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

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

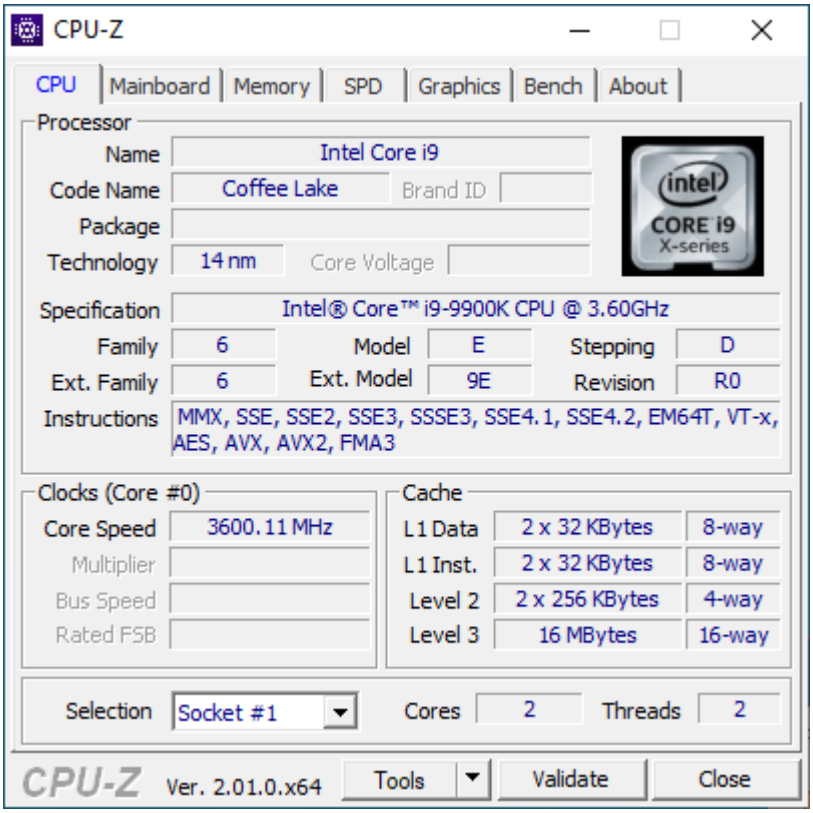


Figure 1: CPU-Z CPU information

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 he 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

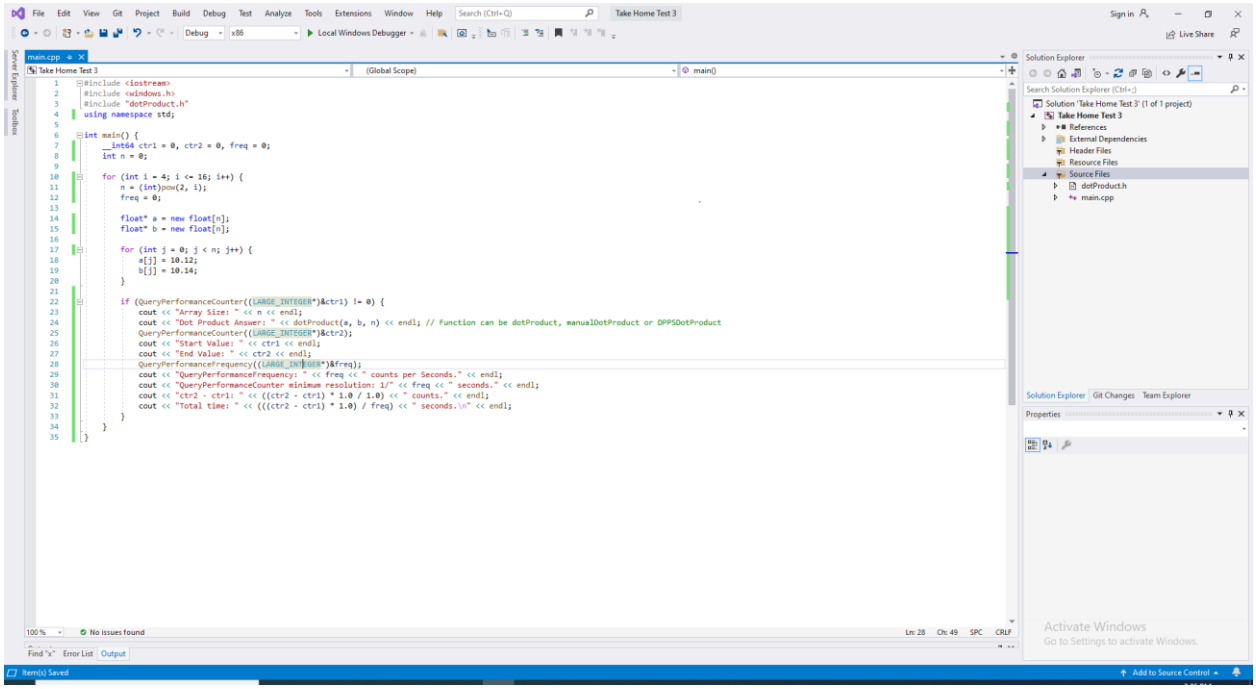


Figure 2: Main.cpp code shown in Visual Studio

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

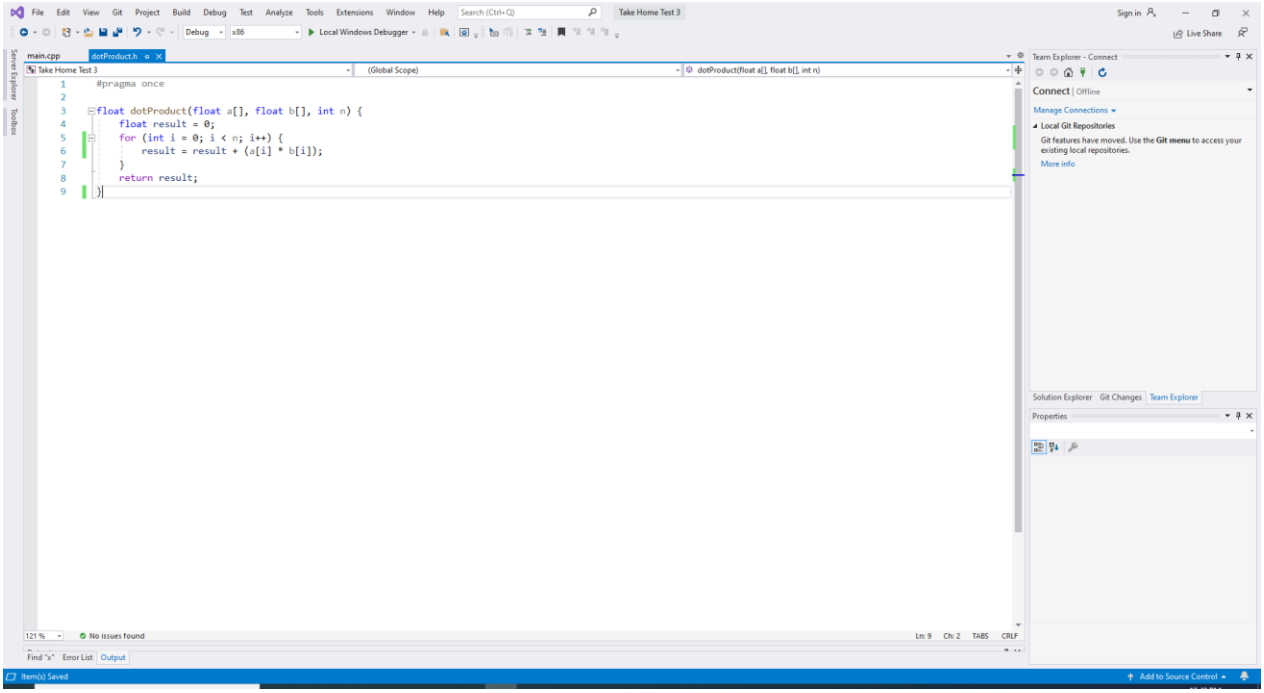


Figure 3: dotProduct function in the dotProduct.h file

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.

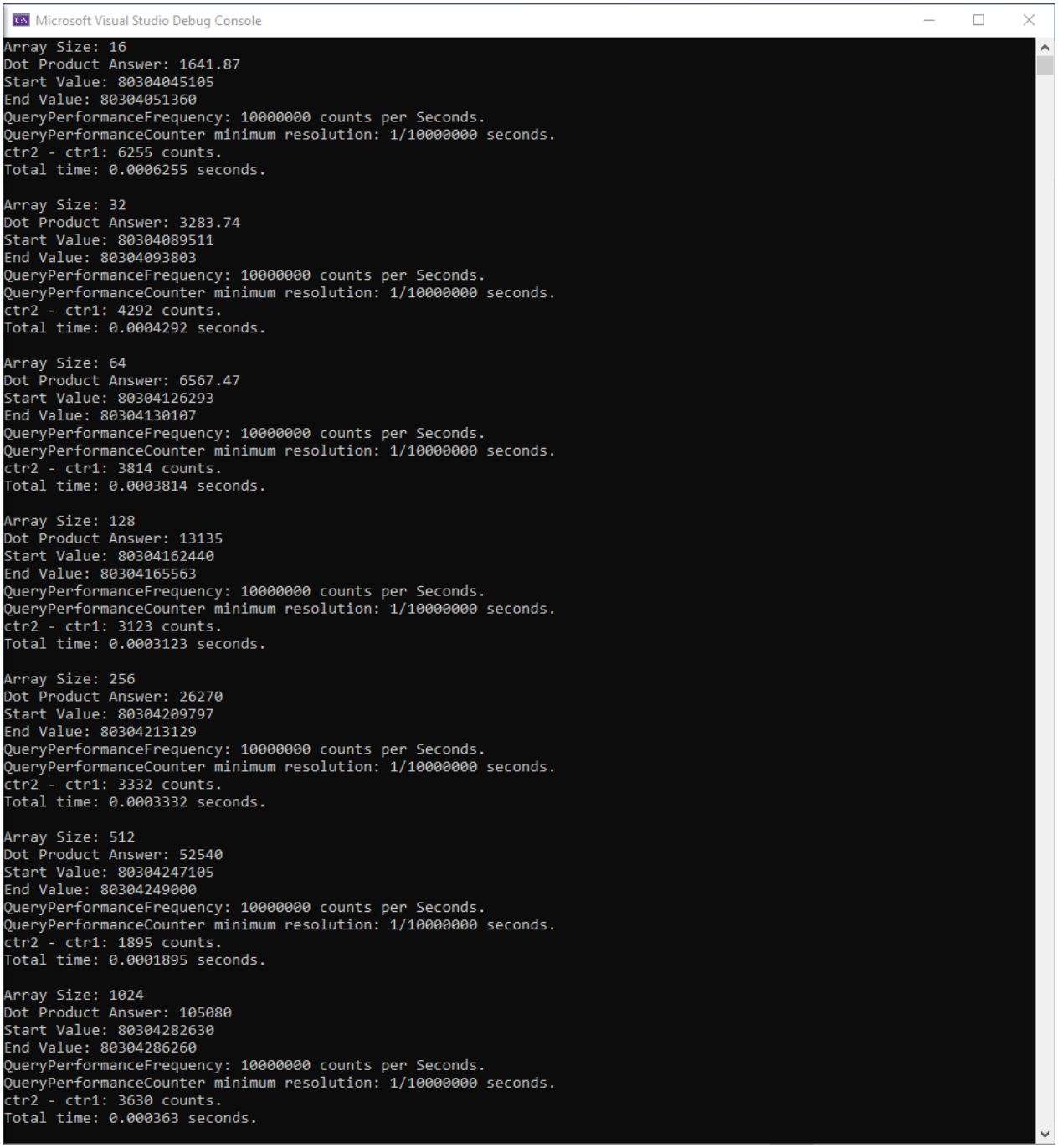


Figure 4: The output after running the function

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.

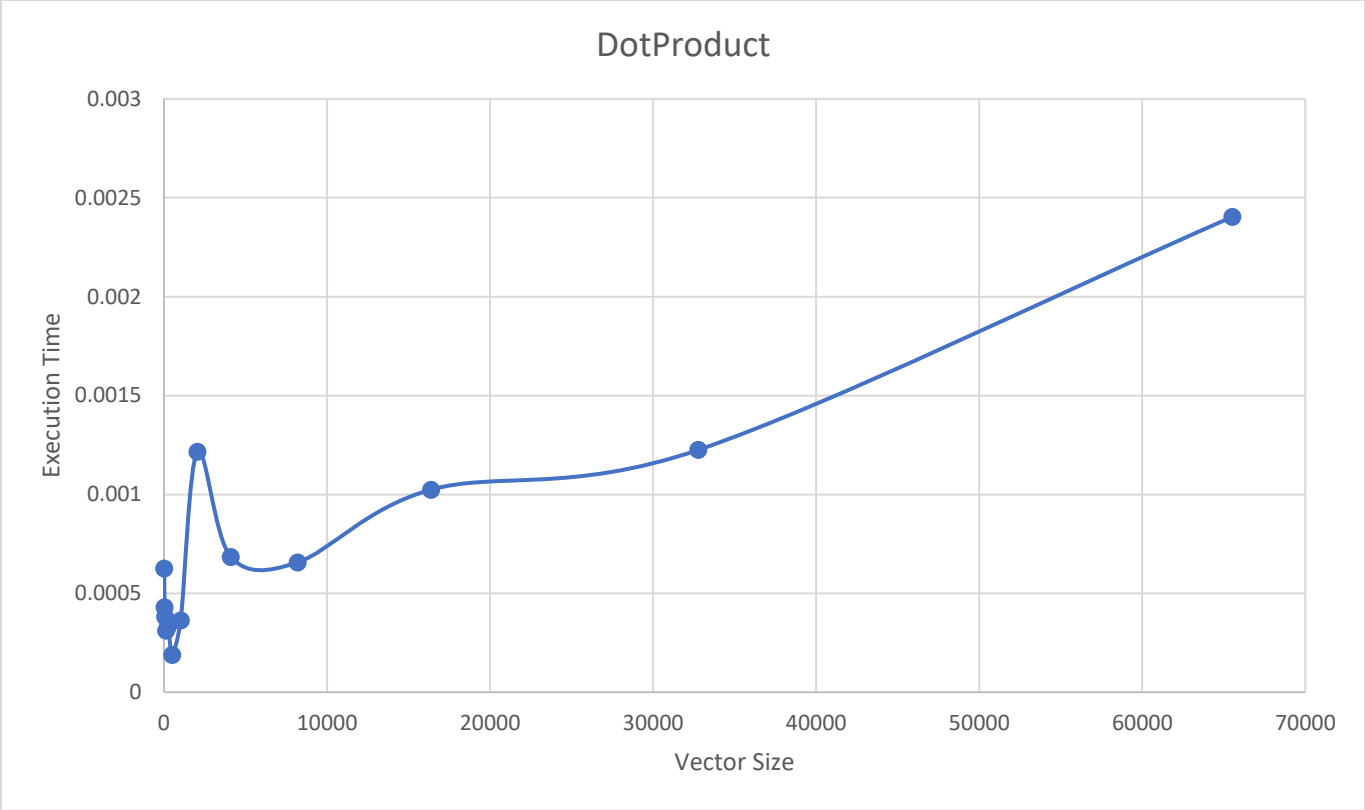


Figure 6: Graph of the dotProduct function execution times vs the vector size.

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

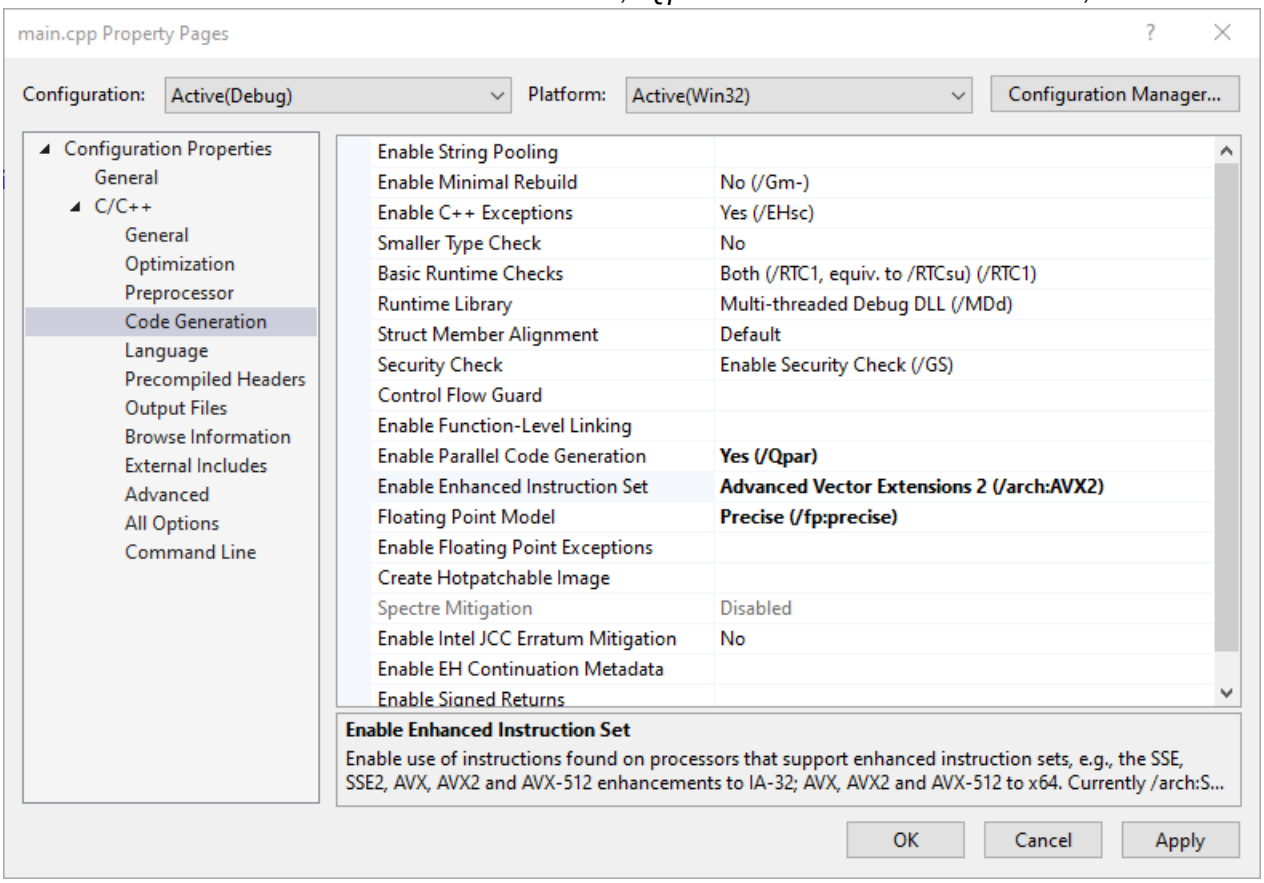


Figure 7: Project Properties Window

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.

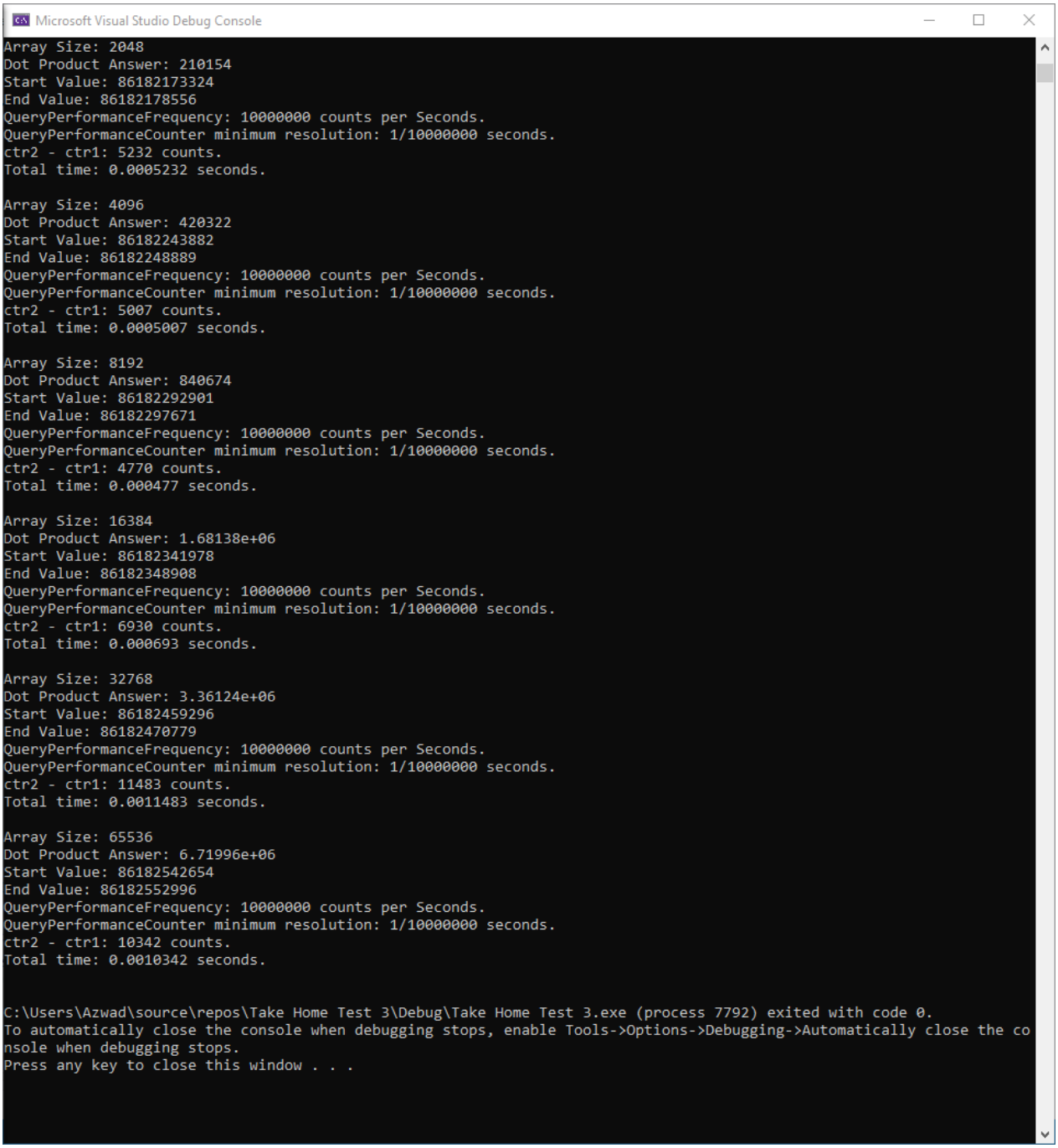


Figure 8: Results of the dotProduct() function with automatic parallelization and vectorization enabled.

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

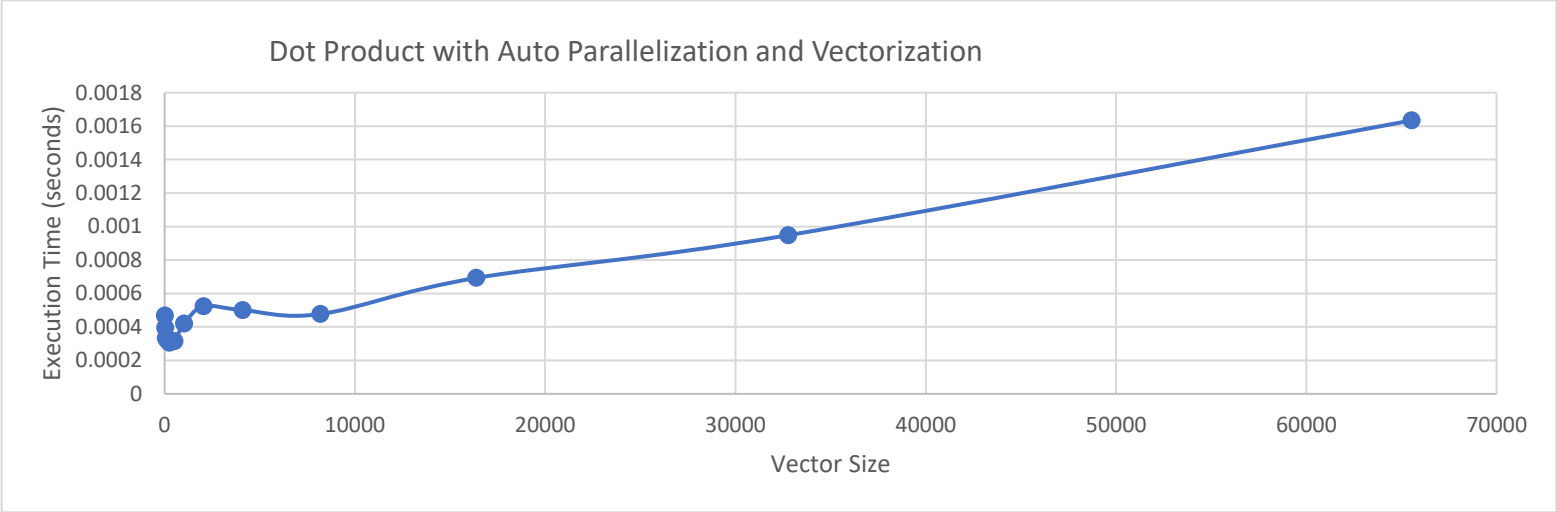
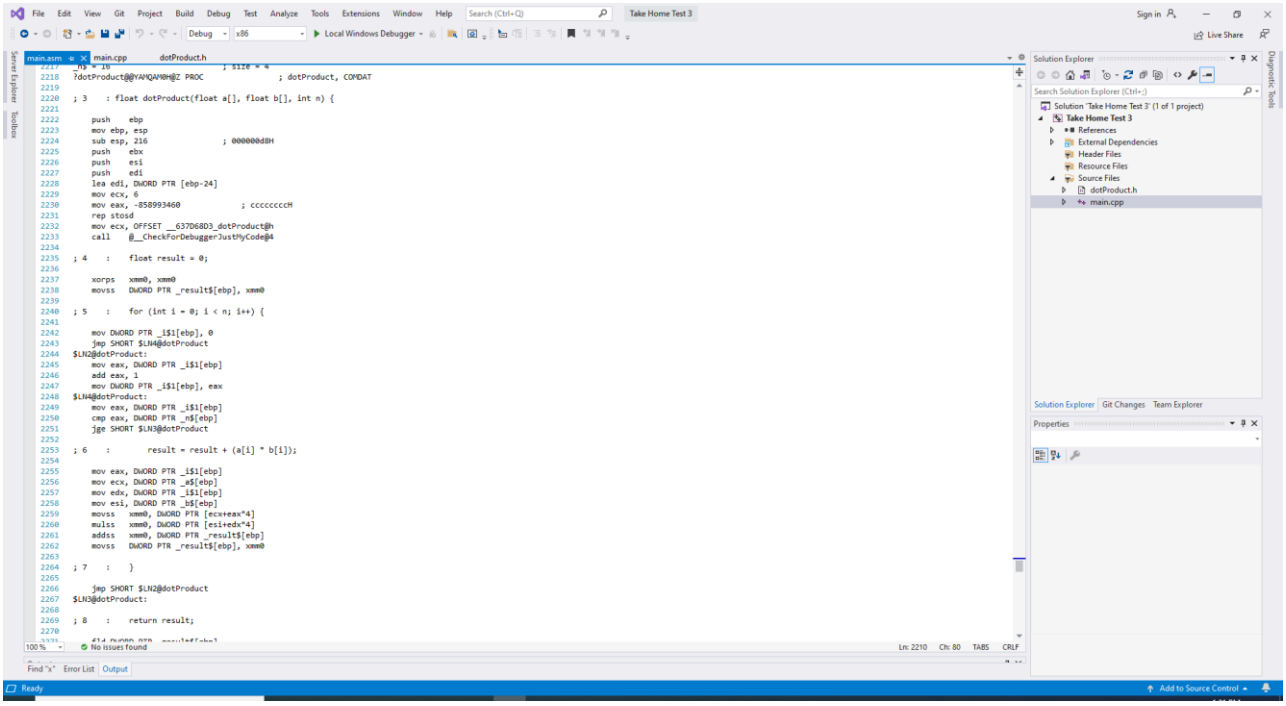


Figure 10: Results of the dotProduct() function with automatic parallelization and vectorization enabled graphed.

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.



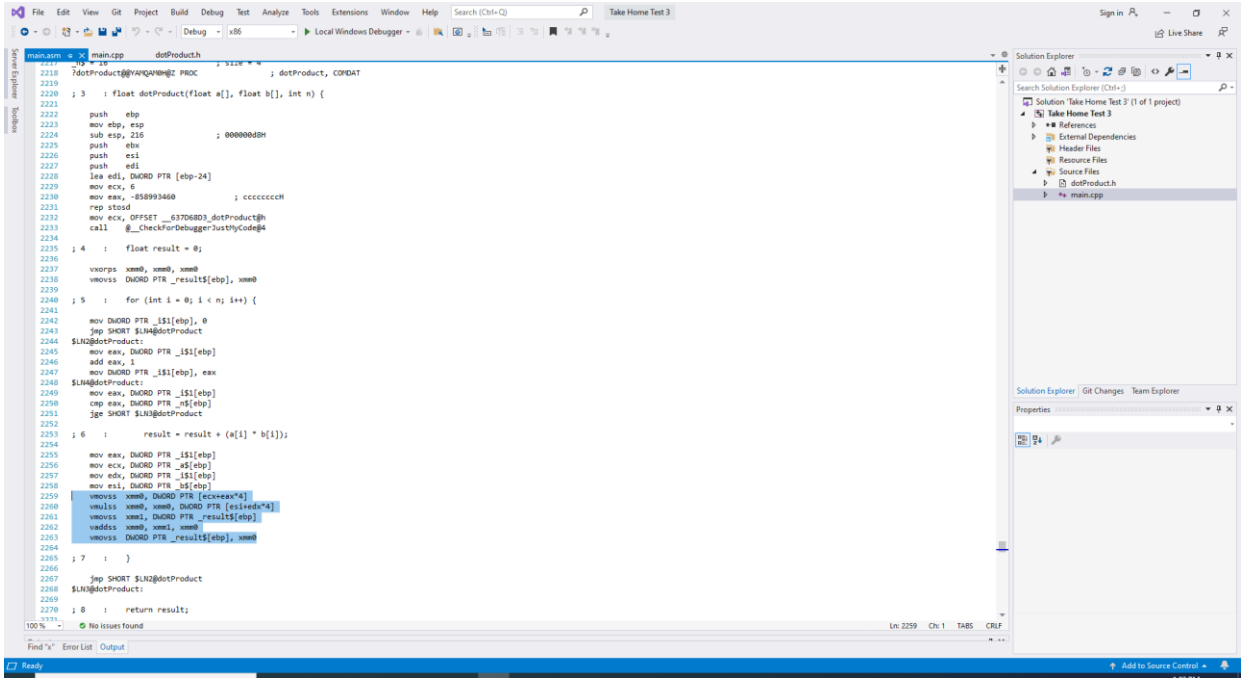


Figure 11: Compiler generated assembly code

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

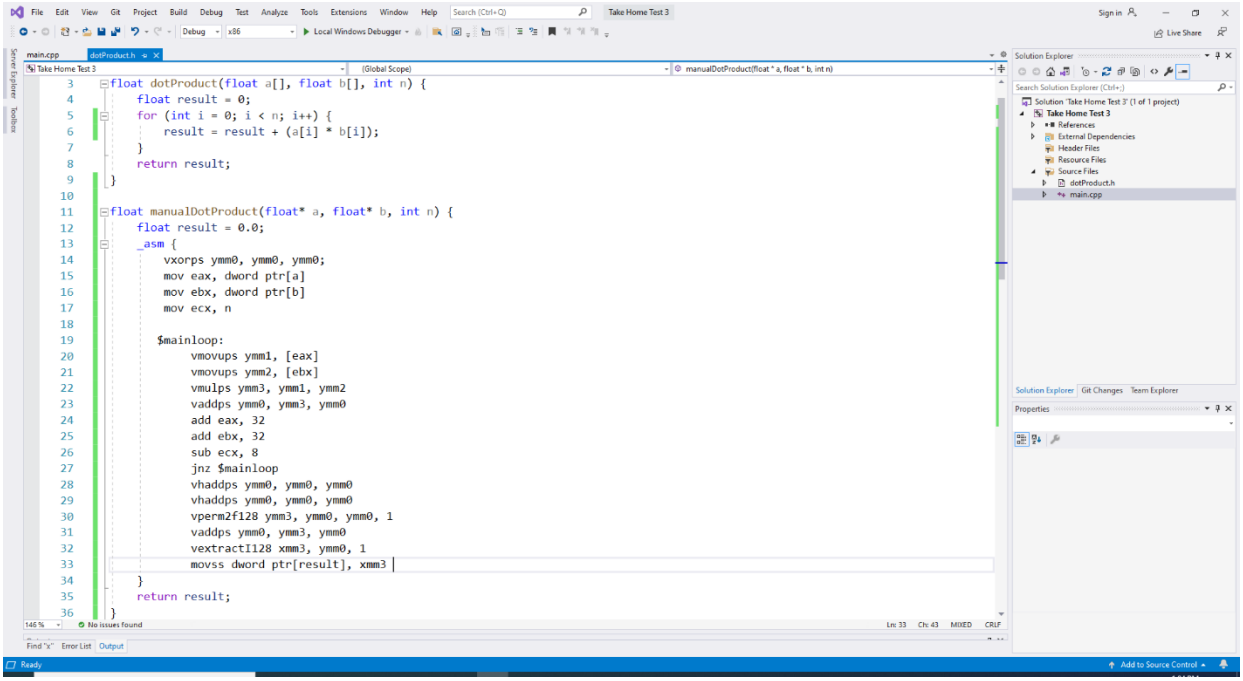


Figure 13: manualDotProduct function

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.

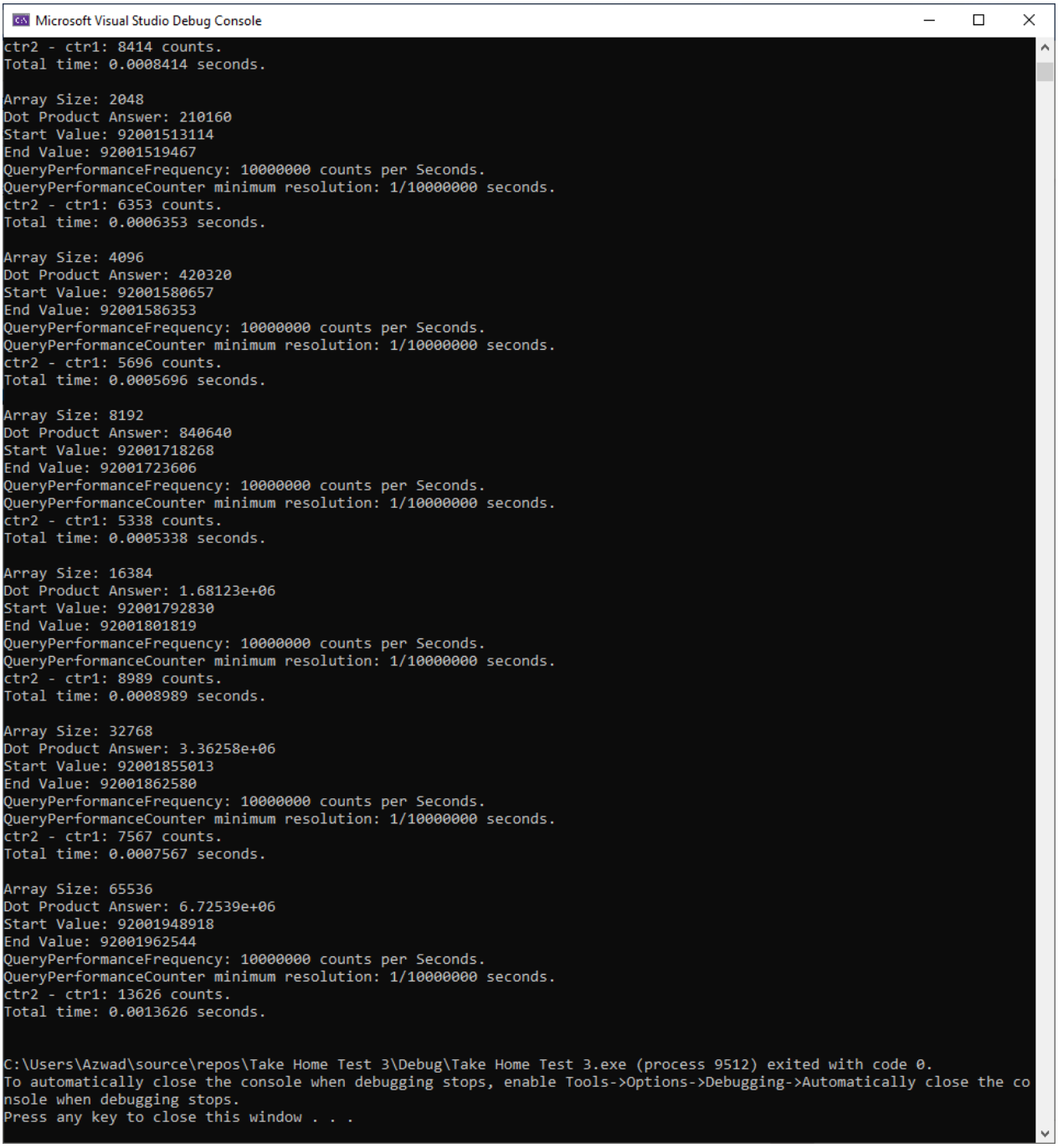


Figure 14: manualDotProduct function output

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

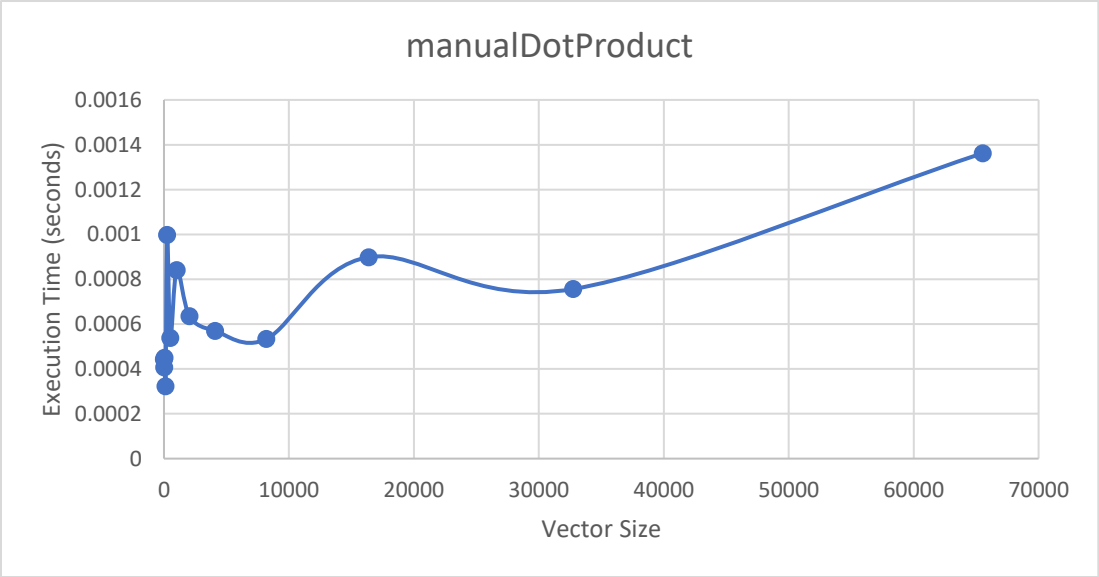


Figure 16: manualDotProduct function results graphed

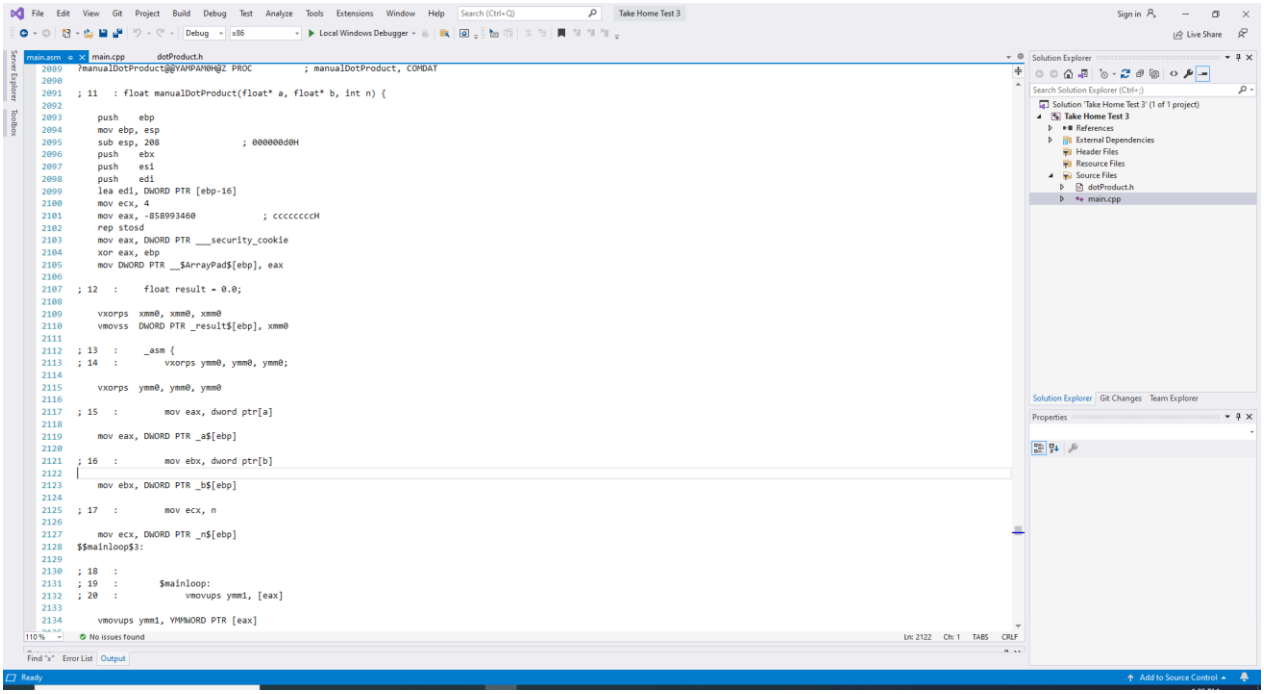


Figure 17: compiler generated assembly code for manualDotProduct()

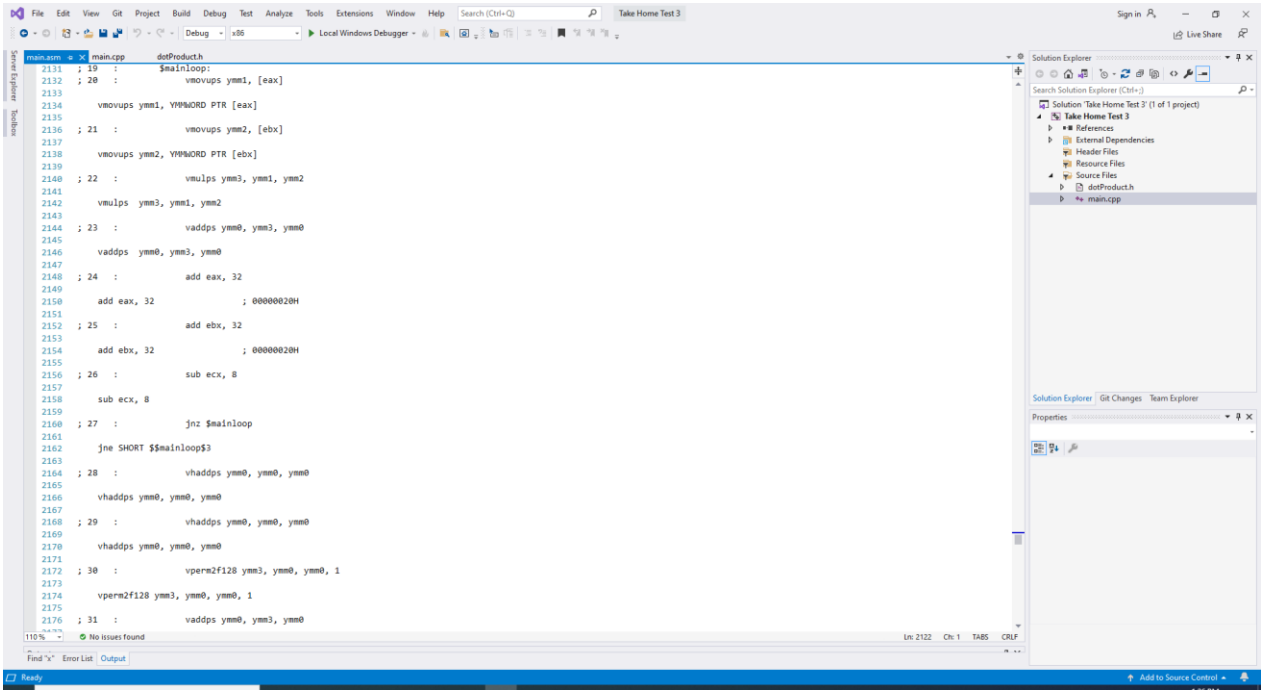


Figure 18: compiler generated assembly code for manualDotProduct()

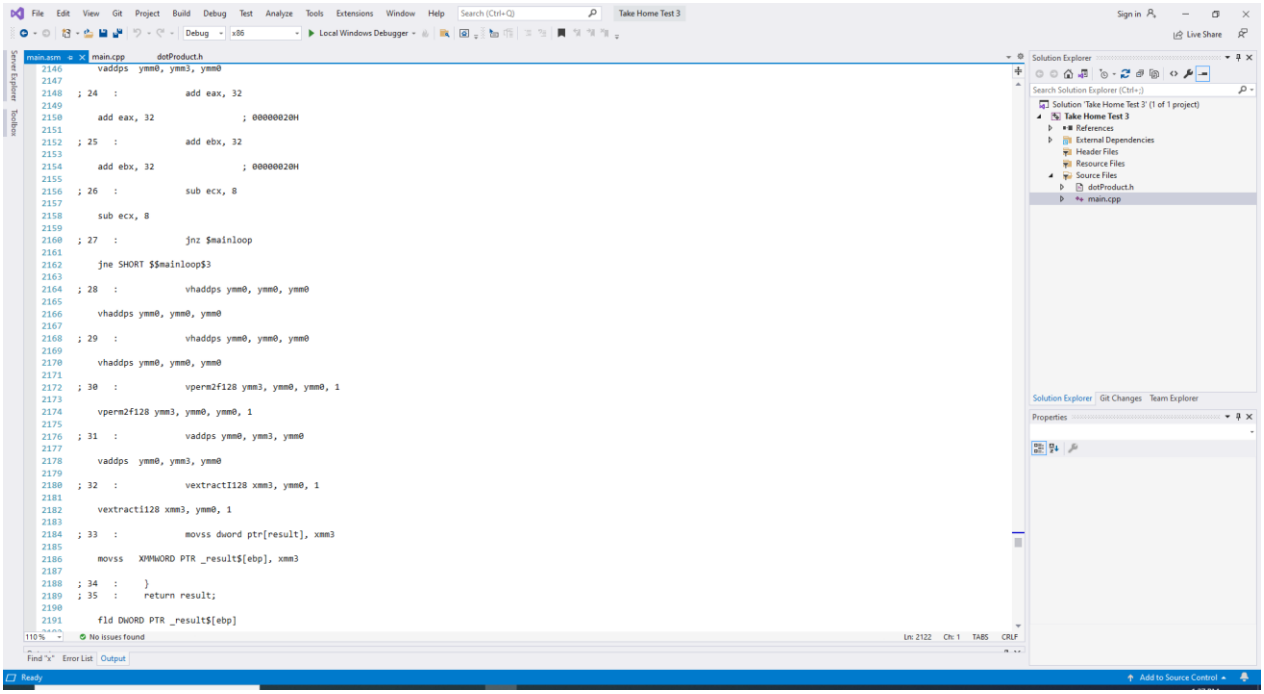


Figure 19: compiler generated assembly code for manualDotProduct()

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

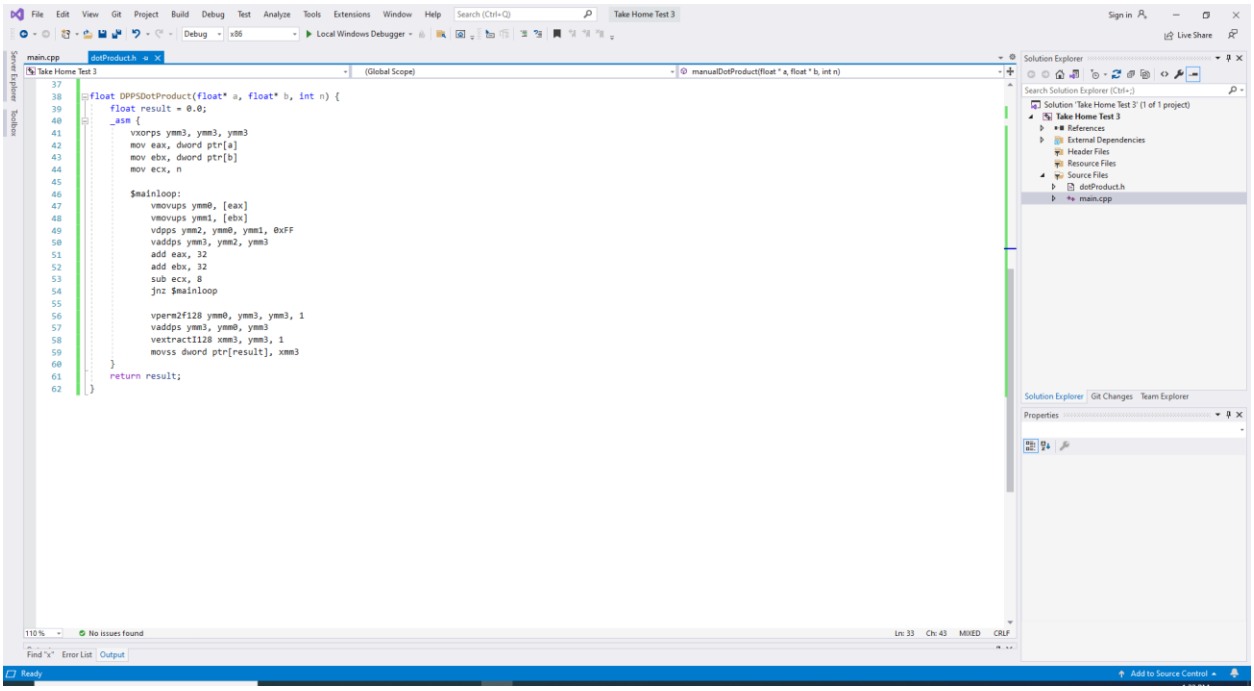


Figure 20: DPPSDotProduct function code in Visual Studio

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 functon DPPSDotProduct we are making the dotProduct function with DPPS vector instructions instead.

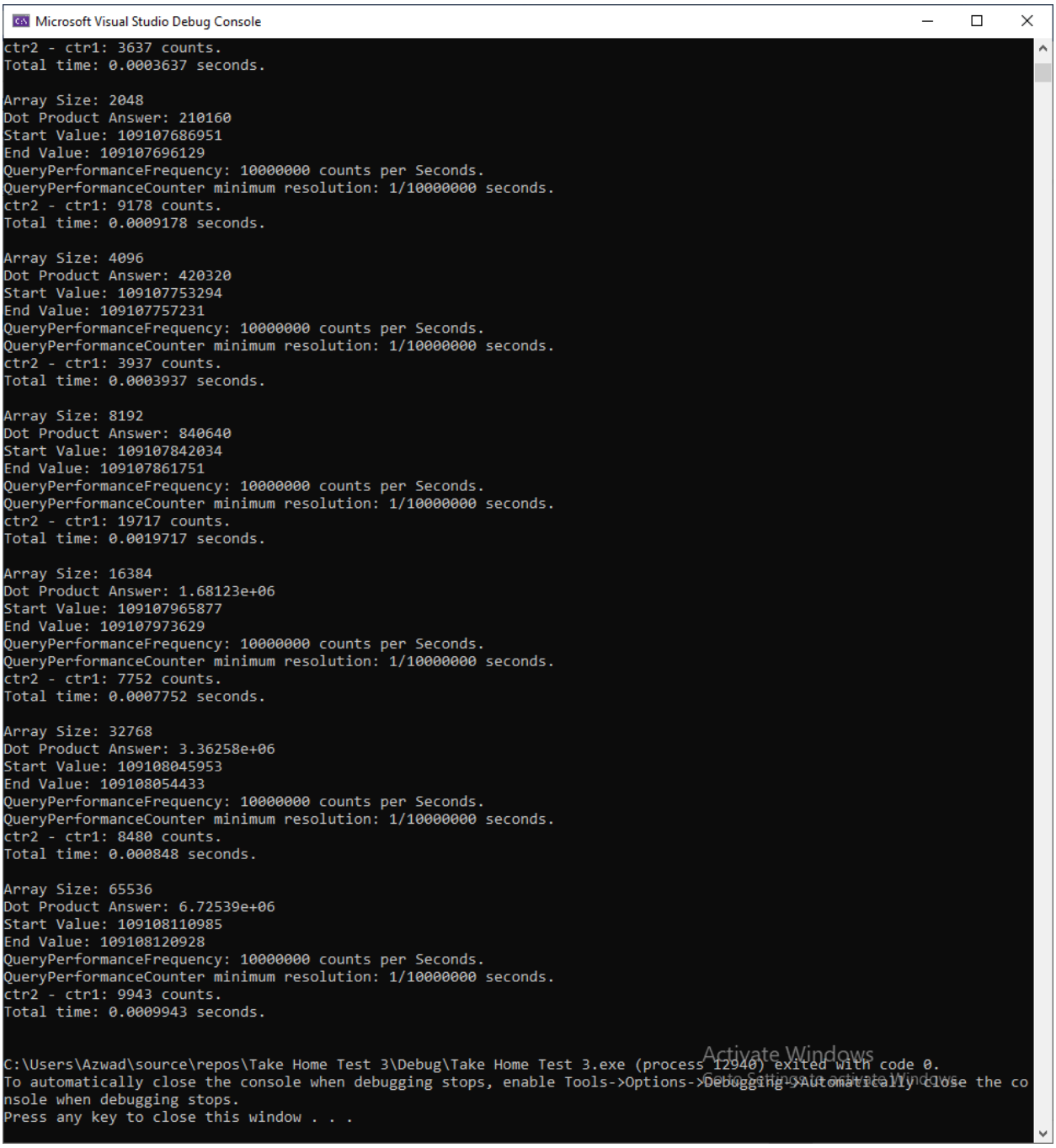


Figure 21: DPPSDotProduct function output.

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

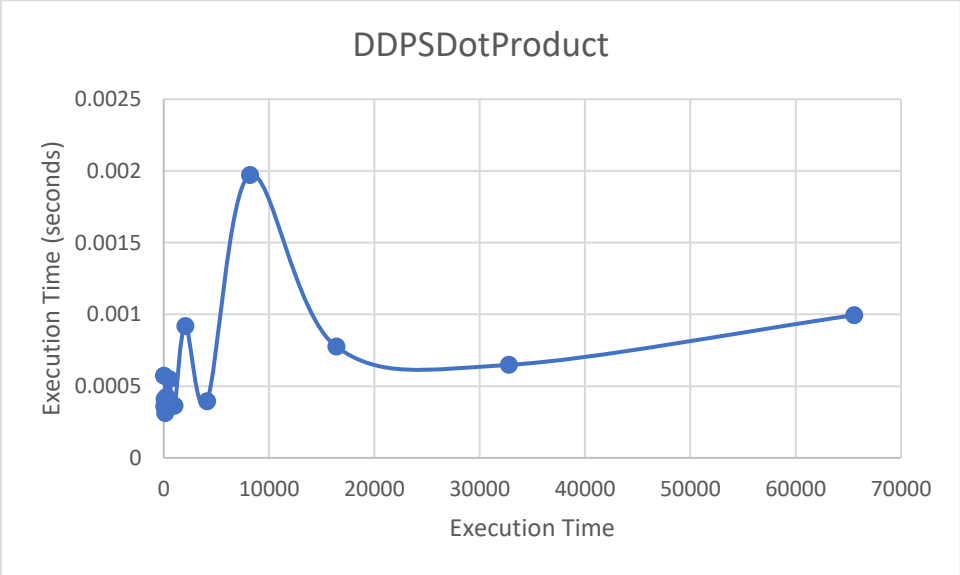


Figure 23: DPPSDotProduct function results in a graph

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 graphes.

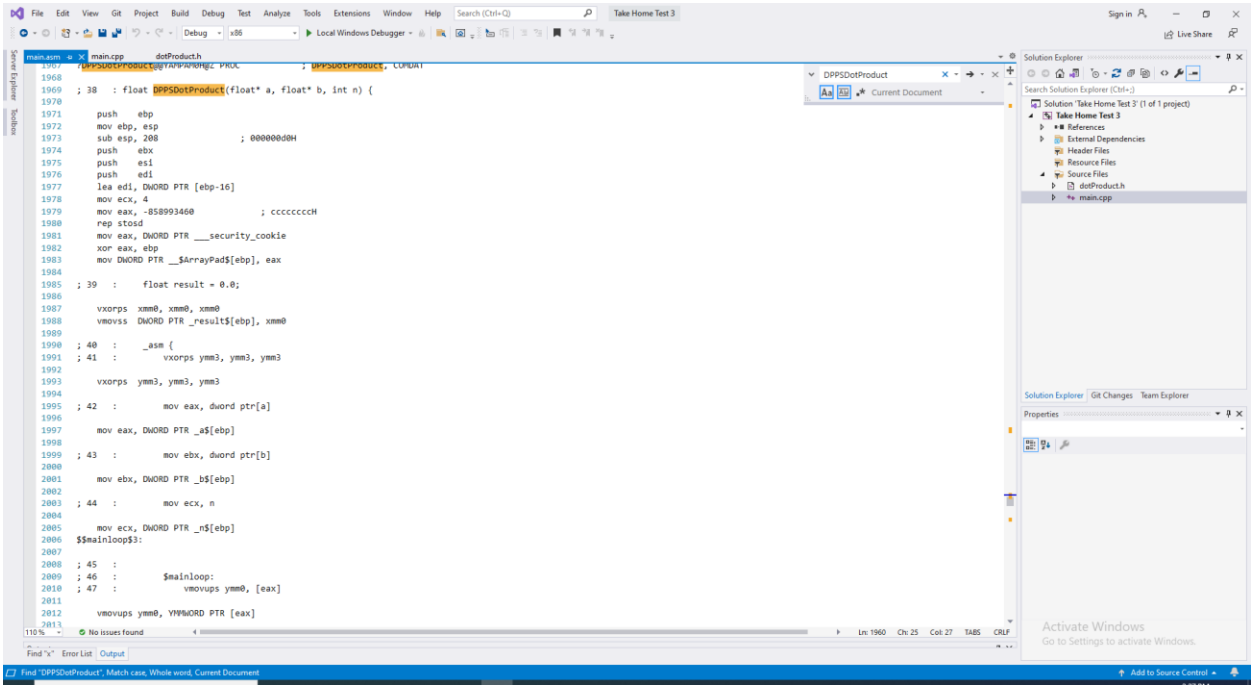


Figure 24: DPPSDotProduct compiler generated assembly code

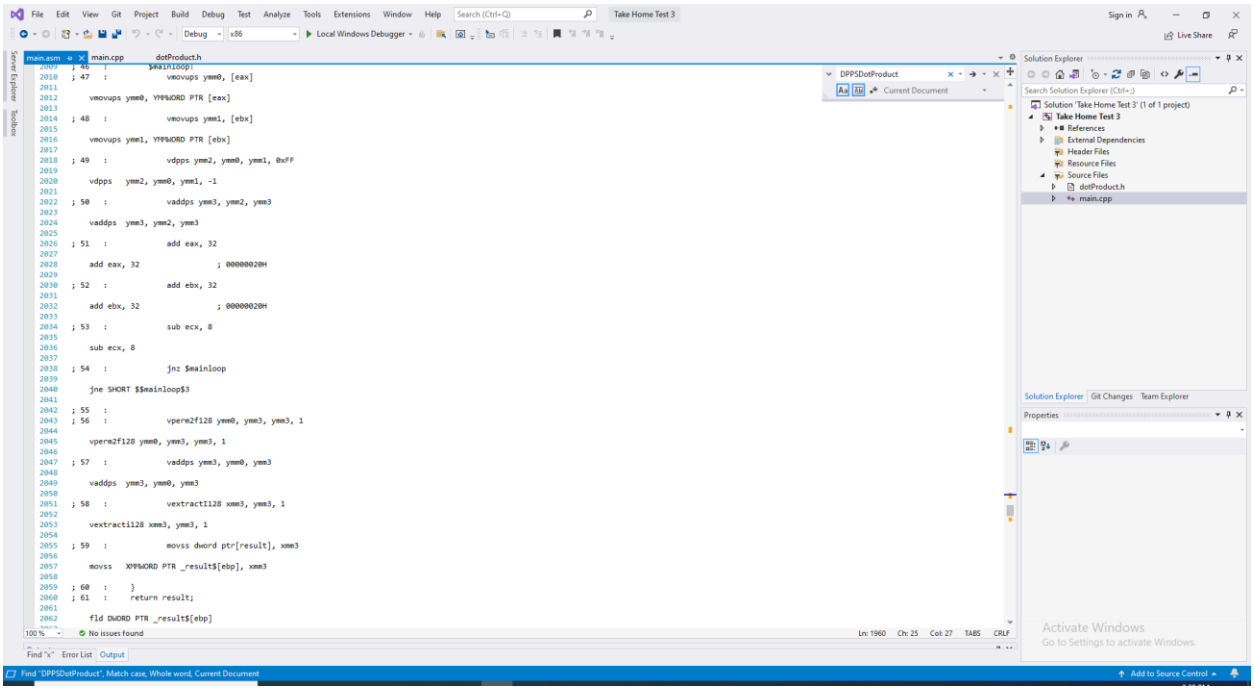


Figure 25: DPPSDotProduct compiler generated assembly code

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.

Comparison

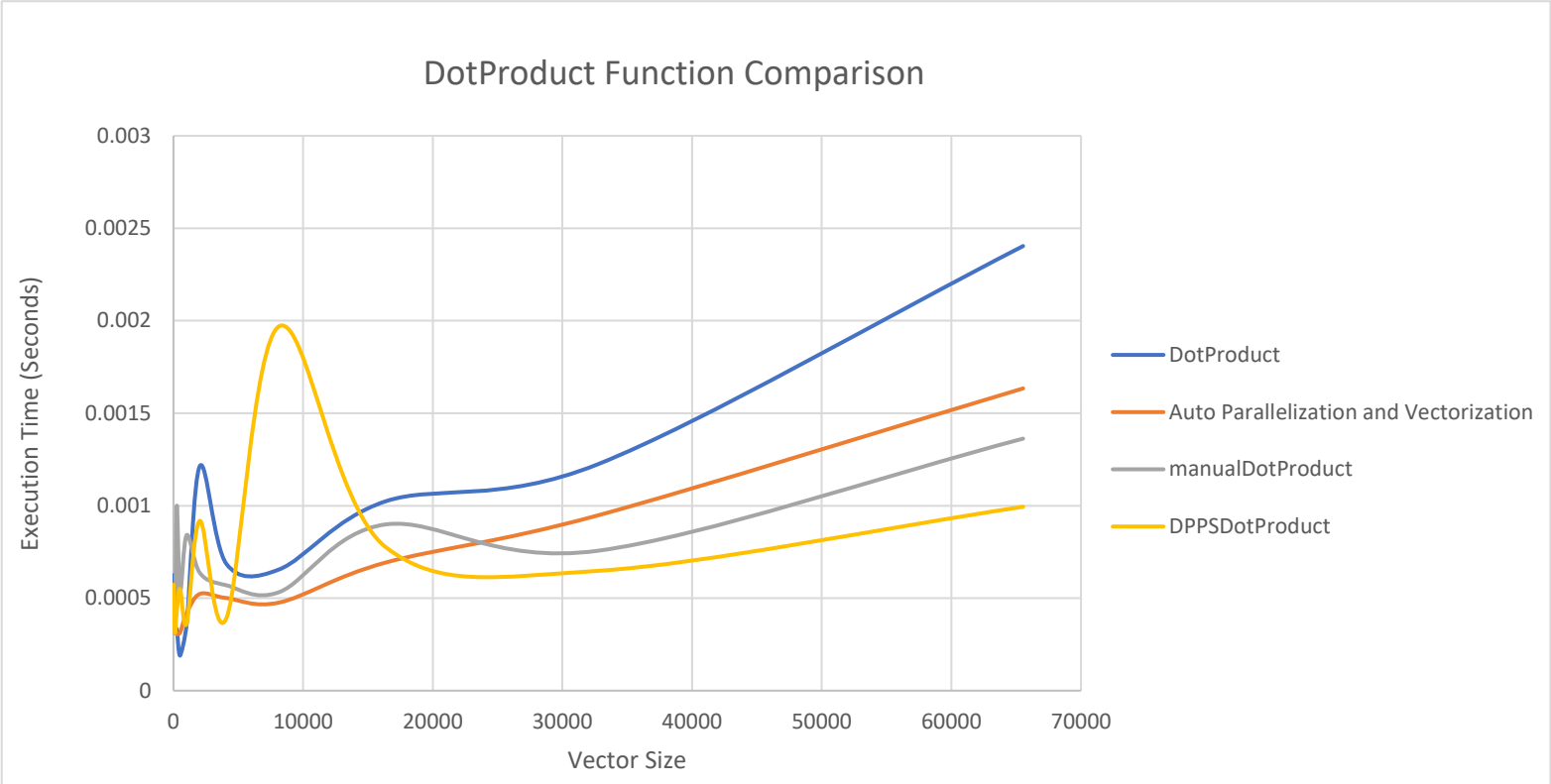


Figure 26: This is the graph of that contains all of the previous functions and their results shown in a graph.

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

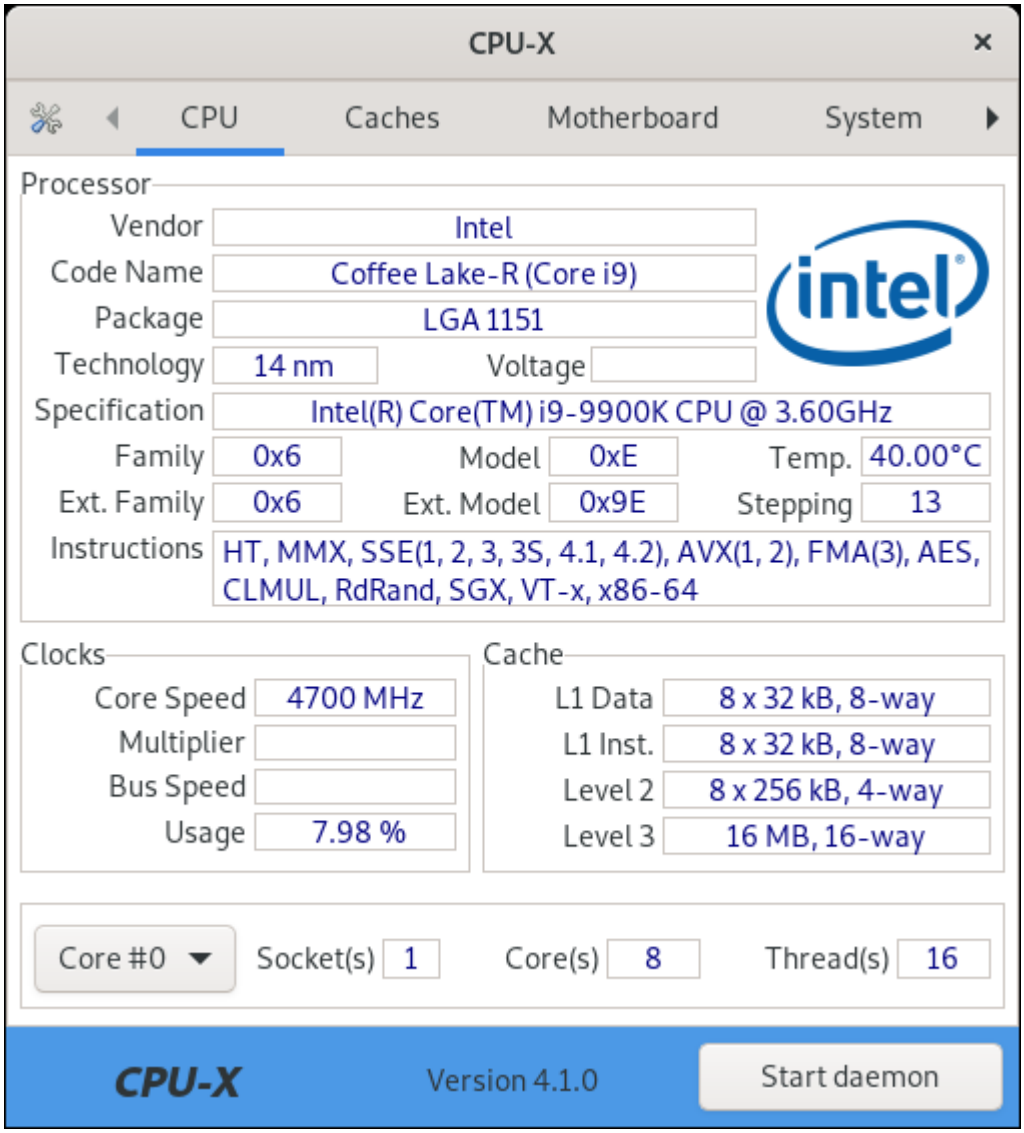


Figure 27: CPU specifications on CPU-X Linux

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 he 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

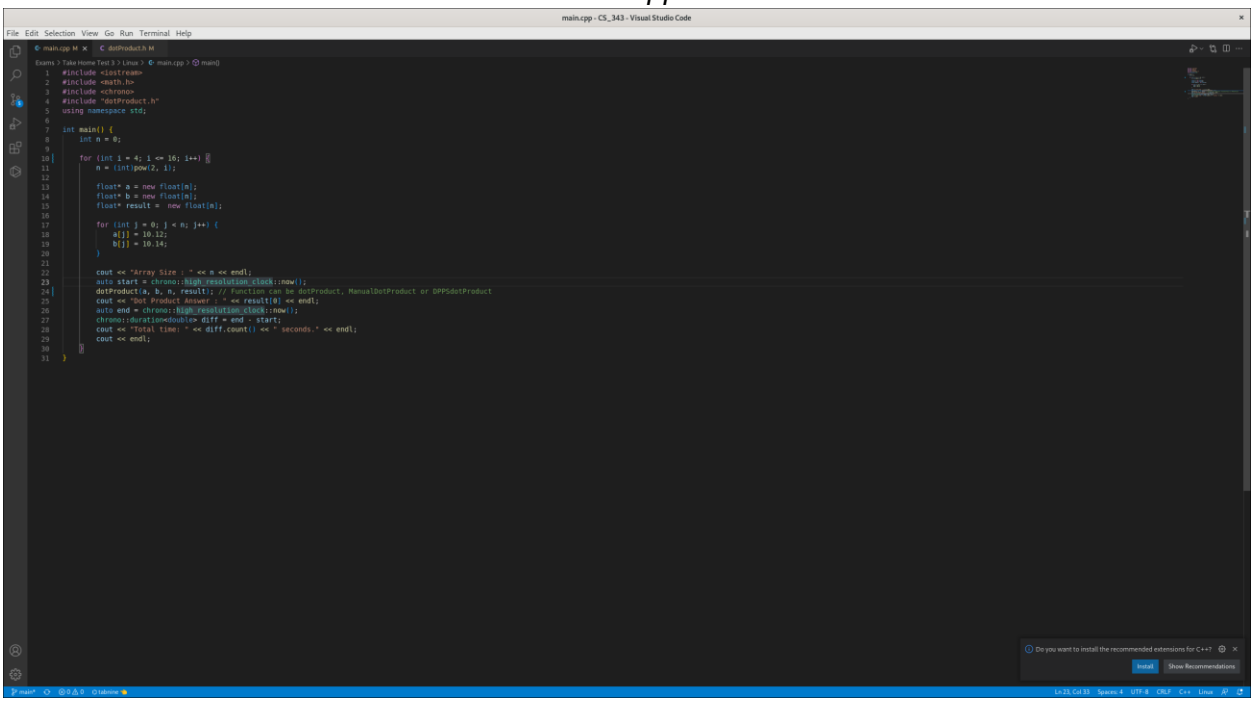


Figure 28: main.cpp code in Linux

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

```cpp
#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
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

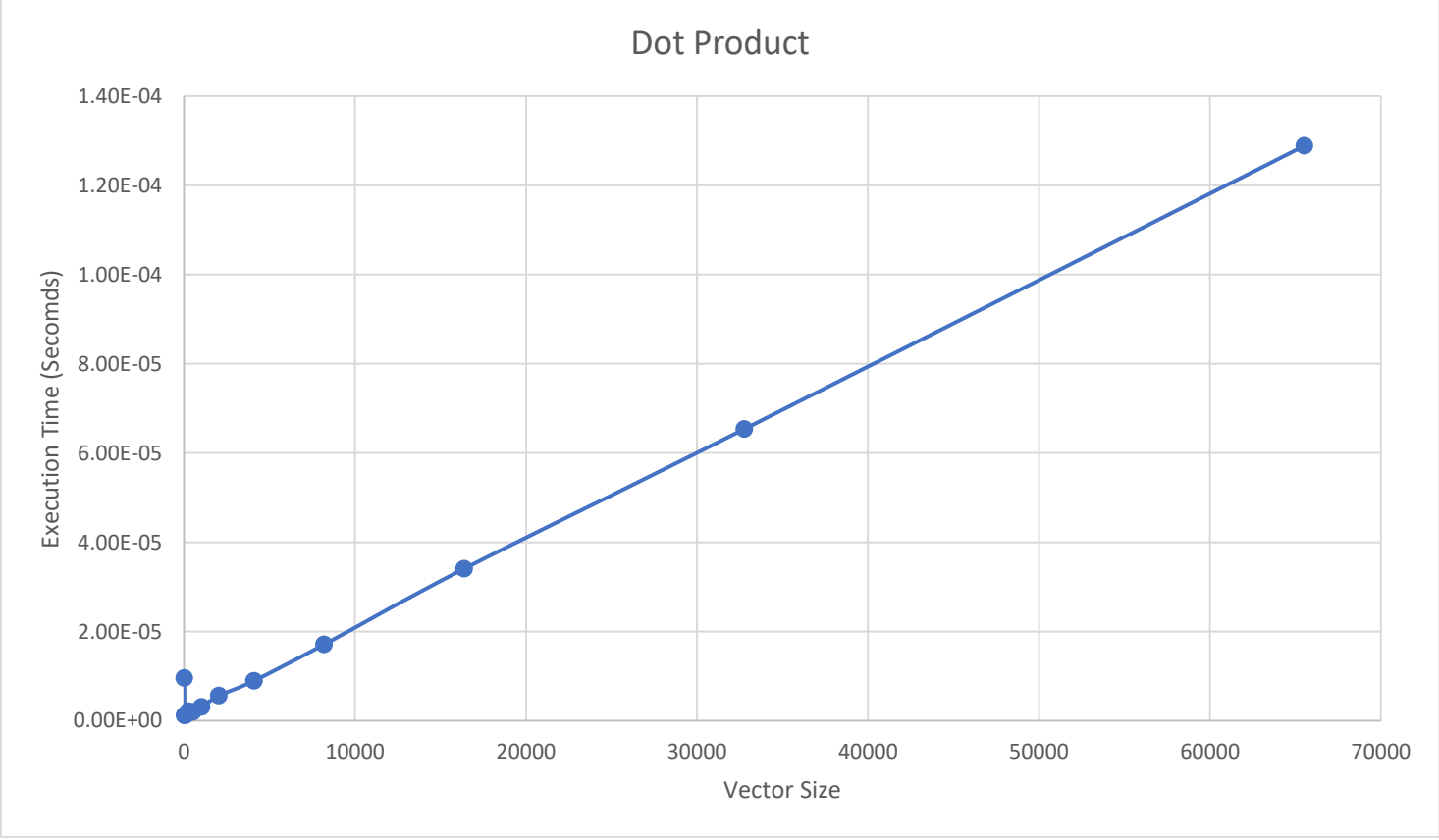


Figure 31: dotProduct function results in a graph

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 Vectorization

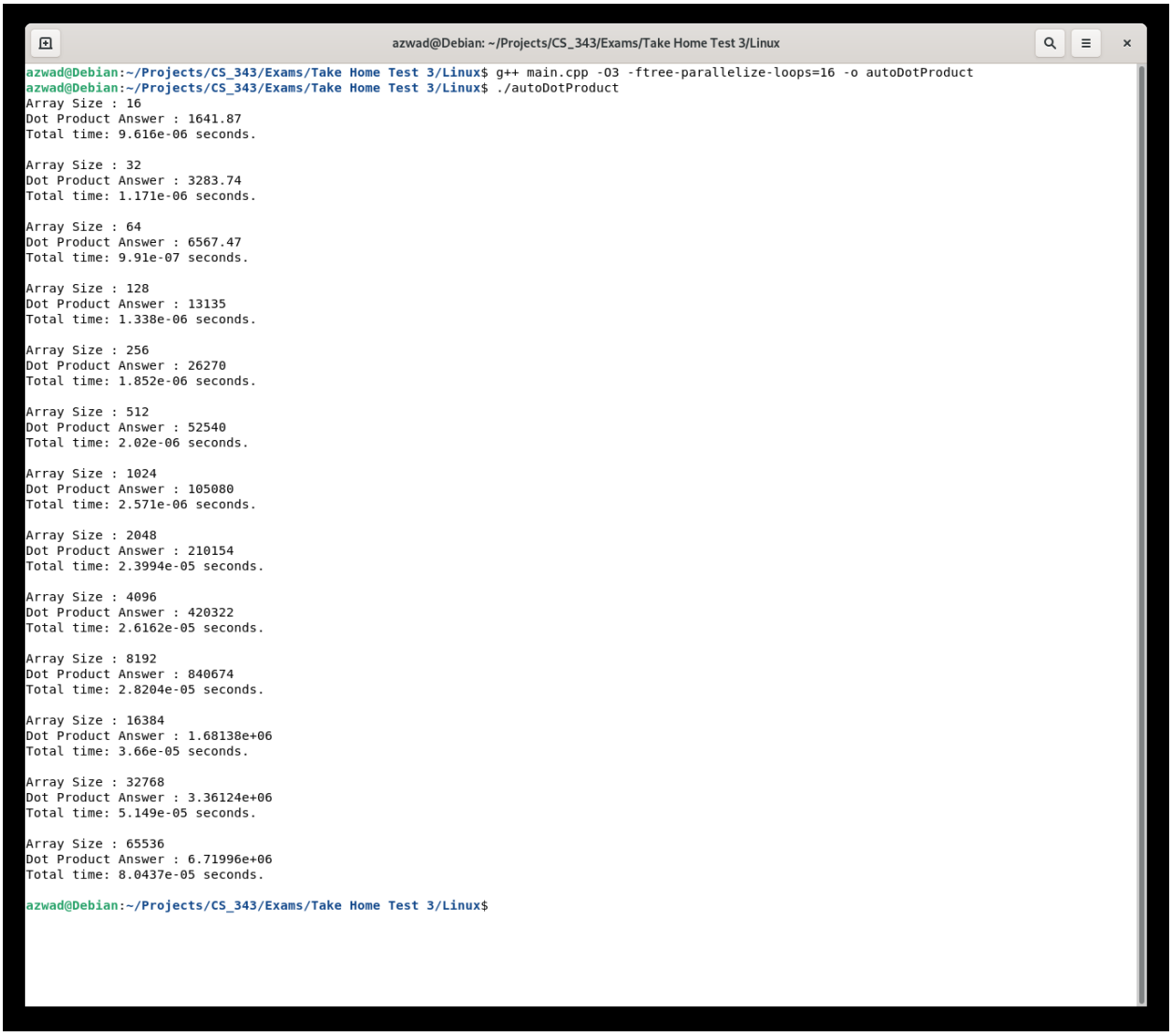


Figure 33: dotProduct function output in terminal

The terminal shows that we use the flags -O3 -ftree-parralelize-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

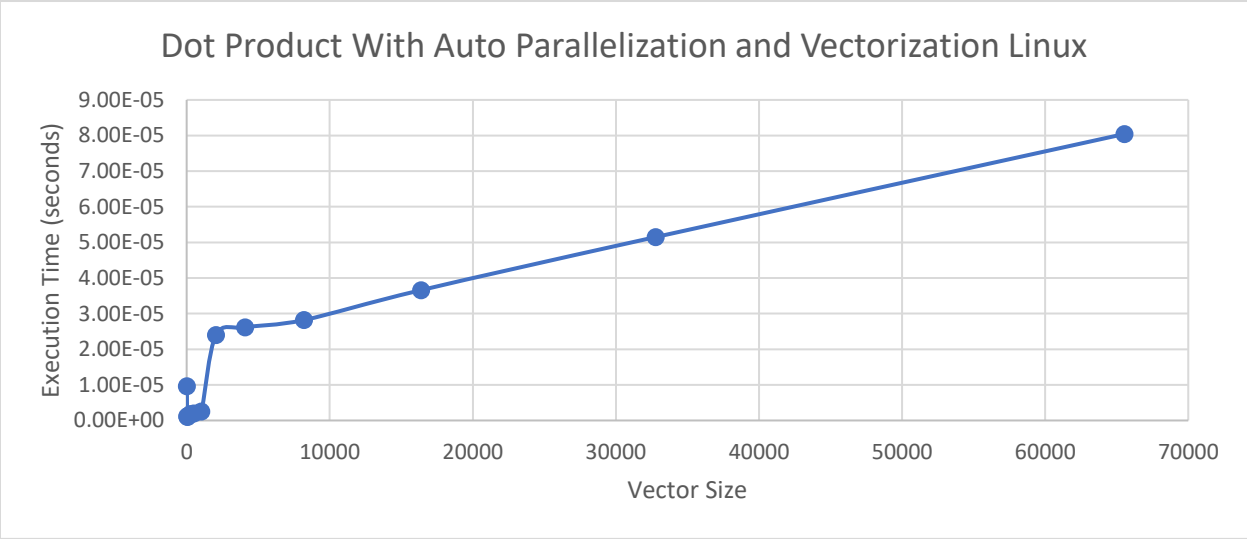
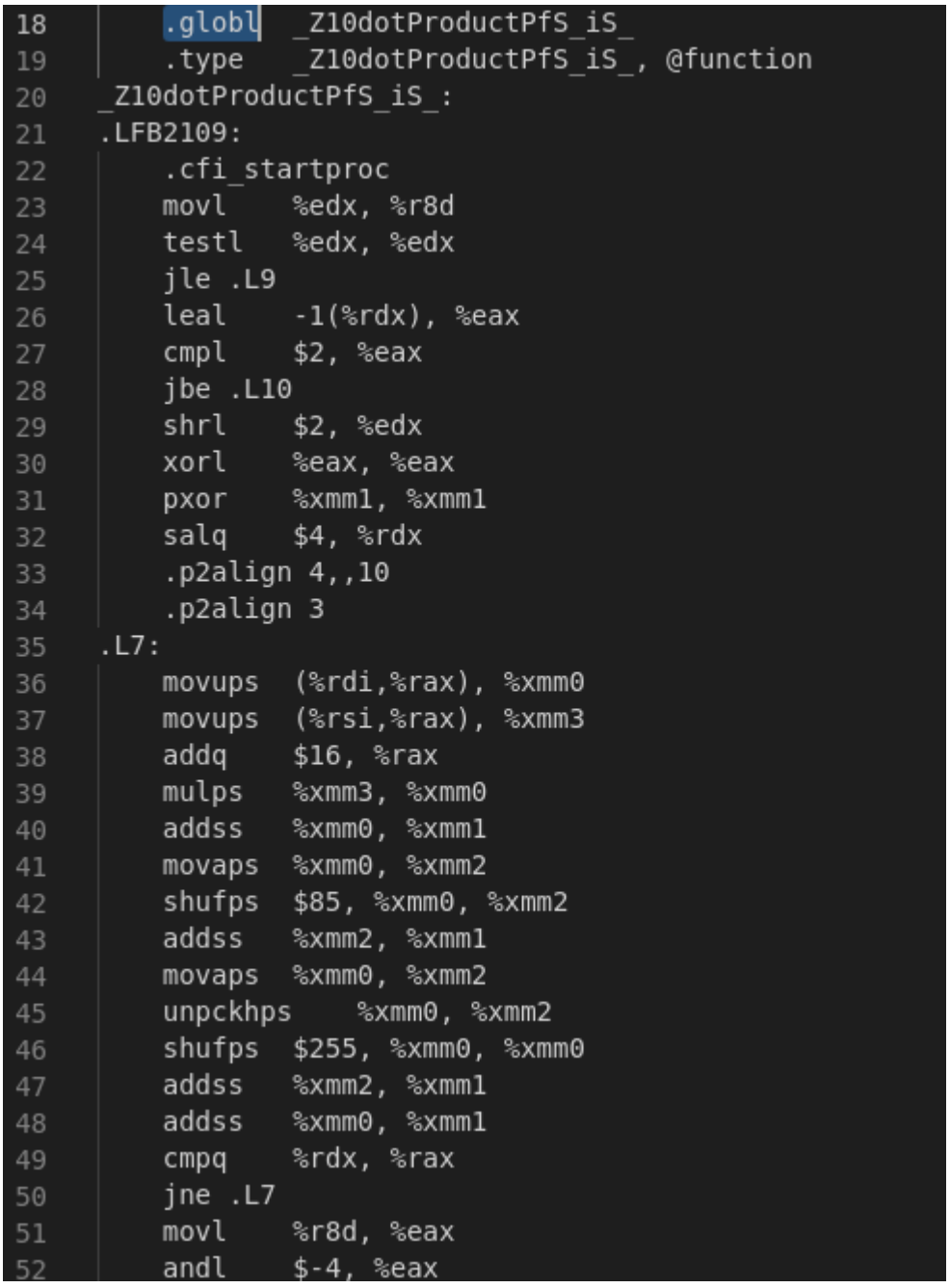


Figure 36: dotProduct function results in a graph

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.

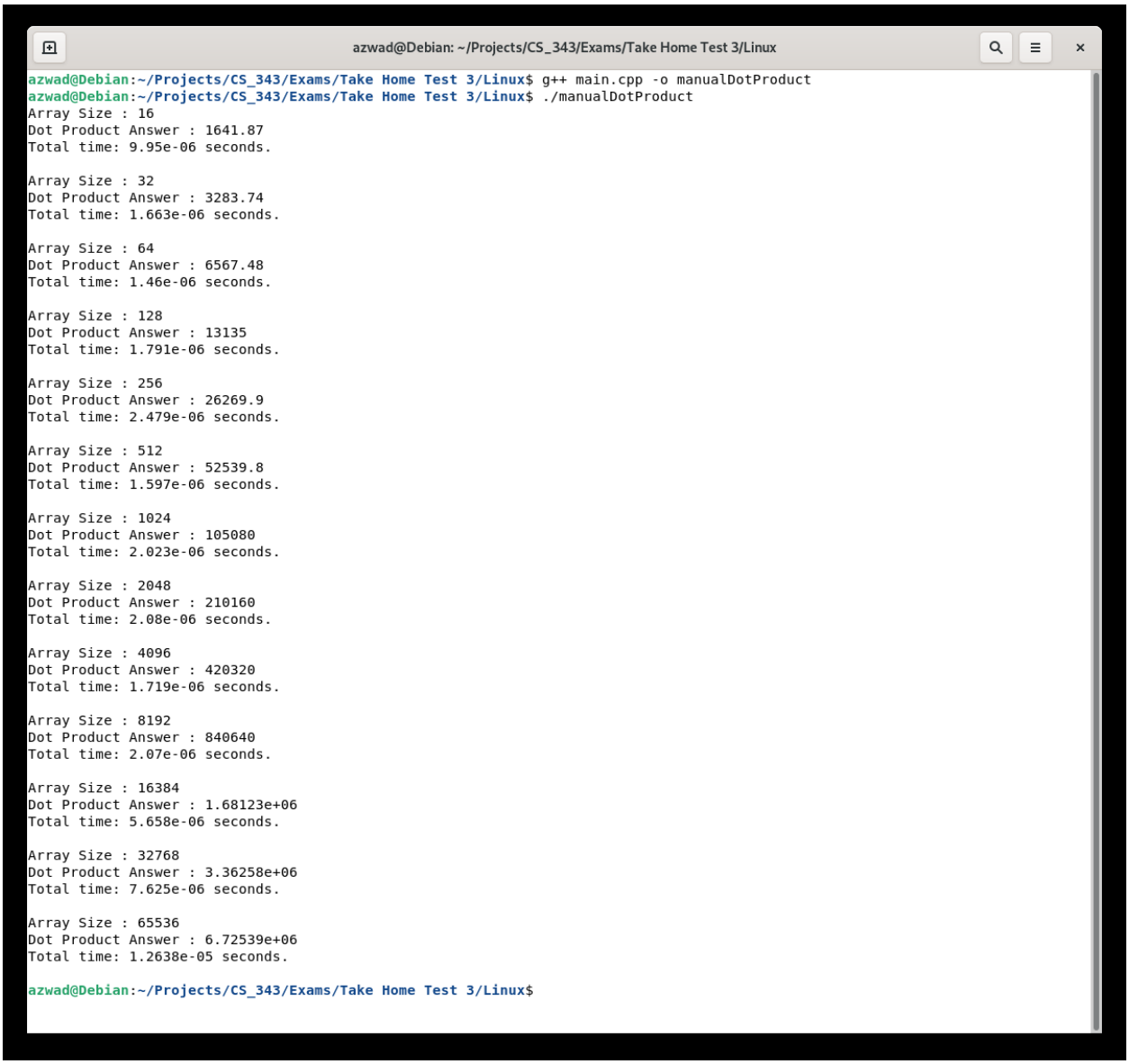


Figure 38: manualDotProduct output in terminal

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

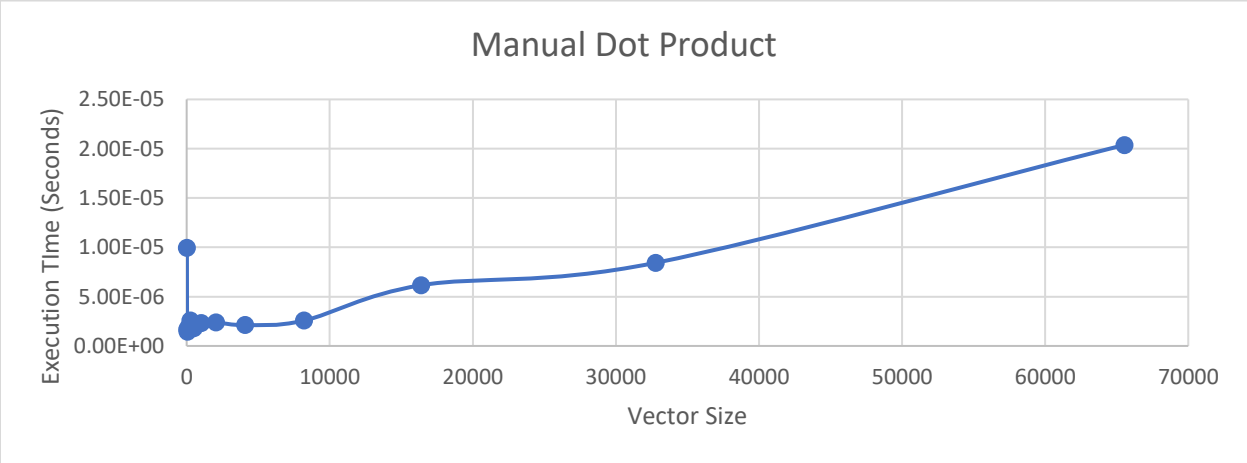


Figure 40: manualDotProduct results in graph

```
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)
```

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 functon DPPSDotProduct we are making the dotProduct function with DPPS vector instructions instead.

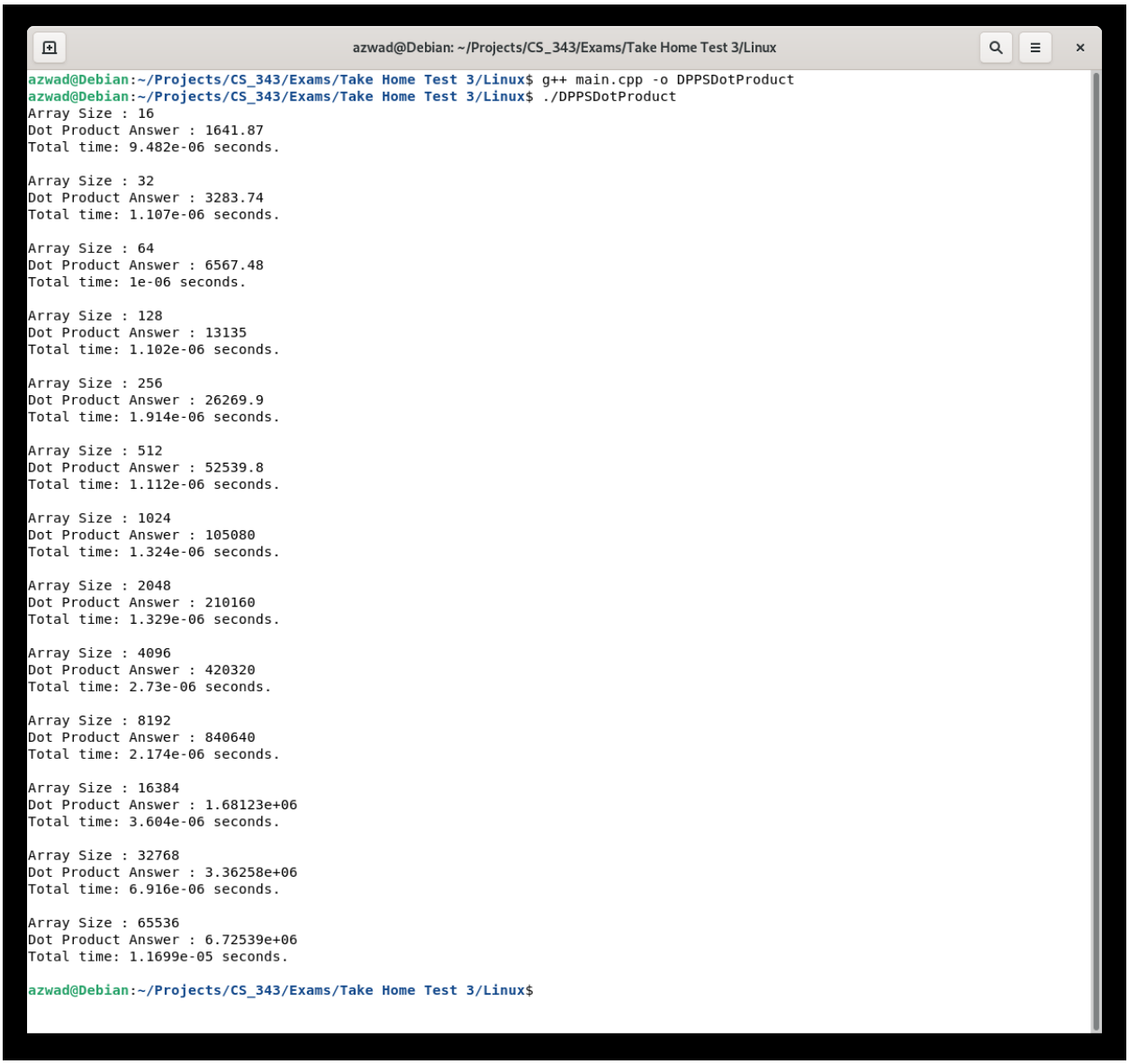


Figure 43: DPPSDotProduct function output

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

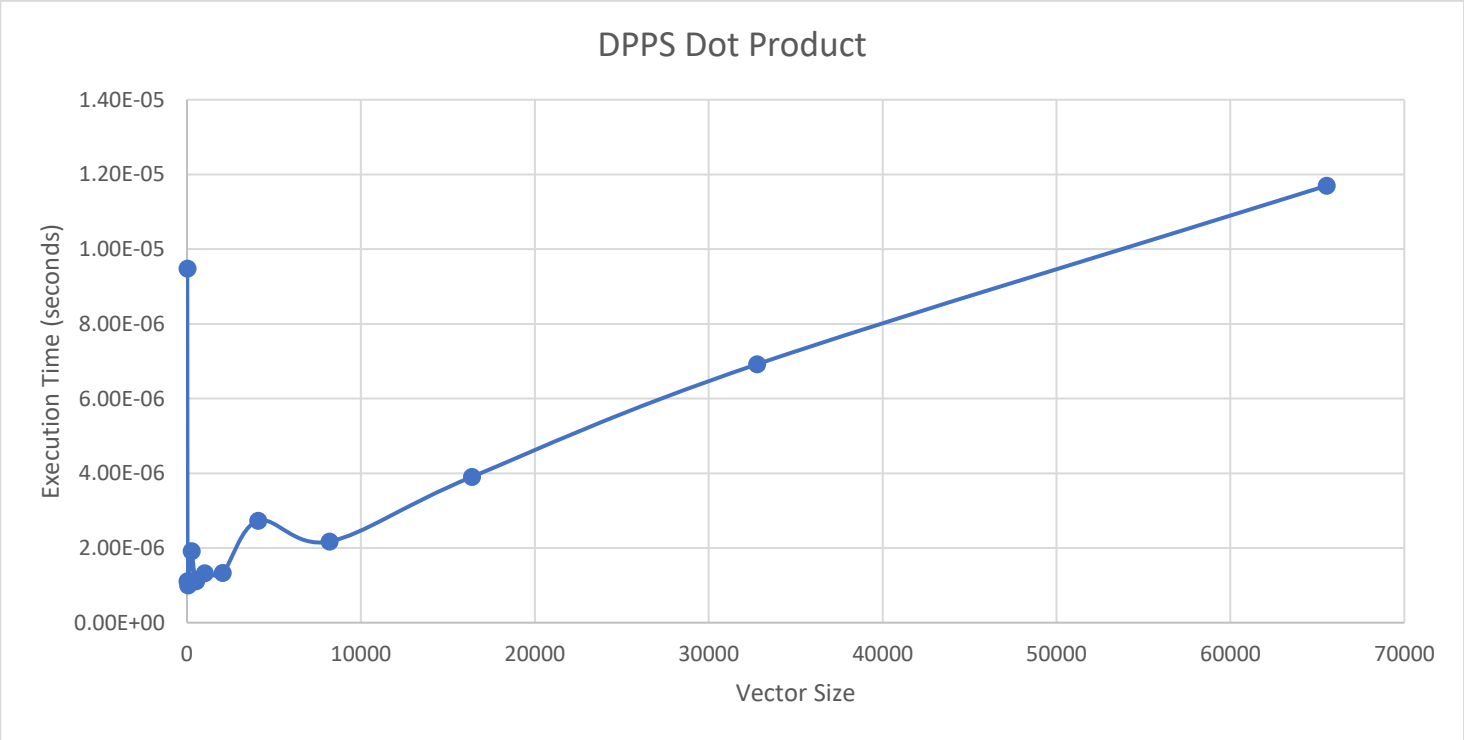


Figure 45: DPPSDotProduct function results in graph

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.

Comparison

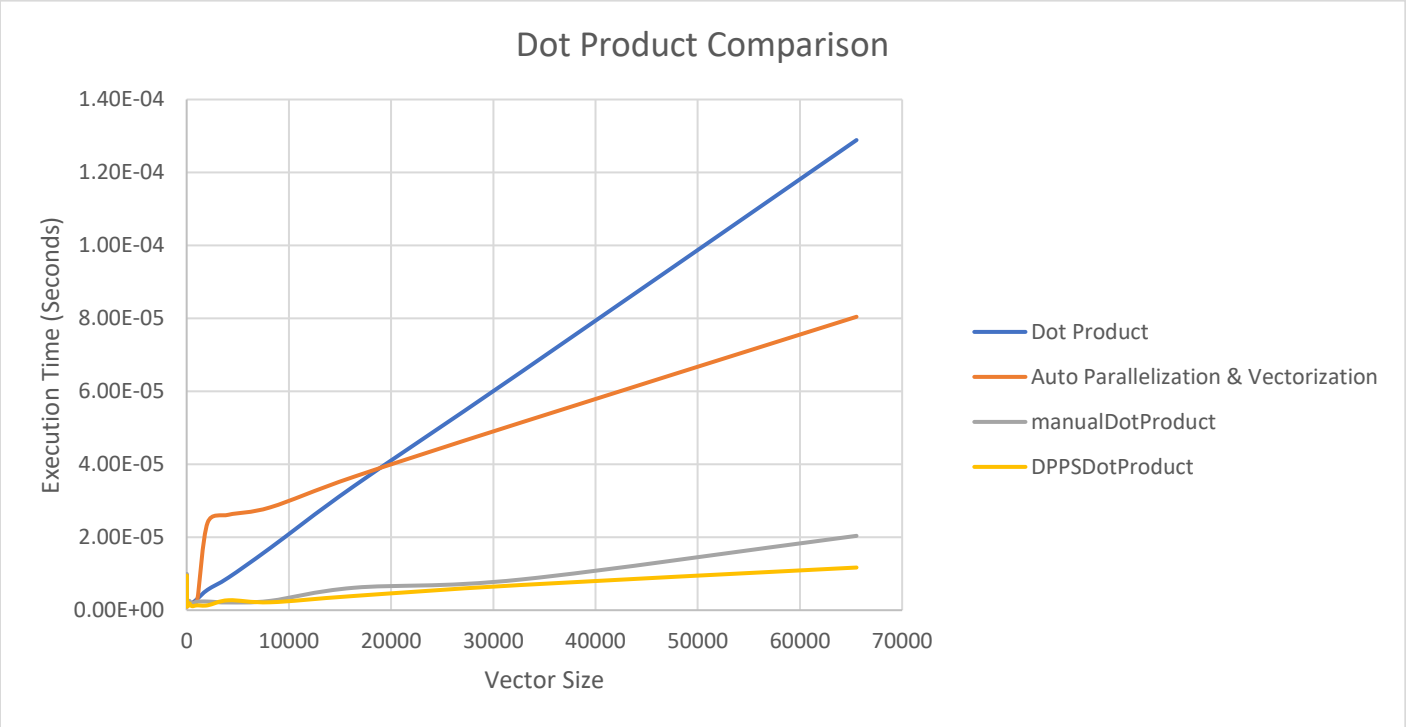


Figure 47: This is the graph of that contains all of the previous functions and their results shown in a graph.

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.