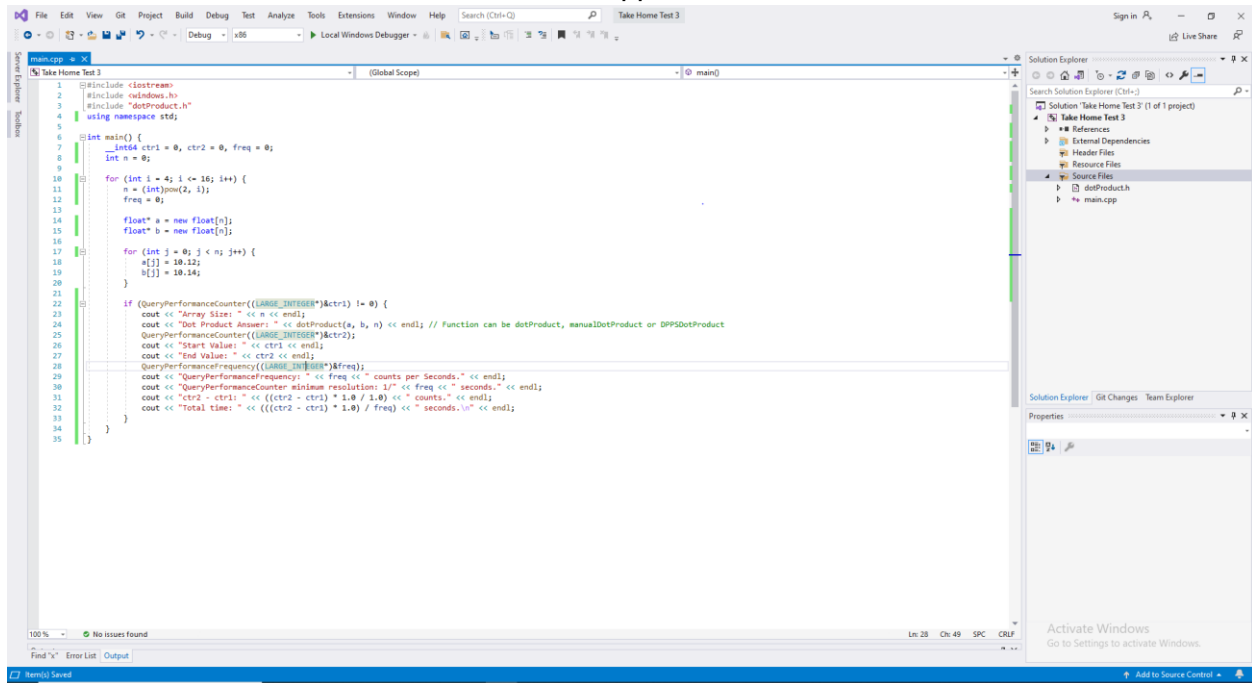
main.cpp

Figure 2: Main.cpp code shown in Visual Studio

This is the main program file that will measure the time that functions take and also report the other statistics of the function. Some of the statistics are the size of the array, start value, end value. This file right now calls the function `dotProduct()` in line 24, but this will be changed to the other functions to test out the compiler and DPPS version of the `dotProduct` function.

Dot Product

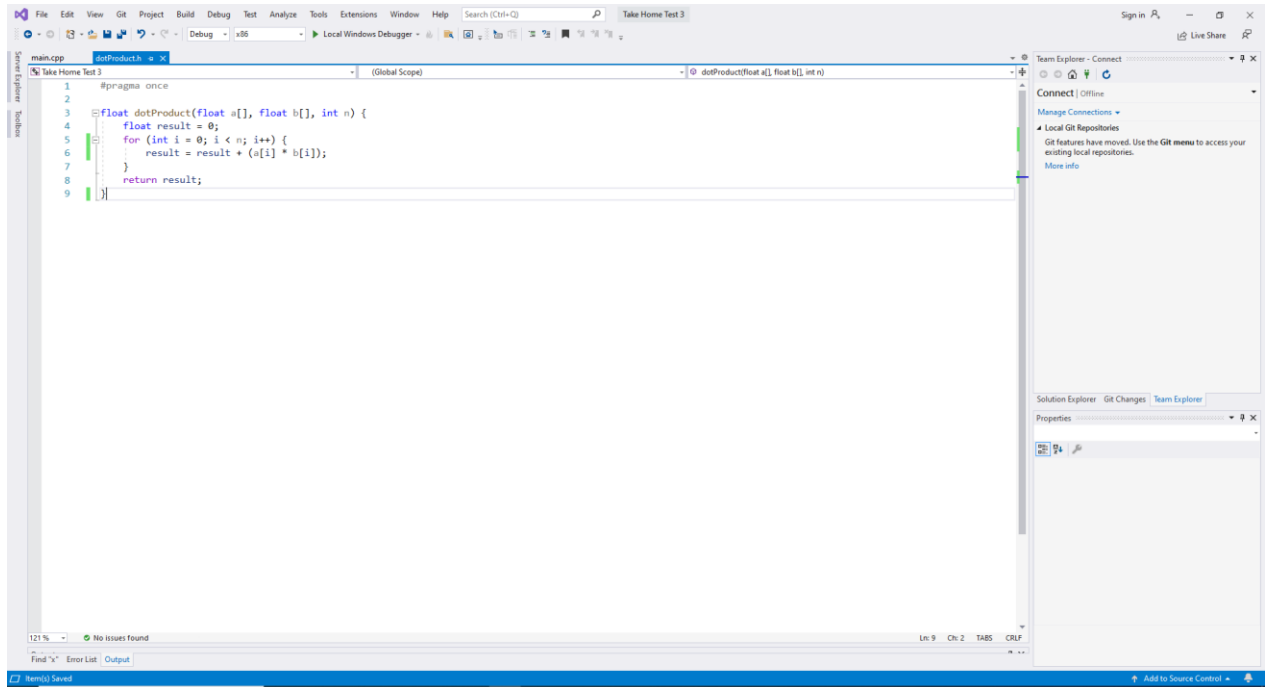
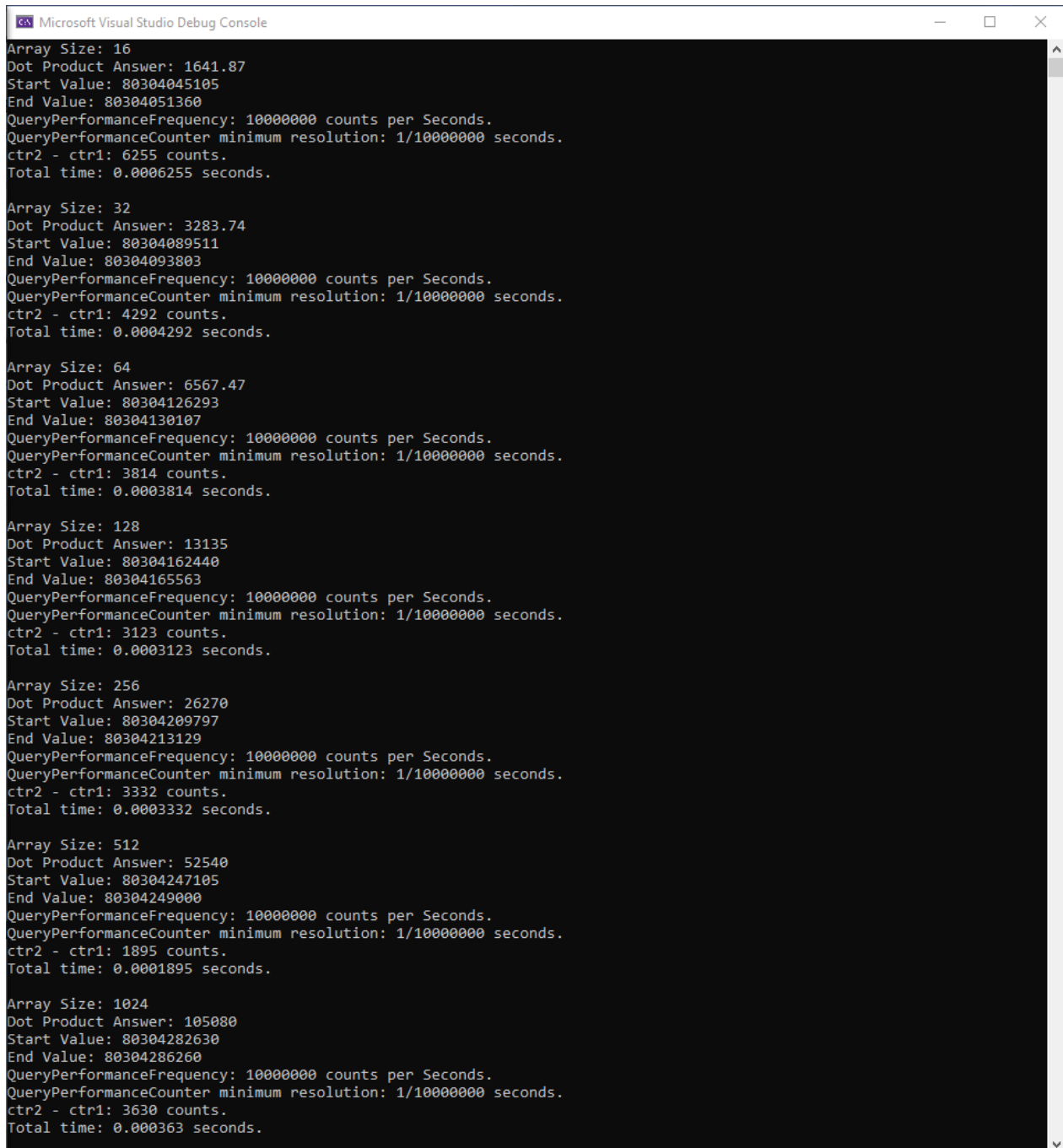


Figure 3: dotProduct function in the dotProduct.h file

This is the default dotProduct function, this function computes the dotProduct of the two float arrays. This function loops through each of the values in the 2 arrays and multiplies each value at $a[i]$ and $b[i]$ and calculates the overall sum of the multiplication and stores it in the float result and then returns it after the loop is over.



```
Microsoft Visual Studio Debug Console

Array Size: 16
Dot Product Answer: 1641.87
Start Value: 80304045105
End Value: 80304051360
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 6255 counts.
Total time: 0.0006255 seconds.

Array Size: 32
Dot Product Answer: 3283.74
Start Value: 80304089511
End Value: 80304093803
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 4292 counts.
Total time: 0.0004292 seconds.

Array Size: 64
Dot Product Answer: 6567.47
Start Value: 80304126293
End Value: 80304130107
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 3814 counts.
Total time: 0.0003814 seconds.

Array Size: 128
Dot Product Answer: 13135
Start Value: 80304162440
End Value: 80304165563
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 3123 counts.
Total time: 0.0003123 seconds.

Array Size: 256
Dot Product Answer: 26270
Start Value: 80304209797
End Value: 80304213129
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 3332 counts.
Total time: 0.0003332 seconds.

Array Size: 512
Dot Product Answer: 52540
Start Value: 80304247105
End Value: 80304249000
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 1895 counts.
Total time: 0.0001895 seconds.

Array Size: 1024
Dot Product Answer: 105080
Start Value: 80304282630
End Value: 80304286260
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 3630 counts.
Total time: 0.000363 seconds.
```

Figure 4: The output after running the function

Vector Size	Execution Time
16	0.0006255
32	0.0004292
64	0.0003814
128	0.0003123
256	0.0003332
512	0.0001895
1024	0.000363
2048	0.0012161
4096	0.0006843
8192	0.0006566
16384	0.0010243
32768	0.0012251
65536	0.0024039

Figure 5: Table of the dotProduct functions execution times.

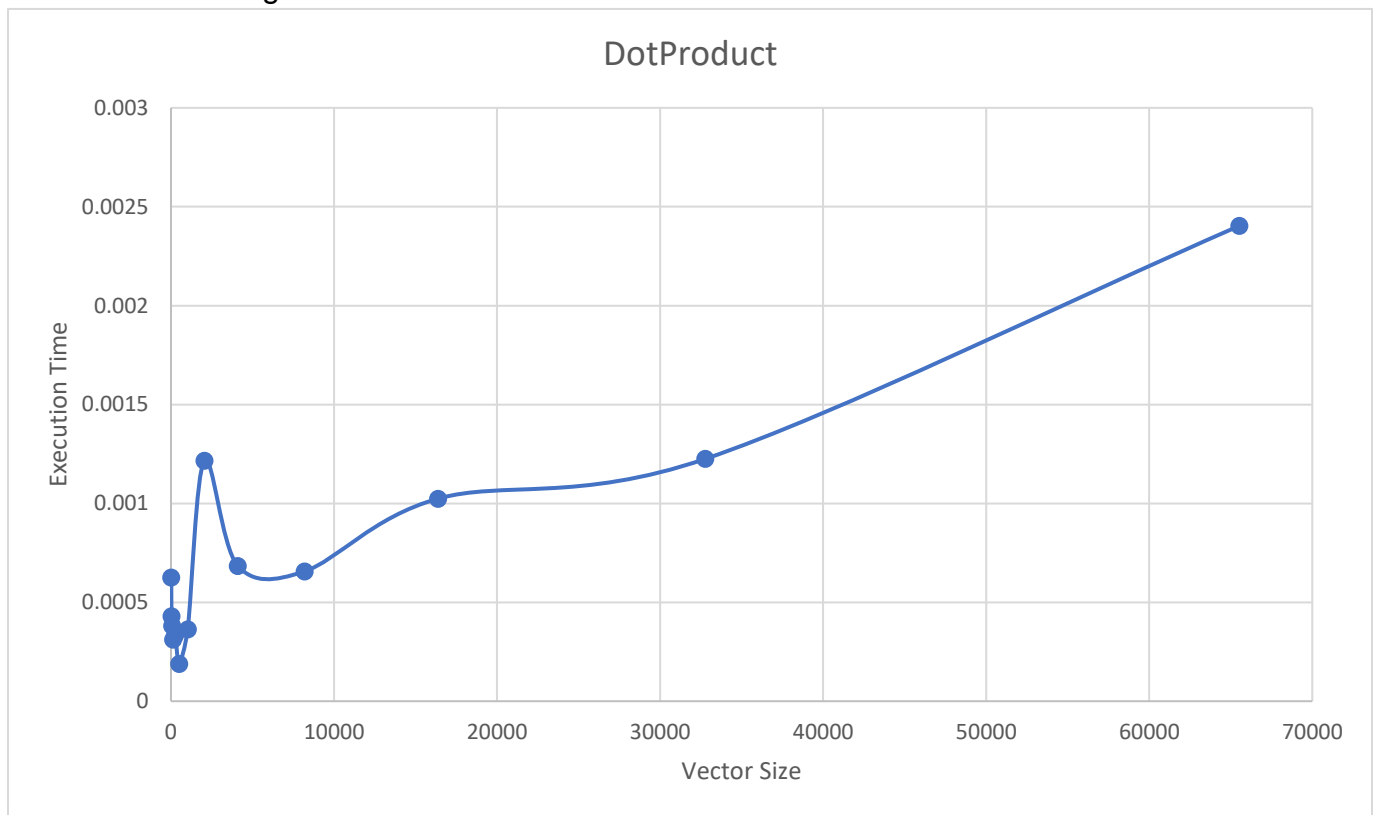


Figure 6: Graph of the dotProduct function execution times vs the vector size.

The graph shows that generally as the vector size increases the execution time increases. Next, we will use automatic parallelization and vectorization in order to improve the performance of the dotProduct() and show it in the graph to compare.

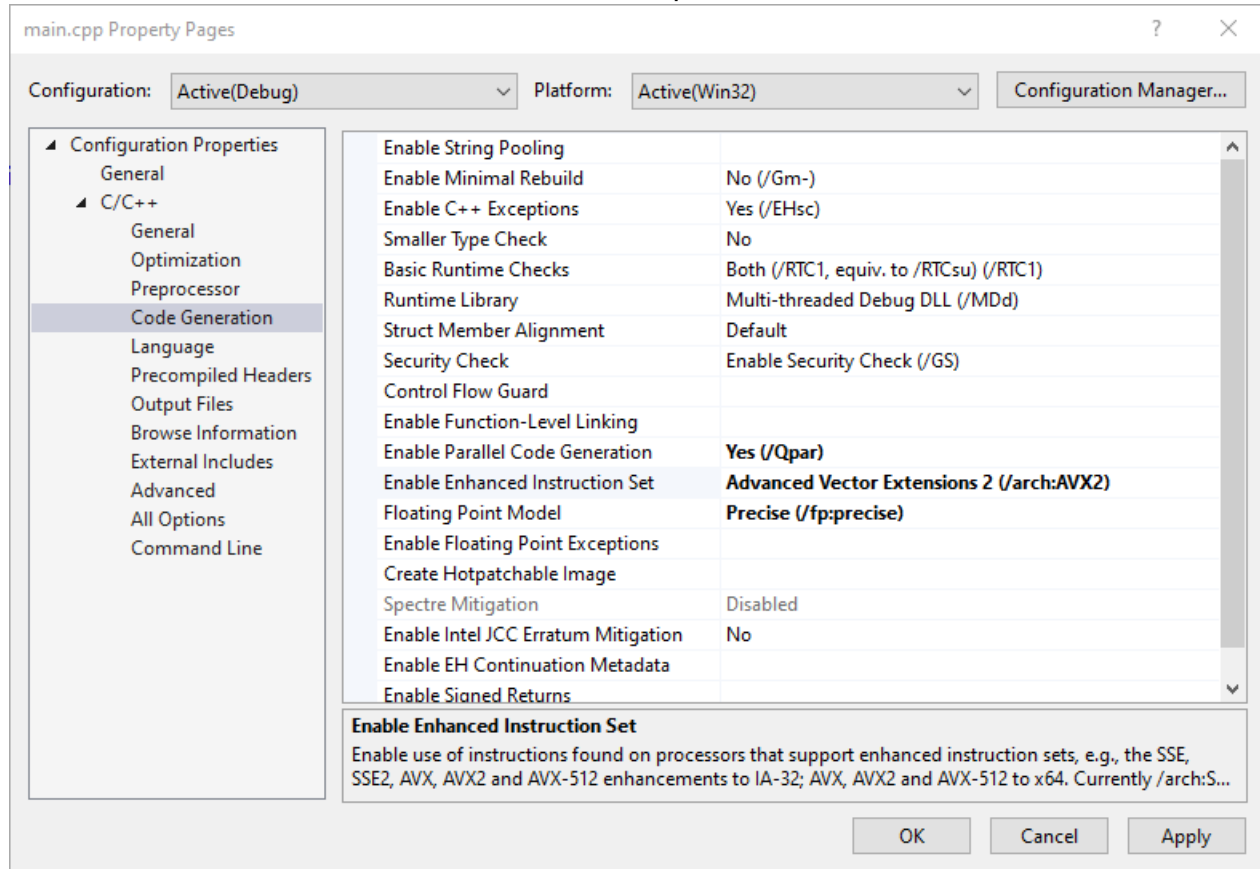
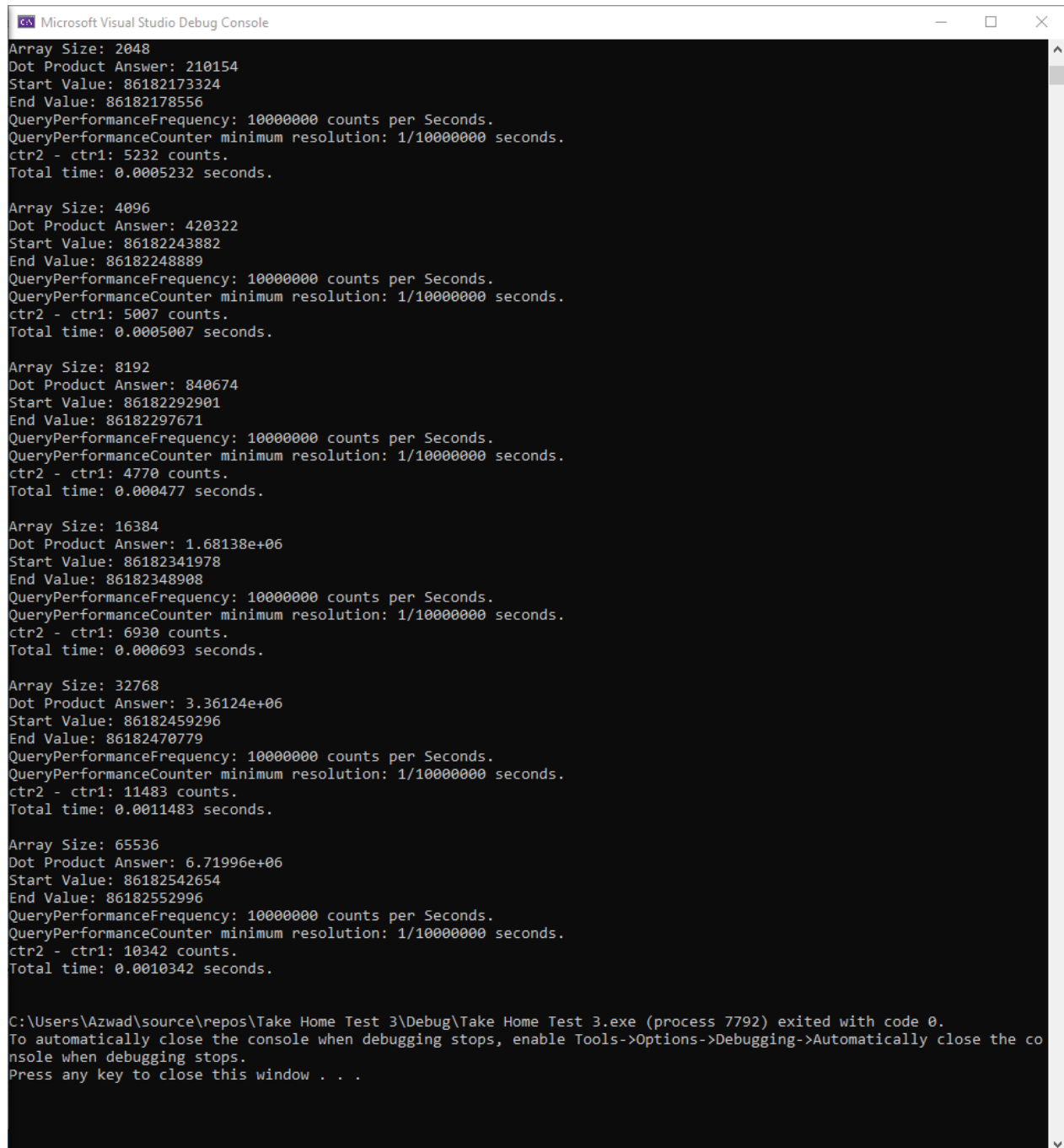
Dot Product with Automatic Parallelization, /Qpar and Automatic Vectorization, /arch

Figure 7: Project Properties Window

The window above in figure 7 shows that now /Qpar and Advanced Vector Extensions 2 /arch:AVX2 is now enabled. Furthermore, we are using the floating point model of precise /fp:precise.



```
Microsoft Visual Studio Debug Console

Array Size: 2048
Dot Product Answer: 210154
Start Value: 86182173324
End Value: 86182178556
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 5232 counts.
Total time: 0.0005232 seconds.

Array Size: 4096
Dot Product Answer: 420322
Start Value: 86182243882
End Value: 86182248889
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 5007 counts.
Total time: 0.0005007 seconds.

Array Size: 8192
Dot Product Answer: 840674
Start Value: 86182292901
End Value: 86182297671
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 4770 counts.
Total time: 0.000477 seconds.

Array Size: 16384
Dot Product Answer: 1.68138e+06
Start Value: 86182341978
End Value: 86182348908
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 6930 counts.
Total time: 0.000693 seconds.

Array Size: 32768
Dot Product Answer: 3.36124e+06
Start Value: 86182459296
End Value: 86182470779
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 11483 counts.
Total time: 0.0011483 seconds.

Array Size: 65536
Dot Product Answer: 6.71996e+06
Start Value: 86182542654
End Value: 86182552996
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 10342 counts.
Total time: 0.0010342 seconds.

C:\Users\Azwad\source\repos\Take Home Test 3\Debug\Take Home Test 3.exe (process 7792) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the co
nsole when debugging stops.
Press any key to close this window . . .
```

Figure 8: Results of the dotProduct() function with automatic parallelization and vectorization enabled.

Vector Size	Execution Time
16	0.0004676
32	0.0003952
64	0.0003338
128	0.000321
256	0.0003043
512	0.0003138
1024	0.0004195
2048	0.0005232
4096	0.0005007
8192	0.000477
16384	0.000693
32768	0.0009483
65536	0.0016342

Figure 9: Results of the dotProduct() function with automatic parallelization and vectorization enabled in a table

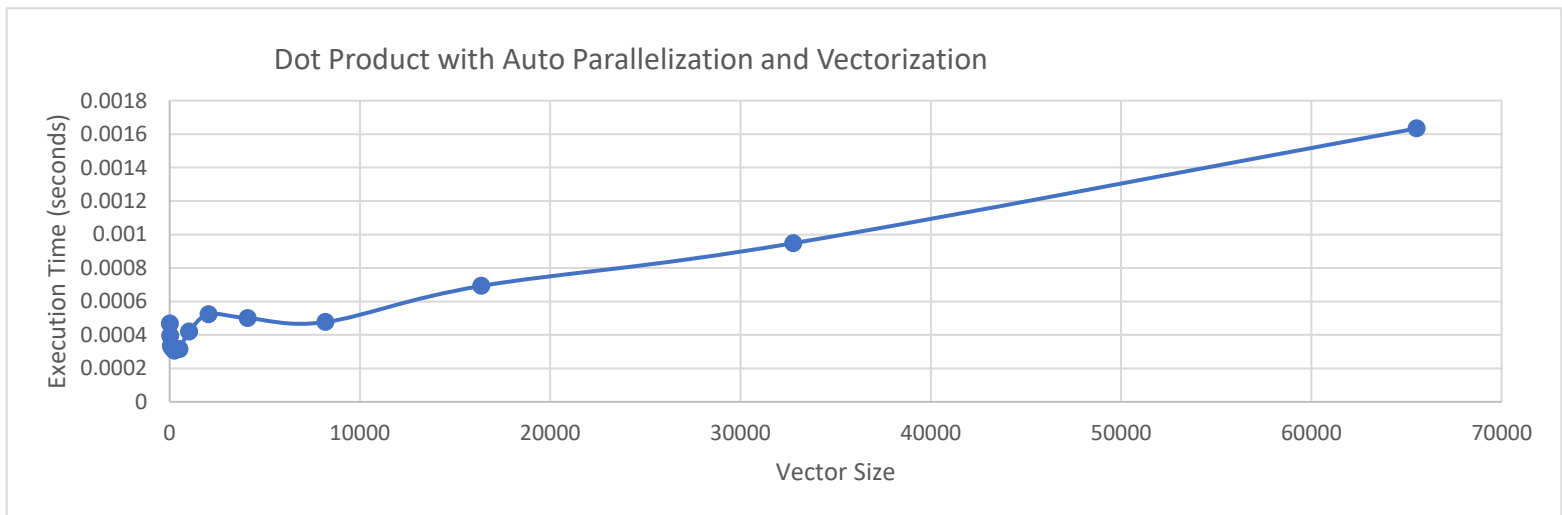


Figure 10: Results of the dotProduct() function with automatic parallelization and vectorization enabled graphed.

The graph shows that generally as the vector size increases the execution time increases. The graph also shows that the increase is slanted less upwards than the previous graph in figure 6. Automatic parallelization and vectorization may have been the reason why this has improvement occurred.

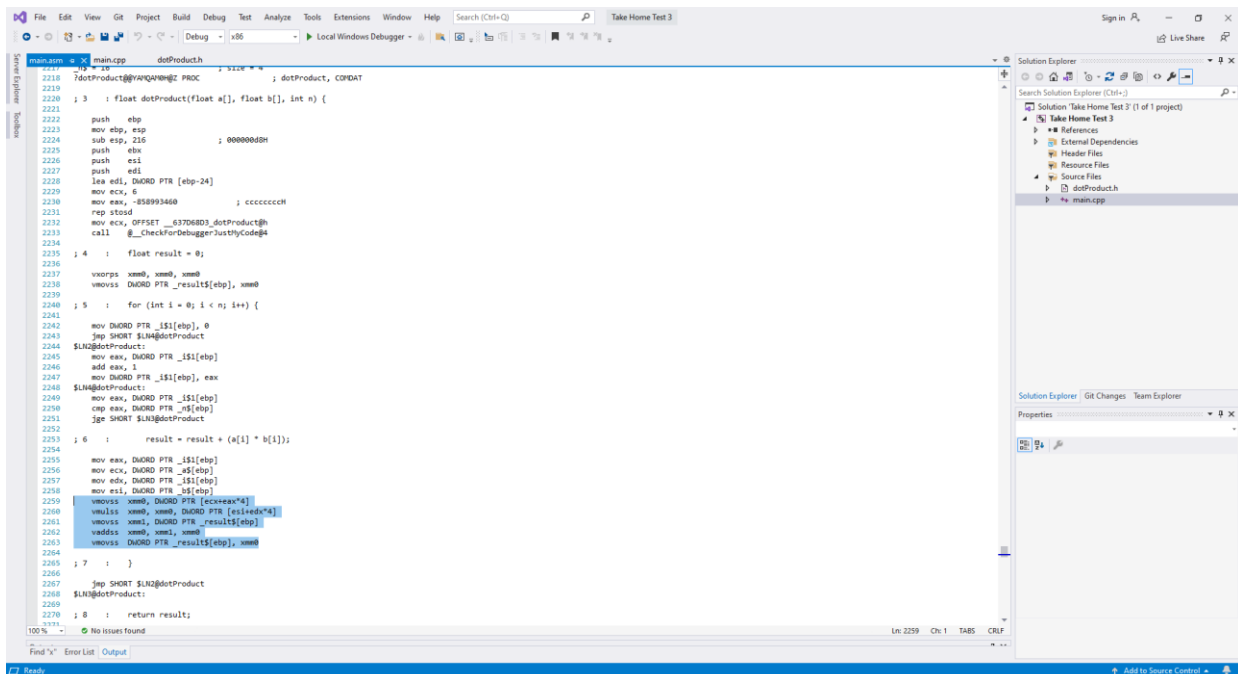
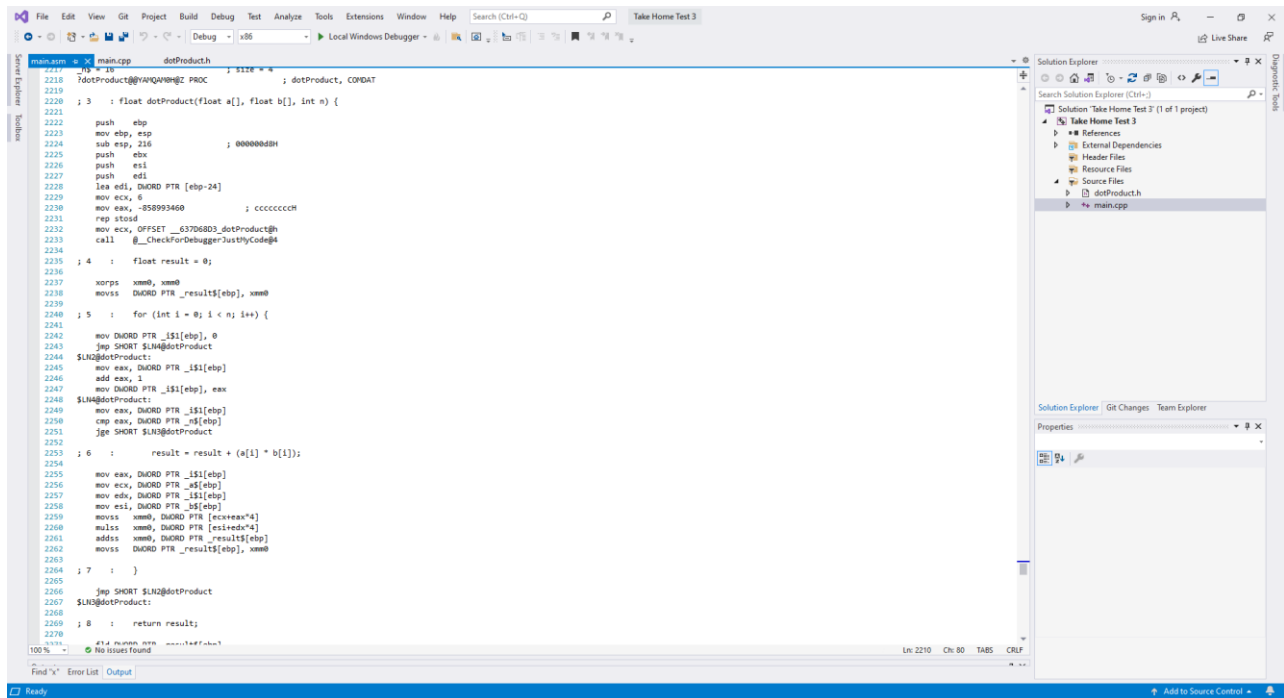


Figure 11: Compiler generated assembly code

Figure 12: Compiler generated assembly code with Figure 11: Compiler generated assembly code

In lines 2259 to 2262 we can see the compiler makes use of vector instructions which means that automatic parallelization and vectorization was the reason why the dotProduct() function had lower execution time per vector size in figure 10 compared to figure 6. This use of vector instructions by the compiler lead to the improved the performance in the dotProduct() function.

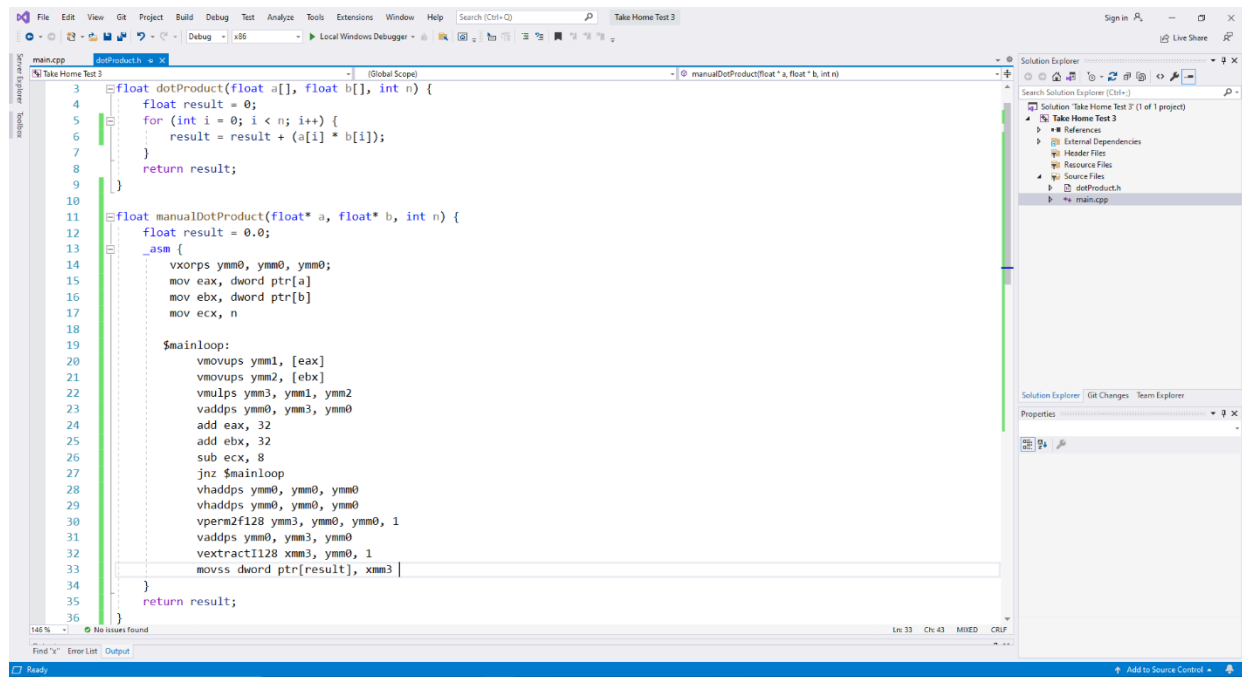
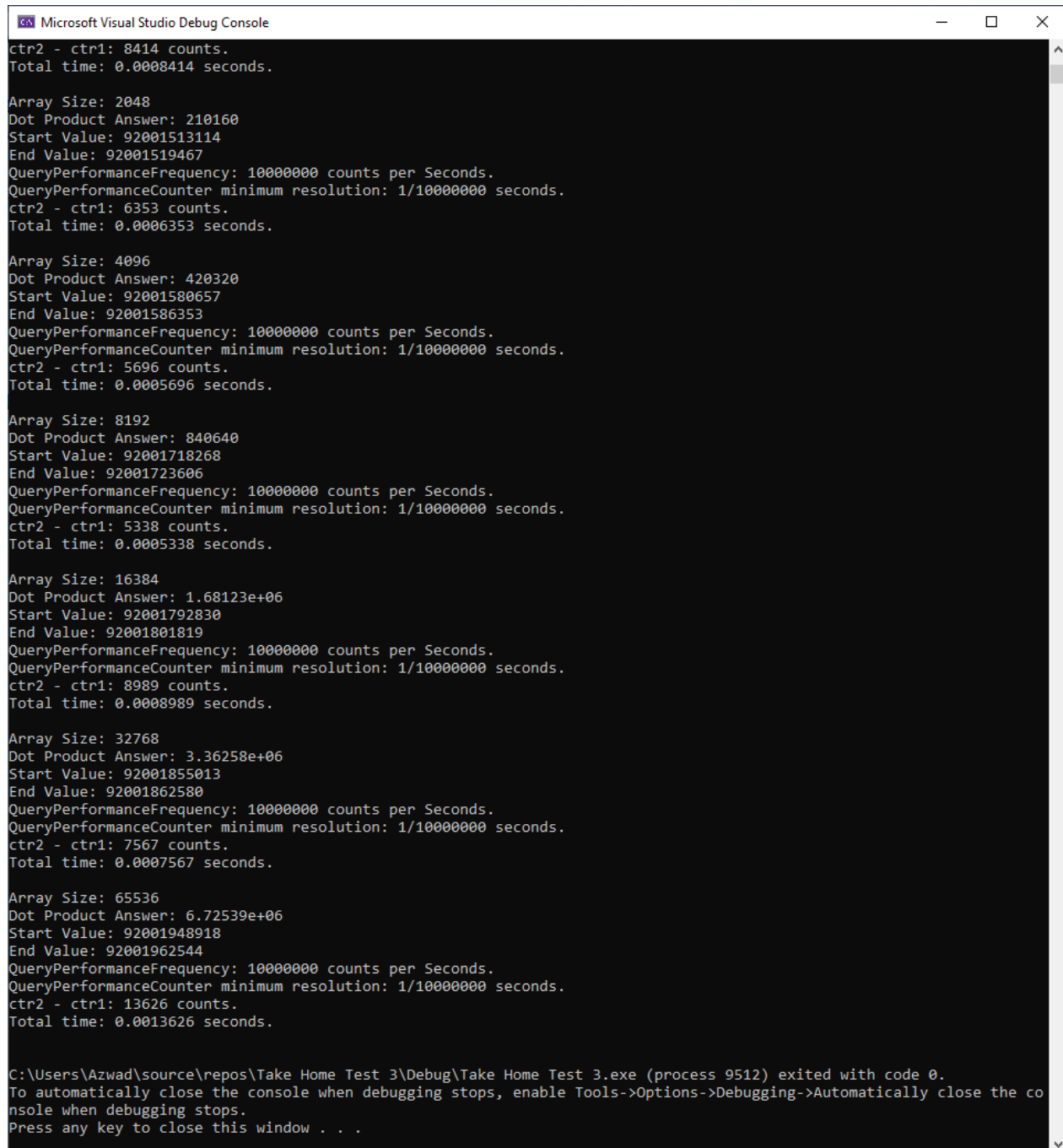
manualDotProduct

Figure 13: manualDotProduct function

This function contains the compiler generated code by automatic parallelization and vectorization enabled. In addition, the function has some changes made to it so that it would work properly and did not produce an error.



```

Microsoft Visual Studio Debug Console
ctr2 - ctr1: 8414 counts.
Total time: 0.0008414 seconds.

Array Size: 2048
Dot Product Answer: 210160
Start Value: 92001513114
End Value: 92001519467
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 6353 counts.
Total time: 0.0006353 seconds.

Array Size: 4096
Dot Product Answer: 420320
Start Value: 92001580657
End Value: 92001586353
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 5696 counts.
Total time: 0.0005696 seconds.

Array Size: 8192
Dot Product Answer: 840640
Start Value: 92001718268
End Value: 92001723606
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 5338 counts.
Total time: 0.0005338 seconds.

Array Size: 16384
Dot Product Answer: 1.68123e+06
Start Value: 92001792830
End Value: 92001801819
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 8989 counts.
Total time: 0.0008989 seconds.

Array Size: 32768
Dot Product Answer: 3.36258e+06
Start Value: 92001855013
End Value: 92001862580
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 7567 counts.
Total time: 0.0007567 seconds.

Array Size: 65536
Dot Product Answer: 6.72539e+06
Start Value: 92001948918
End Value: 92001962544
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 13626 counts.
Total time: 0.0013626 seconds.

C:\Users\Azwad\source\repos\Take Home Test 3\Debug\Take Home Test 3.exe (process 9512) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the co
nsole when debugging stops.
Press any key to close this window . . .

```

Figure 14: manualDotProduct function output

Vector Size	Execution Time
16	0.0004429
32	0.000407
64	0.0004497
128	0.0003229
256	0.0009981
512	0.0005392
1024	0.0008414
2048	0.0006353
4096	0.0005696
8192	0.0005338
16384	0.0008989
32768	0.0007567
65536	0.0013626

Figure 15: manualDotProduct function results in a table

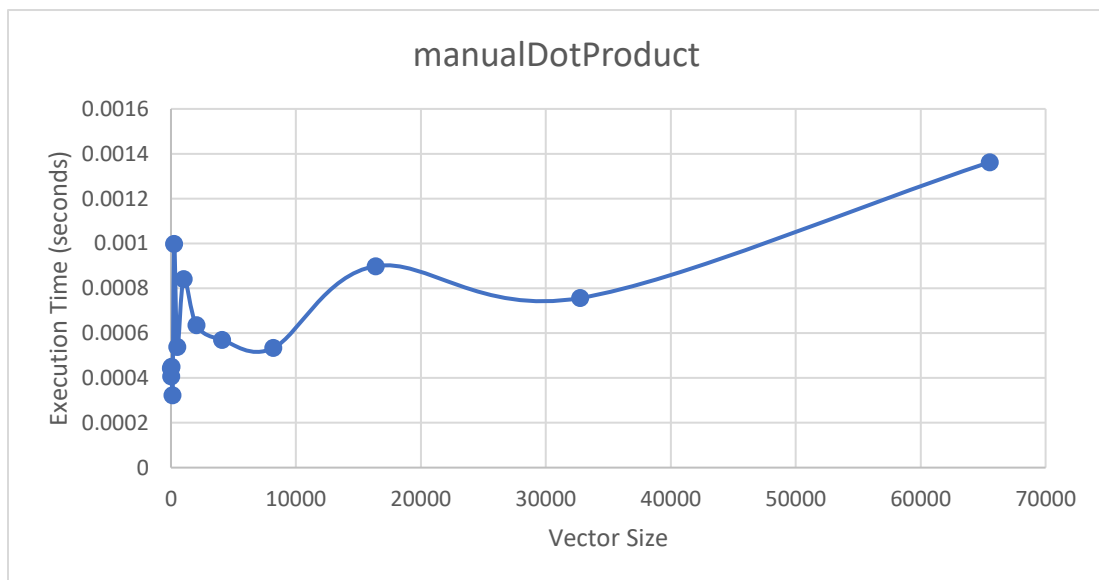


Figure 16: manualDotProduct function results graphed

The screenshot shows the Visual Studio IDE with the assembly code for the `manualDotProduct` function. The code is displayed in the main editor window, and the Solution Explorer on the right shows the project structure.

```

mainasm:  main.cpp  dotProduct.h
2089  /manualDotProduct@YAHPAHNM@Z PROC      ; manualDotProduct, COHDA
2090
2091  ; 11 : float manualDotProduct(float* a, float* b, int n) {
2092
2093      push    ebp
2094      mov     ebp, esp
2095      sub     esp, 20h                ; 00000020h
2096      push    ebx
2097      push    esi
2098      push    edi
2099      lea     edi, DWORD PTR [ebp-16]
2100      mov     ecx, 4
2101      mov     eax, -858993460          ; ccccccccH
2102      rep     stosd
2103      mov     eax, DWORD PTR ___security_cookie
2104      xor     eax, ebp
2105      mov     DWORD PTR ___$ArrayPad$[ebp], eax
2106
2107  ; 12 : float result = 0.0;
2108
2109      xorps   xmm0, xmm0
2110      vmovss  DWORD PTR _results[ebp], xmm0
2111
2112  ; 13 : _asm {
2113  ; 14 :     xorps ymm0, ymm0;
2114
2115      xorps   ymm0, ymm0
2116
2117  ; 15 :     mov eax, dword ptr[a]
2118
2119      mov     eax, DWORD PTR _a$[ebp]
2120
2121  ; 16 :     mov ebx, dword ptr[b]
2122
2123      mov     ebx, DWORD PTR _b$[ebp]
2124
2125  ; 17 :     mov ecx, n
2126
2127      mov     ecx, DWORD PTR _n$[ebp]
2128  $$mainloop$3:
2129
2130  ; 18 :
2131  ; 19 :     $mainloop:
2132  ; 20 :     vmovups ymm1, [eax]
2133
2134      vmovups ymm1, YMMWORD PTR [eax]
2135
2136  ; 21 :     vmovups ymm2, [ebx]
2137
2138      vmovups ymm2, YMMWORD PTR [ebx]
2139
2140  ; 22 :     vmulps ymm3, ymm1, ymm2
2141
2142      vmulps  ymm3, ymm1, ymm2
2143
2144  ; 23 :     vaddps ymm0, ymm3, ymm0
2145
2146      vaddps  ymm0, ymm3, ymm0
2147
2148  ; 24 :     add eax, 32
2149
2150      add     eax, 32                ; 00000020h
2151
2152  ; 25 :     add ebx, 32
2153
2154      add     ebx, 32                ; 00000020h
2155
2156  ; 26 :     sub ecx, 8
2157
2158      sub     ecx, 8
2159
2160  ; 27 :     jnz $mainloop
2161
2162      jnz     SHORT $$mainloop$3
2163
2164  ; 28 :     vhaddps ymm0, ymm0, ymm0
2165
2166      vhaddps ymm0, ymm0, ymm0
2167
2168  ; 29 :     vhaddps ymm0, ymm0, ymm0
2169
2170      vhaddps ymm0, ymm0, ymm0
2171
2172  ; 30 :     vperm2f128 ymm3, ymm0, ymm0, 1
2173
2174      vperm2f128 ymm3, ymm0, ymm0, 1
2175
2176  ; 31 :     vaddps ymm0, ymm3, ymm0
2177
2178      vaddps  ymm0, ymm3, ymm0
2179
2180      ret
2181  }
2182
2183  ; 11 : }
2184
2185  manualDotProduct ENDP

```

Figure 17: compiler generated assembly code for manualDotProduct()

The screenshot shows the Visual Studio IDE with the assembly code for the `manualDotProduct` function. The code is displayed in the main editor window, and the Solution Explorer on the right shows the project structure.

```

mainasm:  main.cpp  dotProduct.h
2131  ; 19 :     $mainloop:
2132  ; 20 :     vmovups ymm1, [eax]
2133
2134      vmovups ymm1, YMMWORD PTR [eax]
2135
2136  ; 21 :     vmovups ymm2, [ebx]
2137
2138      vmovups ymm2, YMMWORD PTR [ebx]
2139
2140  ; 22 :     vmulps ymm3, ymm1, ymm2
2141
2142      vmulps  ymm3, ymm1, ymm2
2143
2144  ; 23 :     vaddps ymm0, ymm3, ymm0
2145
2146      vaddps  ymm0, ymm3, ymm0
2147
2148  ; 24 :     add eax, 32
2149
2150      add     eax, 32                ; 00000020h
2151
2152  ; 25 :     add ebx, 32
2153
2154      add     ebx, 32                ; 00000020h
2155
2156  ; 26 :     sub ecx, 8
2157
2158      sub     ecx, 8
2159
2160  ; 27 :     jnz $mainloop
2161
2162      jnz     SHORT $$mainloop$3
2163
2164  ; 28 :     vhaddps ymm0, ymm0, ymm0
2165
2166      vhaddps ymm0, ymm0, ymm0
2167
2168  ; 29 :     vhaddps ymm0, ymm0, ymm0
2169
2170      vhaddps ymm0, ymm0, ymm0
2171
2172  ; 30 :     vperm2f128 ymm3, ymm0, ymm0, 1
2173
2174      vperm2f128 ymm3, ymm0, ymm0, 1
2175
2176  ; 31 :     vaddps ymm0, ymm3, ymm0
2177
2178      vaddps  ymm0, ymm3, ymm0
2179
2180      ret
2181  }
2182
2183  ; 11 : }
2184
2185  manualDotProduct ENDP

```

Figure 18: compiler generated assembly code for manualDotProduct()

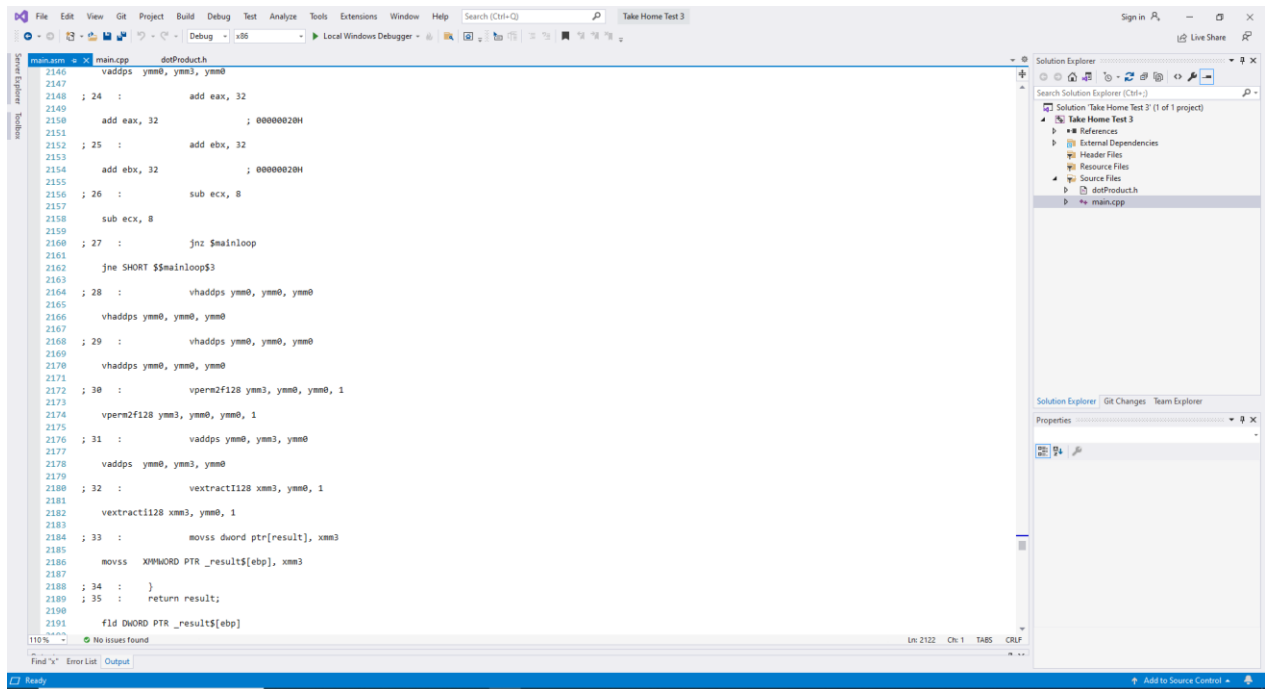


Figure 19: compiler generated assembly code for manualDotProduct()

The compiler generated assembly code shows that the assembly instructions that was written in the `manualDotProduct()` function was used by the compiler. This shows that the function was able to optimize the dot product computation.

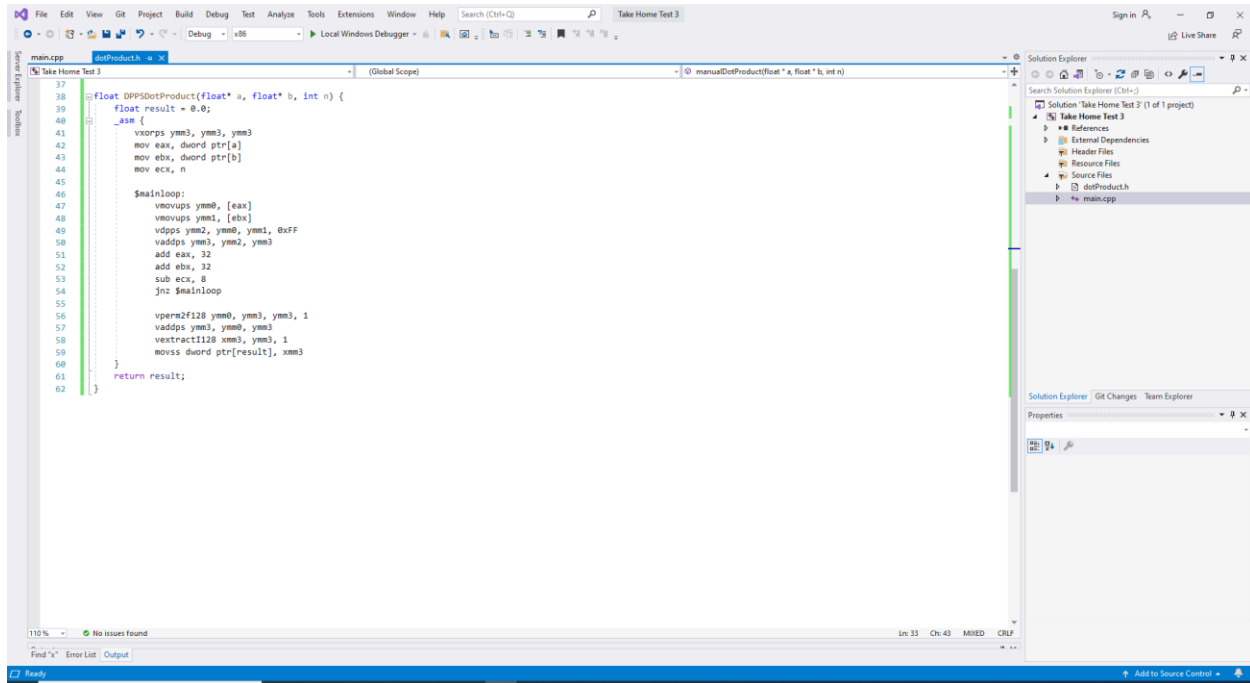
DPPSDotProduct

Figure 20: DPPSDotProduct function code in Visual Studio

The DPPSDotProduct function utilizes the DPPS vector instructions in order to boost performance. In the previous function, manualDotProduct we used the compilers generated code with automatic parallelization and vectorization and in this function DPPSDotProduct we are making the dotProduct function with DPPS vector instructions instead.

```

Microsoft Visual Studio Debug Console

ctr2 - ctr1: 3637 counts.
Total time: 0.0003637 seconds.

Array Size: 2048
Dot Product Answer: 210160
Start Value: 109107686951
End Value: 109107696129
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 9178 counts.
Total time: 0.0009178 seconds.

Array Size: 4096
Dot Product Answer: 420320
Start Value: 109107753294
End Value: 109107757231
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 3937 counts.
Total time: 0.0003937 seconds.

Array Size: 8192
Dot Product Answer: 840640
Start Value: 109107842034
End Value: 109107861751
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 19717 counts.
Total time: 0.0019717 seconds.

Array Size: 16384
Dot Product Answer: 1.68123e+06
Start Value: 109107965877
End Value: 109107973629
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 7752 counts.
Total time: 0.0007752 seconds.

Array Size: 32768
Dot Product Answer: 3.36258e+06
Start Value: 109108045953
End Value: 109108054433
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 8480 counts.
Total time: 0.000848 seconds.

Array Size: 65536
Dot Product Answer: 6.72539e+06
Start Value: 109108110985
End Value: 109108120928
QueryPerformanceFrequency: 1000000 counts per Seconds.
QueryPerformanceCounter minimum resolution: 1/1000000 seconds.
ctr2 - ctr1: 9943 counts.
Total time: 0.0009943 seconds.

C:\Users\Azwad\source\repos\Take Home Test 3\Debug\Take Home Test 3.exe (process 12940) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the co
nsole when debugging stops.
Press any key to close this window . . .

```

Figure 21: DPPSDotProduct function output.

Vector Size	Execution Time
16	0.0005732
32	0.0003594
64	0.0004087
128	0.0003121
256	0.0004261
512	0.0005491
1024	0.0003637
2048	0.0009178
4096	0.0003937
8192	0.0019717
16384	0.0007752
32768	0.000648
65536	0.0009943

Figure 22: DPPSDotProduct function results in a table

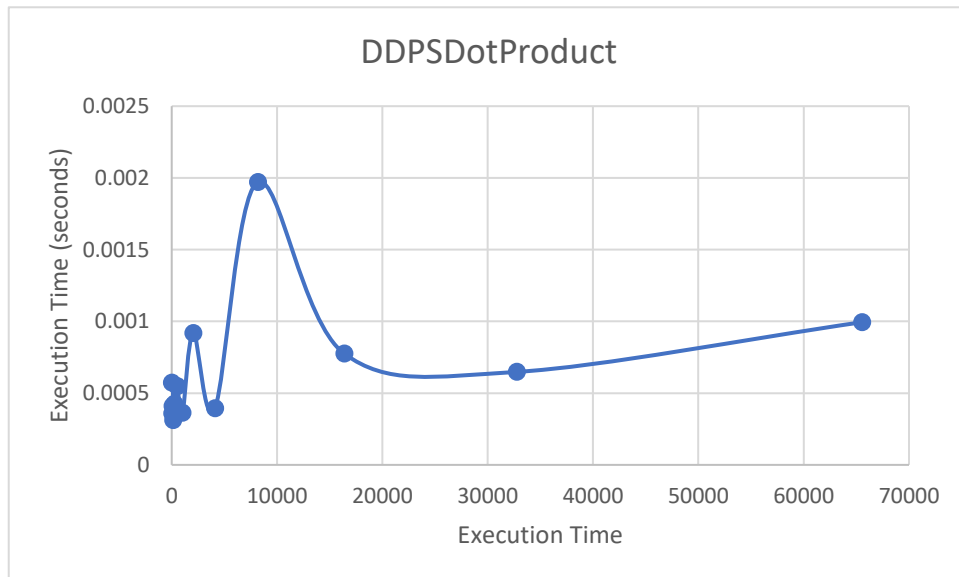


Figure 23: DPPSDotProduct function results in a graph

The results of the DPPSDotProduct are plotted in a graph. The DPPSDotProduct graph is unique and different from other graphs because towards the end of the graph the line increases at a much smaller rate than the previous functions graphs.

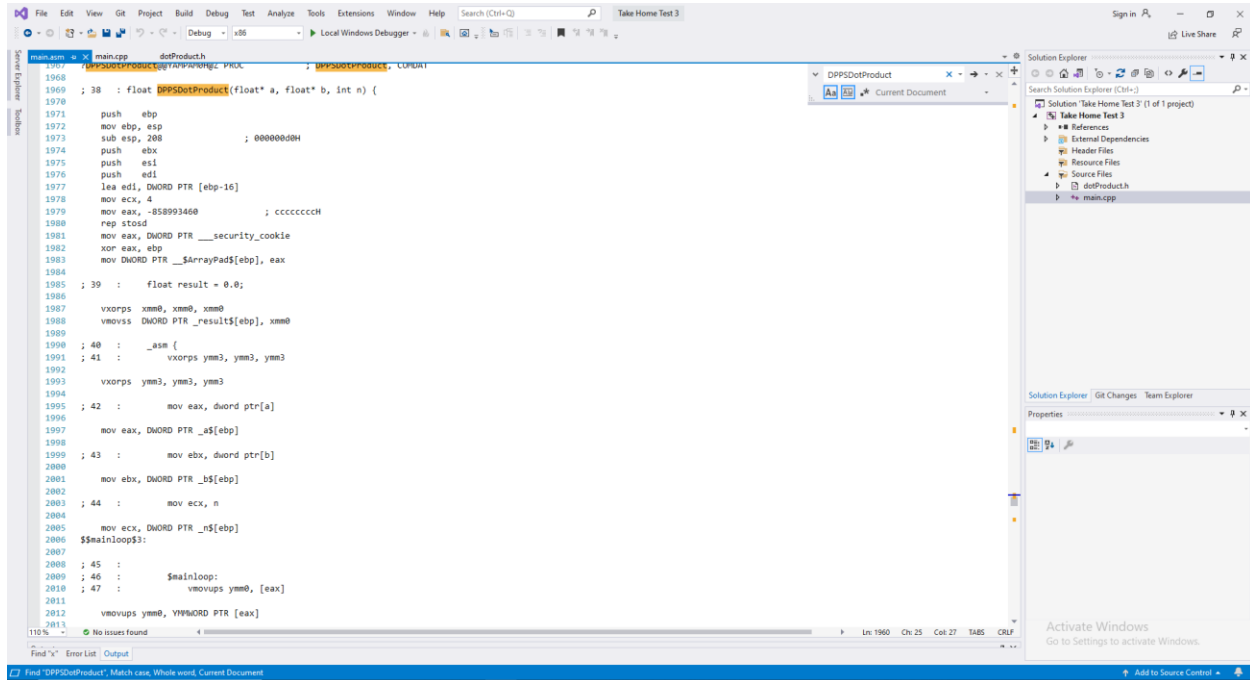
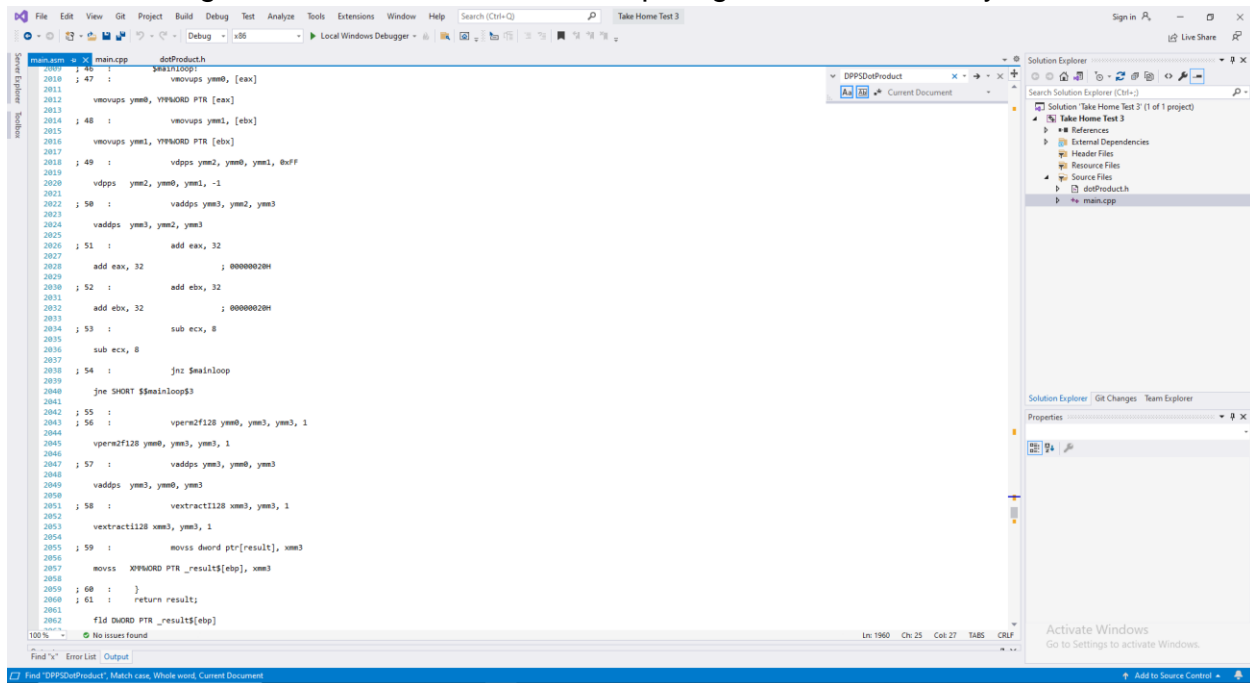


Figure 24: DPPSDotProduct compiler generated assembly code



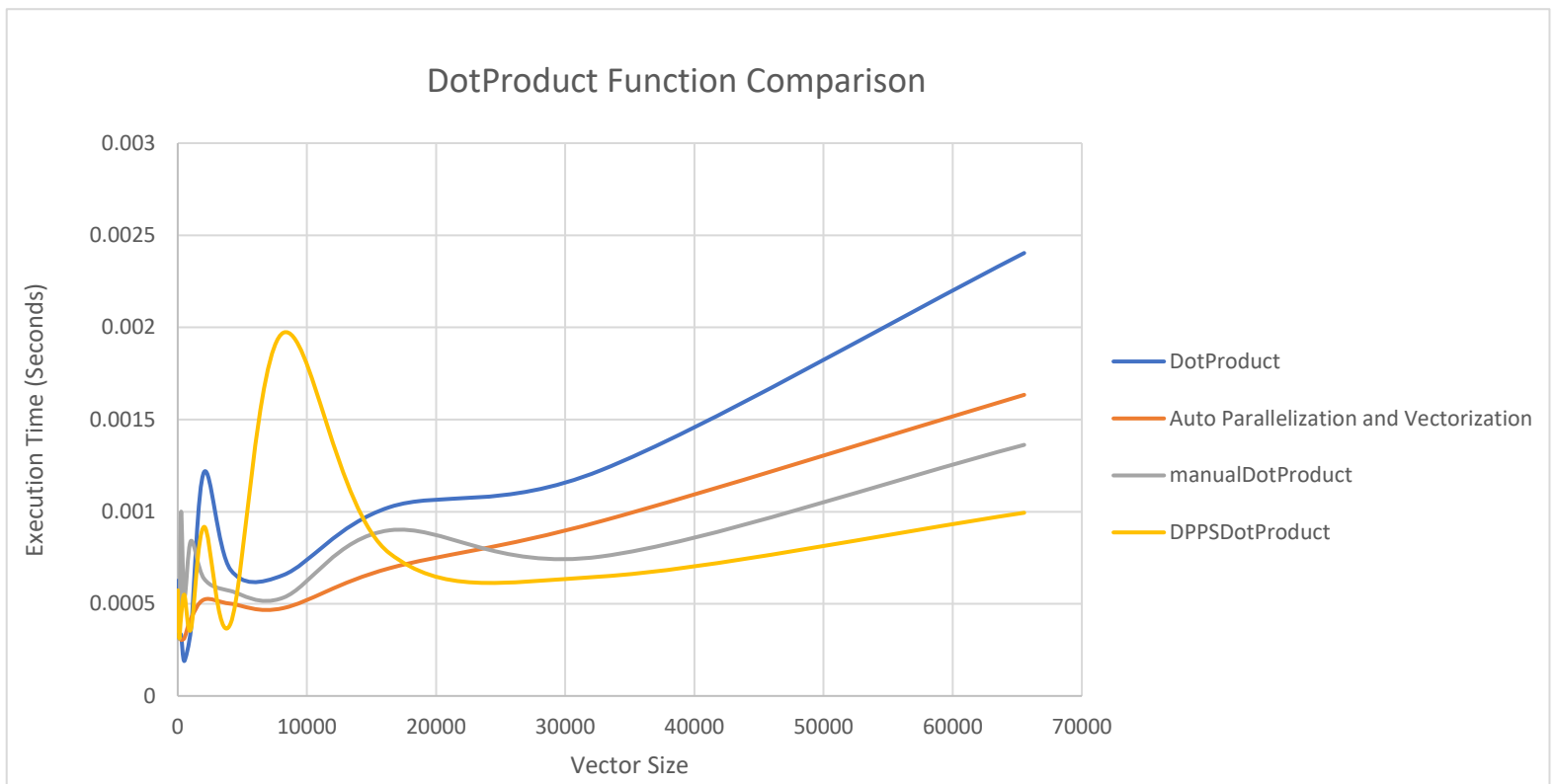
Comparison

Figure 26: This is the graph of that contains all of the previous functions and their results shown in a graph.

After plotting all of the dot product functions execution time, we can compare the different functions execution times. Clearly, we can see that the DPPSDotProduct function line uses the least execution time compared to the other functions towards the end. Next is the manualDotProduct function which has the second least execution time towards the end. Lastly, we have the automatic parallelization and vectorization dotProduct function and the normal dotProduct function taking up the most execution time towards the end. Therefore, we can see that towards the end the DPPS vector instructions become very efficient and are better at larger vector sizes. However, the DPPS vector instructions are not as efficient when the vector sizes are smaller.

Linux
CPU-X

The screenshot shows the CPU-X application window with the following details:

- Processor Section:**
 - Vendor: Intel
 - Code Name: Coffee Lake-R (Core i9)
 - Package: LGA 1151
 - Technology: 14 nm
 - Voltage: (empty)
 - Specification: Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
 - Family: 0x6
 - Model: 0xE
 - Temp.: 40.00°C
 - Ext. Family: 0x6
 - Ext. Model: 0x9E
 - Stepping: 13
 - Instructions: HT, MMX, SSE(1, 2, 3, 3S, 4.1, 4.2), AVX(1, 2), FMA(3), AES, CLMUL, RdRand, SGX, VT-x, x86-64
- Clocks Section:**
 - Core Speed: 4700 MHz
 - Multiplier: (empty)
 - Bus Speed: (empty)
 - Usage: 7.98 %
- Cache Section:**
 - L1 Data: 8 x 32 kB, 8-way
 - L1 Inst.: 8 x 32 kB, 8-way
 - Level 2: 8 x 256 kB, 4-way
 - Level 3: 16 MB, 16-way
- Bottom Section:**
 - Core #0 (dropdown)
 - Socket(s): 1
 - Core(s): 8
 - Thread(s): 16
 - Buttons: CPU-X, Version 4.1.0, Start daemon

Figure 27: CPU specifications on CPU-X Linux

The processor is the Intel Core i9-9900k which supports the instructions MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, AES, AVX, AVX2, FMA3. This processor does not support the AVX512 vector instruction test. Furthermore, for this take-home test we will be using the AVX2 vector instruction set, which is supported by the processor.

Main.cpp

```

1 //main.cpp
2 #include <iostream>
3 #include <math.h>
4 #include <chrono>
5 #include "dotProduct.h"
6 using namespace std;
7
8 int main() {
9     int n = 8;
10    for (int i = 4; i <= 16; i++) {
11        m = intpow(2, i);
12
13        float* a = new float[m];
14        float* b = new float[m];
15        float* result = new float[m];
16
17        for (int j = 0; j < m; j++) {
18            a[j] = 10.12;
19            b[j] = 10.14;
20        }
21
22        cout << "Array Size : " << m << endl;
23        auto start = chrono::high_resolution_clock::now();
24        dotProduct(a, b, m, result); // Function can be dotProduct, ManualDotProduct or DPPSDotProduct
25        auto end = chrono::high_resolution_clock::now();
26        auto diff = end - start;
27        cout << "Total time : " << diff.count() << " seconds." << endl;
28    }
29    return 0;
30 }

```

Figure 28: main.cpp code in Linux

This is the main program file that will measure the time that functions take and also report the other statistics of the function. Some of the statistics are the size of the array, start value, end value. This file right now calls the function dotProduct() in line 24, but this will be changed to the other functions to test out the compiler and DPPS version of the dotProduct function.

Dot Product

```
#pragma once

void dotProduct(float a[], float b[], int n, float* result) {
    float answer = 0;
    for (int i = 0; i < n; i++) {
        answer = answer + (a[i] * b[i]);
    }
    result[0] = answer;
}
```

Figure 29: dotProduct function code

```
azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$ g++ main.cpp -o dotProduct
azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$ ./dotProduct
Array Size : 16
Dot Product Answer : 1641.87
Total time: 9.596e-06 seconds.

Array Size : 32
Dot Product Answer : 3283.74
Total time: 1.215e-06 seconds.

Array Size : 64
Dot Product Answer : 6567.47
Total time: 1.209e-06 seconds.

Array Size : 128
Dot Product Answer : 13135
Total time: 1.426e-06 seconds.

Array Size : 256
Dot Product Answer : 26270
Total time: 2.128e-06 seconds.

Array Size : 512
Dot Product Answer : 52540
Total time: 1.952e-06 seconds.

Array Size : 1024
Dot Product Answer : 105080
Total time: 3.067e-06 seconds.

Array Size : 2048
Dot Product Answer : 210154
Total time: 5.601e-06 seconds.

Array Size : 4096
Dot Product Answer : 420322
Total time: 8.957e-06 seconds.

Array Size : 8192
Dot Product Answer : 840674
Total time: 1.7076e-05 seconds.

Array Size : 16384
Dot Product Answer : 1.68138e+06
Total time: 3.4055e-05 seconds.

Array Size : 32768
Dot Product Answer : 3.36124e+06
Total time: 6.5367e-05 seconds.

Array Size : 65536
Dot Product Answer : 6.71996e+06
Total time: 0.000128884 seconds.

azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$
```

Figure 30: dotProduct function outputs in Linux terminal

Vector Size	Execution Time
16	9.60E-06
32	1.22E-06
64	1.21E-06
128	1.43E-06
256	2.13E-06
512	1.95E-06
1024	3.07E-06
2048	5.60E-06
4096	8.96E-06
8192	1.71E-05
16384	3.41E-05
32768	6.54E-05
65536	0.000128884

Figure 31: dotProducts results in a table

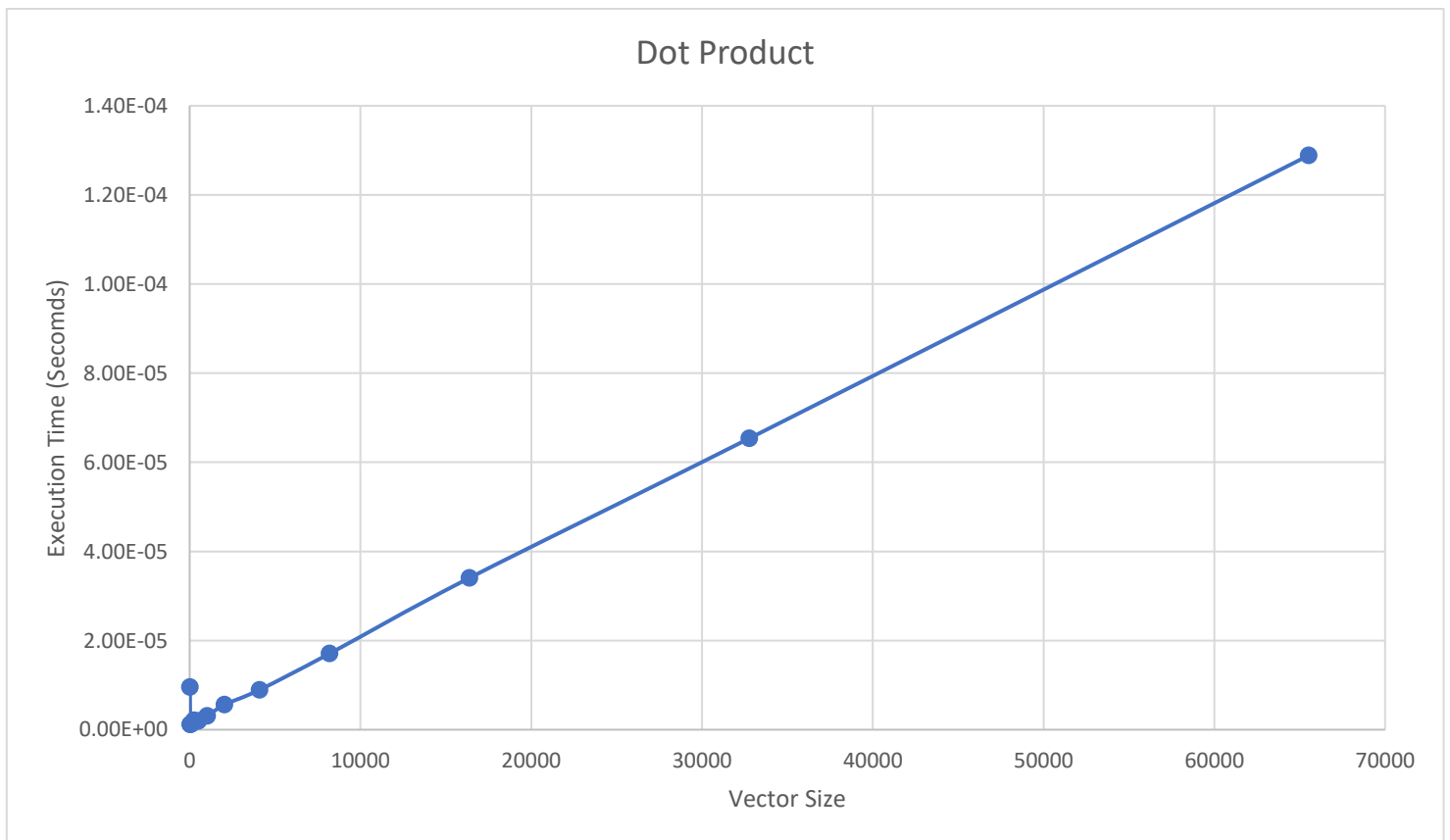
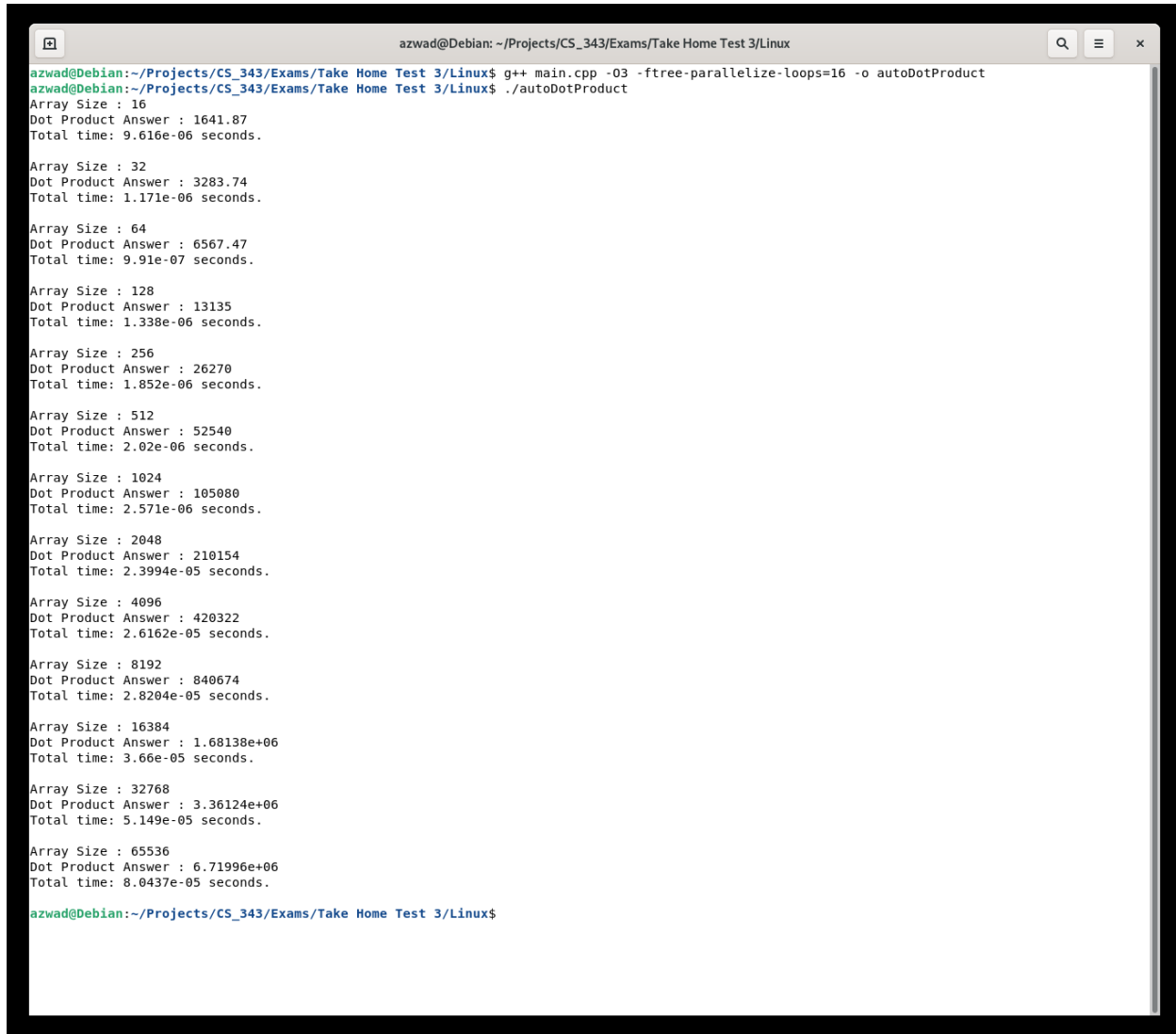


Figure 31: dotProduct function results in a graph

The graph of the dotProduct function shows a somewhat like a linear line, which shows there is not much efficiency here.

Dot Product with Automatic Parallelization and Automatic VectorizationA terminal window titled "azwad@Debian: ~/Projects/CS_343/Exams/Take Home Test 3/Linux" showing the execution of a C++ program. The program calculates dot products for array sizes ranging from 16 to 65536. For each size, it outputs the array size, the dot product answer, and the total time in seconds. The times generally increase with array size, with a noticeable jump at 32768 and 65536. The terminal output is as follows:

```
azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$ g++ main.cpp -O3 -ftree-parallelize-loops=16 -o autoDotProduct
azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$ ./autoDotProduct
Array Size : 16
Dot Product Answer : 1641.87
Total time: 9.616e-06 seconds.

Array Size : 32
Dot Product Answer : 3283.74
Total time: 1.171e-06 seconds.

Array Size : 64
Dot Product Answer : 6567.47
Total time: 9.91e-07 seconds.

Array Size : 128
Dot Product Answer : 13135
Total time: 1.338e-06 seconds.

Array Size : 256
Dot Product Answer : 26270
Total time: 1.852e-06 seconds.

Array Size : 512
Dot Product Answer : 52540
Total time: 2.02e-06 seconds.

Array Size : 1024
Dot Product Answer : 105080
Total time: 2.571e-06 seconds.

Array Size : 2048
Dot Product Answer : 210154
Total time: 2.3994e-05 seconds.

Array Size : 4096
Dot Product Answer : 420322
Total time: 2.6162e-05 seconds.

Array Size : 8192
Dot Product Answer : 840674
Total time: 2.8204e-05 seconds.

Array Size : 16384
Dot Product Answer : 1.68138e+06
Total time: 3.66e-05 seconds.

Array Size : 32768
Dot Product Answer : 3.36124e+06
Total time: 5.149e-05 seconds.

Array Size : 65536
Dot Product Answer : 6.71996e+06
Total time: 8.0437e-05 seconds.

azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$
```

Figure 33: dotProduct function output in terminal

The terminal shows that we use the flags `-O3 -ftree-parallelize-loops=16`, this enables automatic parallelization and vectorization.

Vector Size	Execution Time
16	9.62E-06
32	1.17E-06
64	9.91E-07
128	1.34E-06
256	1.85E-06
512	2.02E-06
1024	2.57E-06
2048	2.40E-05
4096	2.62E-05
8192	2.82E-05
16384	3.66E-05
32768	5.15E-05
65536	8.04E-05

Figure 34: dotProduct function results in a table

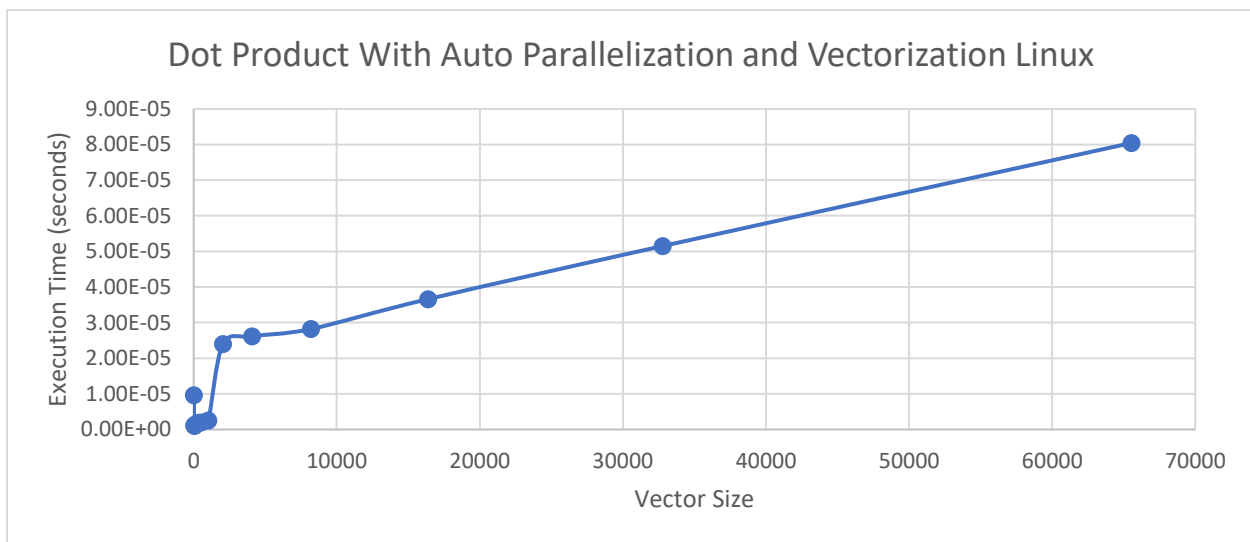


Figure 36: dotProduct function results in a graph

The graph generally shows that as the vector size increases the execution time increases. The dotProduct function without automatic parallelization and vectorization shows the graph as a linear line but the dotProduct function with automatic parallelization and vectorization shows the graph is somewhat a linear line but doesn't increase as fast. Therefore, it is highly likely that the automatic parallelization and vectorization lead to the improvement in efficiency.

```

57     .globl  _Z10dotProductPfS_is_
58     .type   _Z10dotProductPfS_is_, @function
59     _Z10dotProductPfS_is_:
60     .LFB2073:
61         .cfi_startproc
62         pushq   %rbp
63         .cfi_def_cfa_offset 16
64         .cfi_offset 6, -16
65         movq    %rsp, %rbp
66         .cfi_def_cfa_register 6
67         movq    %rdi, -24(%rbp)
68         movq    %rsi, -32(%rbp)
69         movl    %edx, -36(%rbp)
70         movq    %rcx, -48(%rbp)
71         pxor    %xmm0, %xmm0
72         movss   %xmm0, -4(%rbp)
73         movl    $0, -8(%rbp)
74     .L6:
75         movl    -8(%rbp), %eax
76         cmpl    -36(%rbp), %eax
77         jge     .L5
78         movl    -8(%rbp), %eax
79         cltq
80         leaq    0(,%rax,4), %rdx
81         movq    -24(%rbp), %rax
82         addq    %rdx, %rax
83         movss   (%rax), %xmm1
84         movl    -8(%rbp), %eax
85         cltq
86         leaq    0(,%rax,4), %rdx
87         movq    -32(%rbp), %rax
88         addq    %rdx, %rax
89         movss   (%rax), %xmm0
90         mulss   %xmm1, %xmm0
91         movss   -4(%rbp), %xmm1
92         addss   %xmm1, %xmm0
93         movss   %xmm0, -4(%rbp)
94         addl    $1, -8(%rbp)

```

Figure 37: dotProduct function without automatic parallelization and vectorization

```

18  .globl _Z10dotProductPfs_is_
19  .type _Z10dotProductPfs_is_, @function
20  _Z10dotProductPfs_is_:
21  .LFB2109:
22      .cfi_startproc
23      movl    %edx, %r8d
24      testl   %edx, %edx
25      jle     .L9
26      leal    -1(%rdx), %eax
27      cmpl    $2, %eax
28      jbe     .L10
29      shrl    $2, %edx
30      xorl    %eax, %eax
31      pxor    %xmm1, %xmm1
32      salq    $4, %rdx
33      .p2align 4,,10
34      .p2align 3
35  .L7:
36      movups  (%rdi,%rax), %xmm0
37      movups  (%rsi,%rax), %xmm3
38      addq    $16, %rax
39      mulps   %xmm3, %xmm0
40      addss   %xmm0, %xmm1
41      movaps  %xmm0, %xmm2
42      shufps  $85, %xmm0, %xmm2
43      addss   %xmm2, %xmm1
44      movaps  %xmm0, %xmm2
45      unpckhps %xmm0, %xmm2
46      shufps  $255, %xmm0, %xmm0
47      addss   %xmm2, %xmm1
48      addss   %xmm0, %xmm1
49      cmpq    %rdx, %rax
50      jne     .L7
51      movl    %r8d, %eax
52      andl    $-4, %eax

```

Figure 37: dotProduct function without automatic parallelization and vectorization

At line 36 to 46 we can see that the compiler does make use of the vector instructions. This proves that because of the automatic parallelization and vectorization was definitely used and is contributing to the improvement in efficiency for the dotProduct function.

manualDotProduct

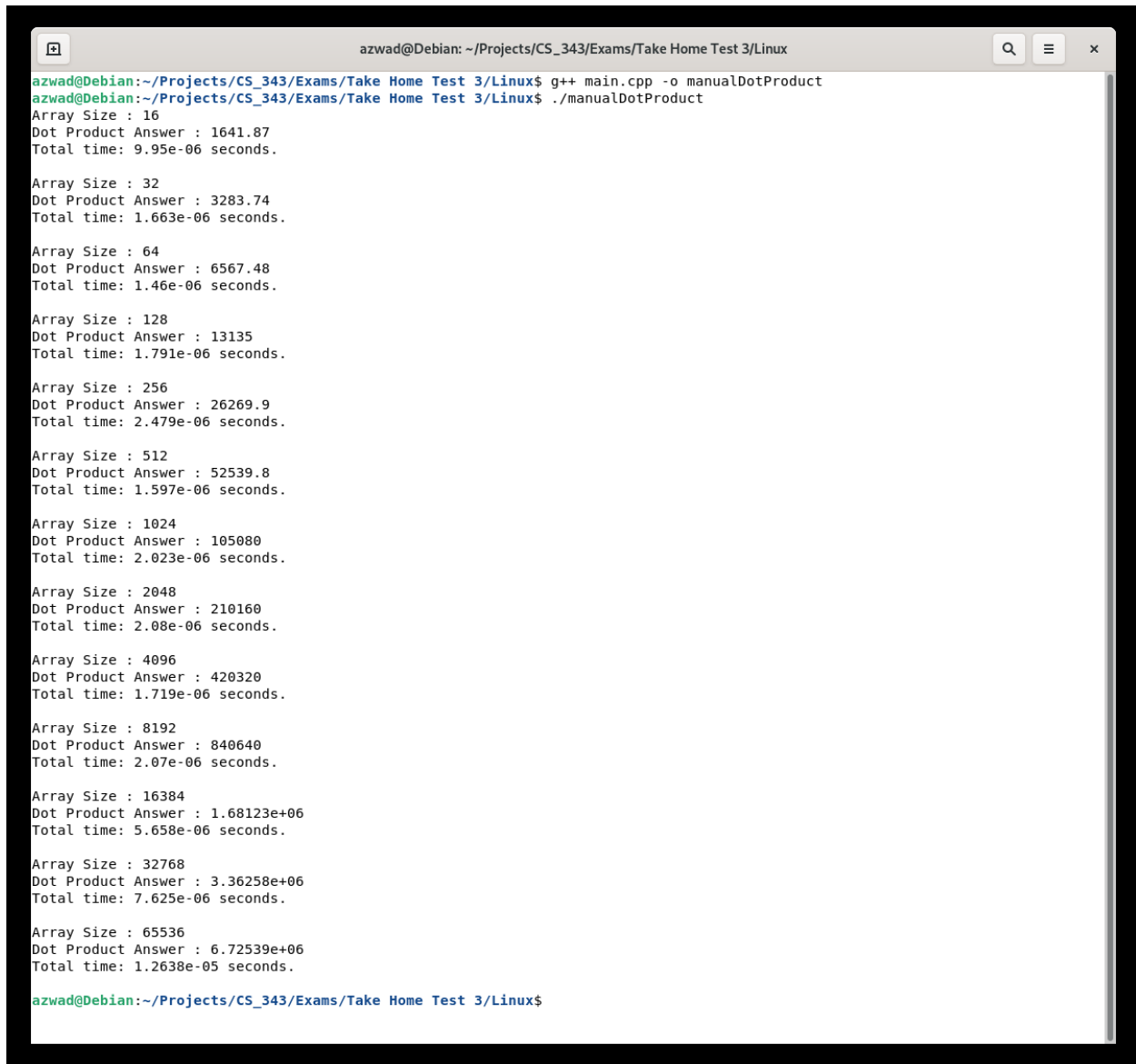
```

void manualDotProduct(float* a, float* b, int n, float* result) {
    asm [
        // arguments are stored in RDI, RSI, RDX, RCX (This is done automatically by Linux)
        "vpxor %ymm0, %ymm0, %ymm0\n"
        "vpxor %ymm3, %ymm3, %ymm3\n"
        ".mainloop:\n"
        "vmovups 0x0(%rdi), %ymm1\n"
        "vmovups 0x0(%rsi), %ymm2\n"
        "vmulps %ymm1, %ymm2, %ymm3\n"
        "vaddps %ymm0, %ymm3, %ymm0\n"
        "add $32, %rdi\n"
        "add $32, %rsi\n"
        "sub $8, %rdx\n"
        "jnz .mainloop\n" |
        "vhaddps %ymm0, %ymm0, %ymm0\n"
        "vhaddps %ymm0, %ymm0, %ymm0\n"
        "vhaddps %ymm0, %ymm0, %ymm0\n"
        "vmovups %ymm0, (%rcx)\n"
    ];
}

```

Figure 37: manualDotProduct function

This function contains the compiler generated code by automatic parallelization and vectorization enabled. In addition, the function has some changes made to it so that it would work properly and did not produce an error.



```
azwad@Debian: ~/Projects/CS_343/Exams/Take Home Test 3/Linux
azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$ g++ main.cpp -o manualDotProduct
azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$ ./manualDotProduct
Array Size : 16
Dot Product Answer : 1641.87
Total time: 9.95e-06 seconds.

Array Size : 32
Dot Product Answer : 3283.74
Total time: 1.663e-06 seconds.

Array Size : 64
Dot Product Answer : 6567.48
Total time: 1.46e-06 seconds.

Array Size : 128
Dot Product Answer : 13135
Total time: 1.791e-06 seconds.

Array Size : 256
Dot Product Answer : 26269.9
Total time: 2.479e-06 seconds.

Array Size : 512
Dot Product Answer : 52539.8
Total time: 1.597e-06 seconds.

Array Size : 1024
Dot Product Answer : 105080
Total time: 2.023e-06 seconds.

Array Size : 2048
Dot Product Answer : 210160
Total time: 2.08e-06 seconds.

Array Size : 4096
Dot Product Answer : 420320
Total time: 1.719e-06 seconds.

Array Size : 8192
Dot Product Answer : 840640
Total time: 2.07e-06 seconds.

Array Size : 16384
Dot Product Answer : 1.68123e+06
Total time: 5.658e-06 seconds.

Array Size : 32768
Dot Product Answer : 3.36258e+06
Total time: 7.625e-06 seconds.

Array Size : 65536
Dot Product Answer : 6.72539e+06
Total time: 1.2638e-05 seconds.

azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$
```

Figure 38: manualDotProduct output in terminal

Vector Size	Execution Time
16	9.95E-06
32	1.66E-06
64	1.46E-06
128	1.79E-06
256	2.48E-06
512	1.60E-06
1024	2.02E-06
2048	2.08E-06
4096	1.72E-06
8192	2.07E-06
16384	5.66E-06
32768	7.63E-06
65536	1.26E-05

Figure 39: manualDotProduct results in table

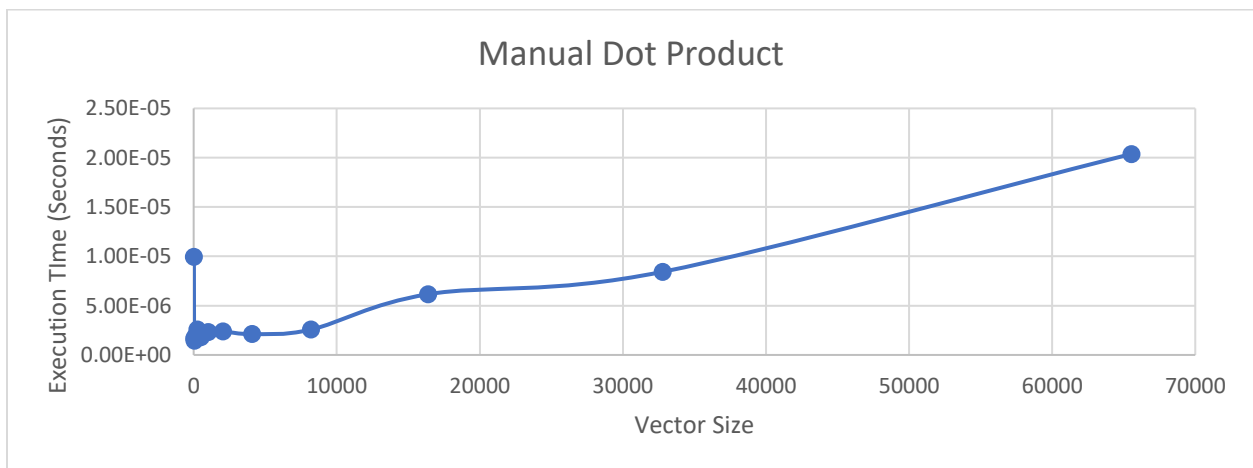


Figure 40: manualDotProduct results in graph

```

104     .cfi_endproc
105     .LFE2073:
106     .size    _Z10dotProductPfS_iS_, .- _Z10dotProductPfS_iS_
107     .globl   _Z16manualDotProductPfS_iS_
108     .type    _Z16manualDotProductPfS_iS_, @function
109     _Z16manualDotProductPfS_iS_:
110     .LFB2074:
111     .cfi_startproc
112     pushq    %rbp
113     .cfi_def_cfa_offset 16
114     .cfi_offset 6, -16
115     movq     %rsp, %rbp
116     .cfi_def_cfa_register 6
117     movq     %rdi, -8(%rbp)
118     movq     %rsi, -16(%rbp)
119     movl     %edx, -20(%rbp)
120     movq     %rcx, -32(%rbp)
121     #APP
122     # 12 "dotProduct.h" 1
123     |    vpxor %ymm0, %ymm0, %ymm0
124     vpxor %ymm3, %ymm3, %ymm3
125     .mainloop:
126     vmovups  0x0(%rdi), %ymm1
127     vmovups  0x0(%rsi), %ymm2
128     vmulps   %ymm1, %ymm2, %ymm3
129     vaddps   %ymm0, %ymm3, %ymm0
130     add $32, %rdi
131     add $32, %rsi
132     sub $8, %rdx
133     jnz .mainloop
134     vhaddps  %ymm0, %ymm0, %ymm0
135     vhaddps  %ymm0, %ymm0, %ymm0
136     vhaddps  %ymm0, %ymm0, %ymm0
137     vmovups  %ymm0, (%rcx)
138

```

Figure 41: manualDotProduct compiler generated code

Looking at the compiler generated code, it is understandable that the instructions that are written in the functions was utilized by the compiler. Therefore, the code worked properly and the dotProduct function is benefiting from this boost in efficiency.

DPPSDotProduct

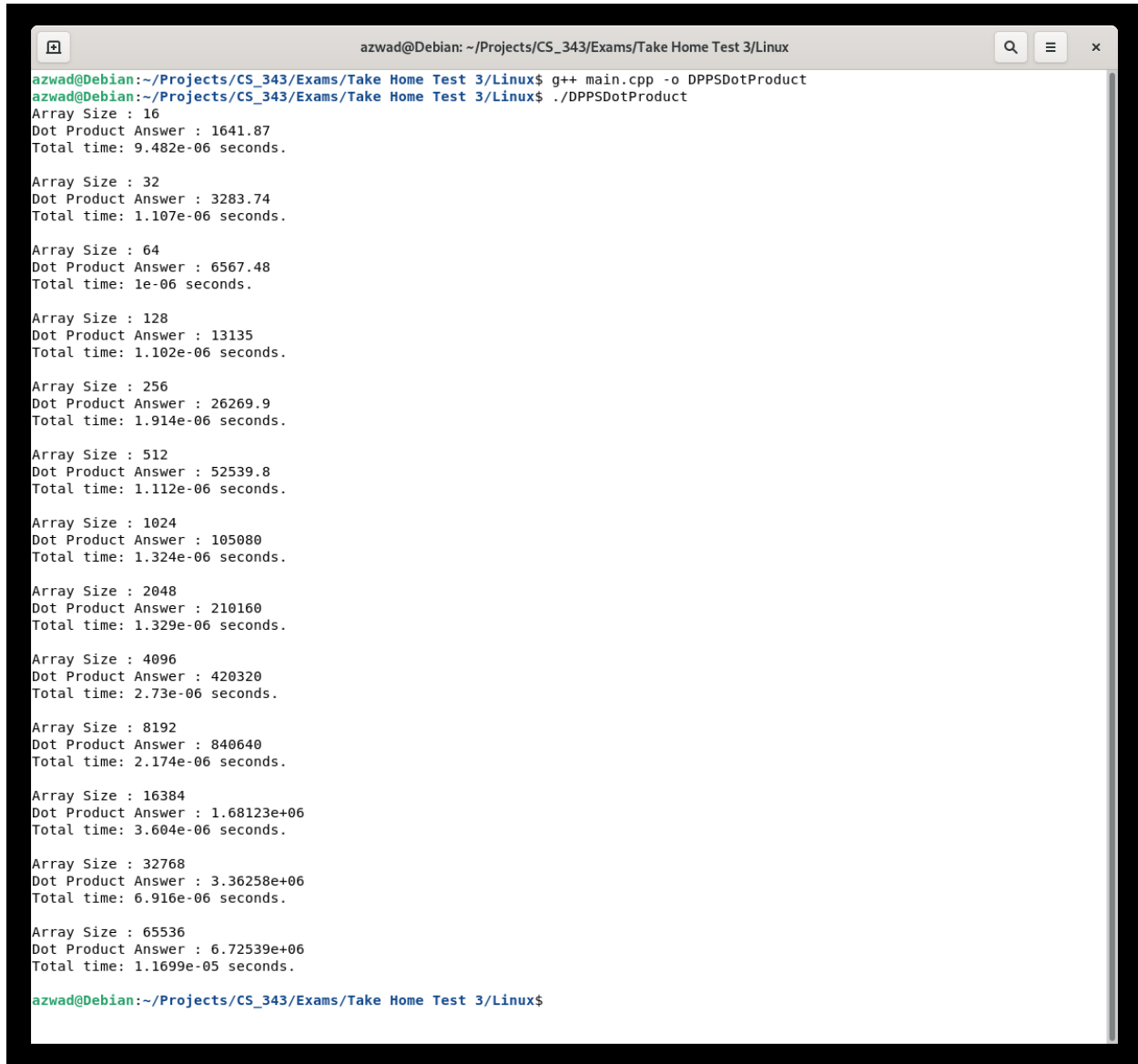
```

void DPPSdotProduct(float* a, float* b, int n, float* result) {
    asm {
        "vpxor %ymm3, %ymm3, %ymm3\n"
        ".main:\n"
        "vmovups 0x0(%rdi), %ymm1\n"
        "vmovups 0x0(%rsi), %ymm2\n"
        "vdpps $0xFF, %ymm1, %ymm2, %ymm0\n"
        "vaddps %ymm0, %ymm3, %ymm3\n"
        "add $32, %rdi\n"
        "add $32, %rsi\n"
        "sub $8, %rdx\n"
        "jnz .main\n"
        "vhaddps %ymm3, %ymm3, %ymm3\n"
        "vmovups %ymm3, (%rcx)\n"
    };
}

```

Figure 42: DPPSDotProduct function code

The DPPSDotProduct function utilizes the DPPS vector instructions in order to boost performance. In the previous function, manualDotProduct we used the compilers generated code with automatic parallelization and vectorization and in this function DPPSDotProduct we are making the dotProduct function with DPPS vector instructions instead.



```
azwad@Debian: ~/Projects/CS_343/Exams/Take Home Test 3/Linux
azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$ g++ main.cpp -o DPPSDotProduct
azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$ ./DPPSDotProduct
Array Size : 16
Dot Product Answer : 1641.87
Total time: 9.482e-06 seconds.

Array Size : 32
Dot Product Answer : 3283.74
Total time: 1.107e-06 seconds.

Array Size : 64
Dot Product Answer : 6567.48
Total time: 1e-06 seconds.

Array Size : 128
Dot Product Answer : 13135
Total time: 1.102e-06 seconds.

Array Size : 256
Dot Product Answer : 26269.9
Total time: 1.914e-06 seconds.

Array Size : 512
Dot Product Answer : 52539.8
Total time: 1.112e-06 seconds.

Array Size : 1024
Dot Product Answer : 105080
Total time: 1.324e-06 seconds.

Array Size : 2048
Dot Product Answer : 210160
Total time: 1.329e-06 seconds.

Array Size : 4096
Dot Product Answer : 420320
Total time: 2.73e-06 seconds.

Array Size : 8192
Dot Product Answer : 840640
Total time: 2.174e-06 seconds.

Array Size : 16384
Dot Product Answer : 1.68123e+06
Total time: 3.604e-06 seconds.

Array Size : 32768
Dot Product Answer : 3.36258e+06
Total time: 6.916e-06 seconds.

Array Size : 65536
Dot Product Answer : 6.72539e+06
Total time: 1.1699e-05 seconds.

azwad@Debian:~/Projects/CS_343/Exams/Take Home Test 3/Linux$
```

Figure 43: DPPSDotProduct function output

Vector Size	Execution Time
16	9.48E-06
32	1.11E-06
64	1.00E-06
128	1.10E-06
256	1.91E-06
512	1.11E-06
1024	1.32E-06
2048	1.33E-06
4096	2.73E-06
8192	2.17E-06
16384	3.60E-06
32768	6.92E-06
65536	1.17E-05

Figure 44: DPPSDotProduct function results in table

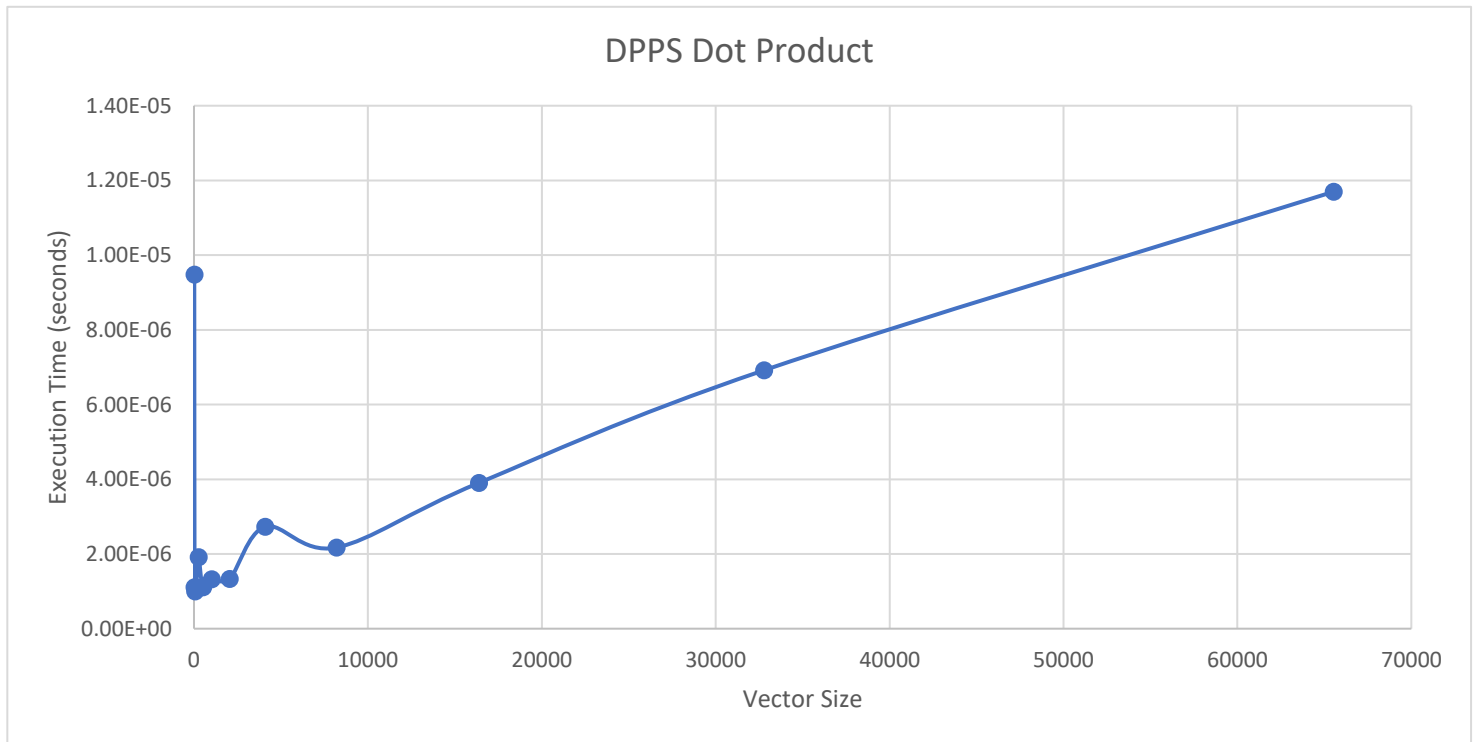


Figure 45: DPPSDotProduct function results in graph

The compiler assembled code shows the DPPS instructions vector instructions utilized by the function which shows that the DPPS instructions are being utilized and are helping the program to be more efficient.

```

.LFB2074:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register 6
    movq     %rdi, -8(%rbp)
    movq     %rsi, -16(%rbp)
    movl     %edx, -20(%rbp)
    movq     %rcx, -32(%rbp)
#APP
# 12 "dotProduct.h" 1
    vpxor %ymm0, %ymm0, %ymm0
    vpxor %ymm3, %ymm3, %ymm3
.mainloop:
    vmovups 0x0(%rdi), %ymm1
    vmovups 0x0(%rsi), %ymm2
    vmulps %ymm1, %ymm2, %ymm3
    vaddps %ymm0, %ymm3, %ymm0
    add $32, %rdi
    add $32, %rsi
    sub $8, %rdx
    jnz .mainloop
    vhaddps %ymm0, %ymm0, %ymm0
    vhaddps %ymm0, %ymm0, %ymm0
    vhaddps %ymm0, %ymm0, %ymm0
    vmovups %ymm0, (%rcx)

```

Figure 46: DPPSDotProduct function compiler generated code

This shows that the compiler actually utilizes the DPPS instructions and proves that the improvement is not due to randomness but to the increase efficiency of using DPPS instructions.

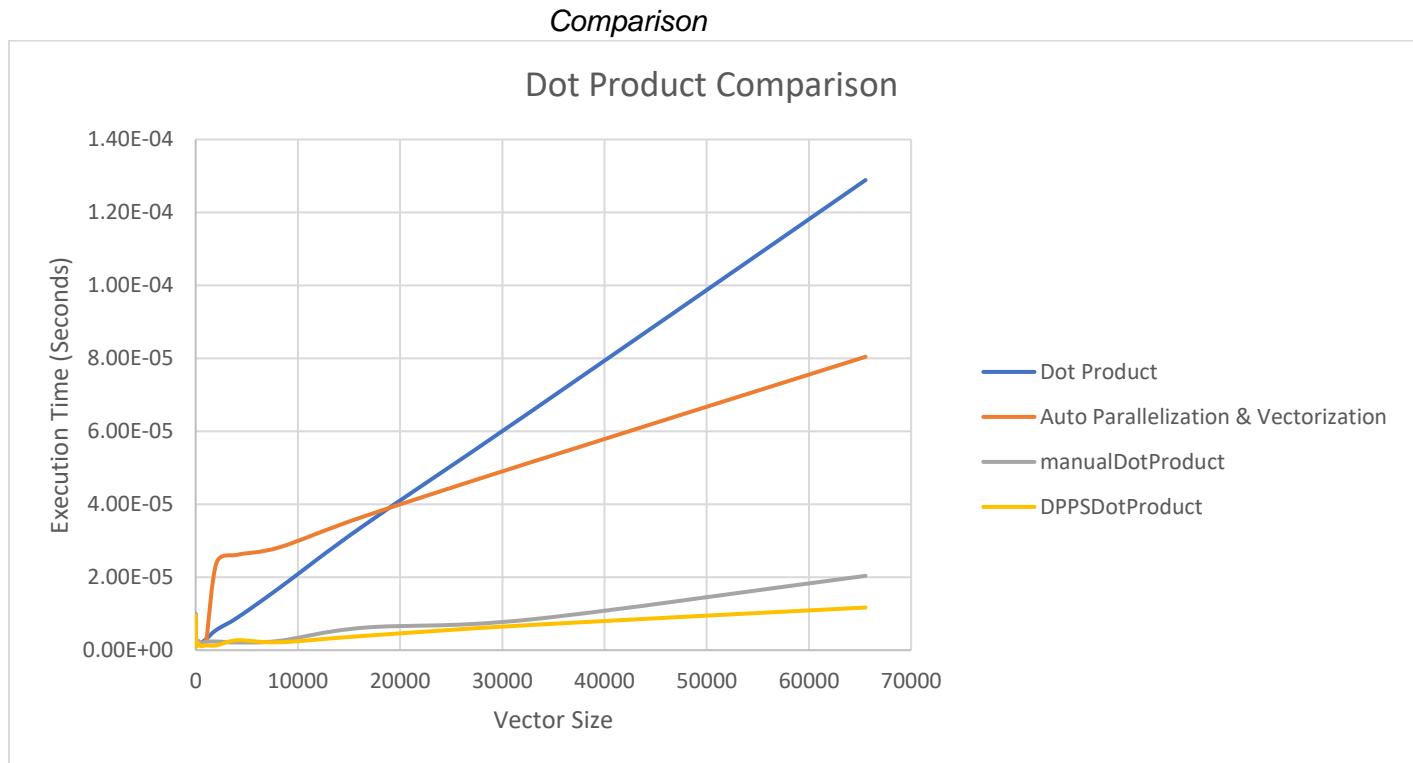


Figure 47: This is the graph of that contains all of the previous functions and their results shown in a graph.

After plotting all of the dot product functions execution time, we can compare the different functions execution times. Clearly, we can see that the DPPSDotProduct function line uses the least execution time compared to the other functions. Also we can see that the manualDotProduct function has the second least execution time. Lastly, we have the automatic parallelization and vectorization of the dotProduct function and the normal dotProduct function which takes the most execution time towards the end. Therefore, we can see that towards the end the DPPS vector instructions become very efficient and are better at larger vector sizes than the other functions.

Conclusion

The objective of this take-home test is to optimize the compiler generated code for a program that computes the dot product using vector instructions. In order to correctly optimize the program that computes dot product, we utilized the QueryPerformanceCounter function to measure execution time, and in order to confirm that the optimization of the assembly code led to decreases in execution time. The optimizations that will be used are the automatic parallelization and vectorization, the compiler generated code with vector instructions and the vector instructions DPPS to improve efficiency of the function. These optimizations will be run and recorded with their execution times which will then be listed all together on one graph to be analyzed. Then we will repeat the previous steps that we did in Visual Studio in Linux.