Take Home Test 3:

OPTIMIZATION OF DOT PRODUCT COMPUTATION OF TWO VECTORS USING VECTOR INSTRUCTIONS

MdShahid Bin Emdad

May 15th, 2022

CSC 34200

I will neither give nor receive unauthorized assistance on this test. I will use only one computing device to perform this TEST, I will not use cell while performing this test

Shahid

Start Time and date: 9:00 am, 5/13/2022

End Time and date: 8:45 pm. 5/13/2022

TABLE OF CONTENTS

Objective:

The objective of this test is to explore, understand and ability to compute and demonstrate dot products using vector instructions and optimize the code by compiler generate in both Intel x86(MS VS) and Intel x64(Linux) environment. The compilers use parallelization, vectorization to optimize the code. We were also instructed to optimate the code by ourselves for better execution runtime. Eventually, we compared the run time from both environments and check which is the most optimal way to execute the program faster.

Description of Specifications, and Functionality:

The digital system I used in this assignment is Microsoft Visual Studio 2019 to get the average time for each array execution time. After that, I used Ubuntu Linux to run all the c files in the same way and got the average time as well for each array size. I did 2^3(8) to 2^12(4096).

Intel x86 (Visual Studio Code):

CPU-Z specifications:

In figure 1, we can see the CPU-Z window with specifications, the highest instructions I can use the AVX2 for asm file generating. We needed to know this to specify generating the asm file.

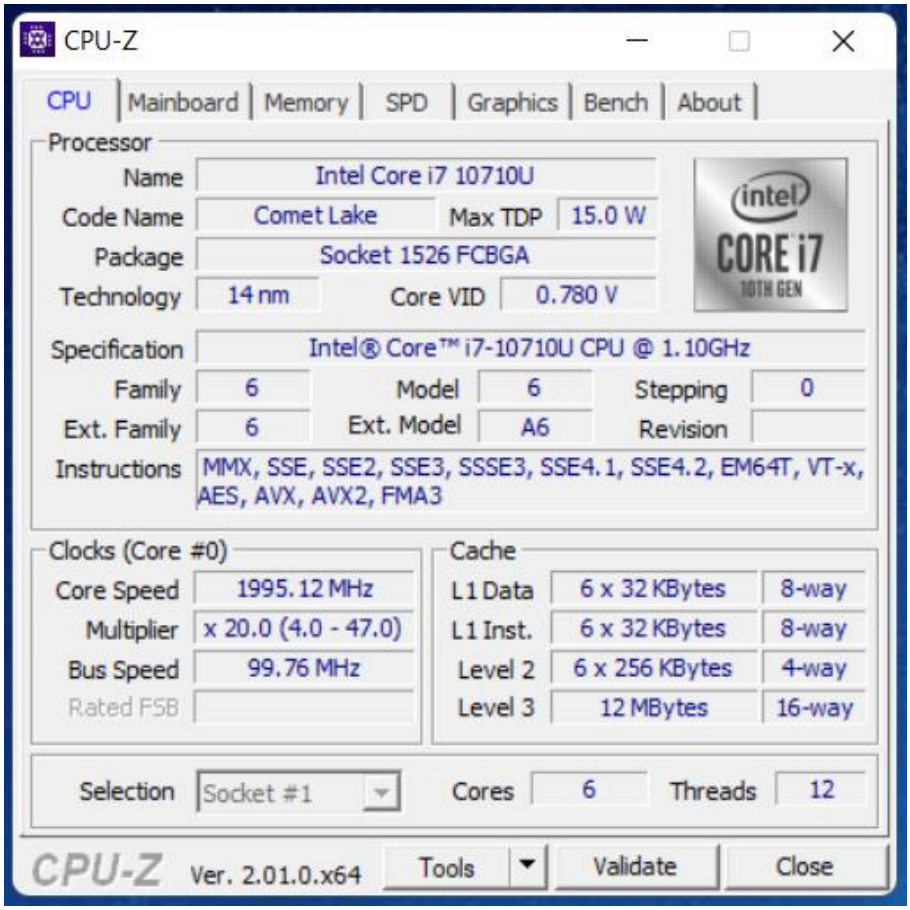


Figure 1: CPU-Z Specs window

In figure 2, we can see the project directory for this project.

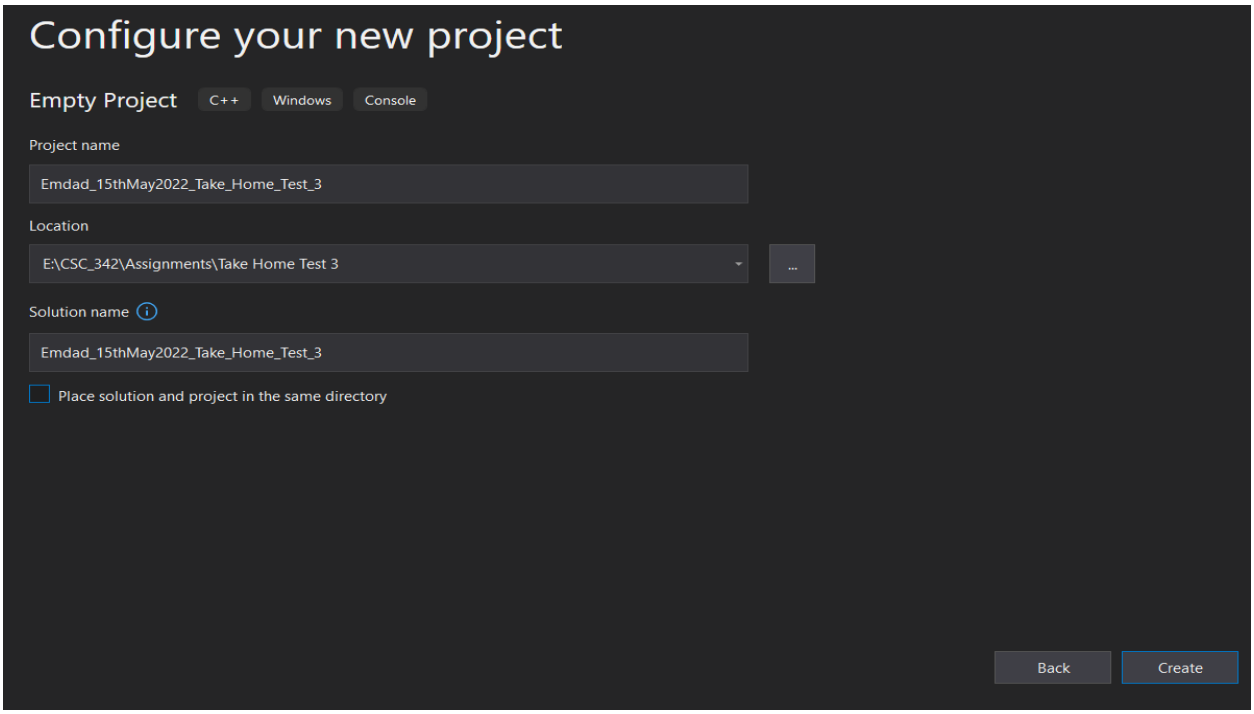


Figure 2: project directory

Dot Product [Non-Optimized]:

In figure 2, we can see the project directory for Emdad_Source.cpp file.

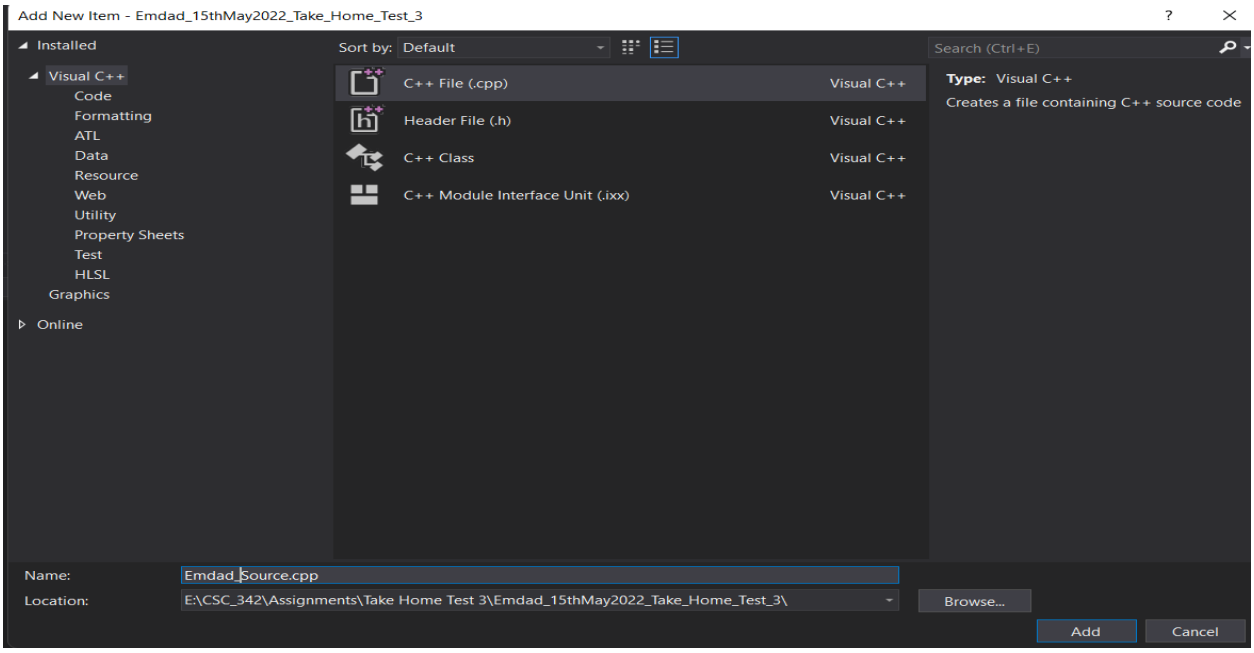


Figure 3: project directory for Emdad_Source.cpp file.

In figure 3, we can see that non optimized c++ code for dot pointer, source code.

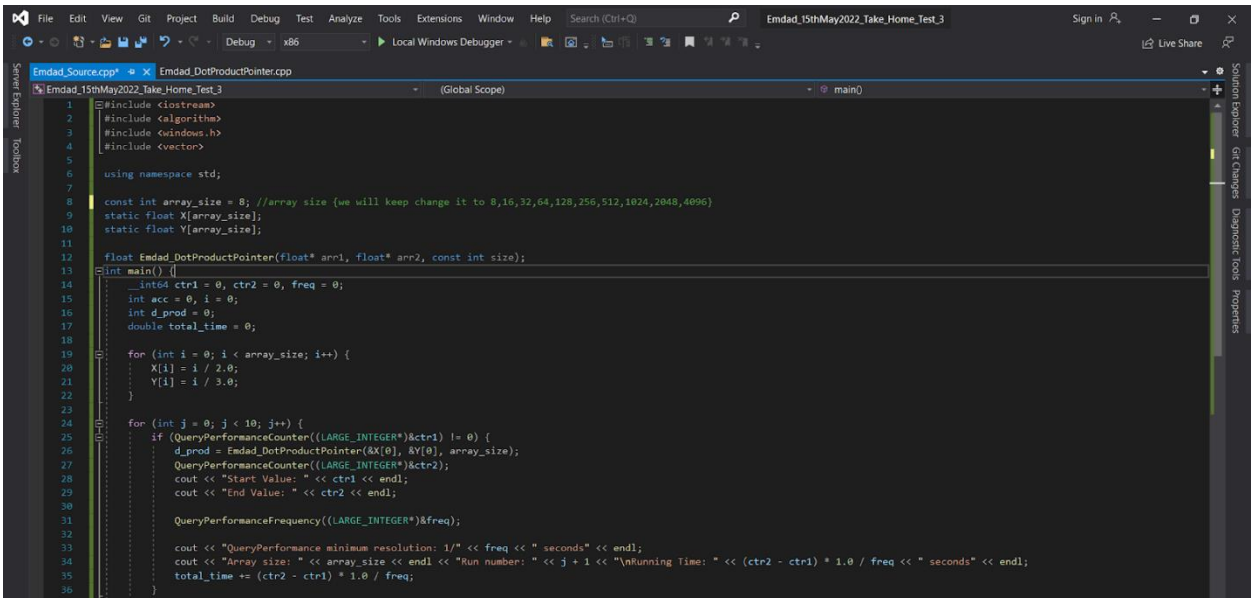


Figure 4: Source code.cpp (Part 1)

Figure 5: Source code.cpp (Part 2)

I used array pointer arithmetic dot production calculation to get the runtime for each run and different array sizes. I manually changed the array size starting from 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096. I also have an average time calculator to find the most accurate time to take from the 10 different runtimes. I used "**QueryPerformanceCounter**" for the runtime calculation.

In figure 6, we can see the project directory for Emdad_DotProductPointer.cpp file.

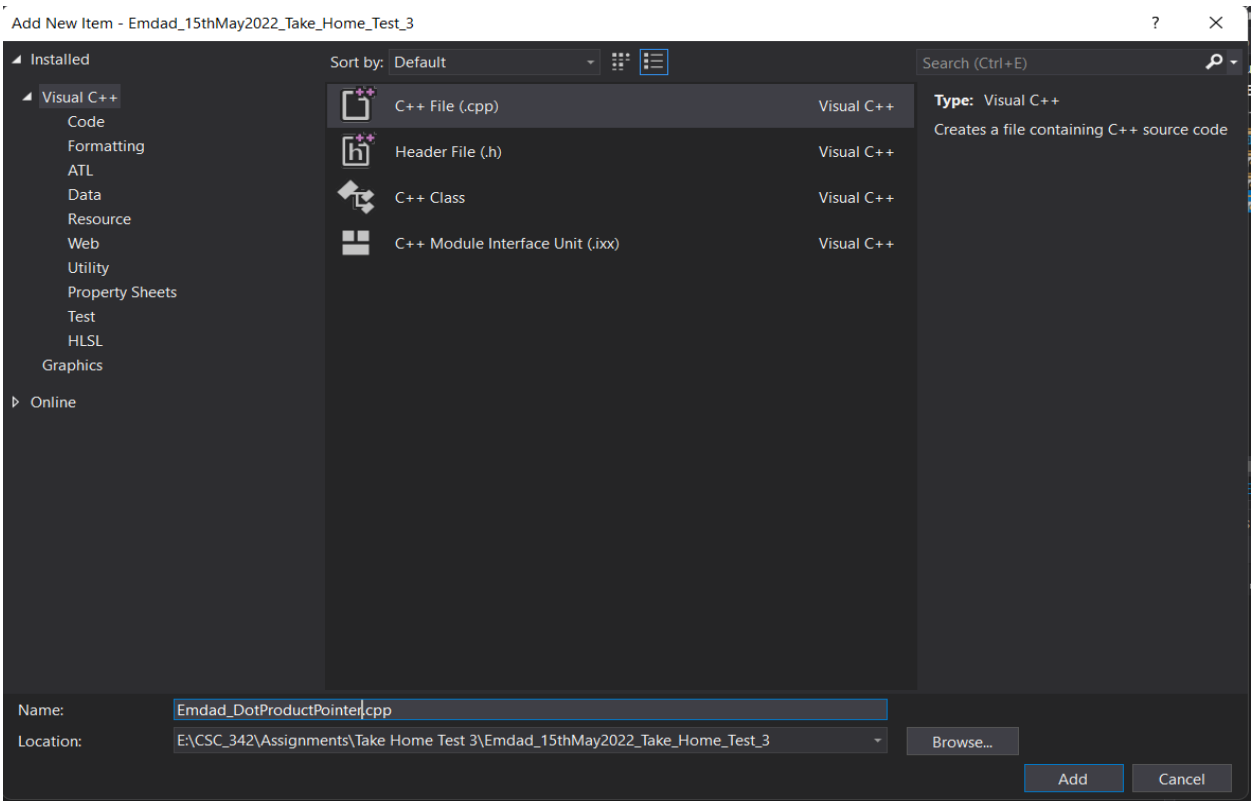


Figure 6: project directory for Emdad_DotProductPointer.cpp

In figure 6, we can see that non optimized c++ code for dot pointer constructor.

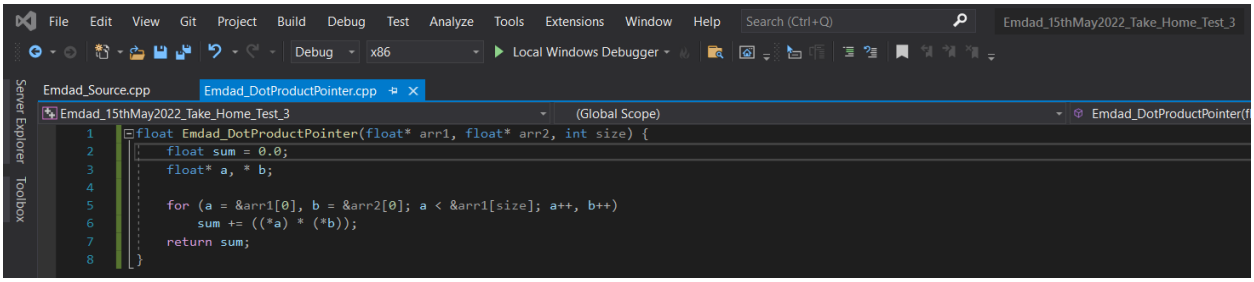


Figure 7: Emdad_DotProductPointer.cpp

Pointer's arithmetic is faster than indexing because indices require multiplication of 4 for each increment which can cause more runtime. It also skips the multiplication step.

In figure 8, we can see the programs compiled without any issues.

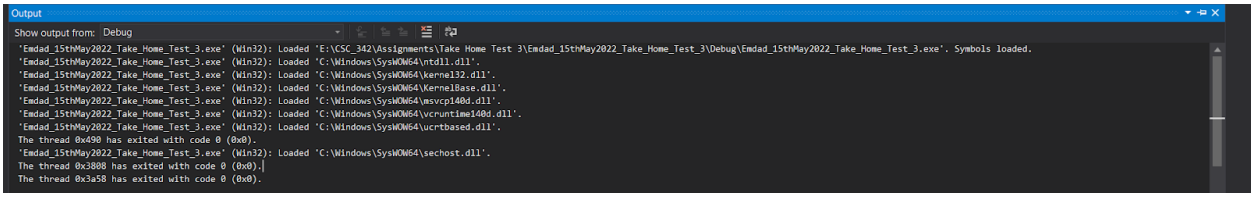


Figure 8: Program compiled

Dot Product [Compiler Generated Optimization]:

In figure 9 and 10, we can see that the file setup to generate asm file from the dotpointer c++ file.

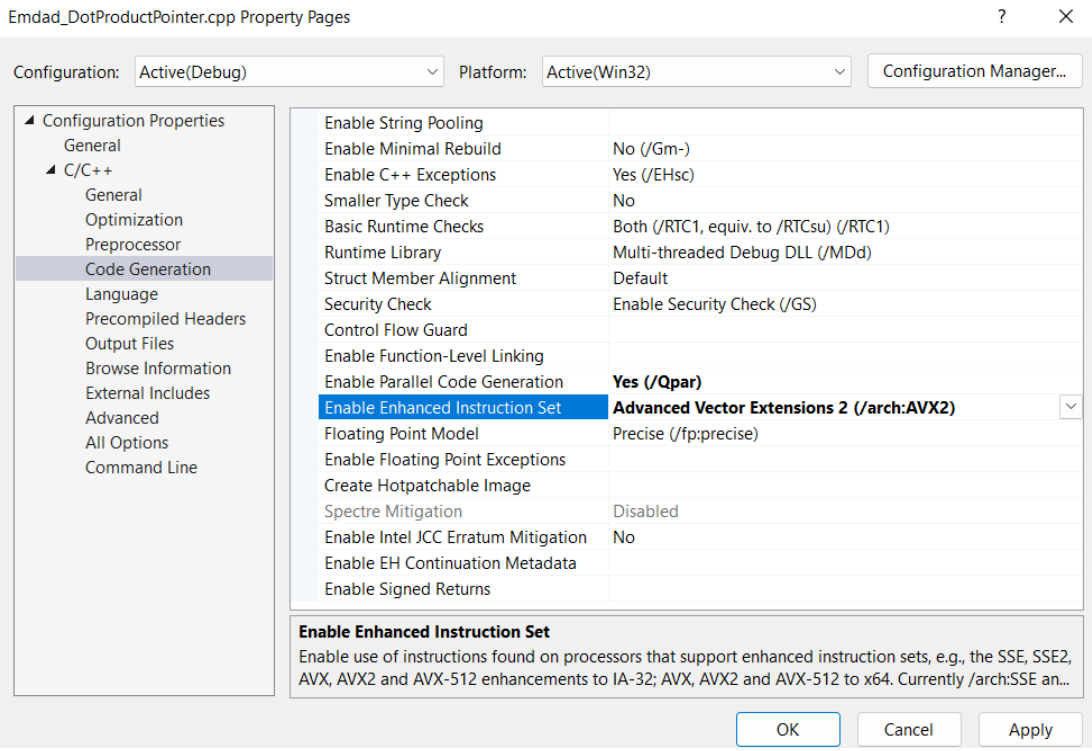


Figure 9: Code Generation Setup

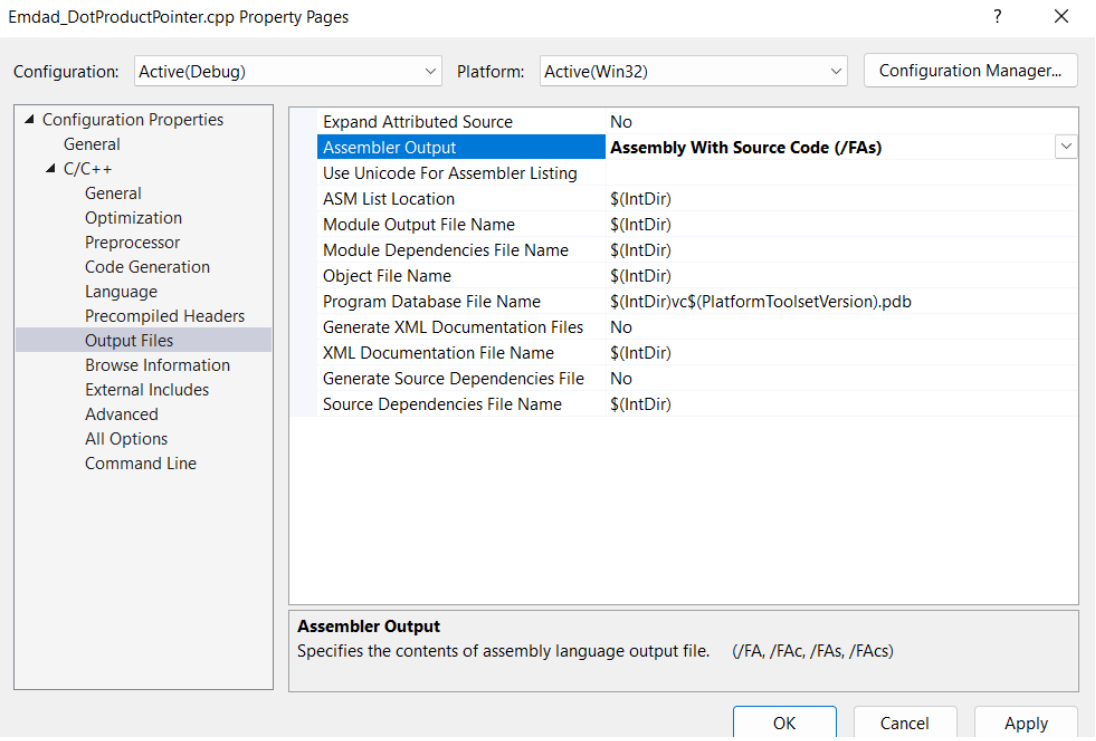


Figure 10: Output files extension

In figure 11, we can see the generated asm file from the source code by the program.

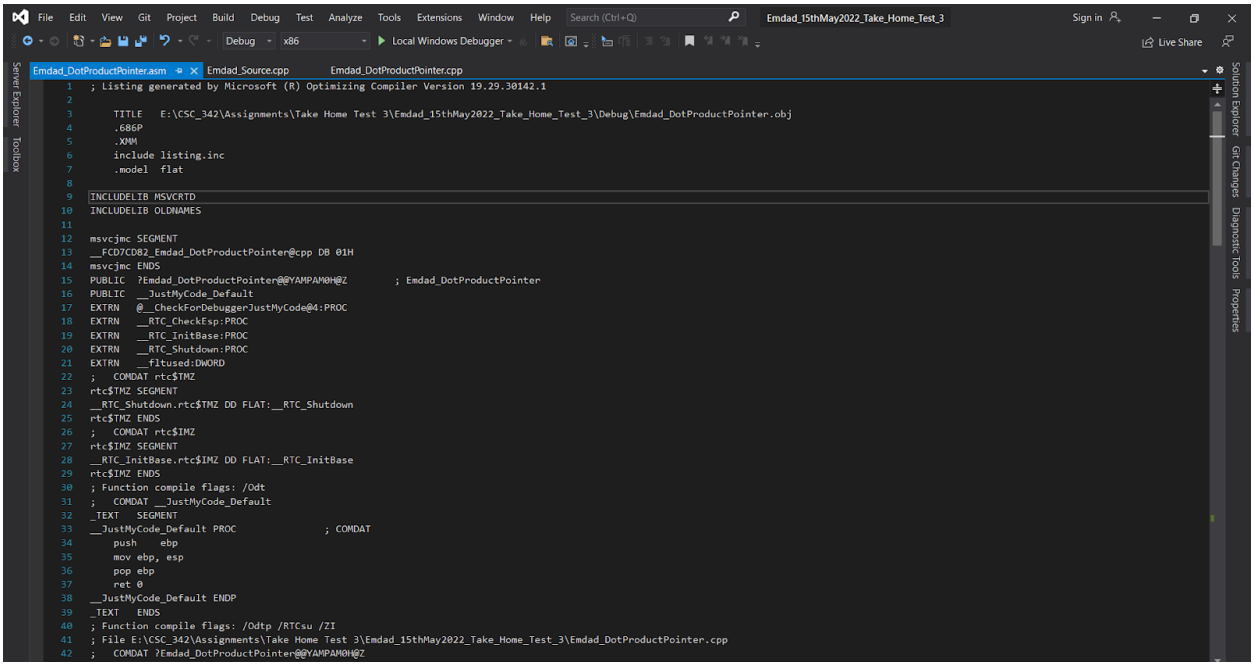
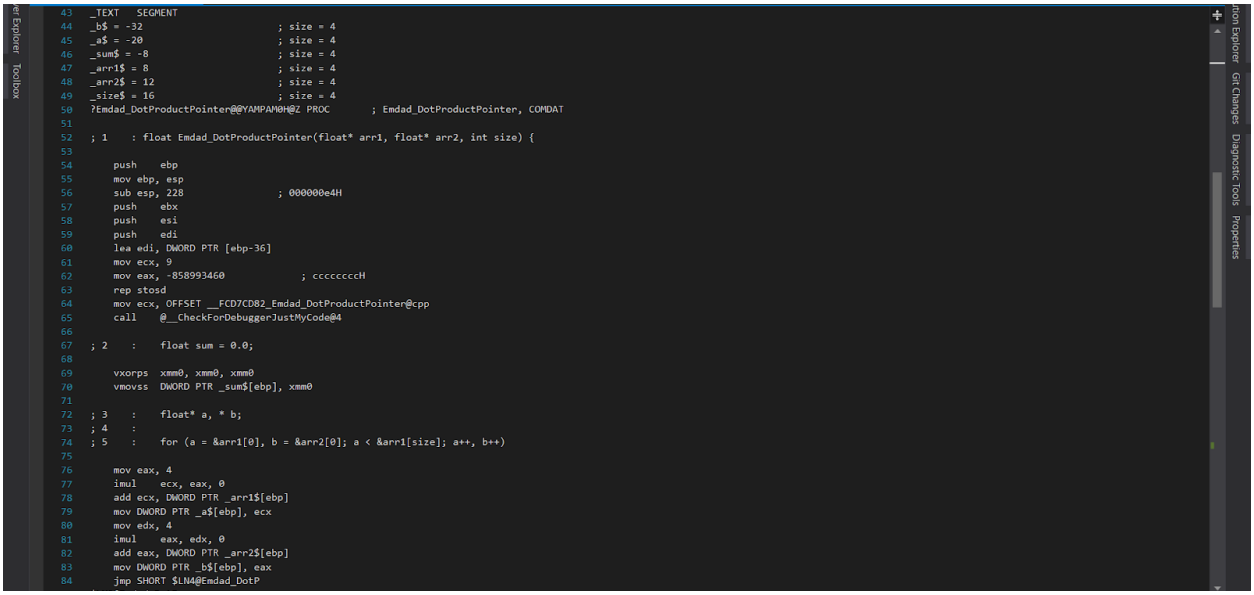


Figure 11: Emdad_DotProductPointer.asm (Part 1)



```
_TEXT   SEGMENT
_b$ = -32                       ; size = 4
_a$ = -20                       ; size = 4
_sum$ = -8                      ; size = 4
_arr1$ = 8                      ; size = 4
_arr2$ = 12                     ; size = 4
_size$ = 16                     ; size = 4
?Emdad_DotProductPointer@@YAMPAM0H@Z PROC       ; Emdad_DotProductPointer, COMDAT

; 1    : float Emdad_DotProductPointer(float* arr1, float* arr2, int size) {

    push    ebp
    mov ebp, esp
    sub esp, 228                ; 000000e4H
    push    ebx
    push    esi
    push    edi
    lea edi, DWORD PTR [ebp-36]
    mov ecx, 9
    mov eax, -858993460             ; ccccccccH
    rep stosd
    mov ecx, OFFSET __FCD7CD82_Emdad_DotProductPointer@cpp
    call    @__CheckForDebuggerJustMyCode@4

; 2    :     float sum = 0.0;

    vxorps  xmm0, xmm0, xmm0
    vmovss  DWORD PTR _sum$[ebp], xmm0

; 3    :     float* a, * b;
; 4    :
; 5    :     for (a = &arr1[0], b = &arr2[0]; a < &arr1[size]; a++, b++)

    mov eax, 4
    imul    ecx, eax, 0
    add ecx, DWORD PTR _arr1$[ebp]
    mov DWORD PTR _a$[ebp], ecx
    mov edx, 4
    imul    eax, edx, 0
    add eax, DWORD PTR _arr2$[ebp]
    mov DWORD PTR _b$[ebp], eax
    jmp SHORT $LN4@Emdad_DotP
```

Figure 12: Emdad_DotProductPointer.asm (Part 2)

```
$LN2@Emdad_DotP:
    mov eax, DWORD PTR _a$[ebp]
    add eax, 4
    mov DWORD PTR _a$[ebp], eax
    mov ecx, DWORD PTR _b$[ebp]
    add ecx, 4
    mov DWORD PTR _b$[ebp], ecx
$LN4@Emdad_DotP:
    mov eax, DWORD PTR _size$[ebp]
    mov ecx, DWORD PTR _arr1$[ebp]
    lea edx, DWORD PTR [ecx+eax*4]
    cmp DWORD PTR _a$[ebp], edx
    jae SHORT $LN3@Emdad_DotP

; 6    :         sum += ((*a) * (*b));

    mov eax, DWORD PTR _a$[ebp]
    mov ecx, DWORD PTR _b$[ebp]
    vmovss  xmm0, DWORD PTR [eax]
    vmulss  xmm0, xmm0, DWORD PTR [ecx]
    vmovss  xmm1, DWORD PTR _sum$[ebp]
    vaddss  xmm0, xmm1, xmm0
    vmovss  DWORD PTR _sum$[ebp], xmm0
    jmp SHORT $LN2@Emdad_DotP
$LN3@Emdad_DotP:

; 7    :     return sum;

    fld DWORD PTR _sum$[ebp]

; 8    : }

    pop edi
    pop esi
    pop ebx
    add esp, 228                ; 000000e4H
    cmp ebp, esp
    call    __RTC_CheckEsp
    mov esp, ebp
    pop ebp
    ret 0
?Emdad_DotProductPointer@@YAMPAM0H@Z ENDP       ; Emdad_DotProductPointer
```

Figure 13: Emdad_DotProductPointer.asm (Part 3)

```
_TEXT   ENDS
END
```

100 % No issues found Ln: 119 Ch: 9 Col: 12 TABS CRLF

Ready Add to Source Control

Figure 14: Emdad_DotProductPointer.asm (Part 4)

In figure 15, we can see the programs compiled without any issues.

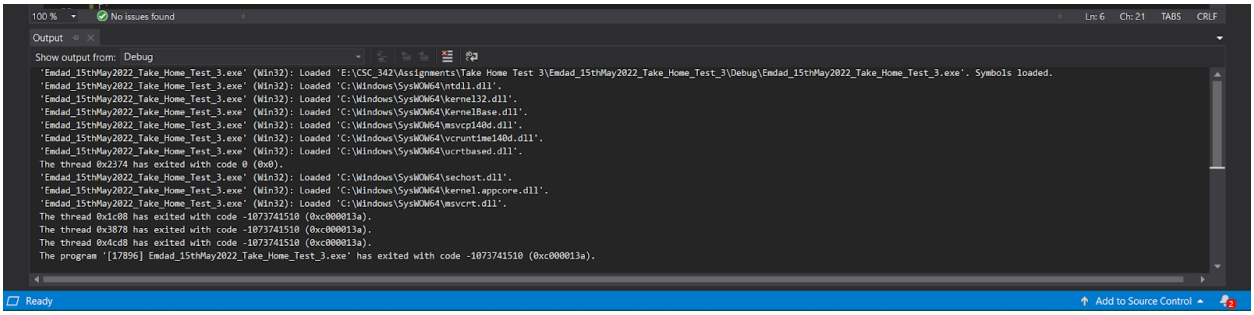


Figure 15: program compilation

Dot Product [Manually Optimized]:

In figure 16, we can see the optimized code for the Emdad_DotProductPointer.asm file. In this code, I removed some of the unnecessary lines which was costing more run time. I removed some repeated lines and moved the code out of loop for less runtime. I used parallelization and vectorization for this.

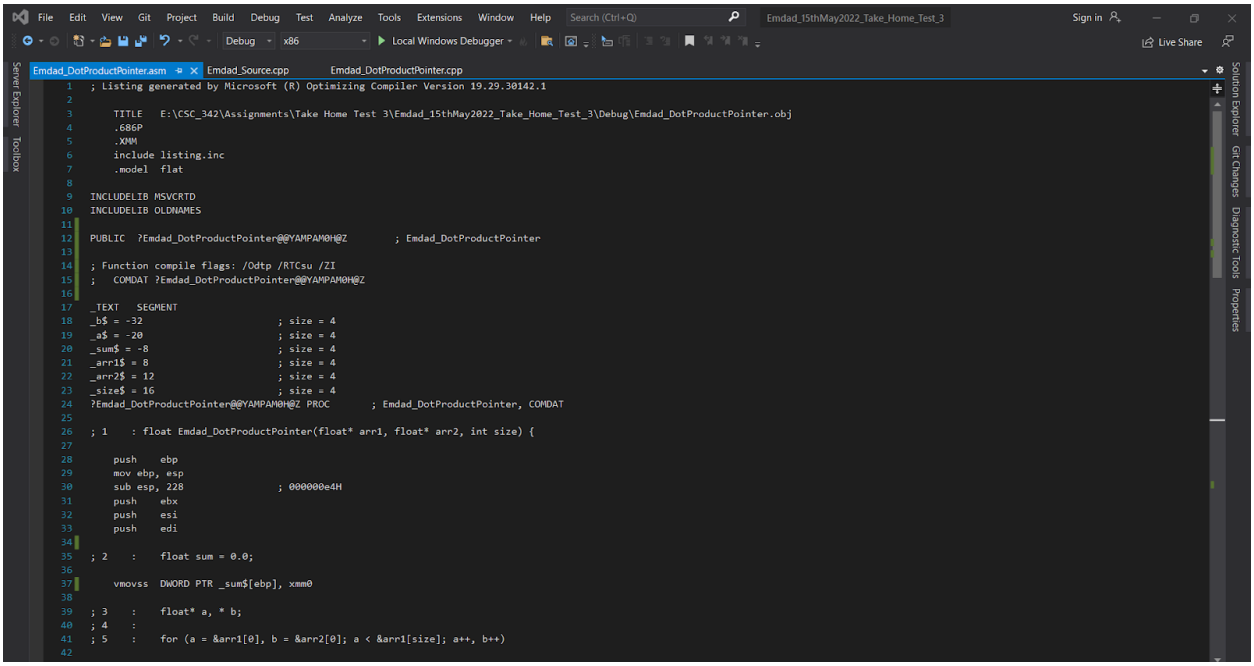
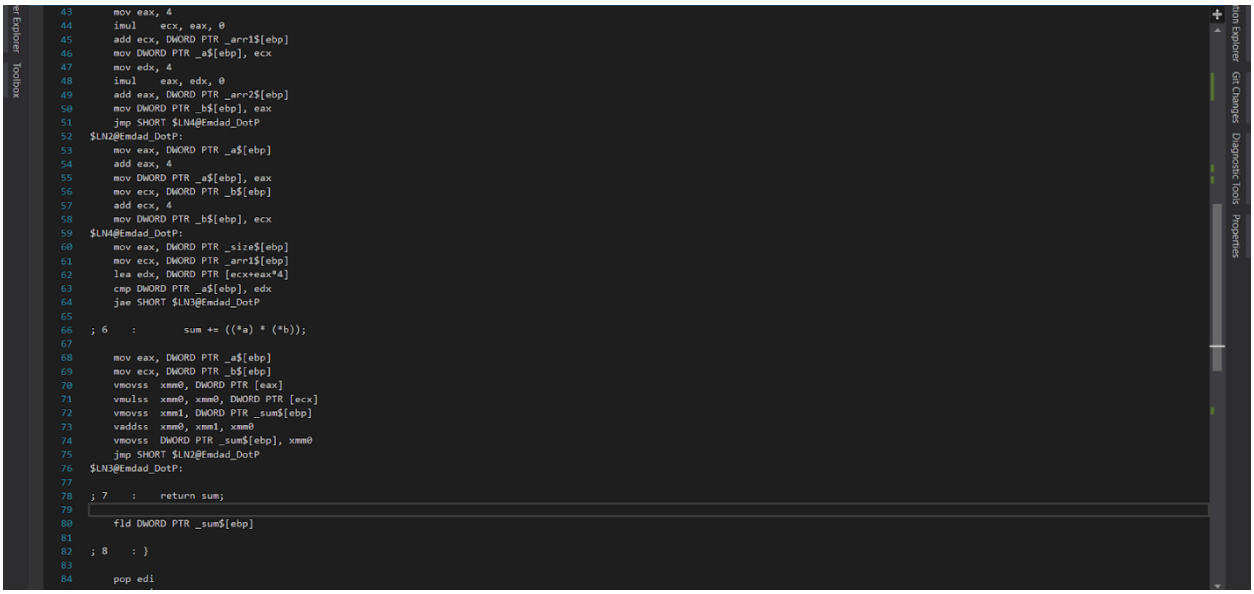


Figure 16: Emdad_DotProductPointer.asm (Part 1)



```
    mov eax, 4
    imul    ecx, eax, 0
    add ecx, DWORD PTR _arr1$[ebp]
    mov DWORD PTR _a$[ebp], ecx
    mov edx, 4
    imul    eax, edx, 0
    add eax, DWORD PTR _arr2$[ebp]
    mov DWORD PTR _b$[ebp], eax
    jmp SHORT $LN4@Emdad_DotP
$LN2@Emdad_DotP:
    mov eax, DWORD PTR _a$[ebp]
    add eax, 4
    mov DWORD PTR _a$[ebp], eax
    mov ecx, DWORD PTR _b$[ebp]
    add ecx, 4
    mov DWORD PTR _b$[ebp], ecx
$LN4@Emdad_DotP:
    mov eax, DWORD PTR _size$[ebp]
    mov ecx, DWORD PTR _arr1$[ebp]
    lea edx, DWORD PTR [ecx+eax*4]
    cmp DWORD PTR _a$[ebp], edx
    jae SHORT $LN3@Emdad_DotP

; 6    :        sum += ((*a) * (*b));

    mov eax, DWORD PTR _a$[ebp]
    mov ecx, DWORD PTR _b$[ebp]
    vmovss  xmm0, DWORD PTR [eax]
    vmulss  xmm0, xmm0, DWORD PTR [ecx]
    vmovss  xmm1, DWORD PTR _sum$[ebp]
    vaddss  xmm0, xmm1, xmm0
    vmovss  DWORD PTR _sum$[ebp], xmm0
    jmp SHORT $LN2@Emdad_DotP
$LN3@Emdad_DotP:

; 7    :    return sum;

    fld DWORD PTR _sum$[ebp]

; 8    : }

    pop edi
```

Figure 17: Emdad_DotProductPointer.asm (Part 2)

```
    pop esi
    pop ebx
    add esp, 228            ; 000000e4H
    cmp ebp, esp
    call    __RTC_CheckEsp
    mov esp, ebp
    pop ebp
    ret 0
?Emdad_DotProductPointer@@YAMPAM0H@Z ENDP       ; Emdad_DotProductPointer
_TEXT   ENDS
END
```

100 % | No issues found | Ln: 79 | Ch: 1 | TABS | CRLF

Ready | Add to Source Control

Figure 18: Emdad_DotProductPointer.asm (Part 3)

VDPPS Vector Instruction:

In figure 19, we can see the code for VDPPS cpp file.

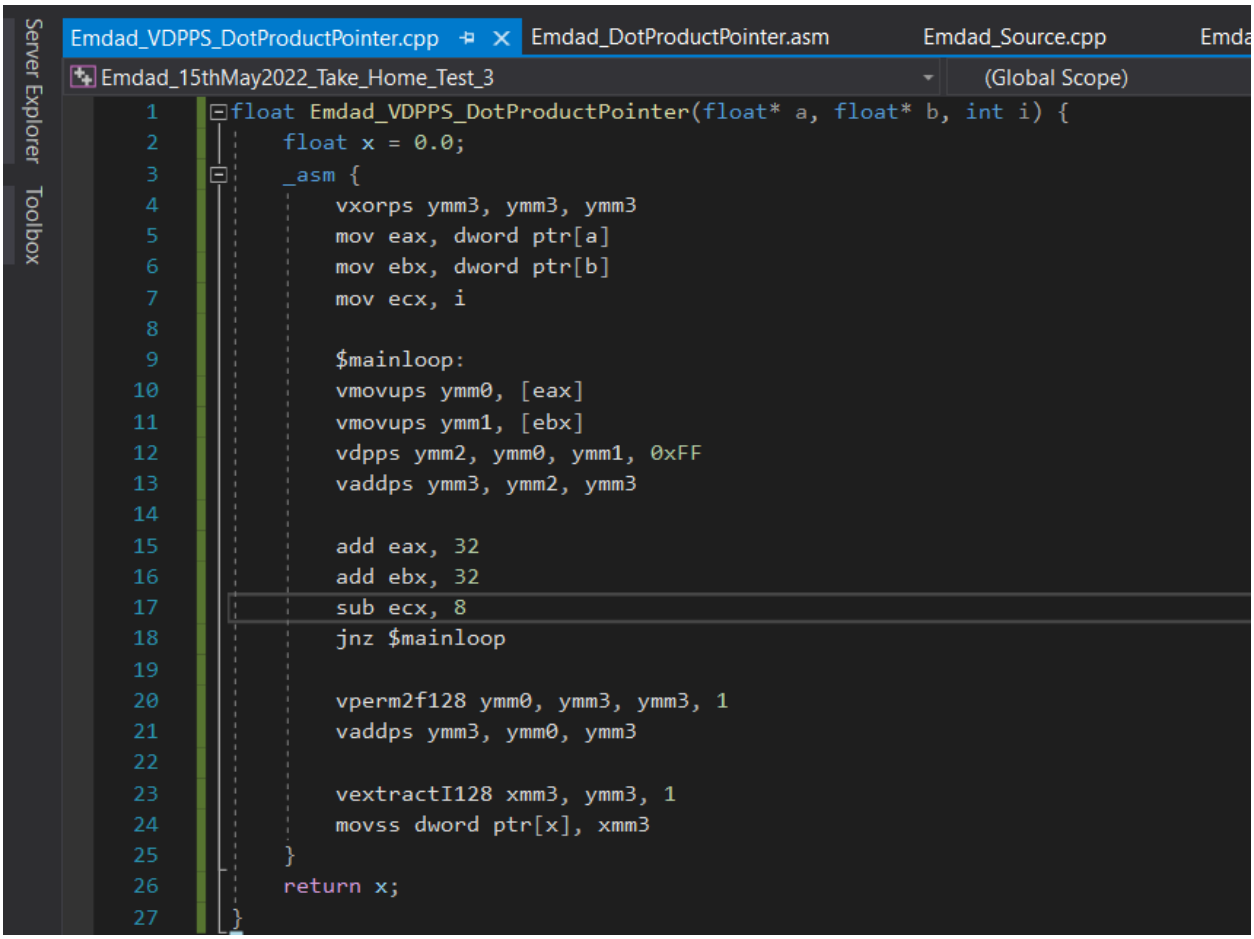


Figure 19: Emdad_VDPPS_DotProductPointer.cpp

I used DPPS vector instruction to optimize the dot product pointer code further. DPPS do not require any unnecessary addition or multiplication to execute the programs in multiple runs. DPPS does all the operations at once and give the output at the end.

In figure 20, we can see the programs compiled without any issues.

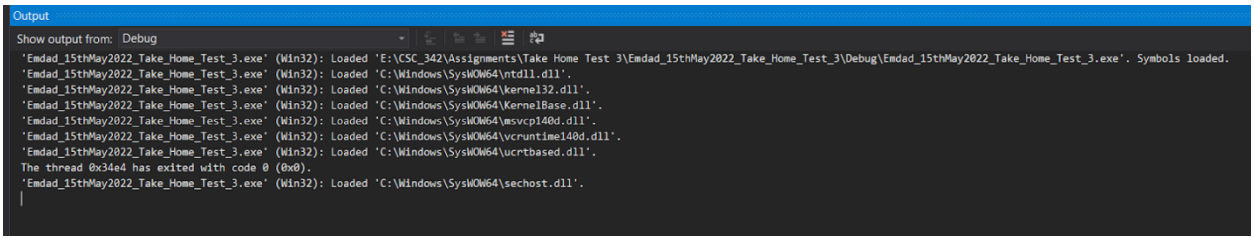


Figure 20: Program Compilation

Intel x64 (Linux g++):

CPU-X specifications:

In figure 21, we can see the CPU-X window with specifications, the highest instructions I can use the AVX2 for asm file generating.

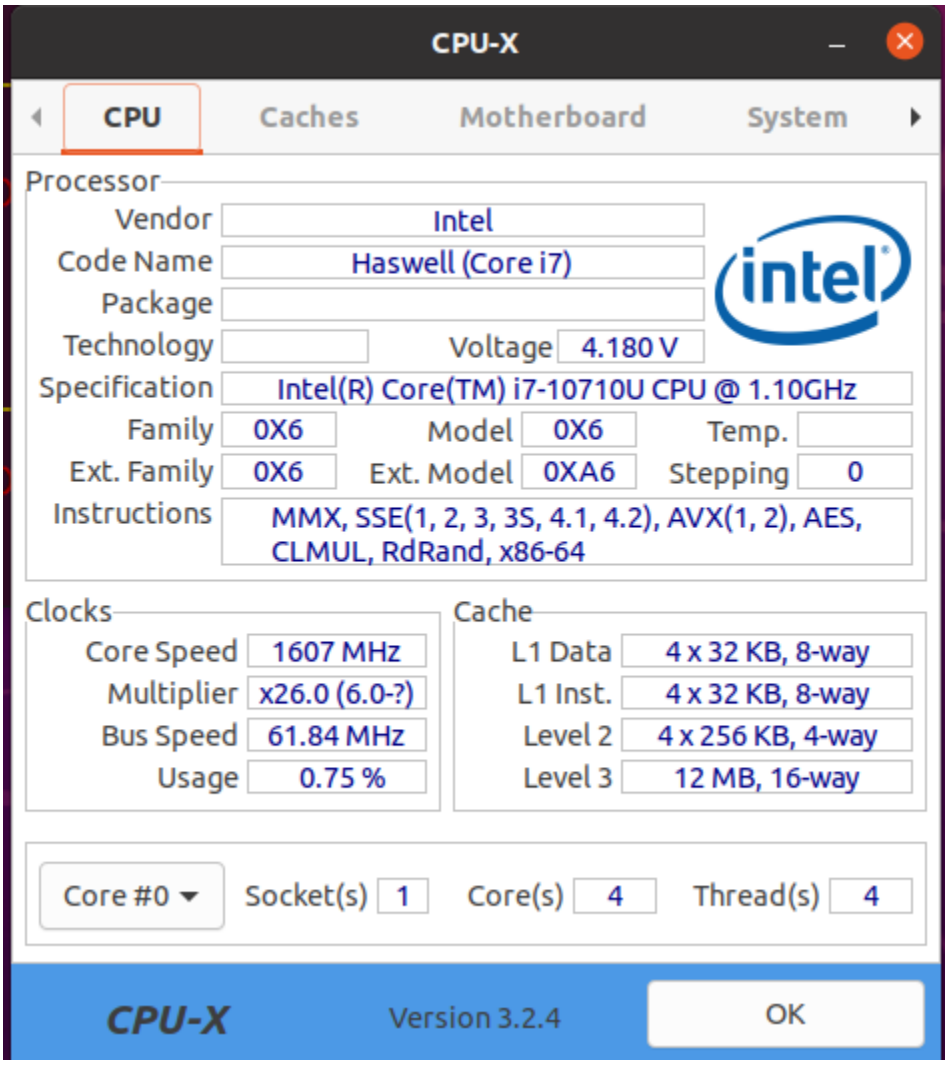


Figure 21: CPU-X Specs window.

I am running Ubuntu in VirtualBox which has different specifications. Therefore, I wanted to have a check by using CPU-X to see if I can use the same instruction in my Linux computer as well.

Dot Product [Non-Optimized]:

In figure 22, we can see the code for Emdad_Source.cpp file.

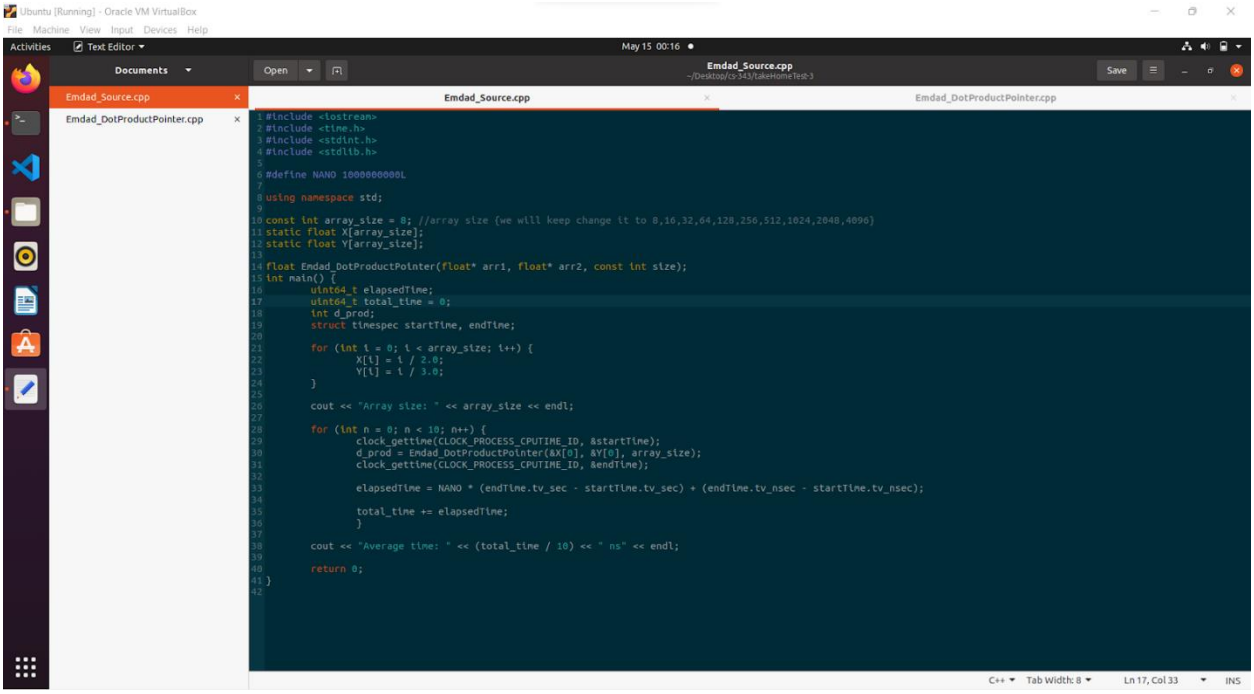


Figure 22: Emdad_Source.cpp

In figure 23, we can see the code for Emdad_DotProductPointer.cpp file.

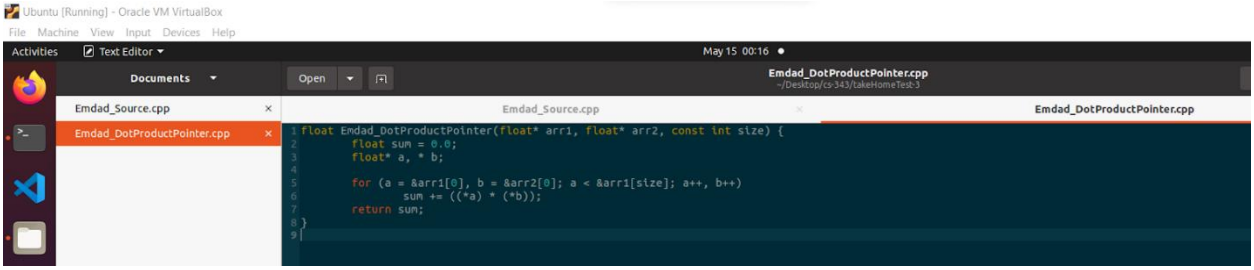


Figure 23: Emdad_DotProductPointer.cpp

Dot Product [Compiler Generated Optimization]:

In figure 24, we can see the Emdad_DotProductPointer.s file which generated from the Emdad_Source.cpp file.

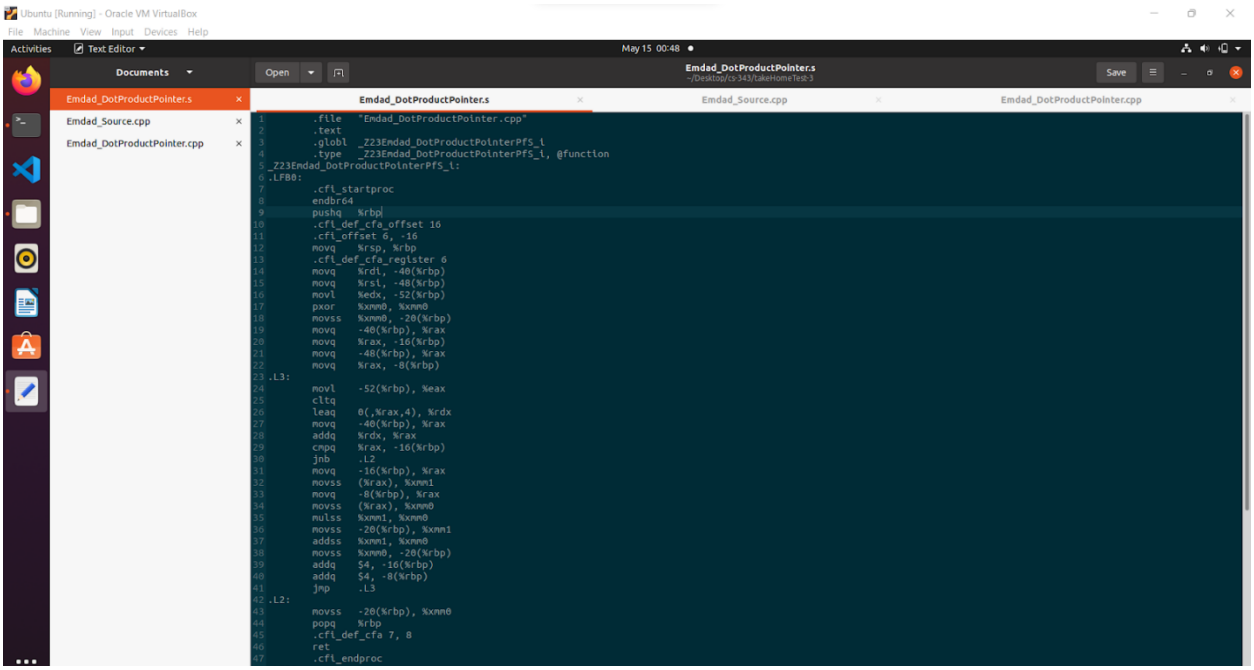


Figure 24: Emdad_DotProductPointer.s (Part 1)

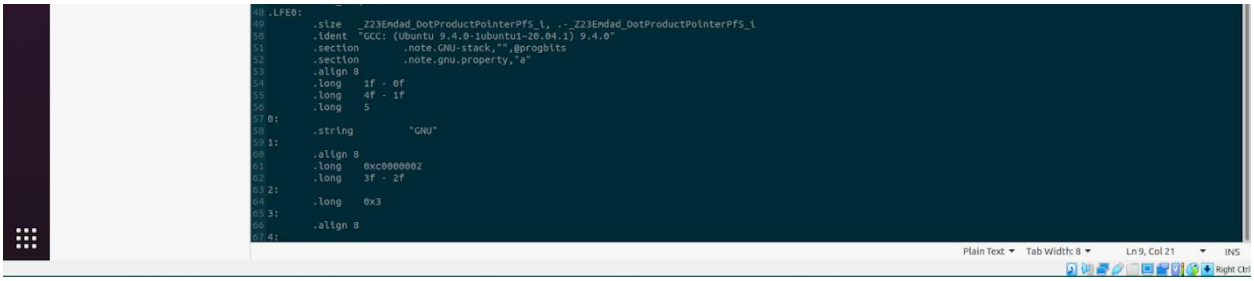


Figure 25: Emdad_DotProductPointer.s (Part 2)

I ran **g++ -S Emdad_DotProductPointer.cpp** to generate the asm file.

Simulation:

Intel x86 (Visual Studio Code):

Dot Product [Non-Optimized]:

In figure 26, we can see the array size average execution running time for array size 8.

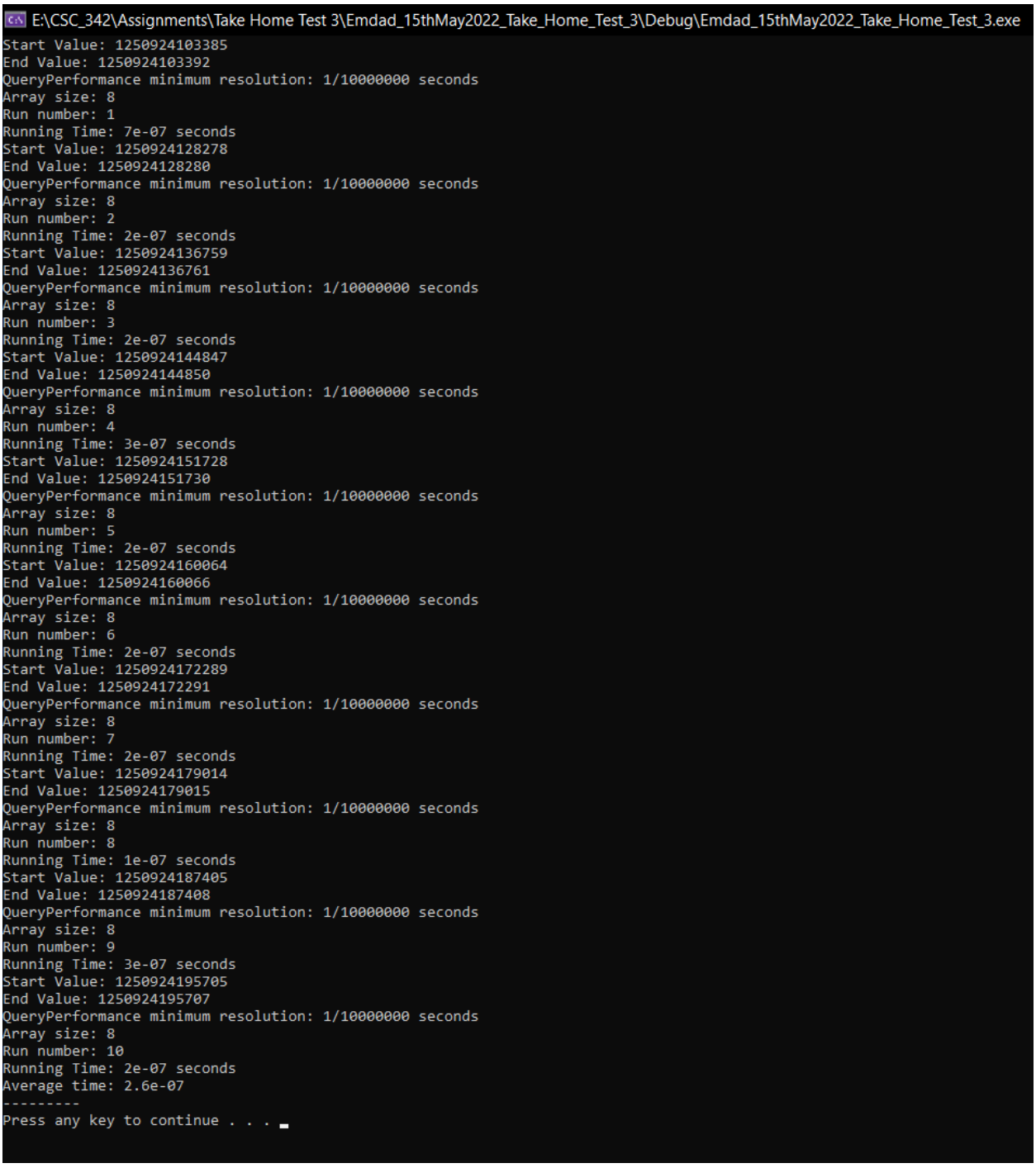


Figure 26: average execution running time for array size 8.

In figure 27, we can see the array size average execution running time for array size 16.

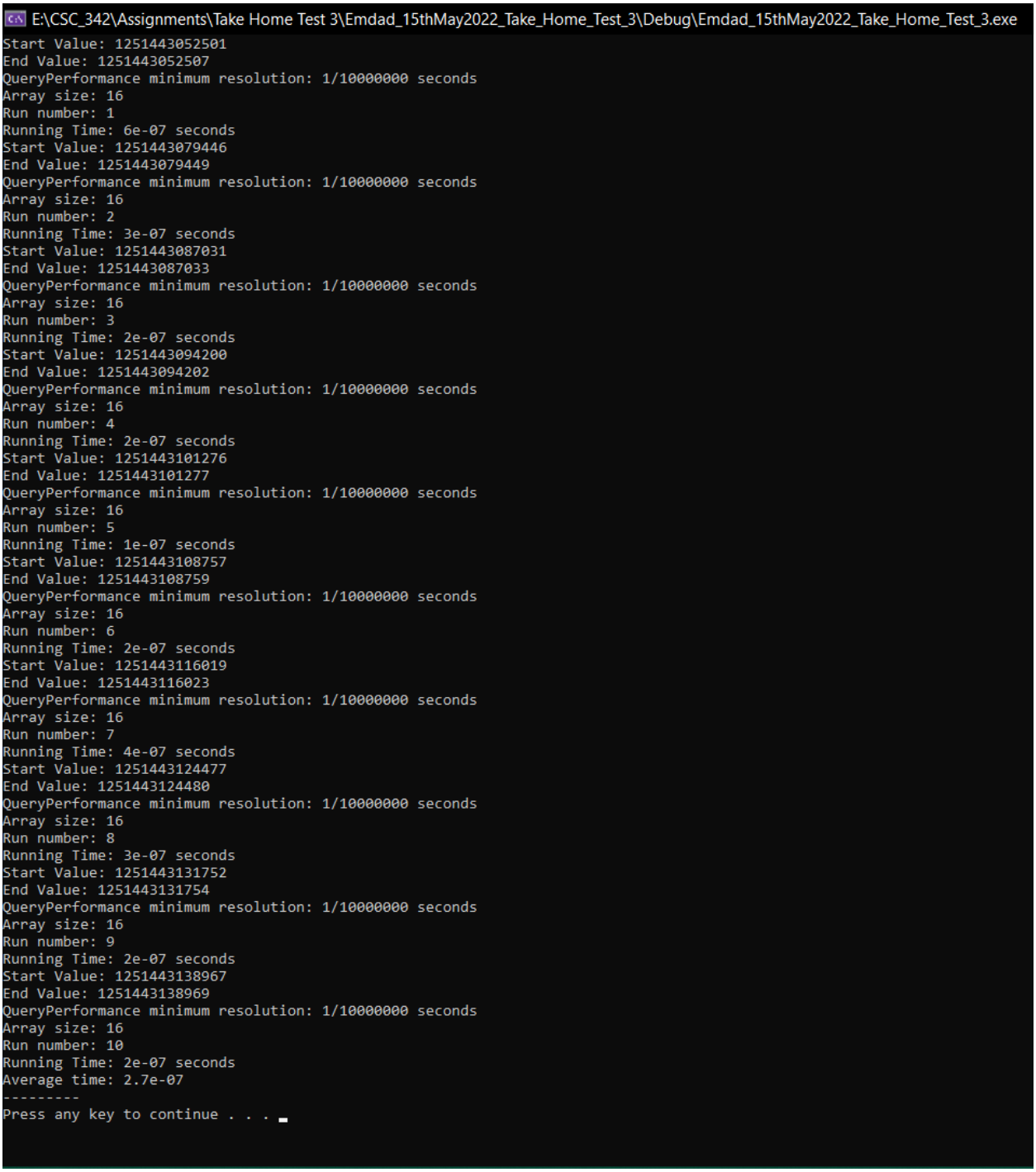


Figure 27: average execution running time for array size 16

In figure 28, we can see the array size average execution running time for array size 32.

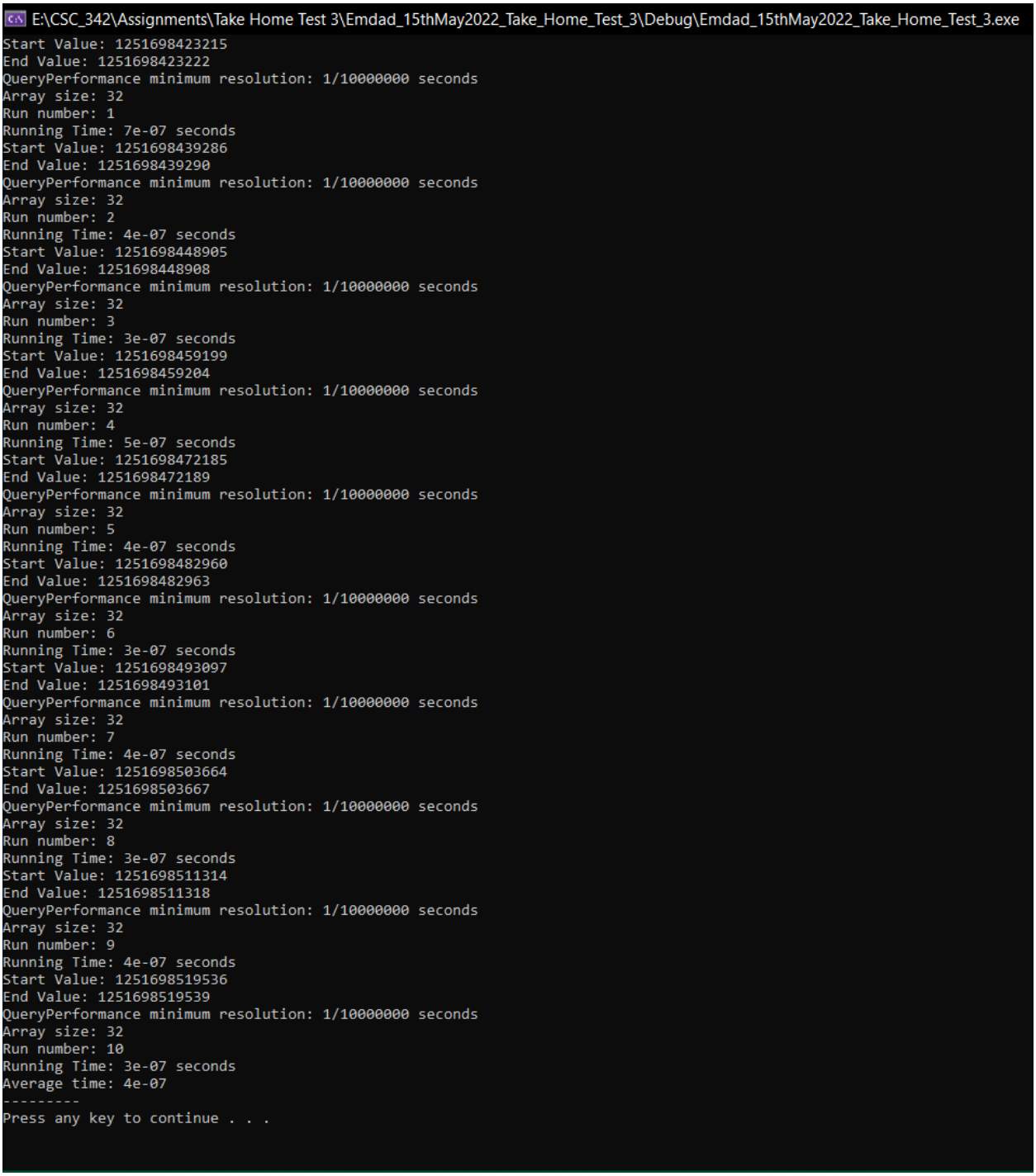


Figure 28: execution running time for array size 32.

In figure 29, we can see the array size average execution running time for array size 64.

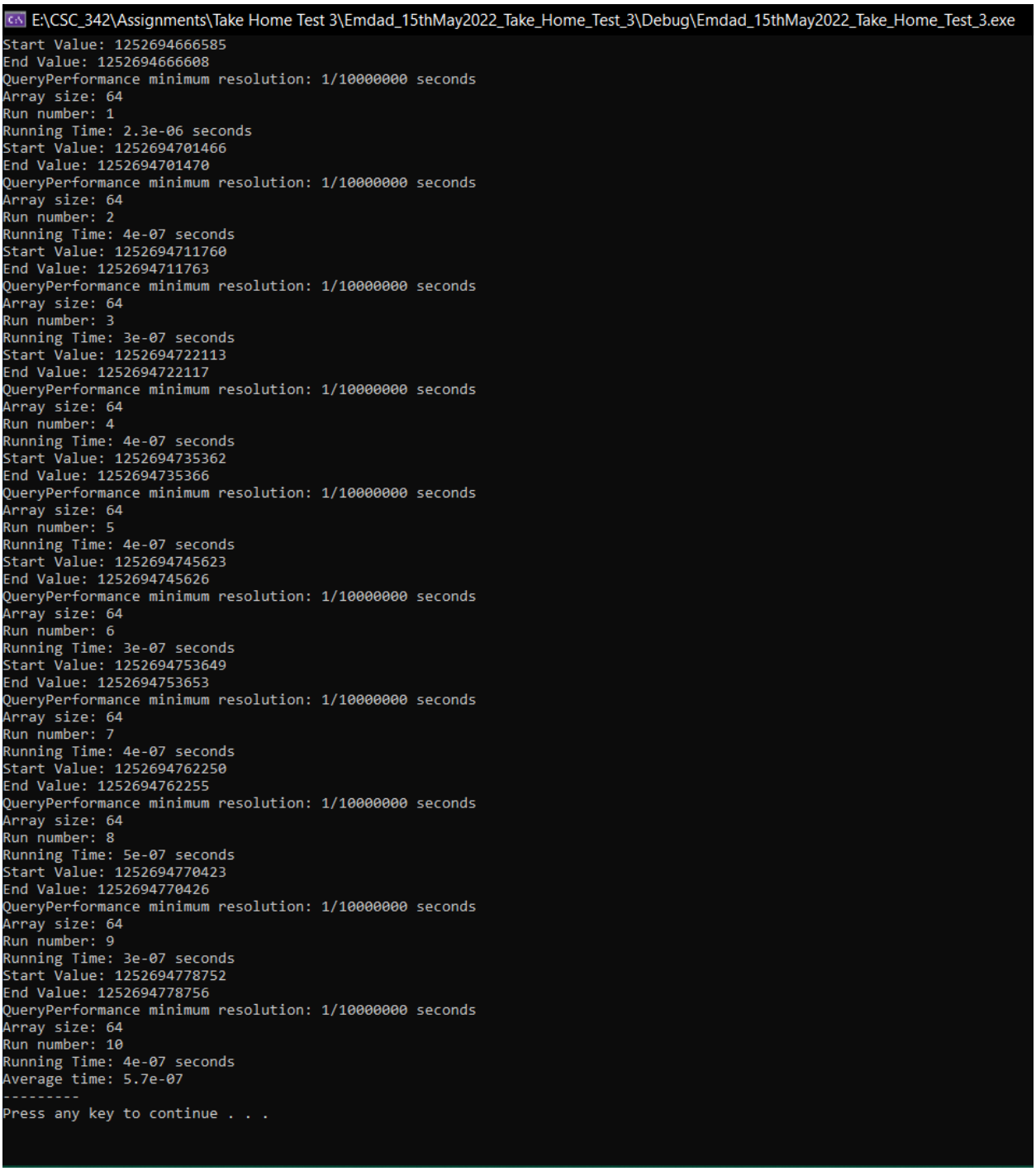


Figure 29: execution running time for array size 64

In figure 30, we can see the array size average execution running time for array size 128.

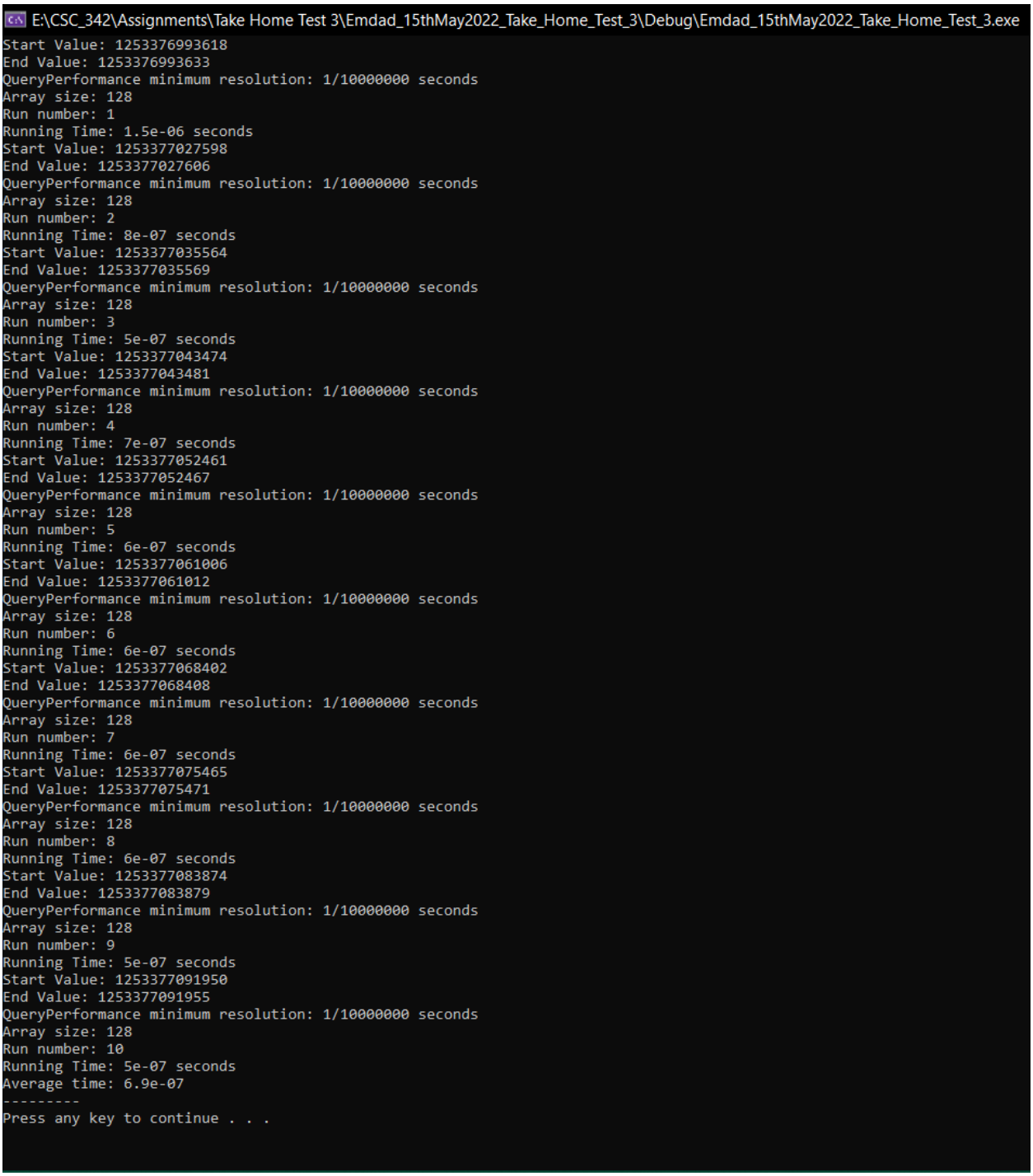


Figure 30: execution running time for array size 128.

In figure 31, we can see the array size average execution running time for array size 256.

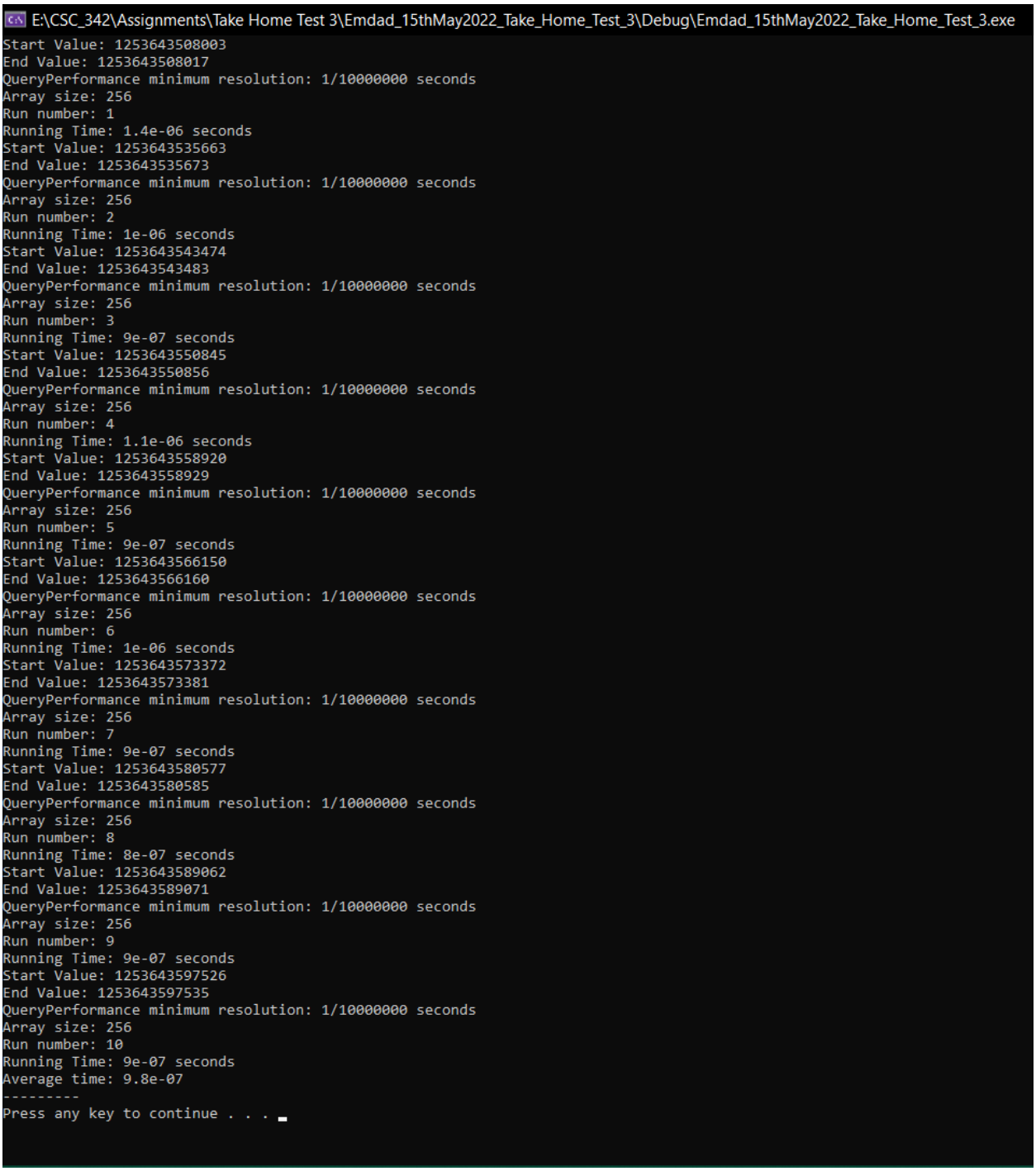


Figure 31: execution running time for array size 256.

In figure 32, we can see the array size average execution running time for array size 512.

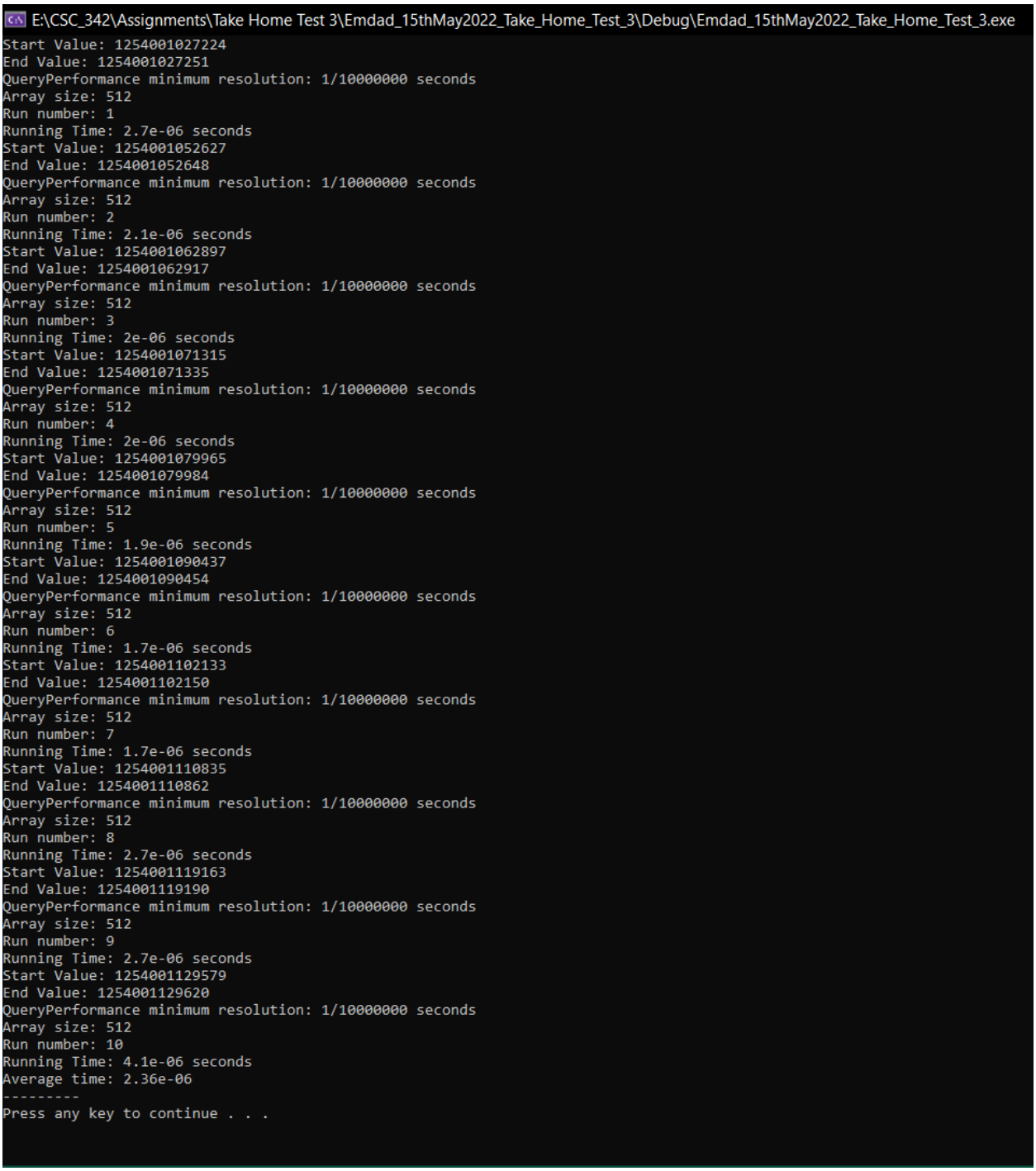


Figure 32: execution running time for array size 512.

In figure 33, we can see the array size average execution running time for array size 1024.

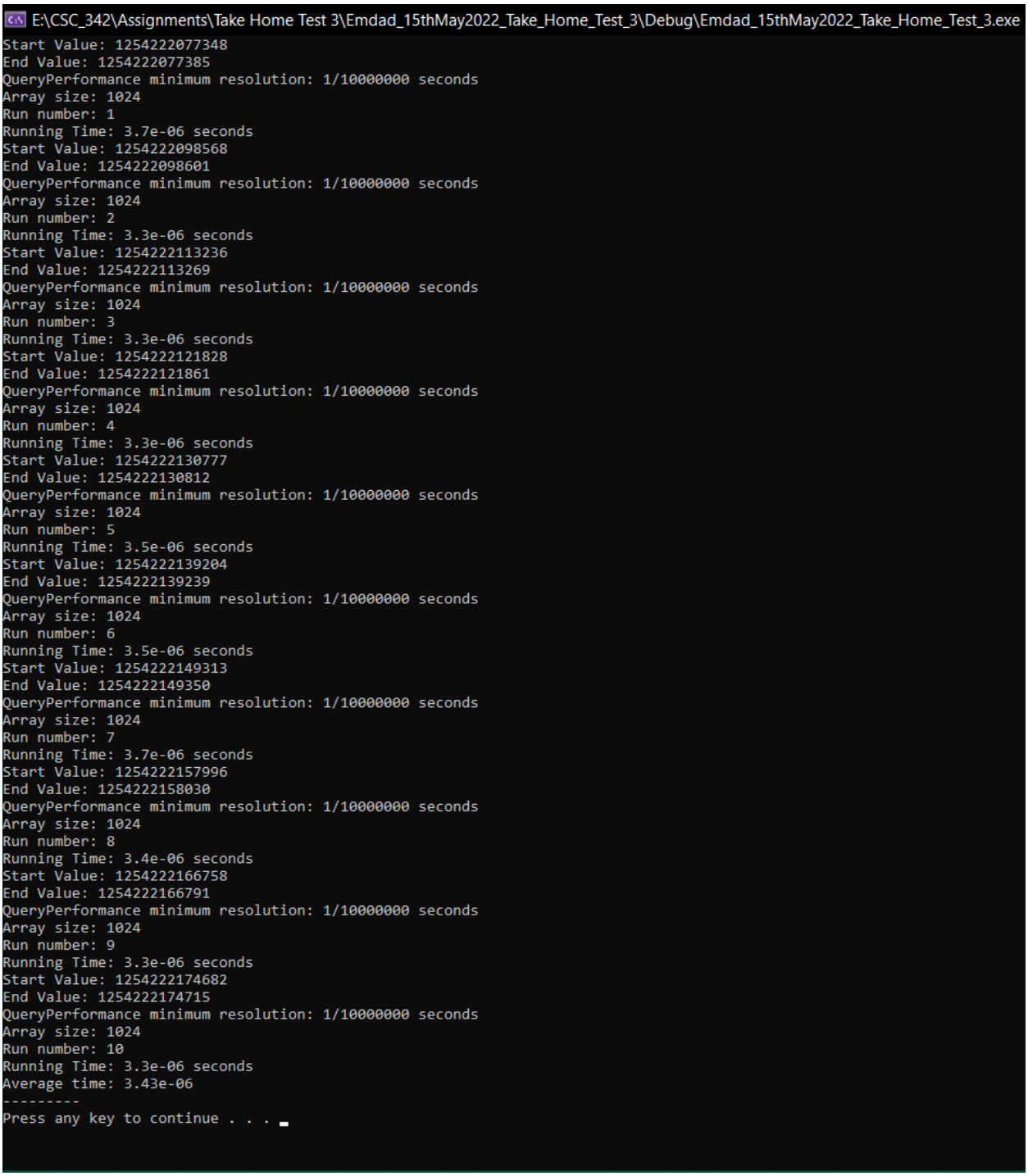


Figure 33: execution running time for array size 1024.

In figure 34, we can see the array size average execution running time for array size 2048.

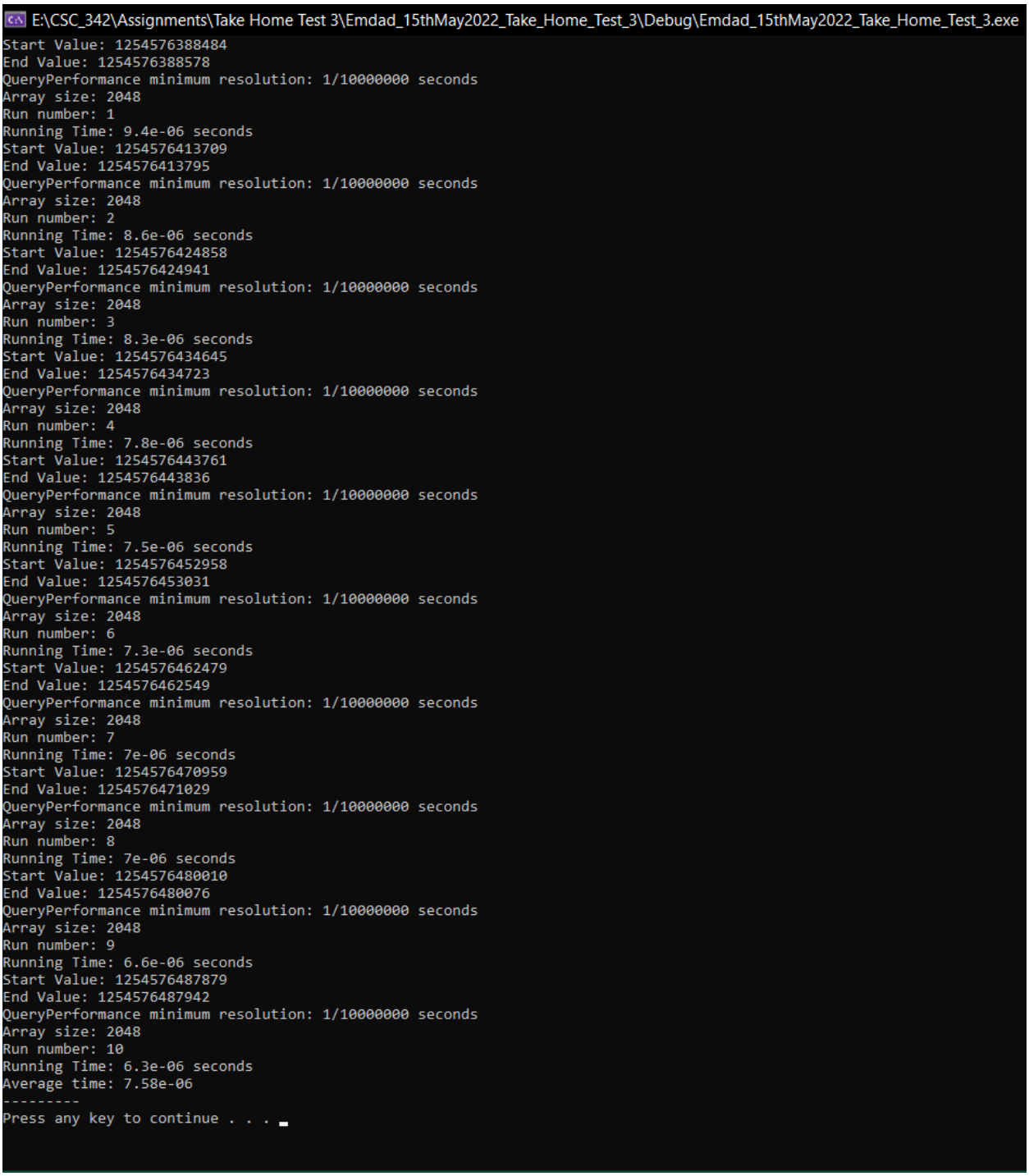


Figure 34: execution running time for array size 2048.

In figure 35, we can see the array size average execution running time for array size 4096.

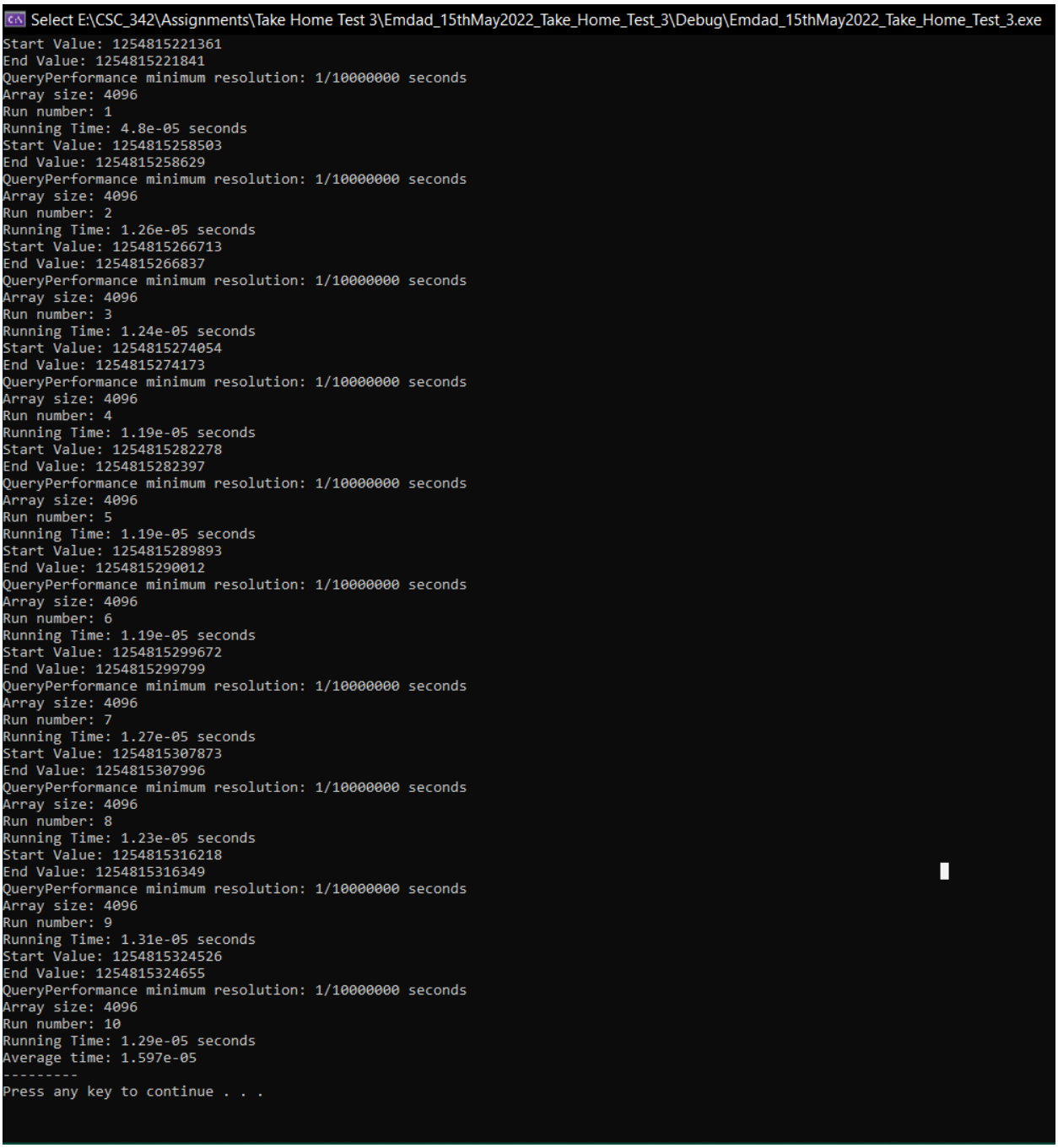


Figure 35: execution running time for array size 4096.

In figure 36, we can see that I made a table for the average runtime based on the array sizes.

Array Size	Average Running Time (seconds)
8	2.60E-07
16	2.70E-07
32	4.00E-07
64	5.70E-07
128	6.90E-07
256	9.80E-07
512	2.36E-06
1024	3.43E-06
2048	7.58E-06
4096	1.60E-05

Figure 36: Average Runtime Table(Non-Optimized)

In figure 37, we can see that I made a graph for the average runtime based on the array sizes. We can how the runtime went as the array size increased.

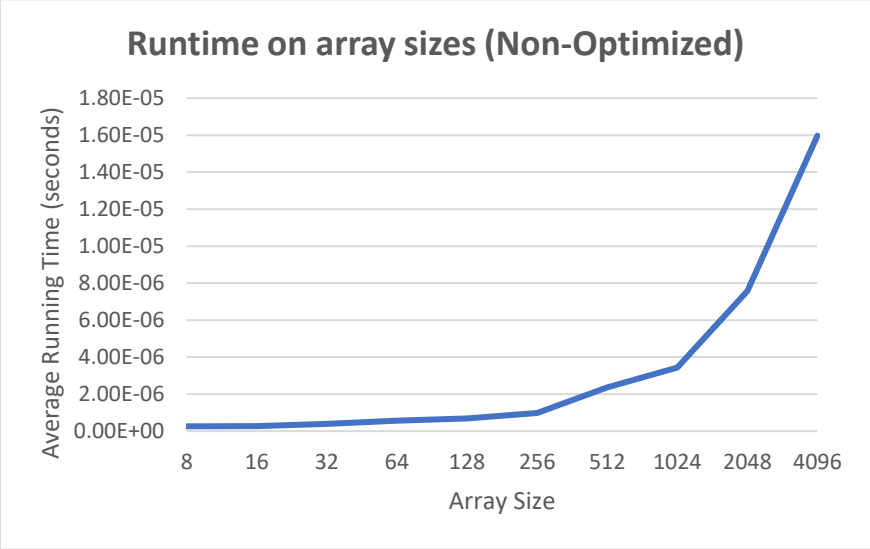


Figure 37: Runtime graph(non-Optimized)

Dot Product [Compiler Generated Optimization]:

In figure 38, we can see the array size average execution running time for array size 8.

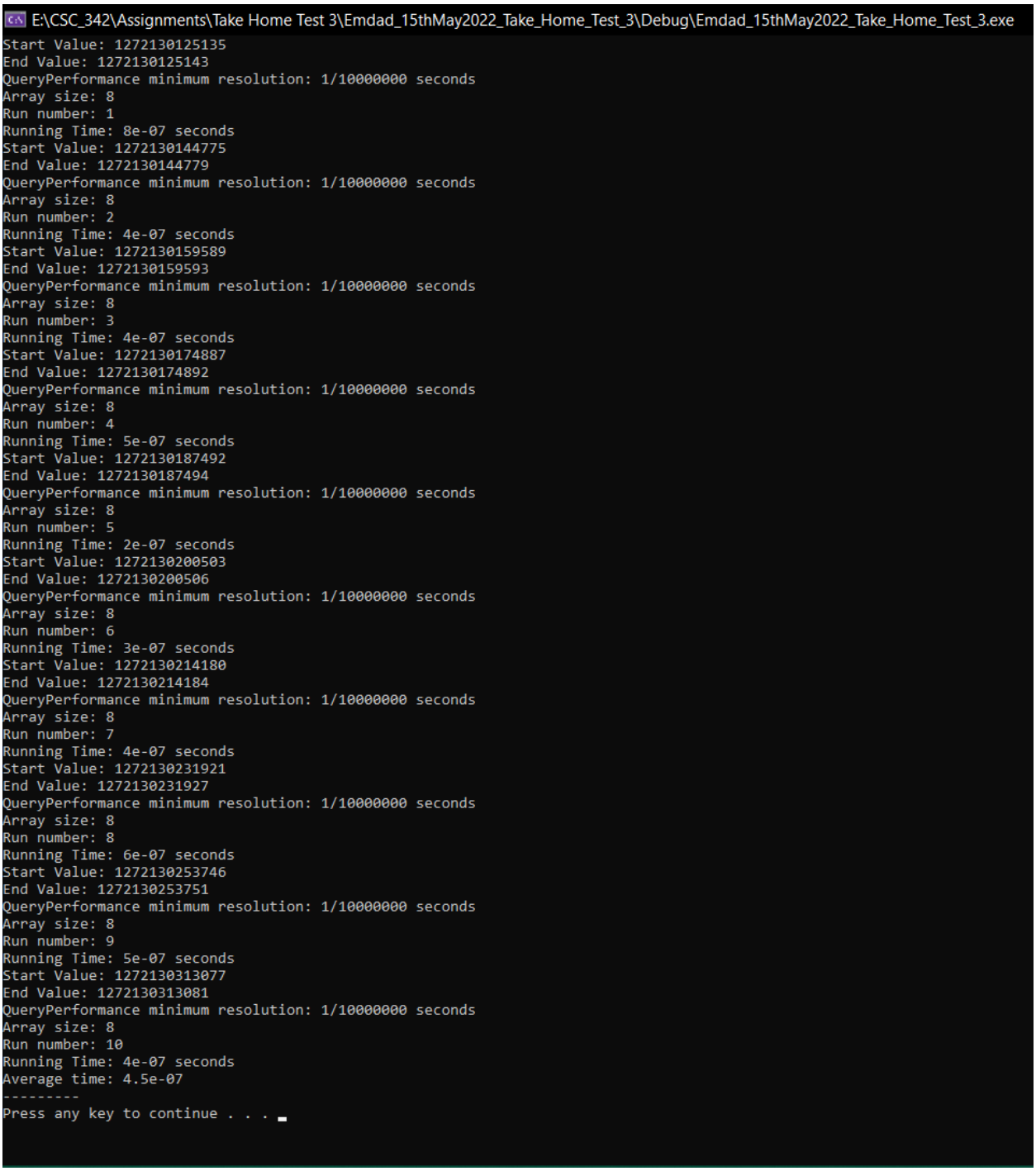


Figure 38: average execution running time for array size 8.

In figure 39, we can see the array size average execution running time for array size 16.

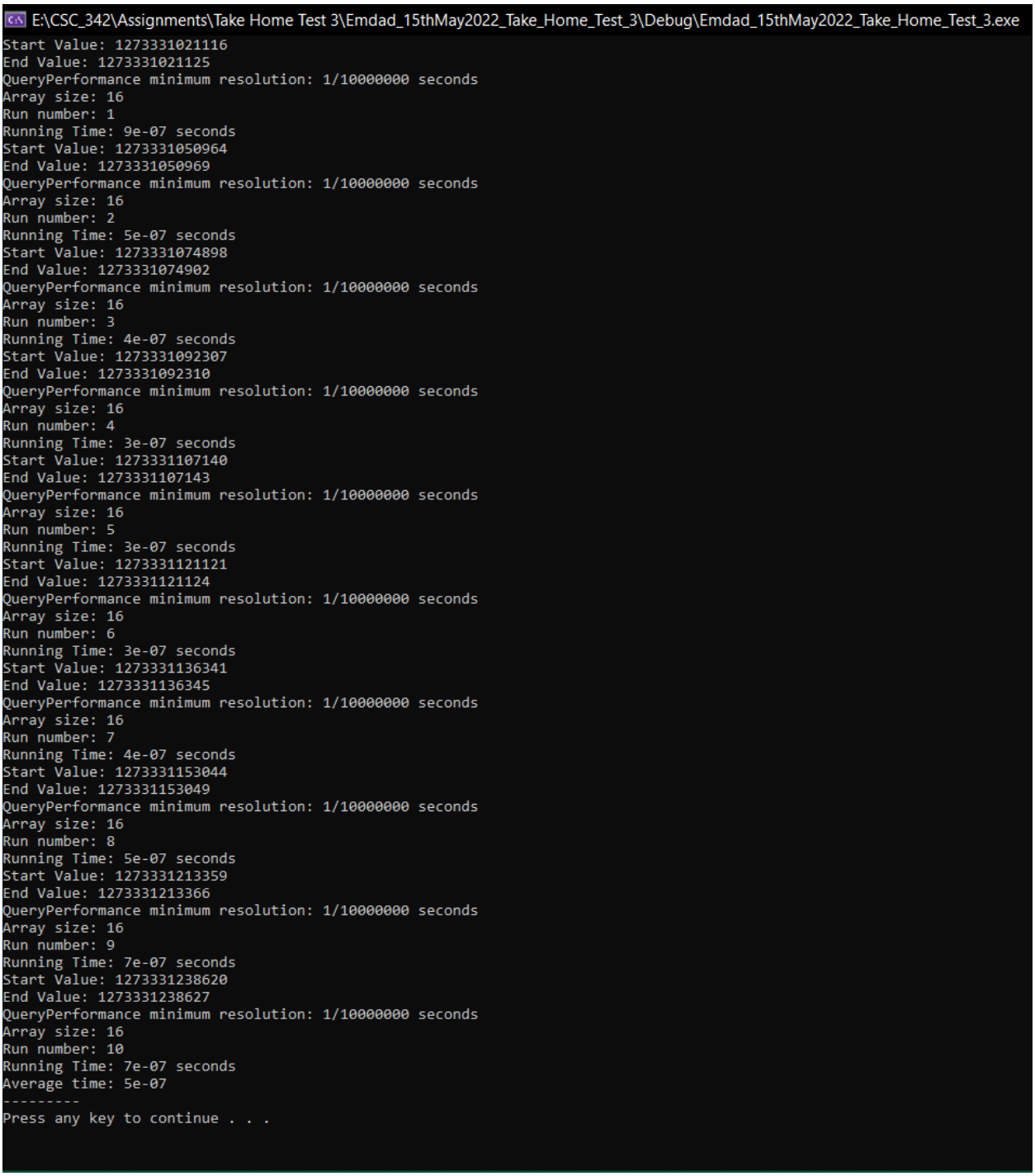


Figure 39: execution running time for array size 16.

In figure 40, we can see the array size average execution running time for array size 32.

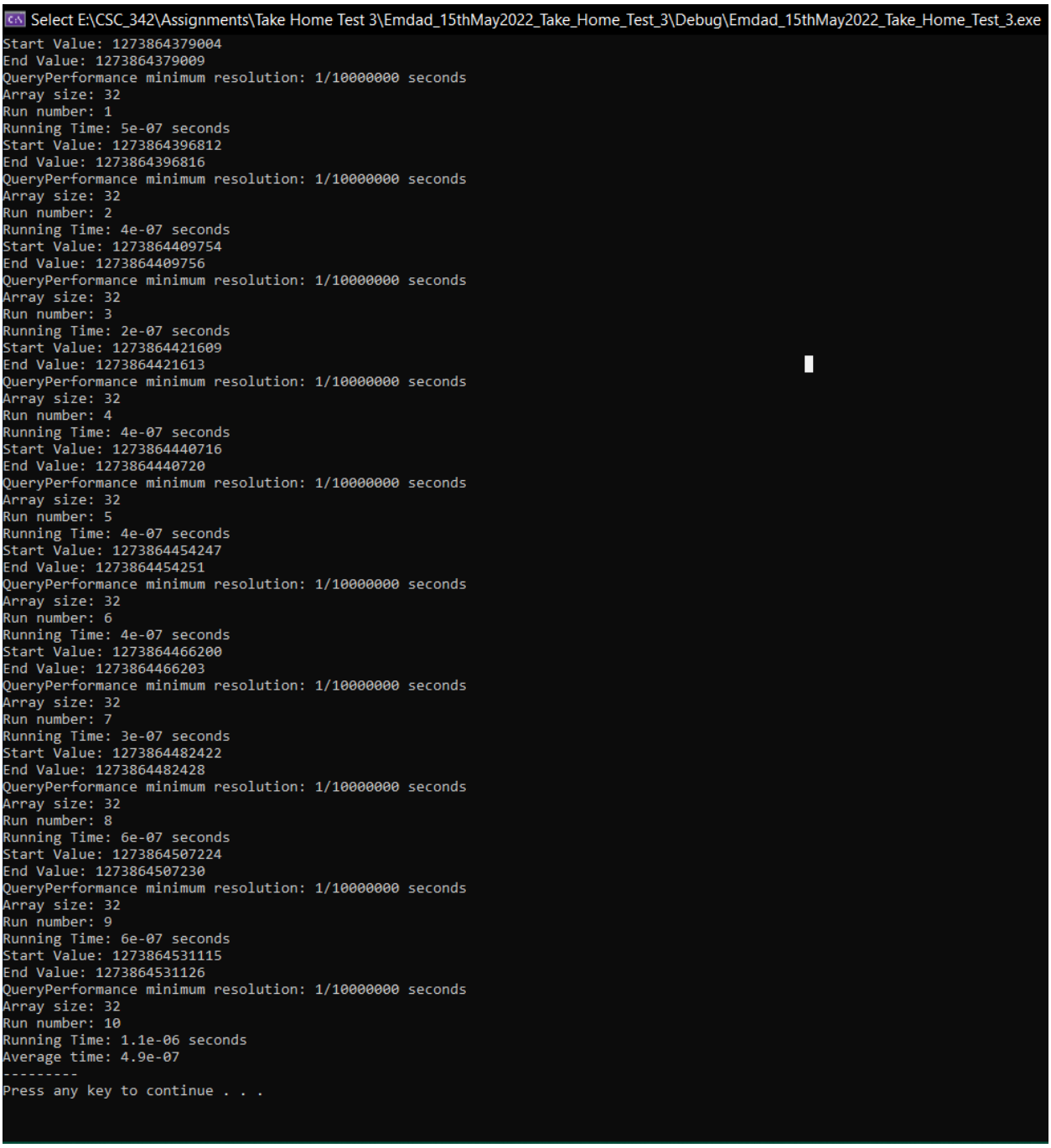


Figure 40: execution running time for array size 32

In figure 41, we can see the array size average execution running time for array size 64.

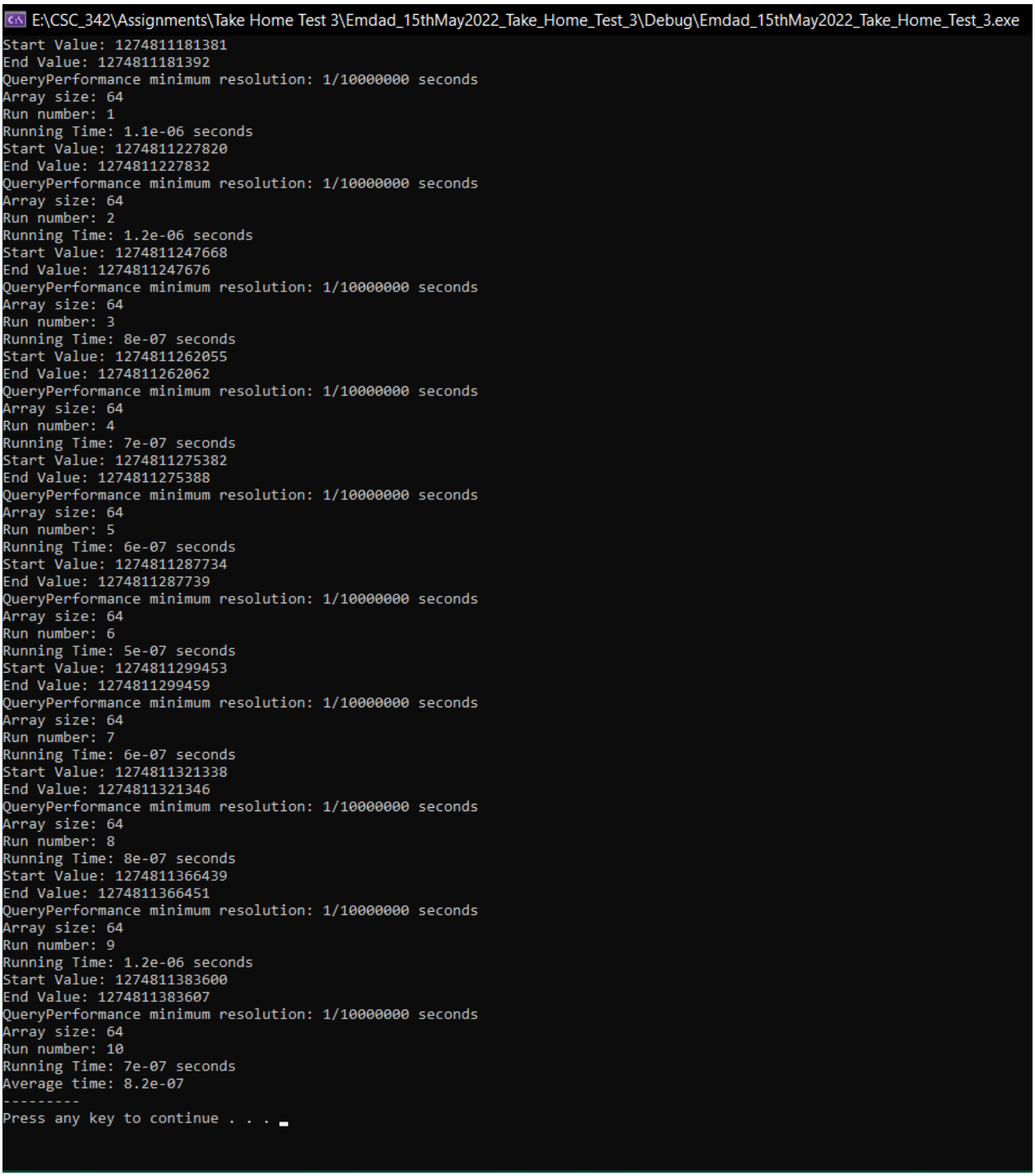


Figure 41: execution running time for array size 64

In figure 42, we can see the array size average execution running time for array size 128.

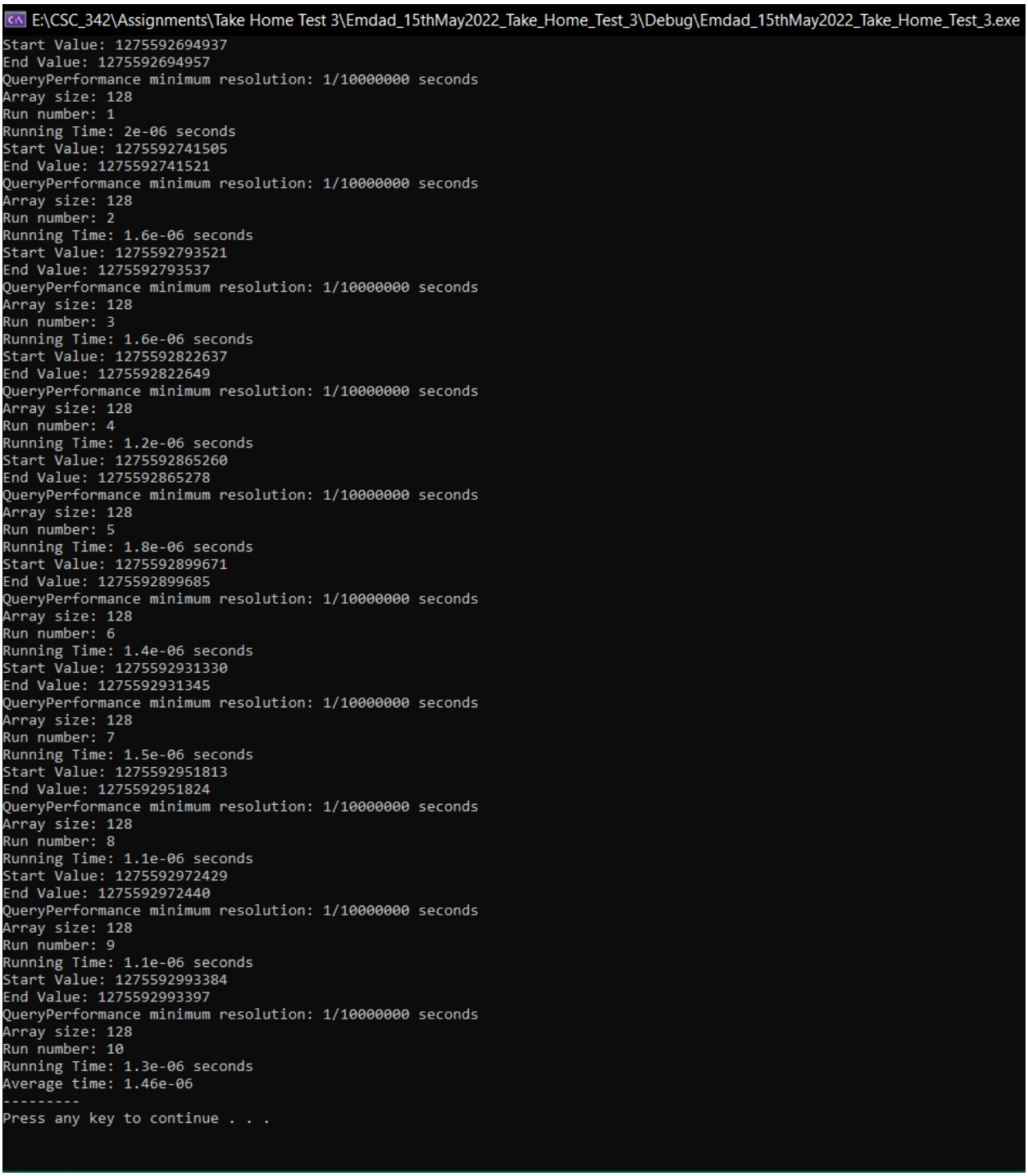


Figure 42: execution running time for array size 128

In figure 43, we can see the array size average execution running time for array size 256.

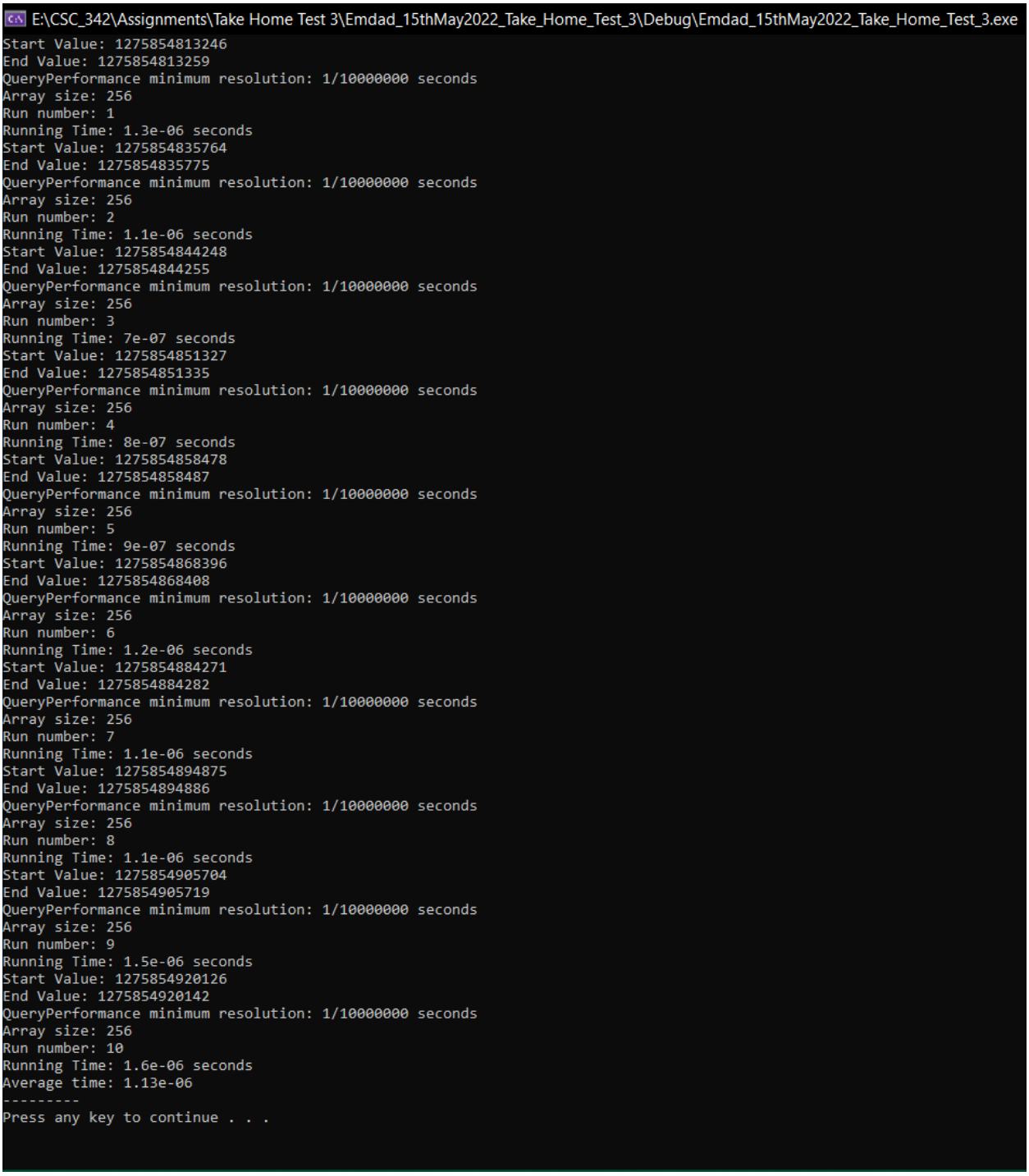


Figure 43: execution running time for array size 256

In figure 44, we can see the array size average execution running time for array size 512.

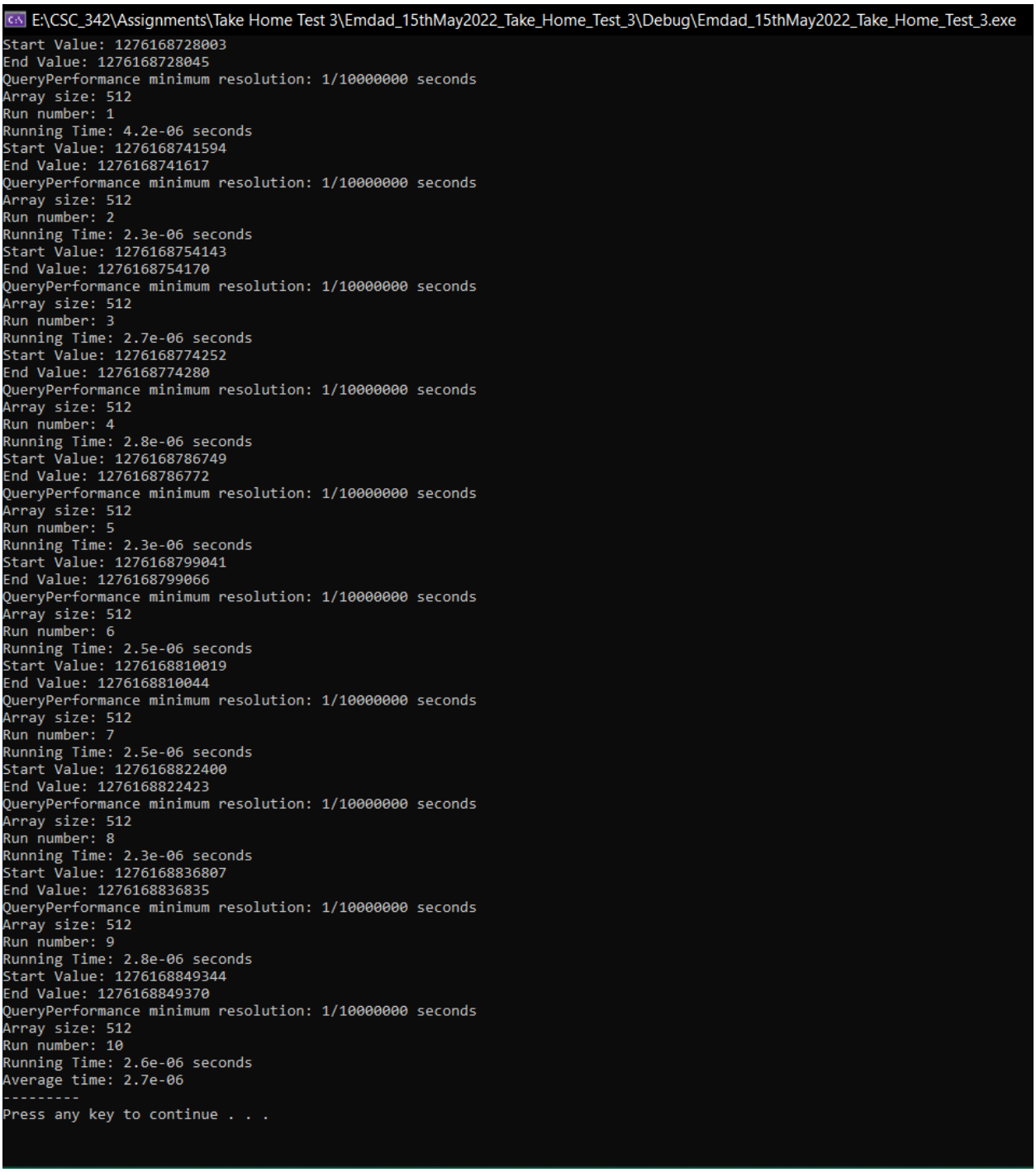


Figure 44: execution running time for array size 512

In figure 45, we can see the array size average execution running time for array size 1024.

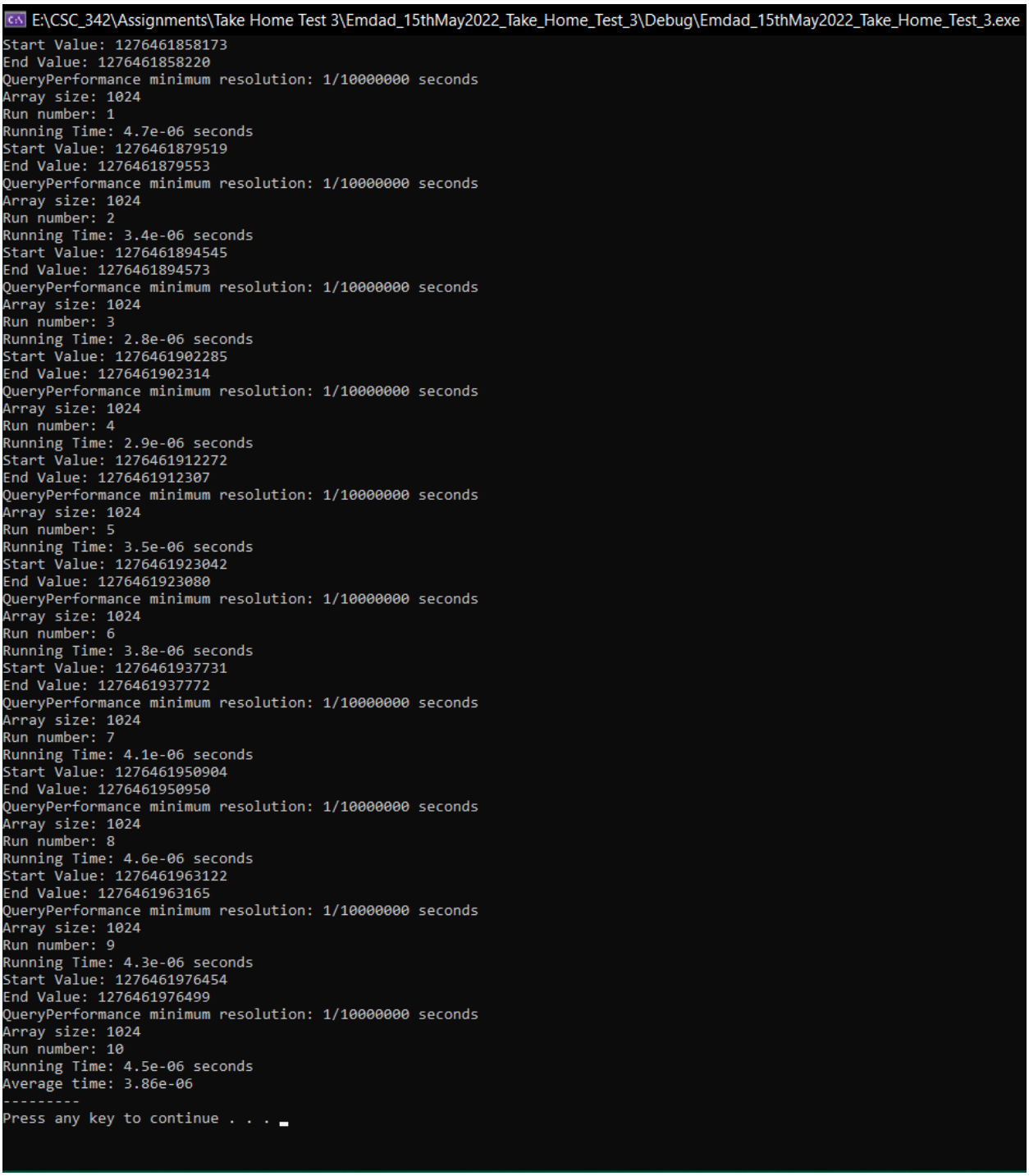


Figure 45: execution running time for array size 1024

In figure 46, we can see the array size average execution running time for array size 2048.

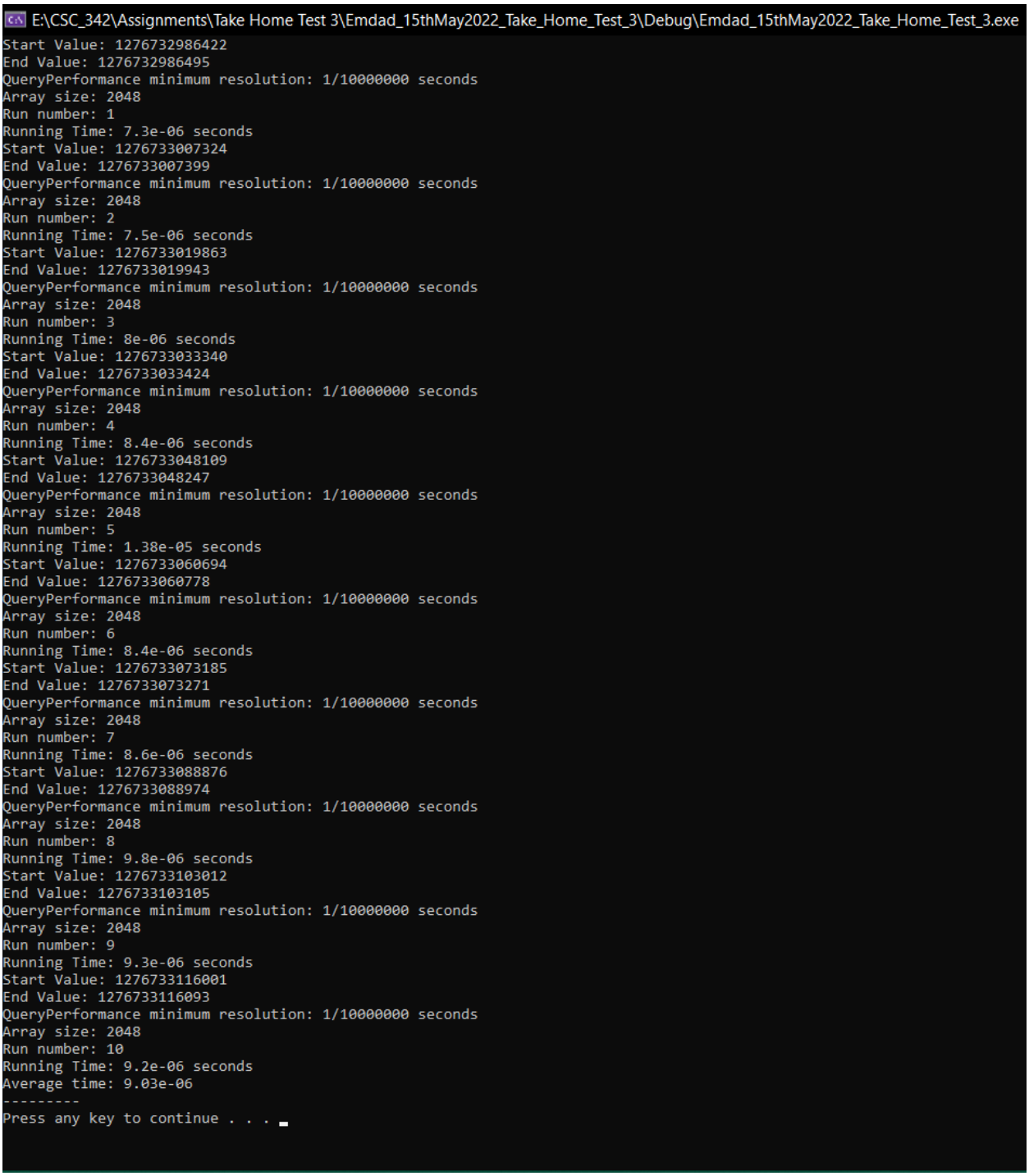


Figure 46: execution running time for array size 2048

In figure 47, we can see the array size average execution running time for array size 4096.

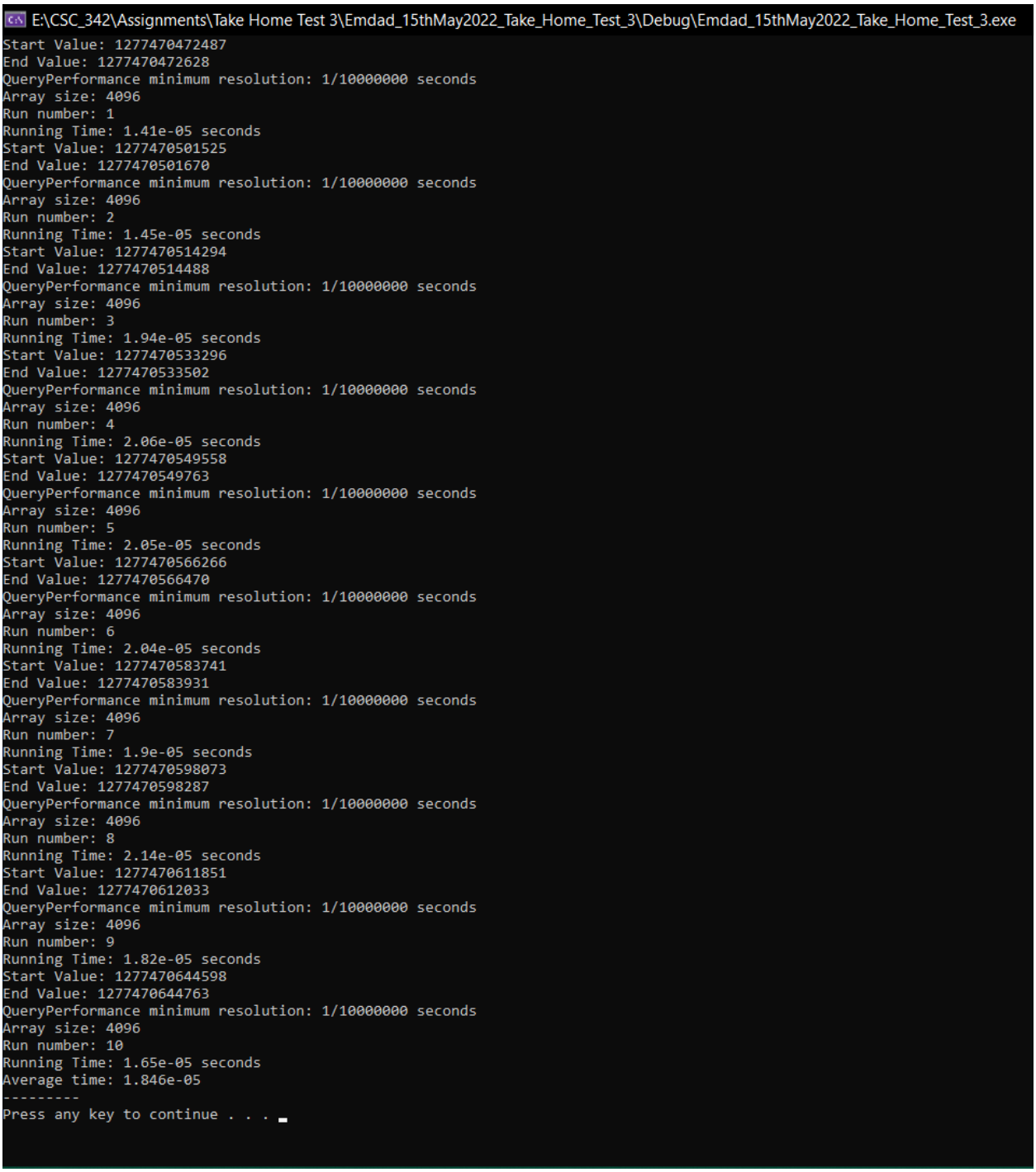


Figure 47: execution running time for array size 4096

In figure 48, we can see that I made a table for the average runtime based on the array sizes.

Array Size	Average Running Time (seconds)
8	4.50E-07
16	5.00E-07
32	4.90E-07
64	8.20E-07
128	1.46E-06
256	1.13E-06
512	2.70E-06
1024	3.86E-06
2048	9.03E-06
4096	1.85E-05

Figure 48: Average Runtime Table (Compiler Generated Optimization)

In figure 49, we can see that I made a graph for the average runtime based on the array sizes. We can how the runtime went as the array size increased.

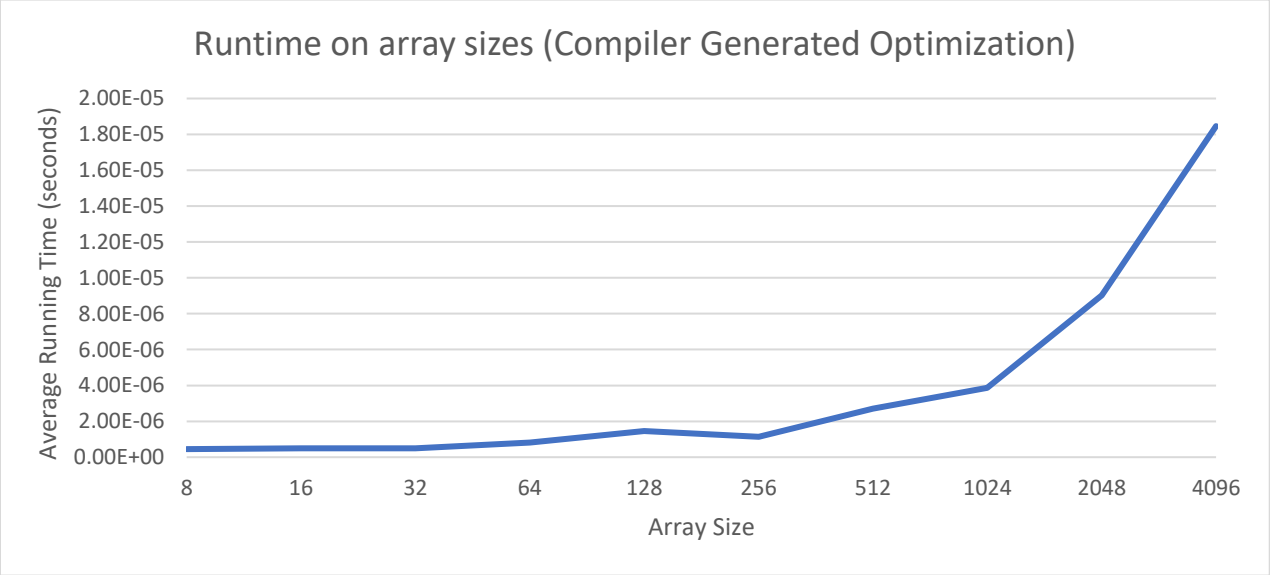


Figure 49: Runtime graph (Compiler Generated Optimization)

Dot Product [Manually Optimized]:

In figure 50, we can see the array size average execution running time for array size 8.

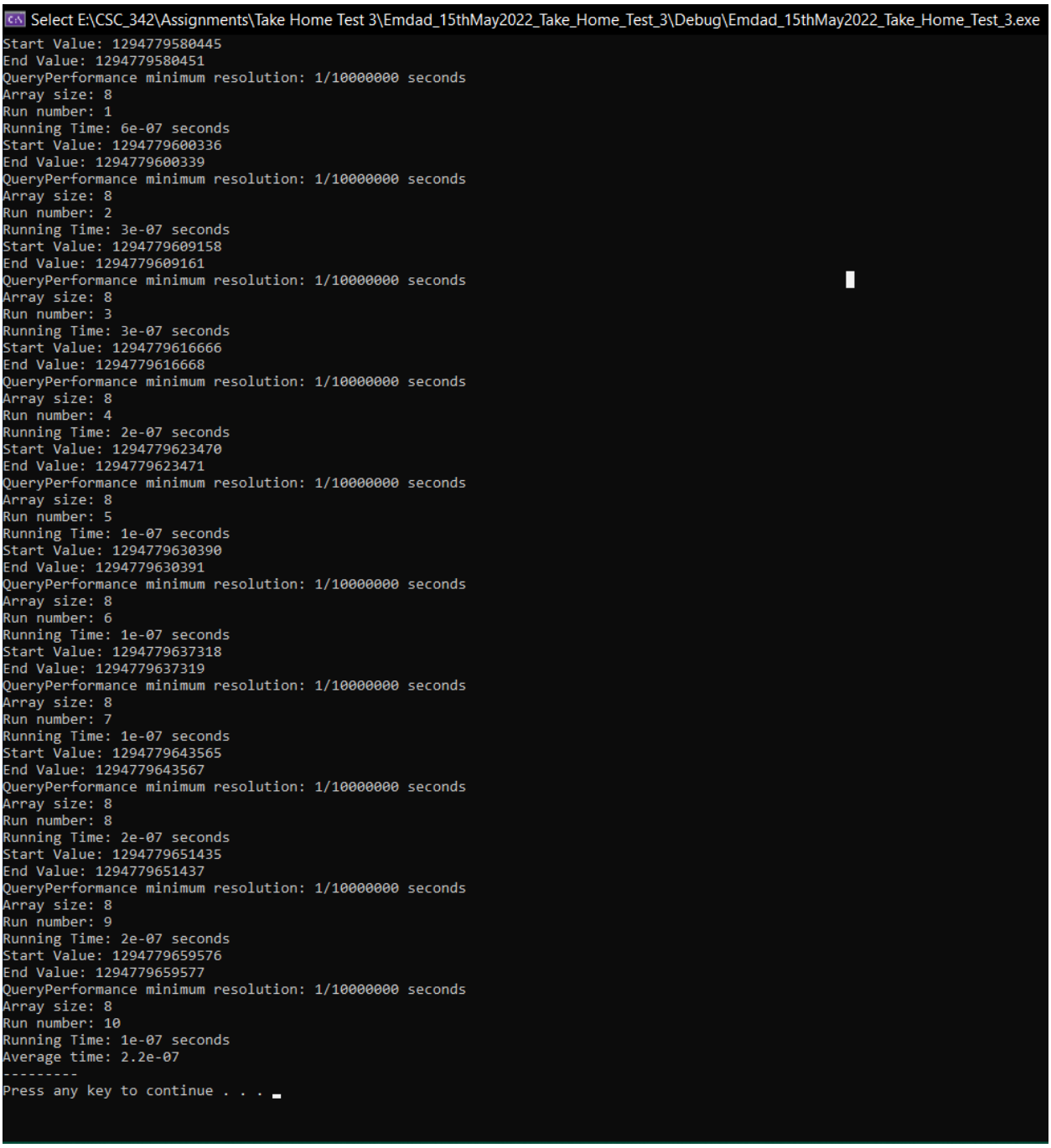


Figure 50: execution running time for array size 8.

In figure 51, we can see the array size average execution running time for array size 16.

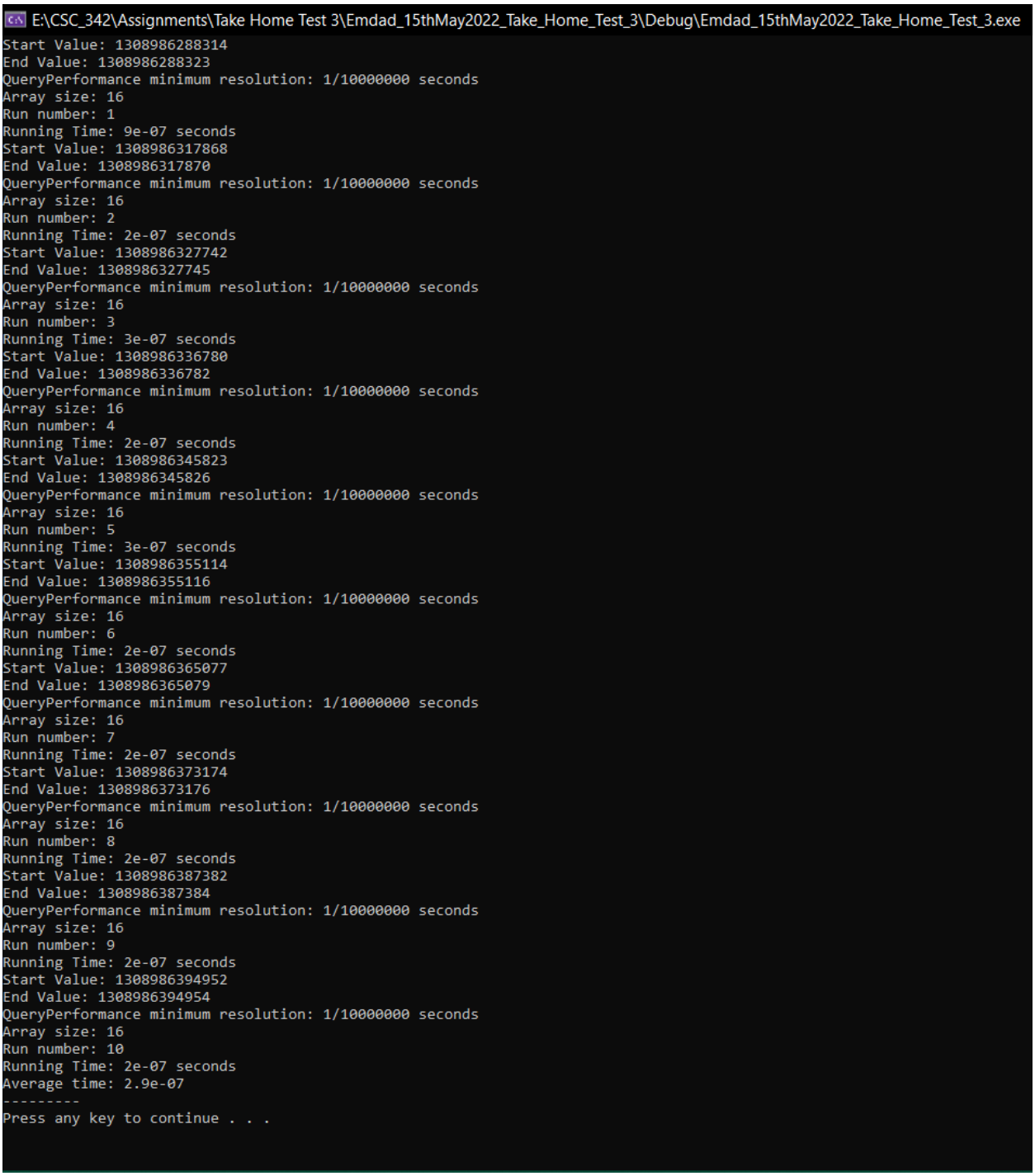


Figure 51: execution running time for array size 16

In figure 52, we can see the array size average execution running time for array size 32.

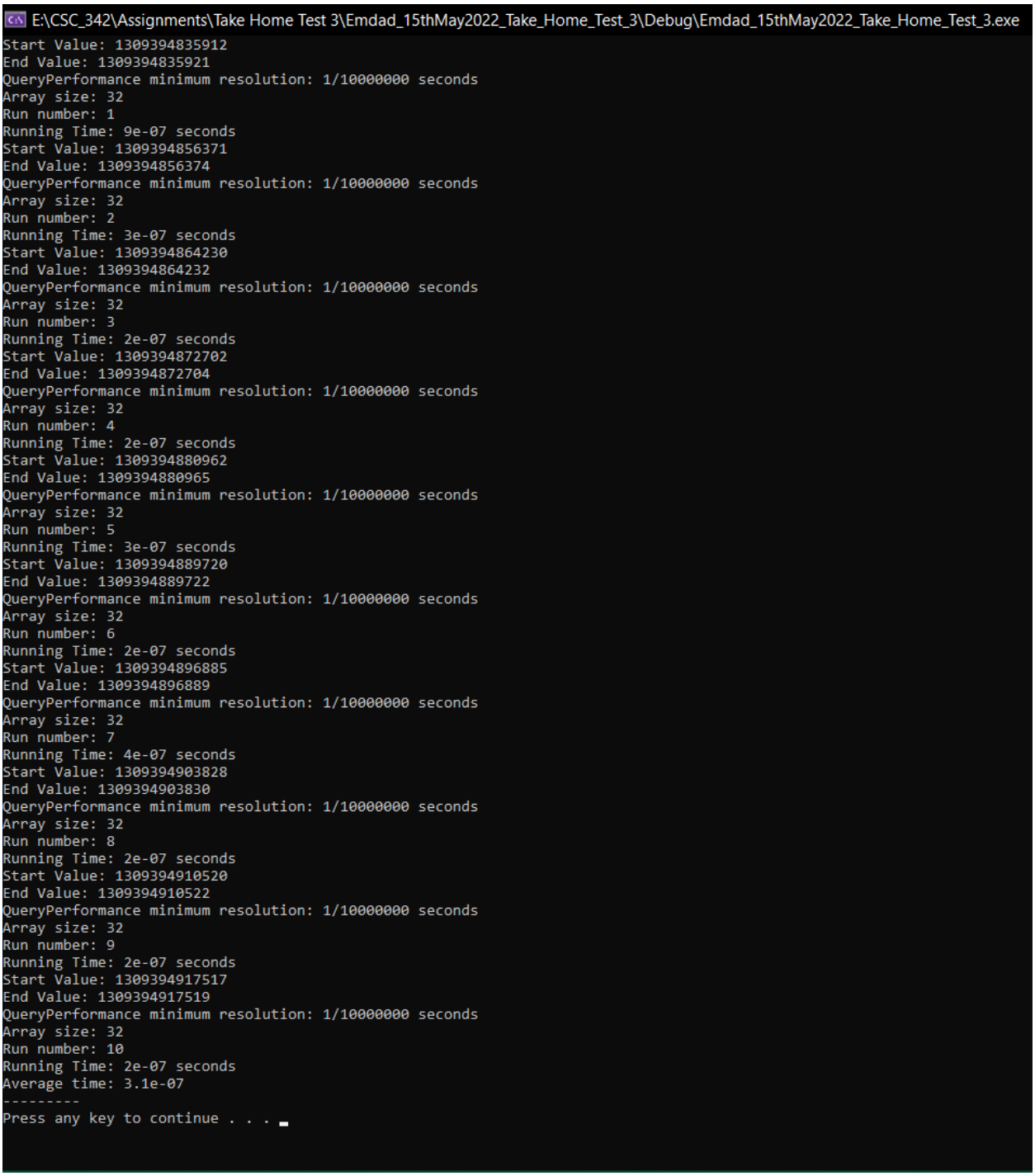


Figure 52: execution running time for array size 32

In figure 53, we can see the array size average execution running time for array size 64.

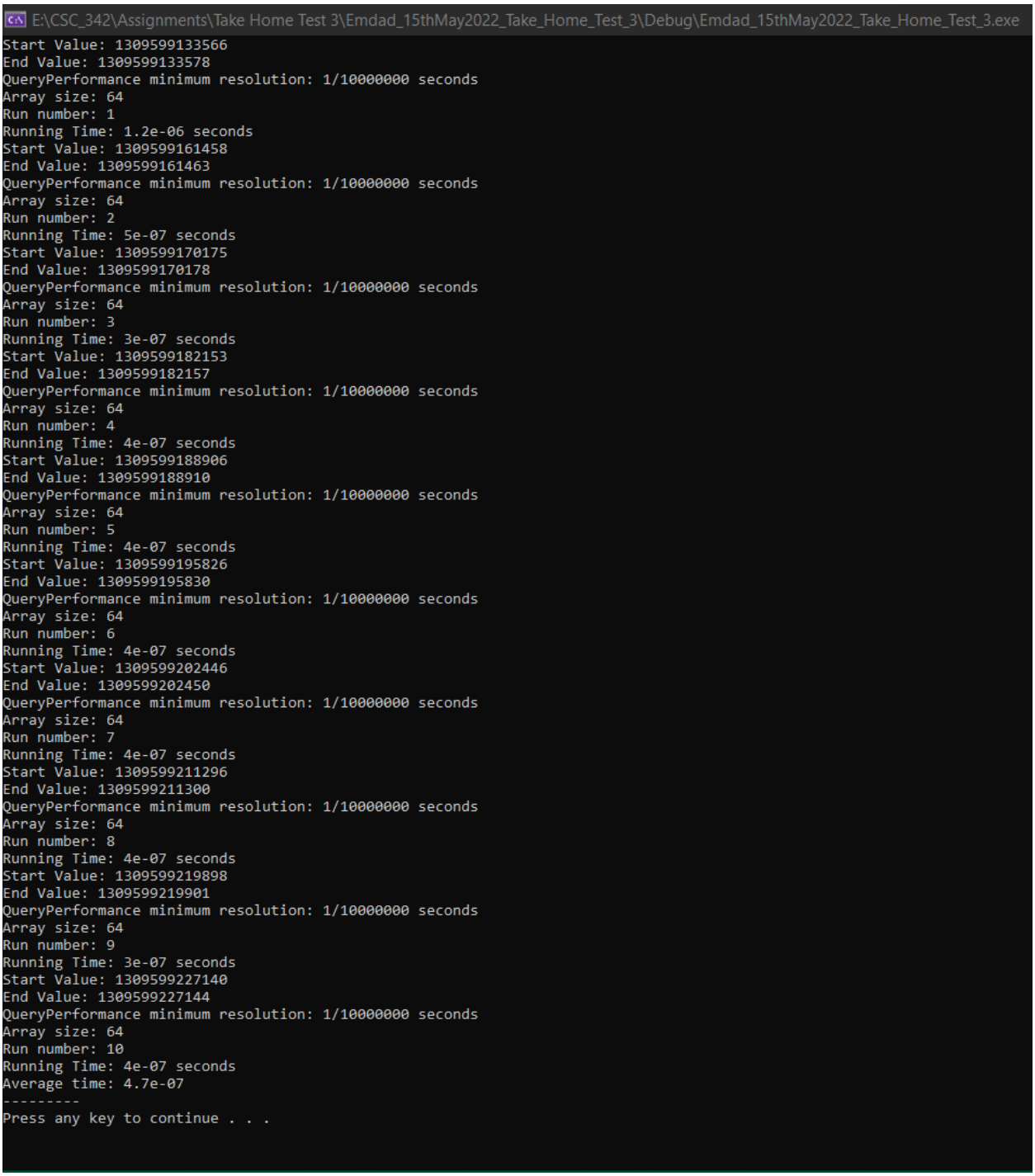


Figure 53: execution running time for array size 64

In figure 54, we can see the array size average execution running time for array size 128.

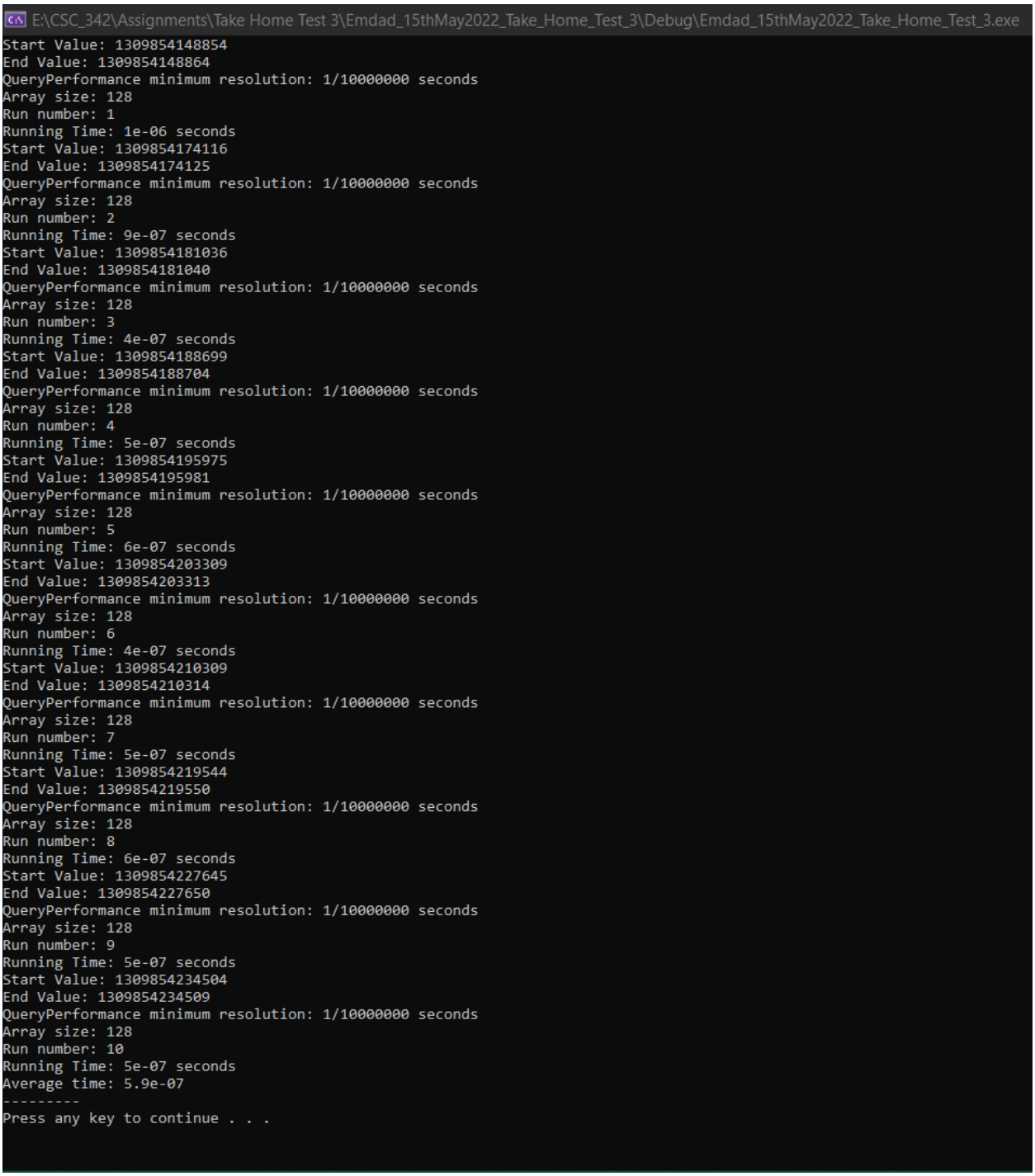


Figure 54: execution running time for array size 128

In figure 55, we can see the array size average execution running time for array size 256.

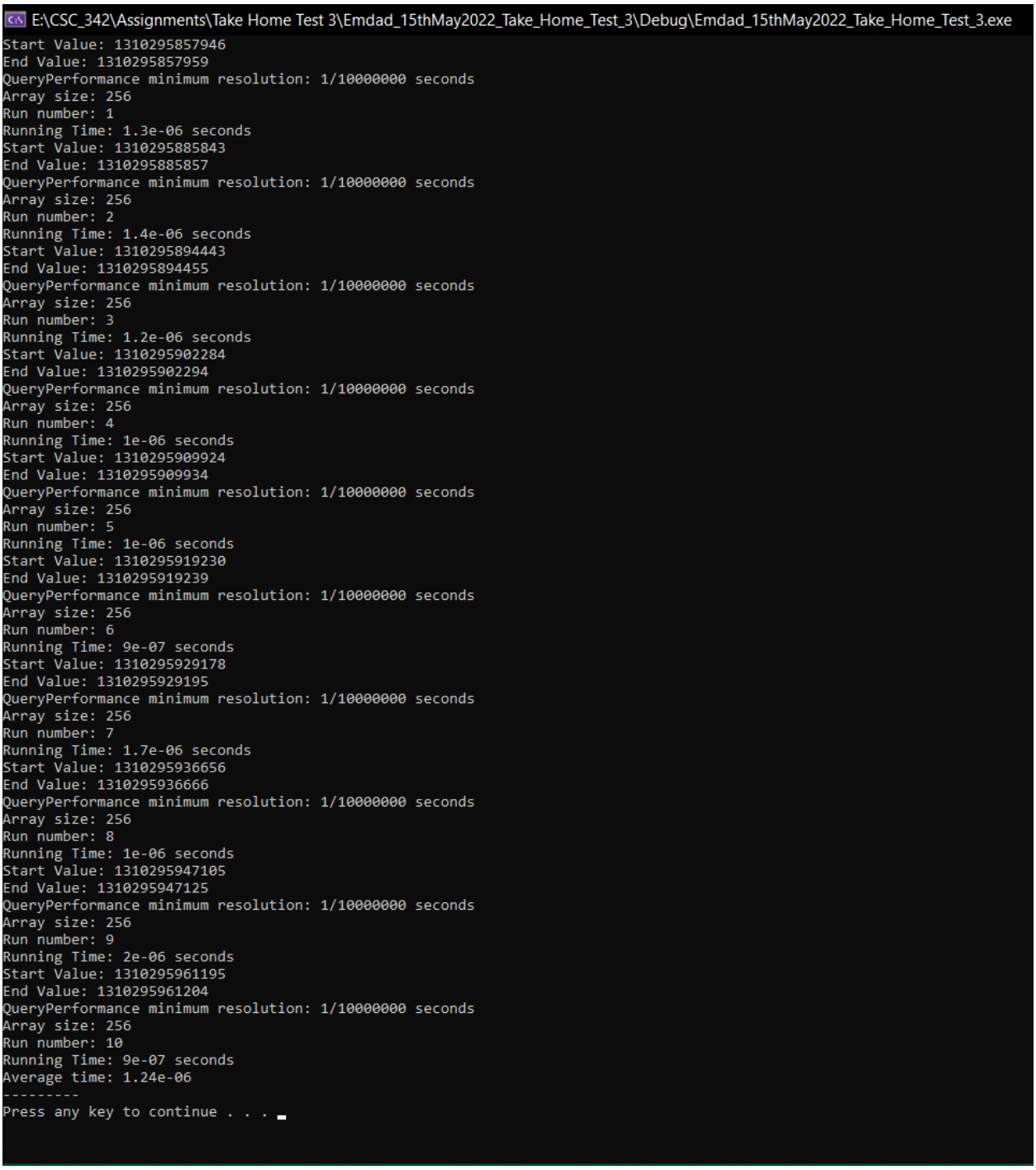


Figure 55: execution running time for array size 256

In figure 56, we can see the array size average execution running time for array size 512.

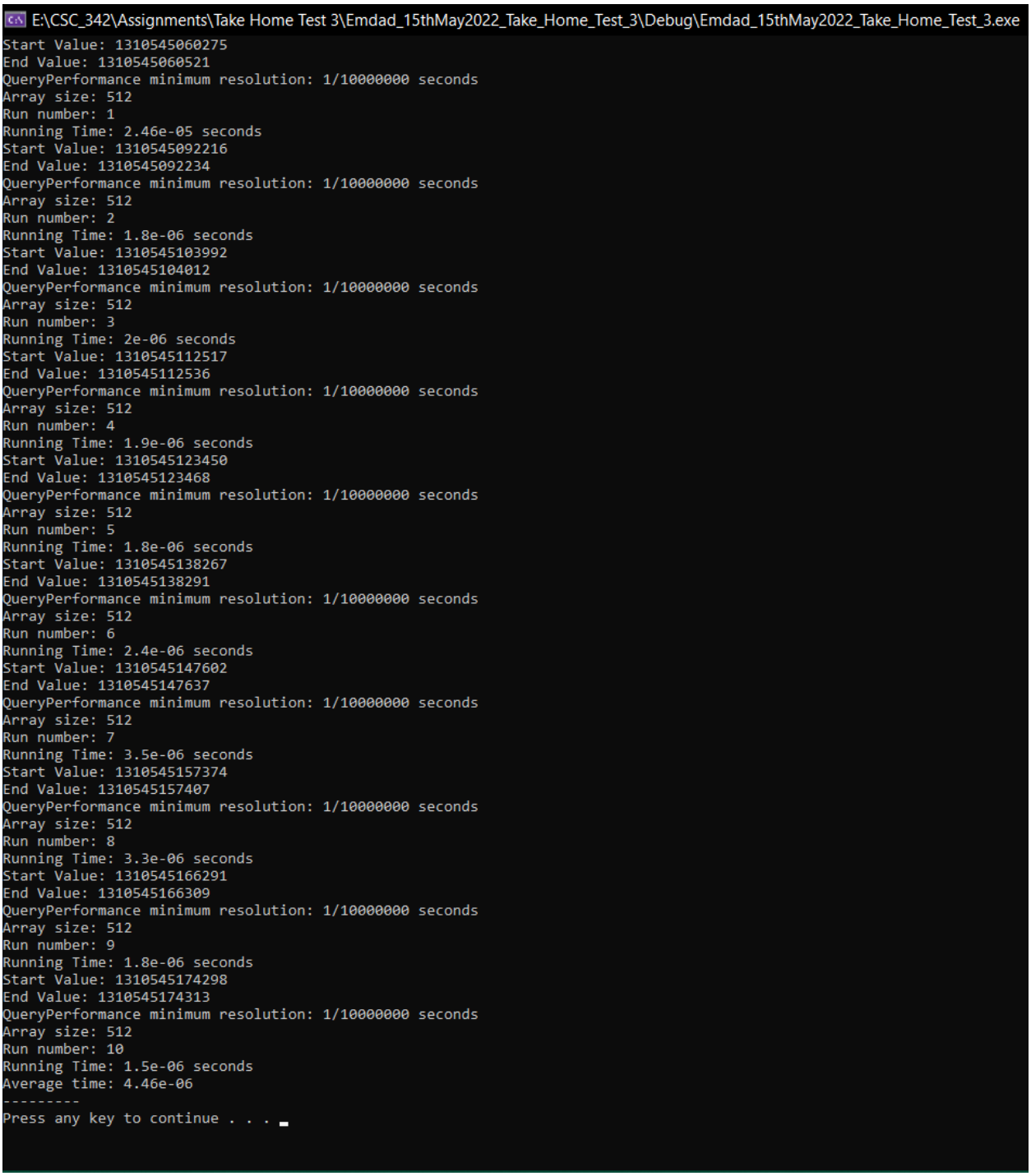


Figure 56: execution running time for array size 512

In figure 57, we can see the array size average execution running time for array size 1024.

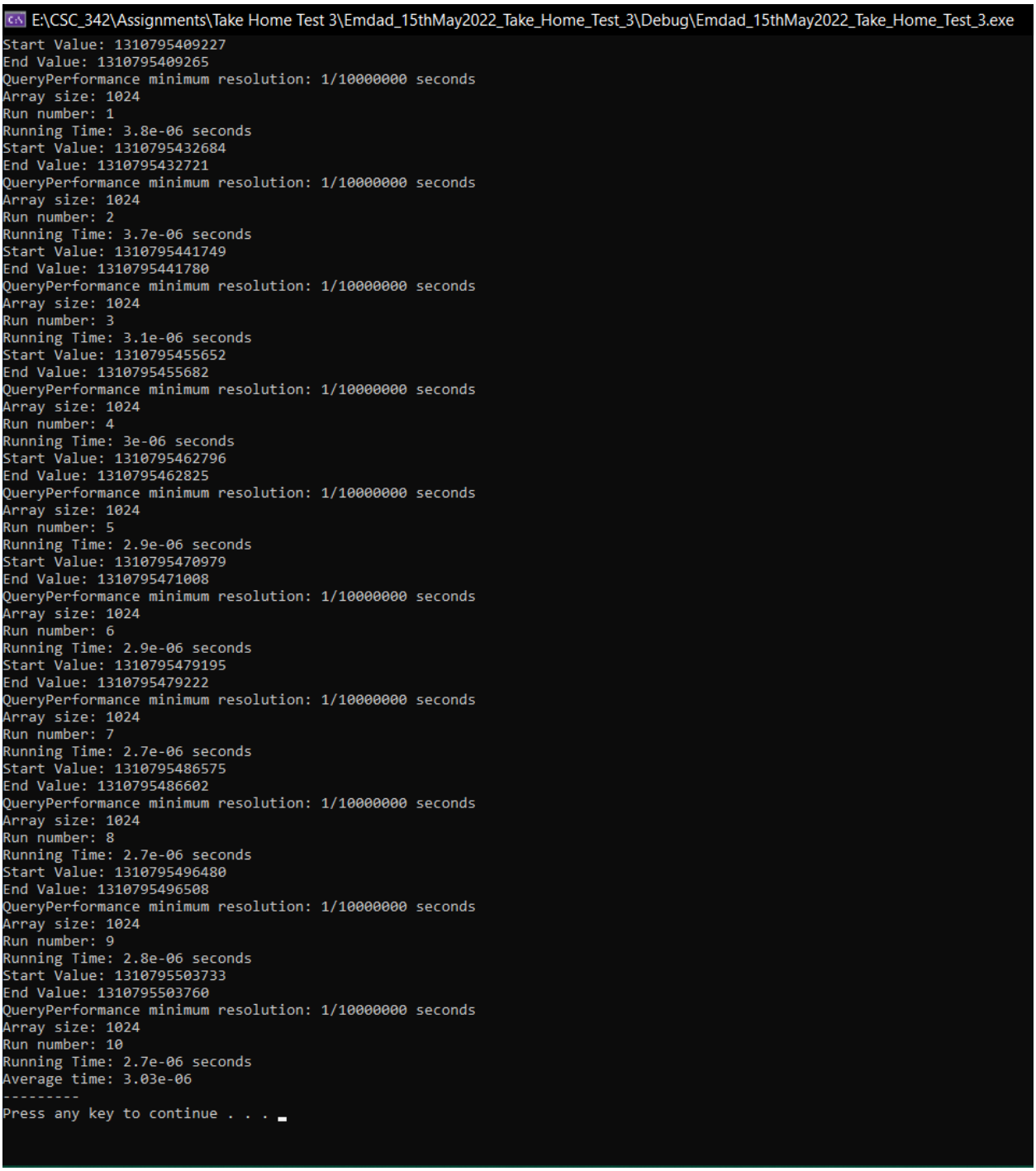


Figure 57: execution running time for array size 1024

In figure 58, we can see the array size average execution running time for array size 2048.

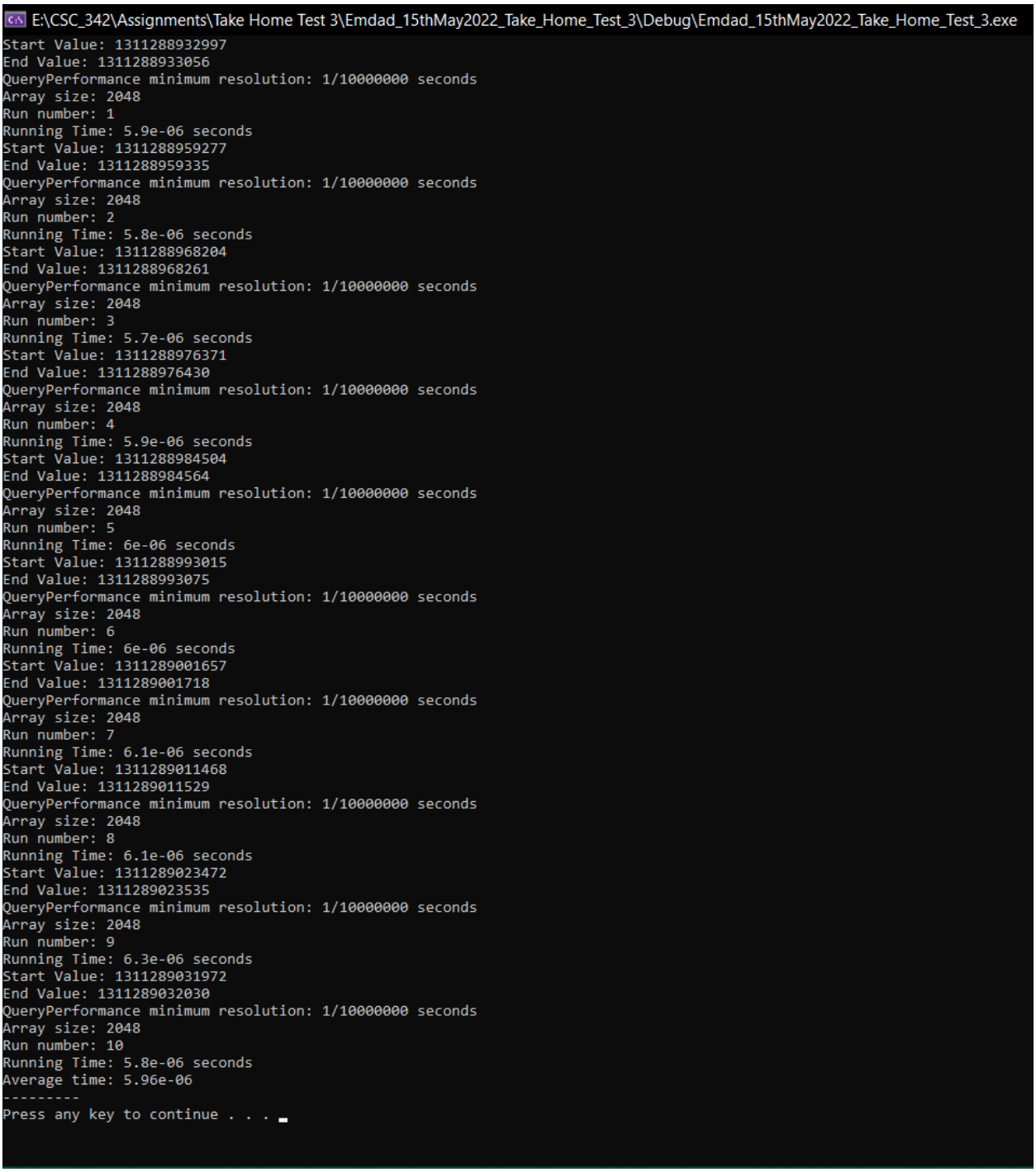


Figure 58: execution running time for array size 2048

In figure 59, we can see the array size average execution running time for array size 4096.

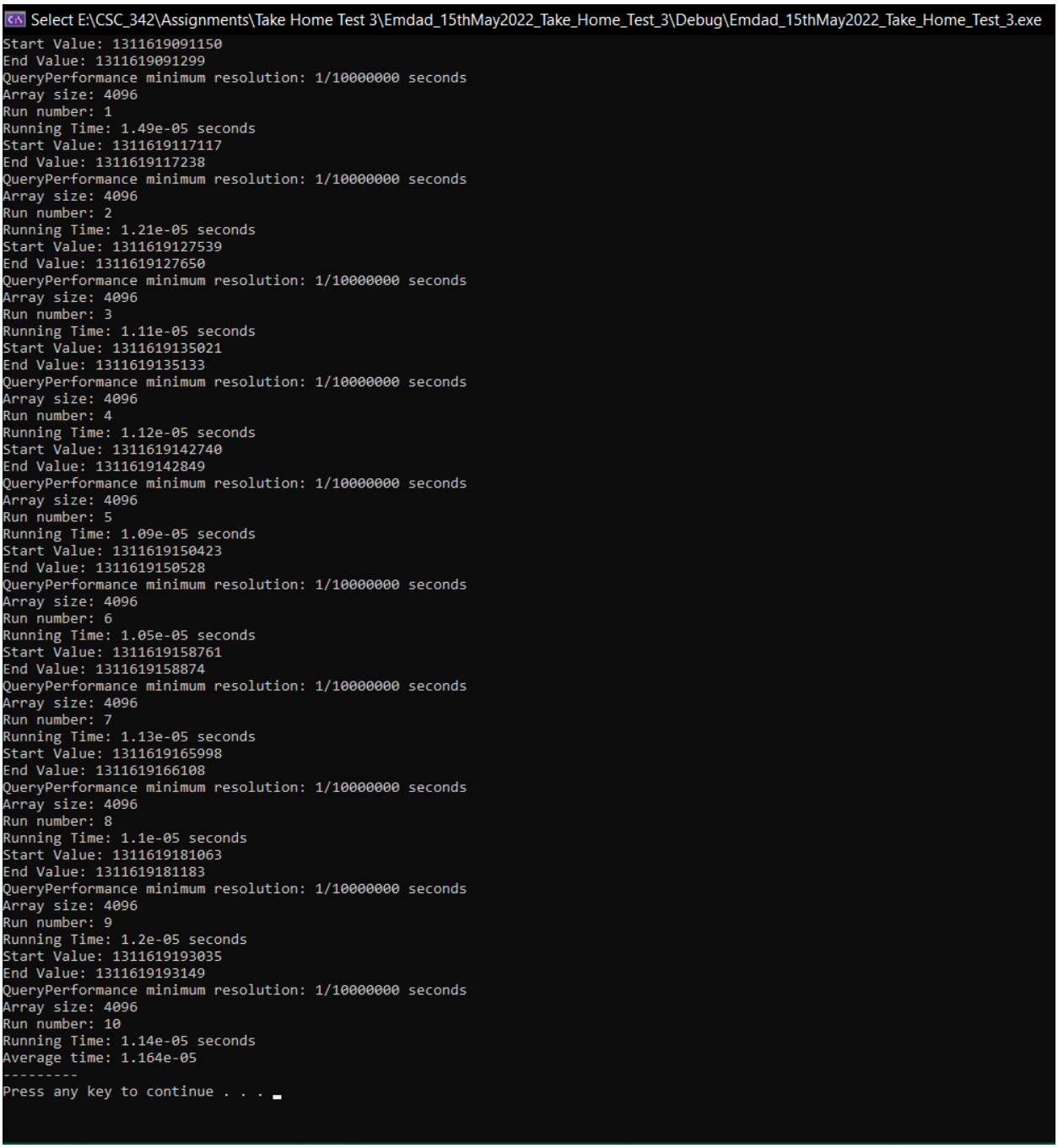


Figure 59: execution running time for array size 4096

In figure 60, we can see that I made a table for the average runtime based on the array sizes.

Array Size	Average Running Time (seconds)
8	2.20E-07
16	2.90E-07
32	3.10E-07
64	4.70E-07
128	5.90E-07
256	1.24E-06
512	2.46E-06
1024	3.03E-06
2048	5.96E-06
4096	1.16E-05

Figure 60: Average Runtime Table (Manually Optimized)

In figure 61, we can see that I made a graph for the average runtime based on the array sizes. We can how the runtime went as the array size increased.

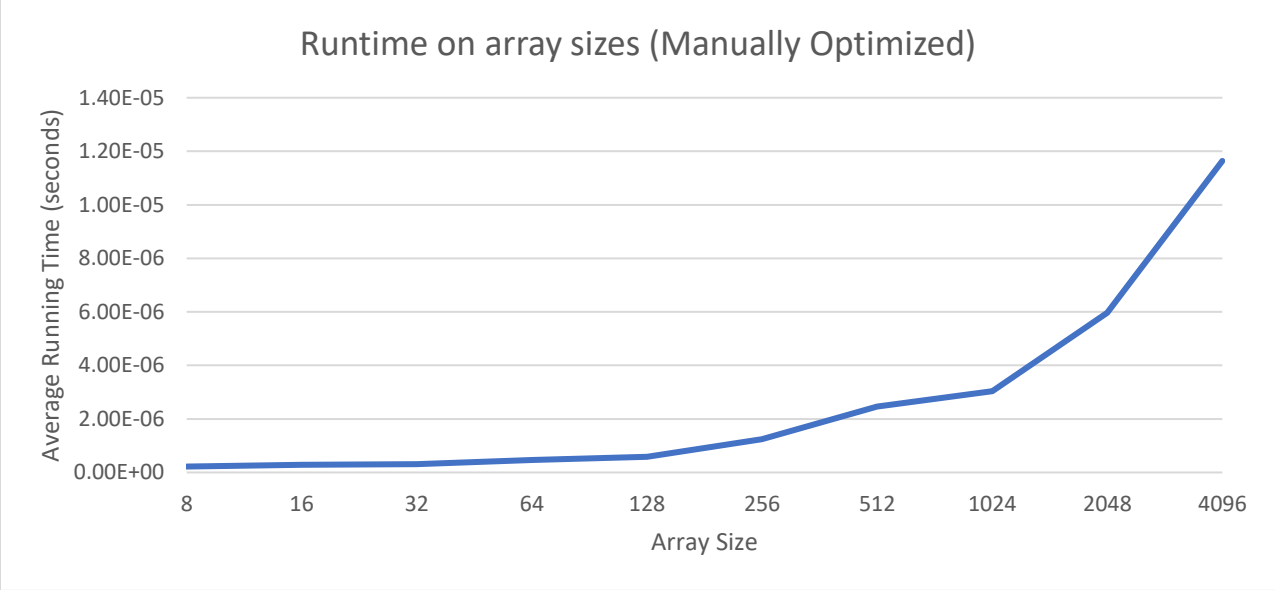


Figure 61: Runtime graph (Manually Optimized)

VDPPS:

In figure 62, we can see the array size average execution running time for array size 8.

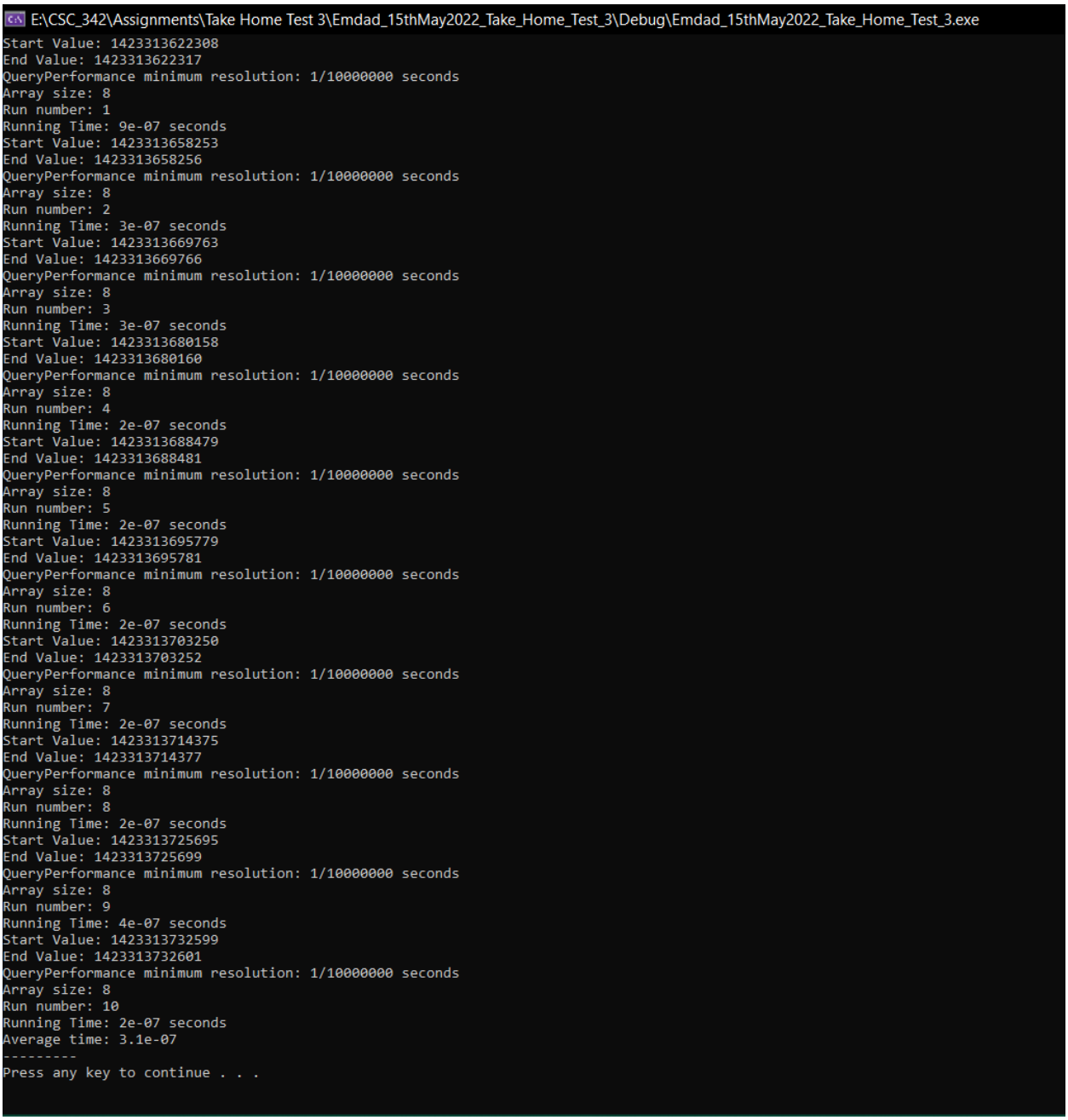


Figure 62: execution running time for array size 8

In figure 63, we can see the array size average execution running time for array size 16.

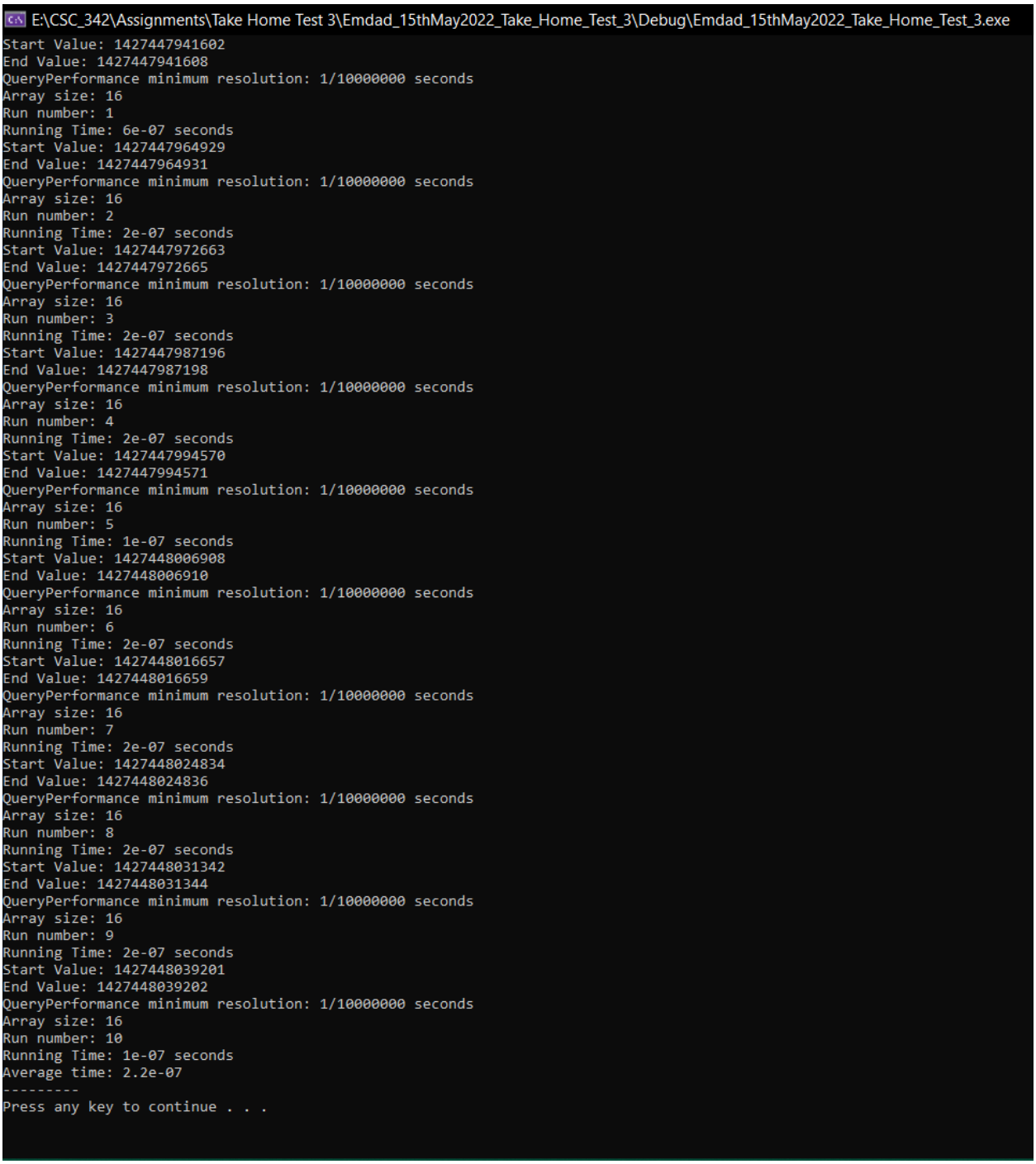


Figure 63: execution running time for array size 16

In figure 64, we can see the array size average execution running time for array size 32.

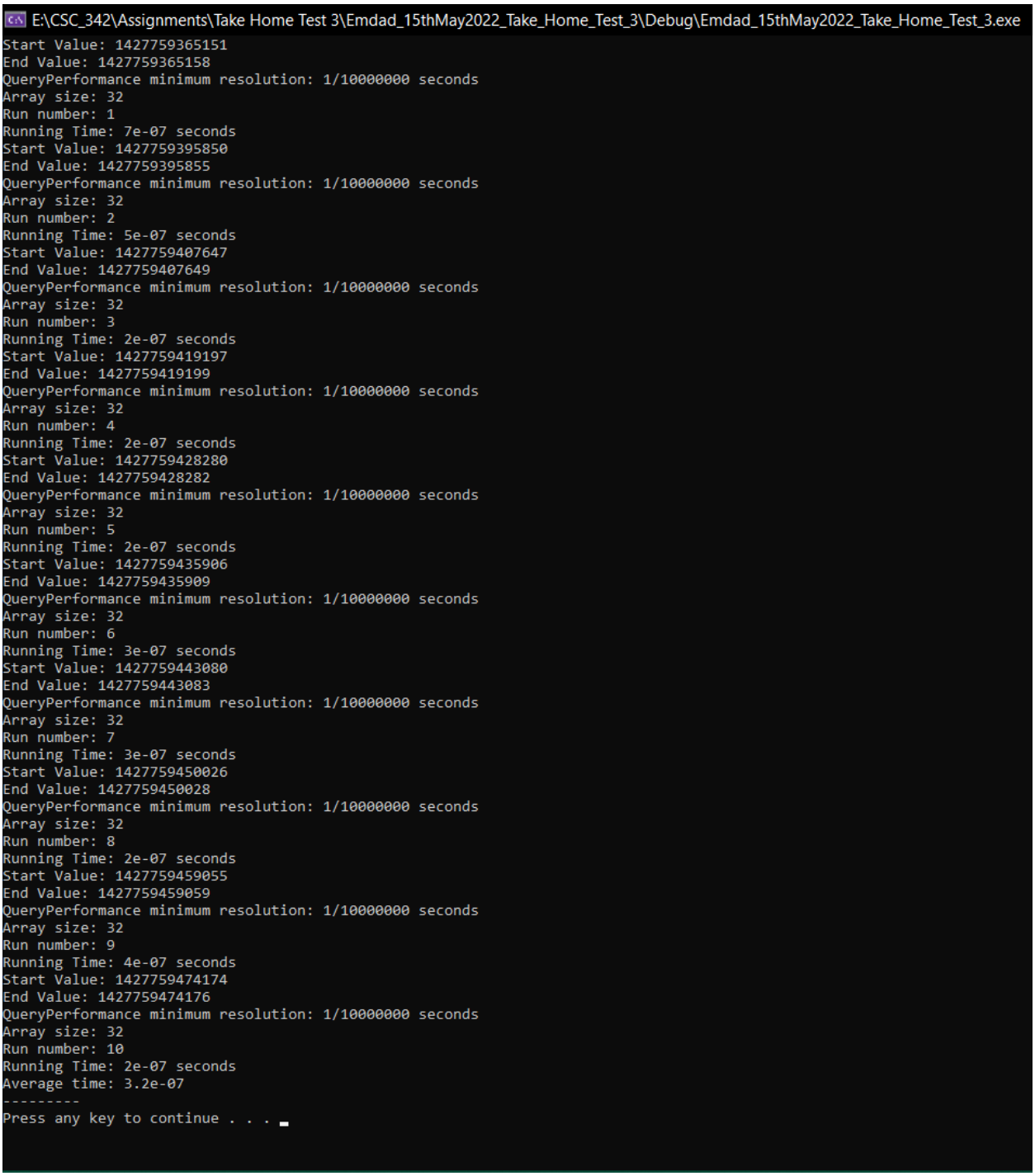


Figure 64: execution running time for array size 32

In figure 65, we can see the array size average execution running time for array size 64.

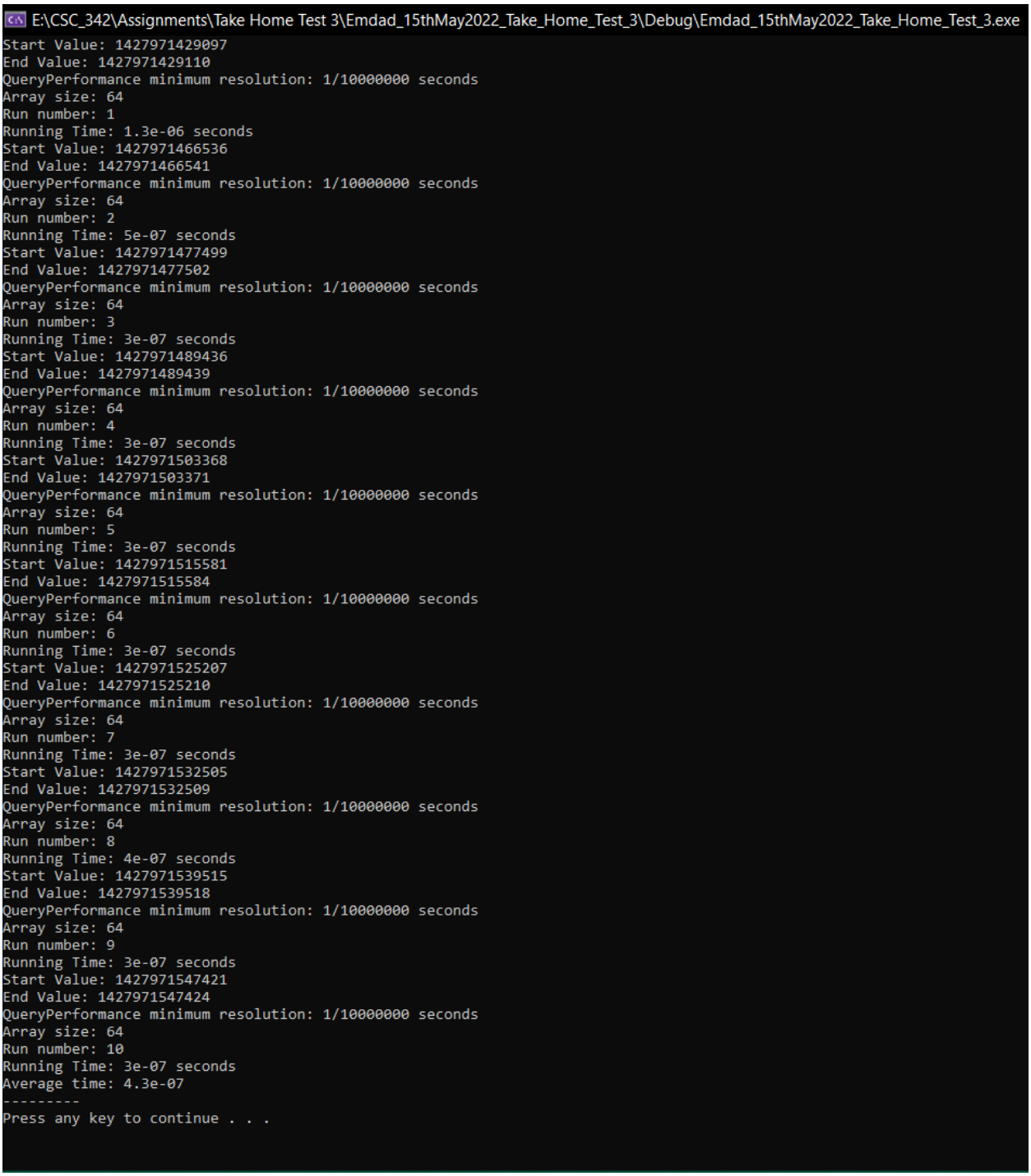


Figure 65: execution running time for array size 64

In figure 66, we can see the array size average execution running time for array size 128.

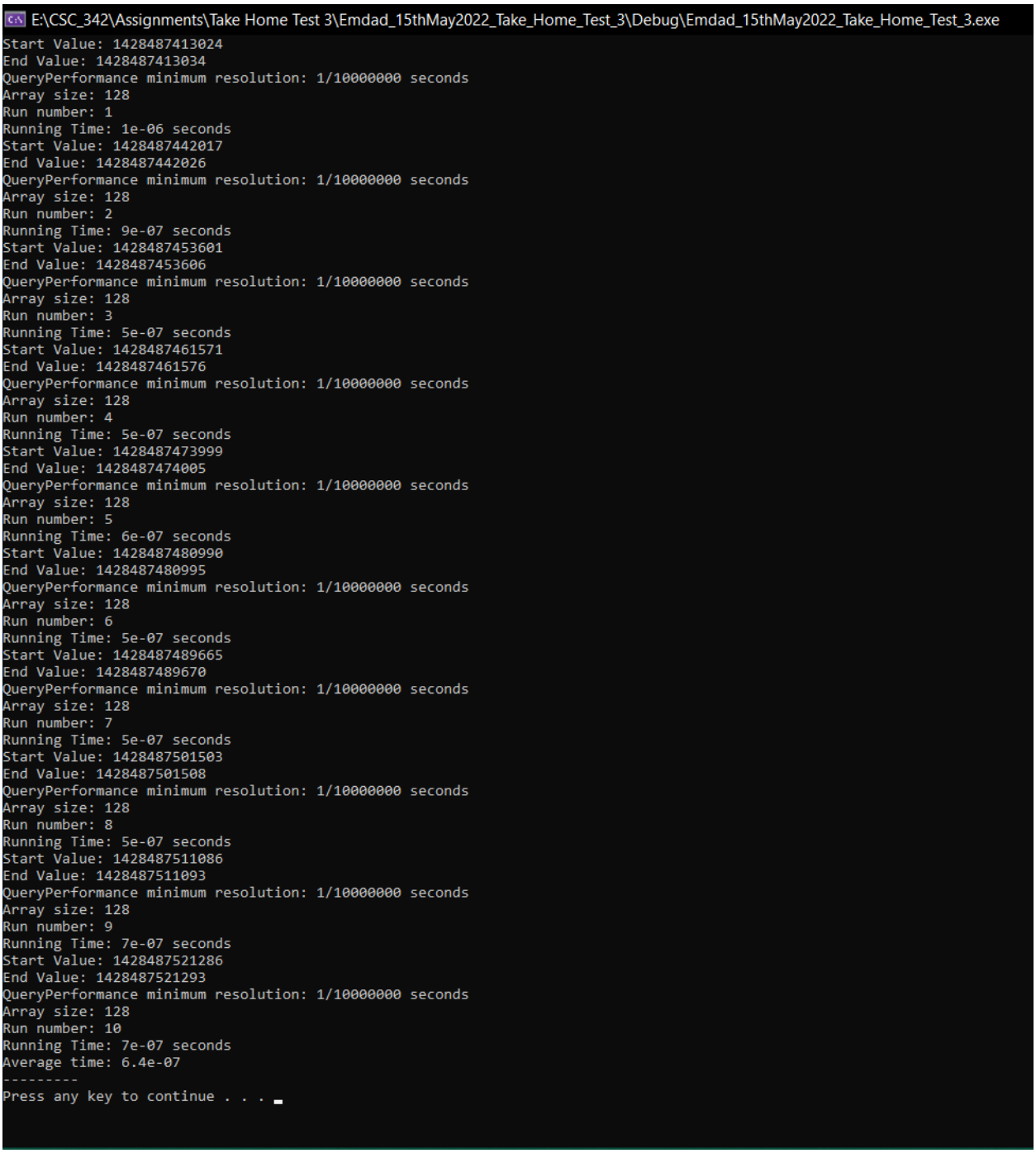


Figure 66: execution running time for array size 128

In figure 67, we can see the array size average execution running time for array size 256.

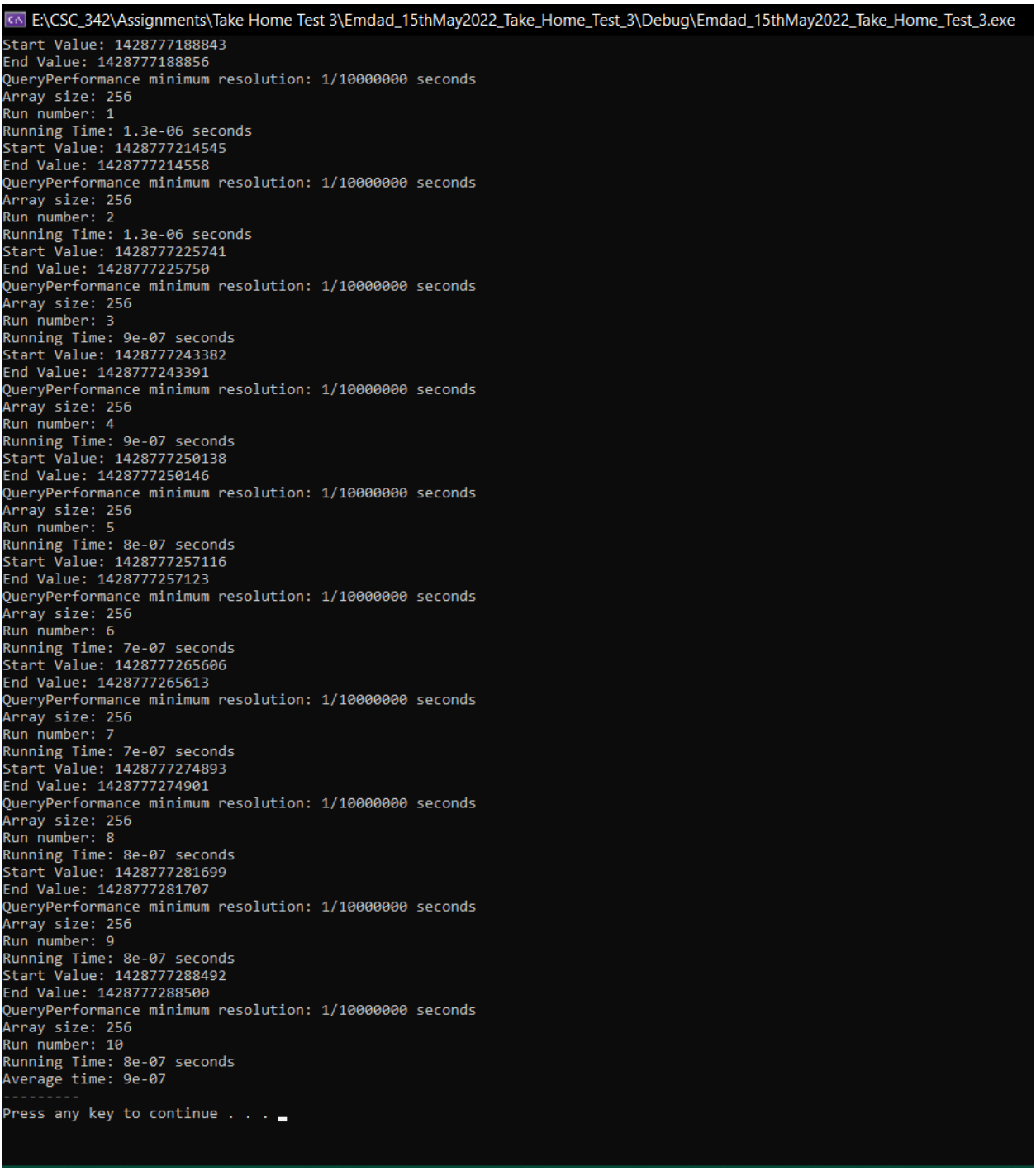


Figure 67: execution running time for array size 256

In figure 68, we can see the array size average execution running time for array size 512.

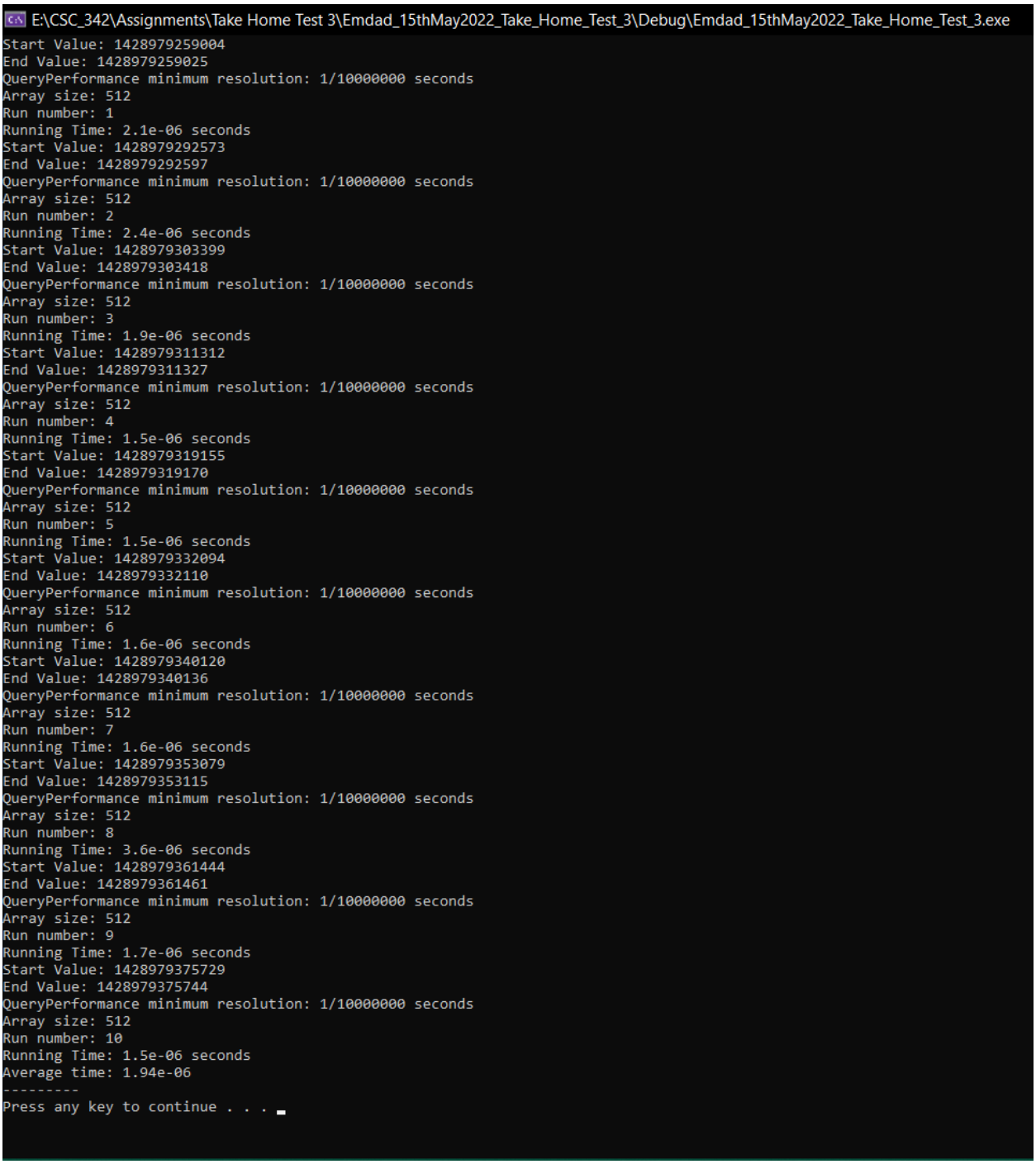


Figure 68: execution running time for array size 512

In figure 69, we can see the array size average execution running time for array size 1024.

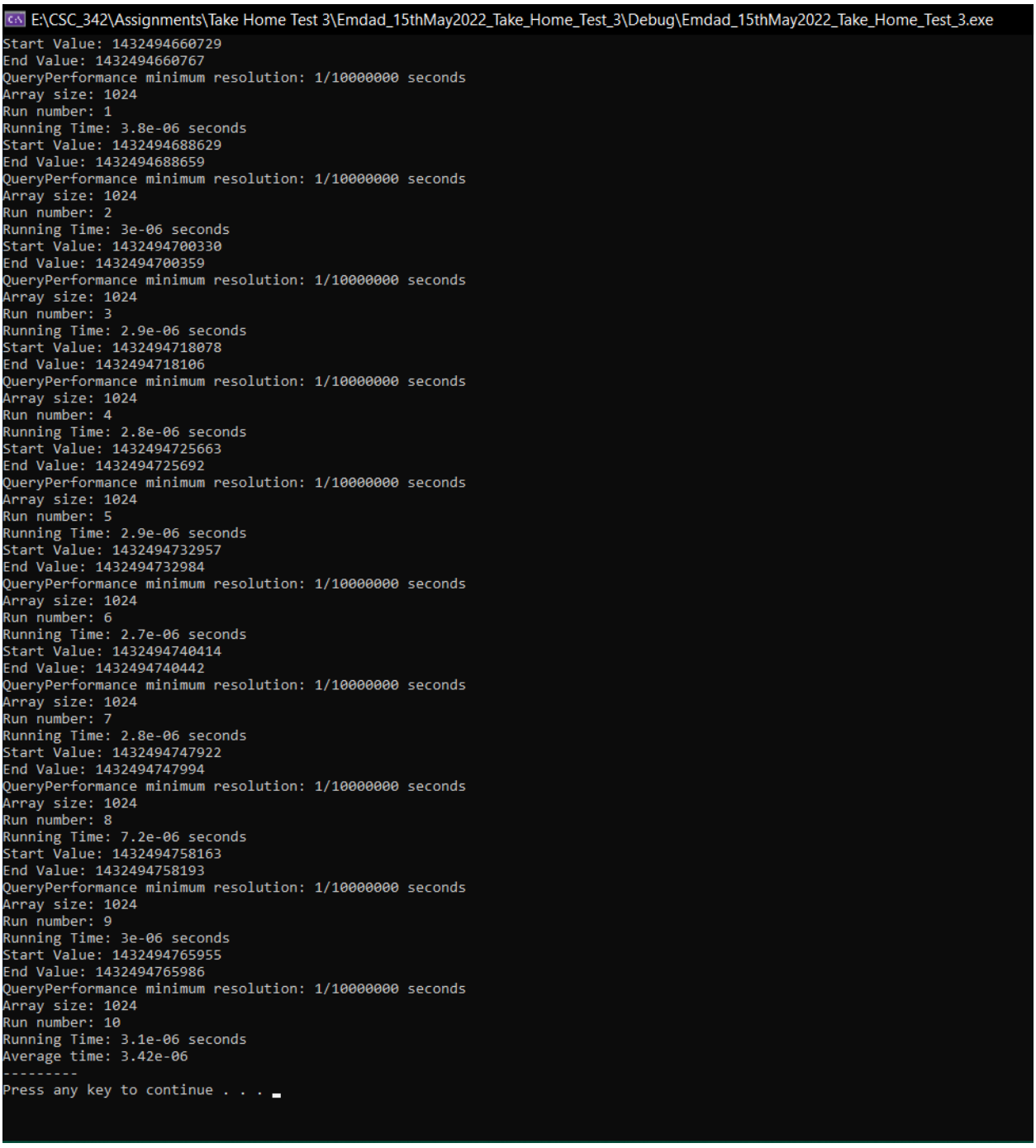


Figure 69: execution running time for array size 1024

In figure 70, we can see the array size average execution running time for array size 2048.

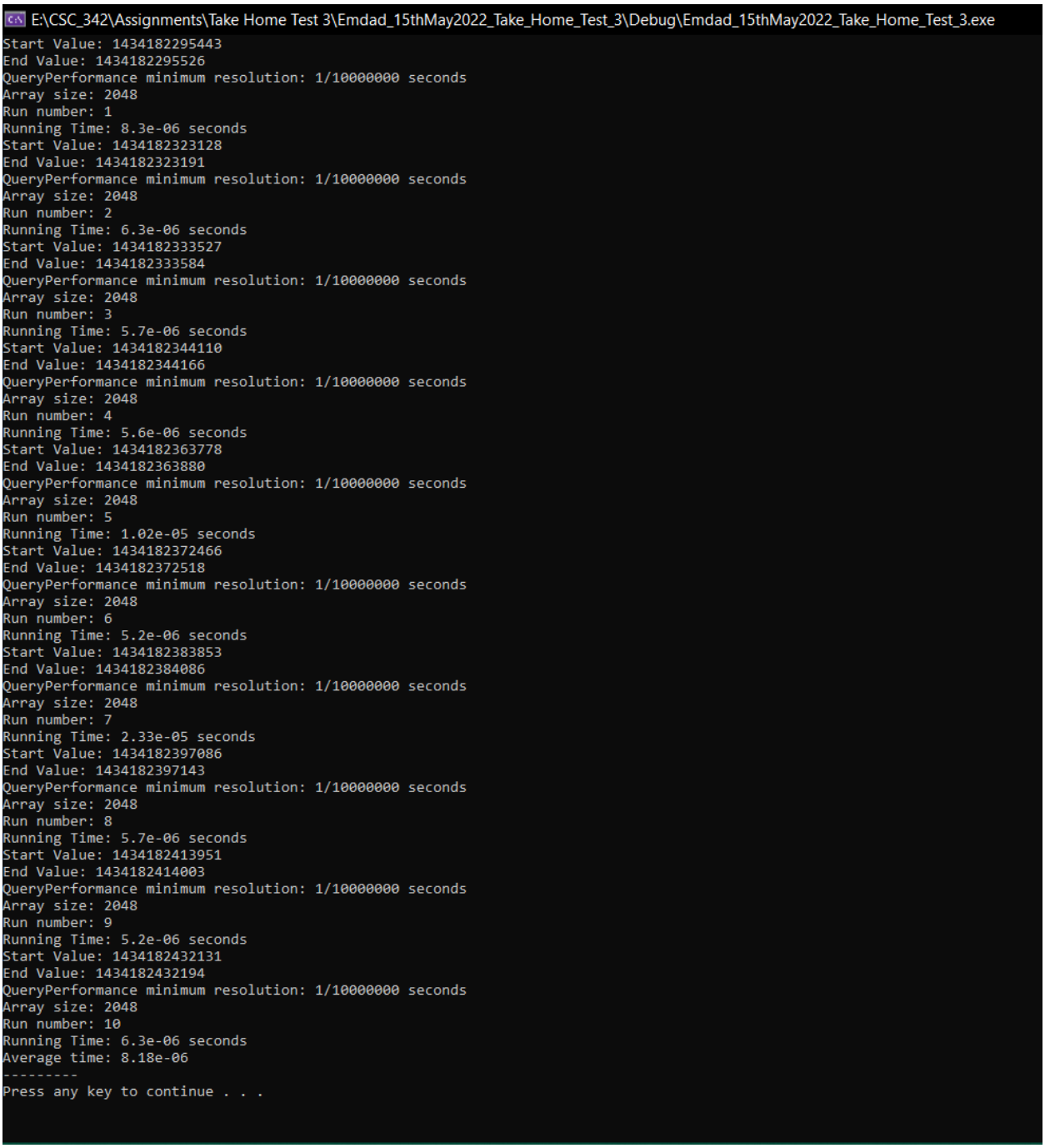


Figure 70: execution running time for array size 2048

In figure 71, we can see the array size average execution running time for array size 4096.

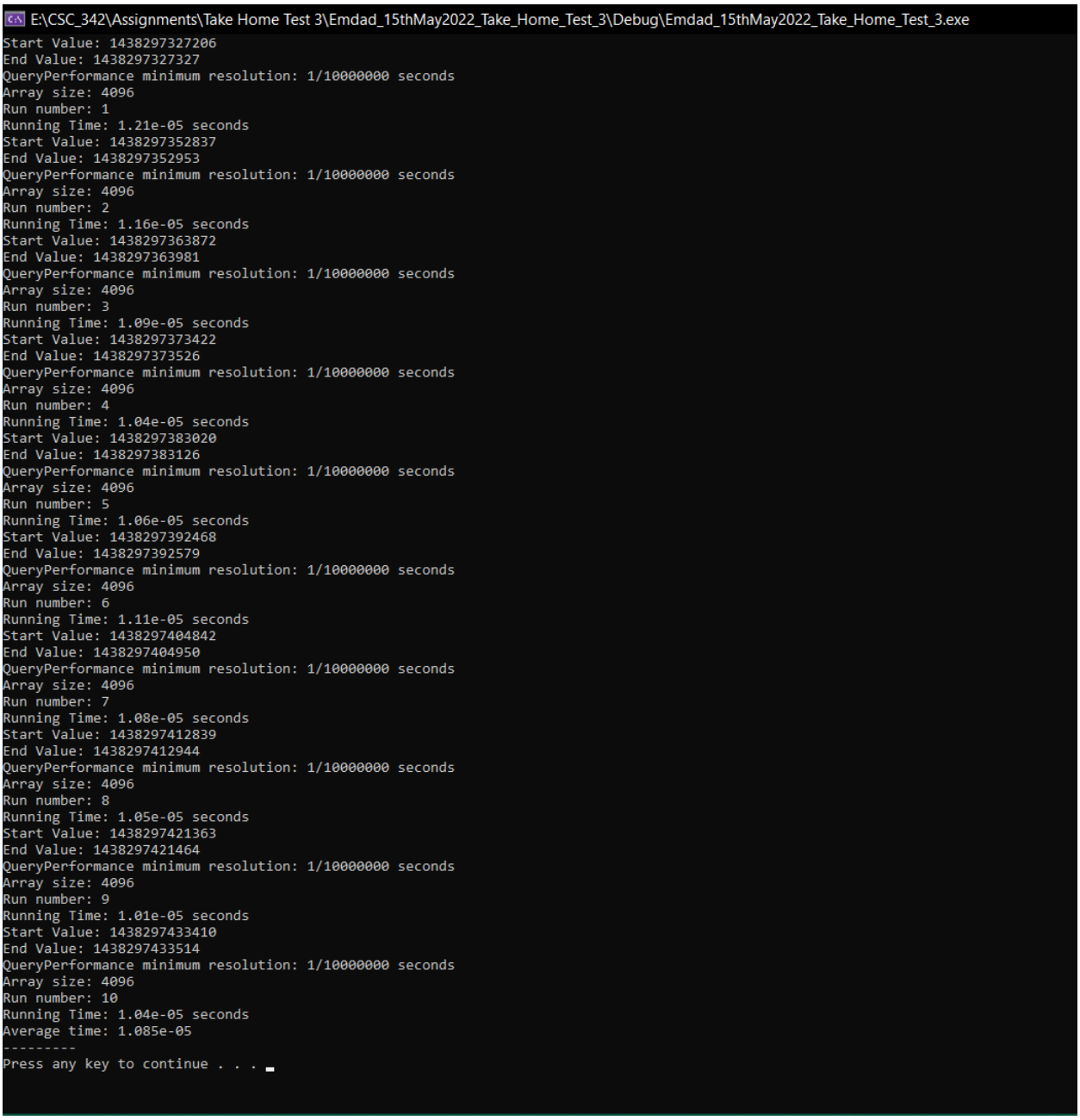


Figure 71: execution running time for array size 4096

In figure 72, we can see that I made a table for the average runtime based on the array sizes.

Array Size	Average Running Time (seconds)
8	3.10E-07
16	2.20E-07
32	3.20E-07
64	4.30E-07
128	6.40E-07
256	9.00E-07
512	1.94E-06
1024	3.42E-06
2048	8.18E-06
4096	1.09E-05

Figure 72: : Average Runtime Table (VDPPS)

In figure 73, we can see that I made a graph for the average runtime based on the array sizes. We can how the runtime went as the array size increased.

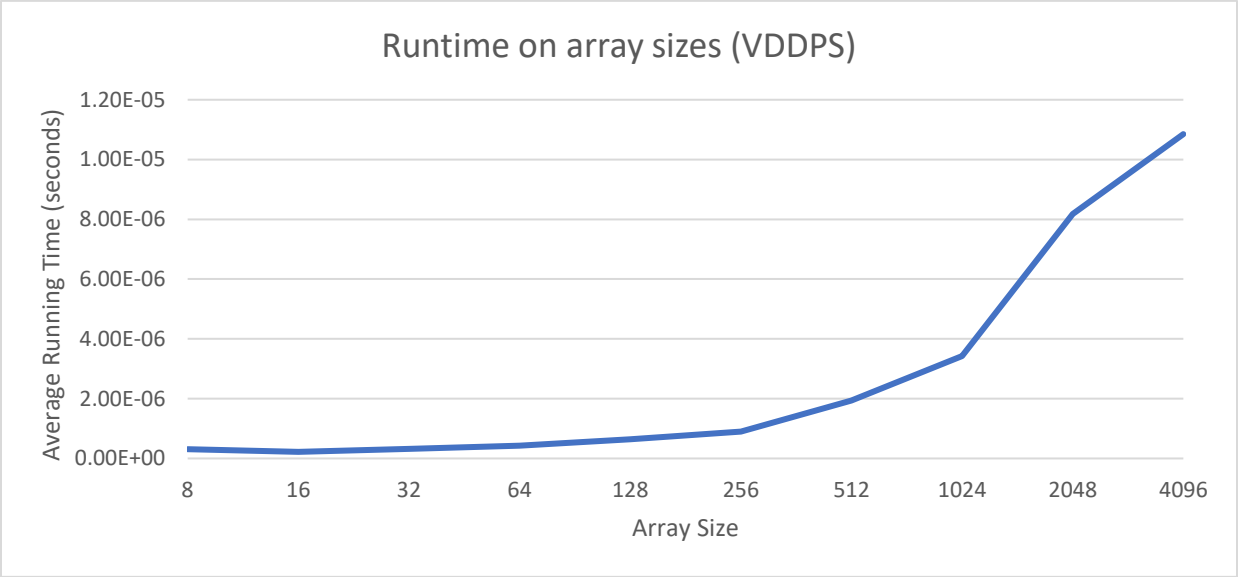


Figure 73: Runtime graph (VDPPS)

Intel x64 (Linux g++):

Dot Product [Non-Optimized]:

In figure 74, we can see the average time for array size {8,16,32,64,128,256,512,1024,2048,4096}. We did the execution in order. We ran **g++ -o0 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out** to compile the files. It generated an a.out output file. Then we did **./a.out** to execute the file.

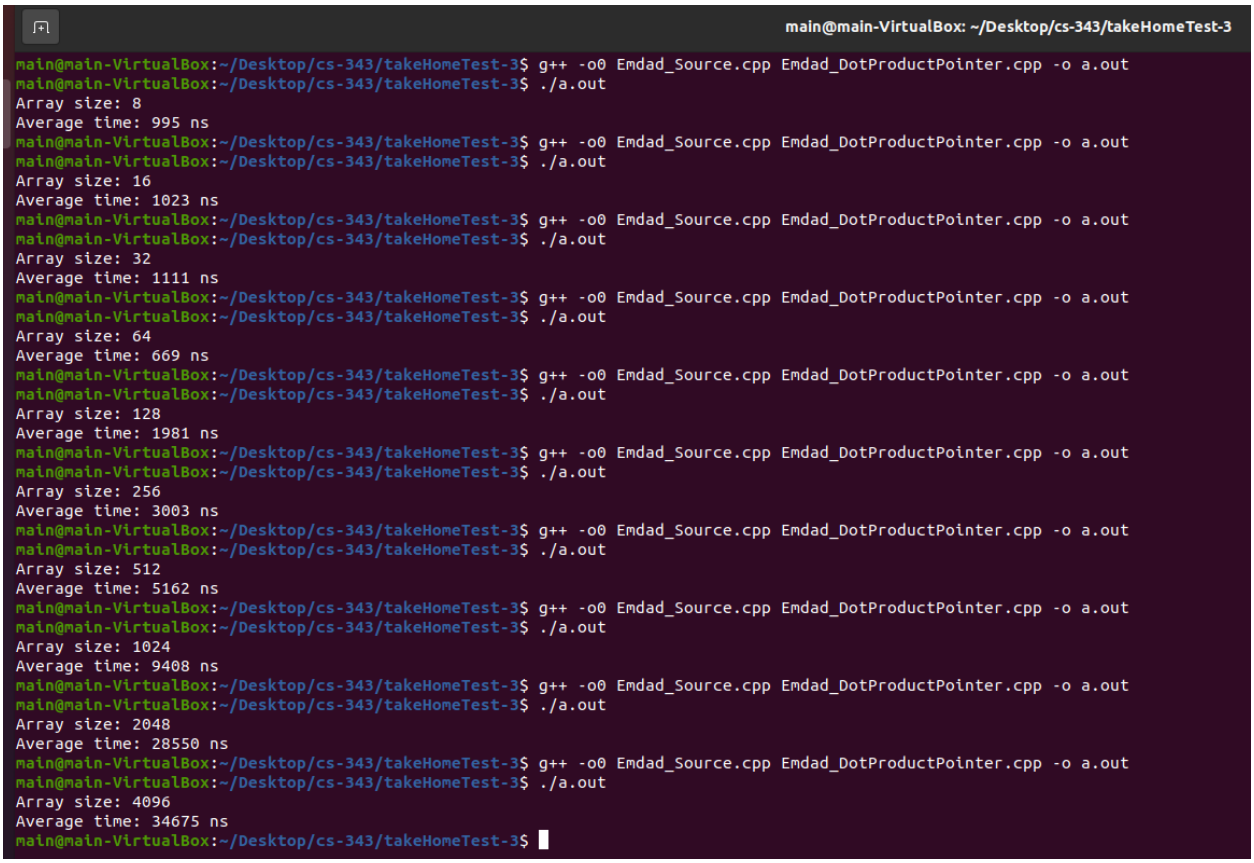


Figure 74: Runtime on array sizes(non-Optimized)

In figure 75, we can see that I made a table for the average runtime based on the array sizes.

Array Size	Average Running Time (ns)
8	995
16	1023
32	1111
64	669
128	1981
256	3003
512	5162
1024	9408
2048	28550
4096	34675

Figure 75: : Average Runtime Table (Non-Optimized)

In figure 76, we can see that I made a graph for the average runtime based on the array sizes. We can how the runtime went as the array size increased.

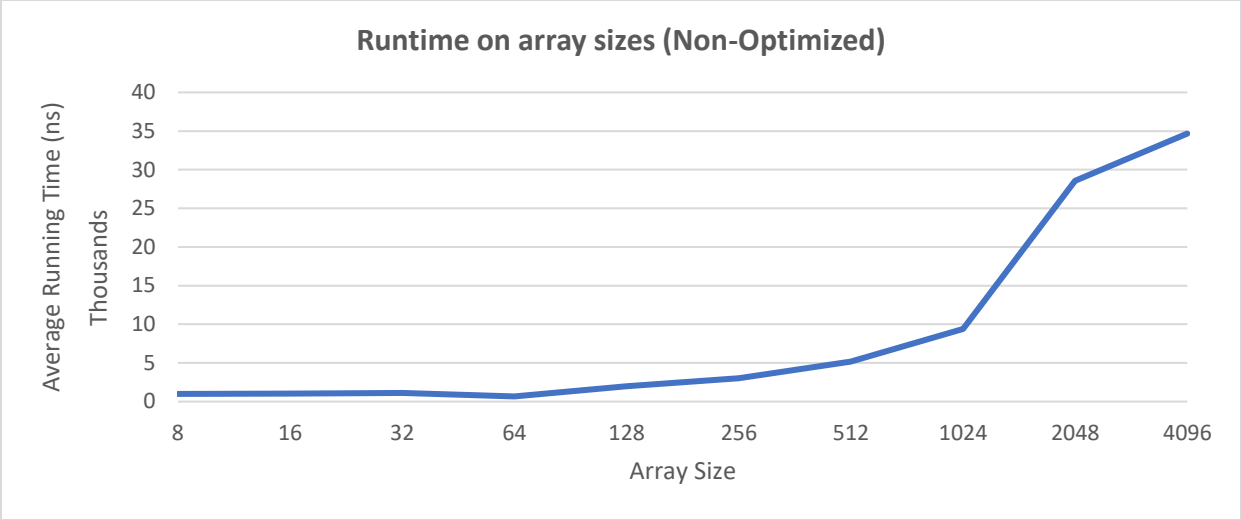


Figure 76: Runtime graph (non-Optimized)

Dot Product [Compiler Generated Optimization]:

In figure 77, we can see the average time for array size {8,16,32,64,128,256,512,1024,2048,4096}. We did the execution in order. We ran **g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out** to compile the files. It generated an a.out output file. Then we did **./a.out** to execute the file.

```
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 8
Average time: 654 ns
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 16
Average time: 744 ns
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 32
Average time: 1202 ns
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 64
Average time: 1425 ns
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 128
Average time: 1943 ns
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 256
Average time: 2778 ns
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 512
Average time: 3256 ns
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 1024
Average time: 9433 ns
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 2048
Average time: 17757 ns
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ g++ -o1 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out
main@main-VirtualBox:~/Desktop/cs-343/takeHomeTest-3$ ./a.out
Array size: 4096
Average time: 47222 ns
```

Figure 77: Runtime on array sizes (Compiler Generated Optimization)

In figure 78, we can see that I made a table for the average runtime based on the array sizes.

Array Size	Average Running Time (ns)
8	654
16	744
32	1202
64	1425
128	1943
256	2778
512	3256
1024	9433
2048	17757
4096	47222

Figure 78: Average Runtime Table (Compiler Generated Optimization)

In figure 79, we can see that I made a graph for the average runtime based on the array sizes. We can how the runtime went as the array size increased.

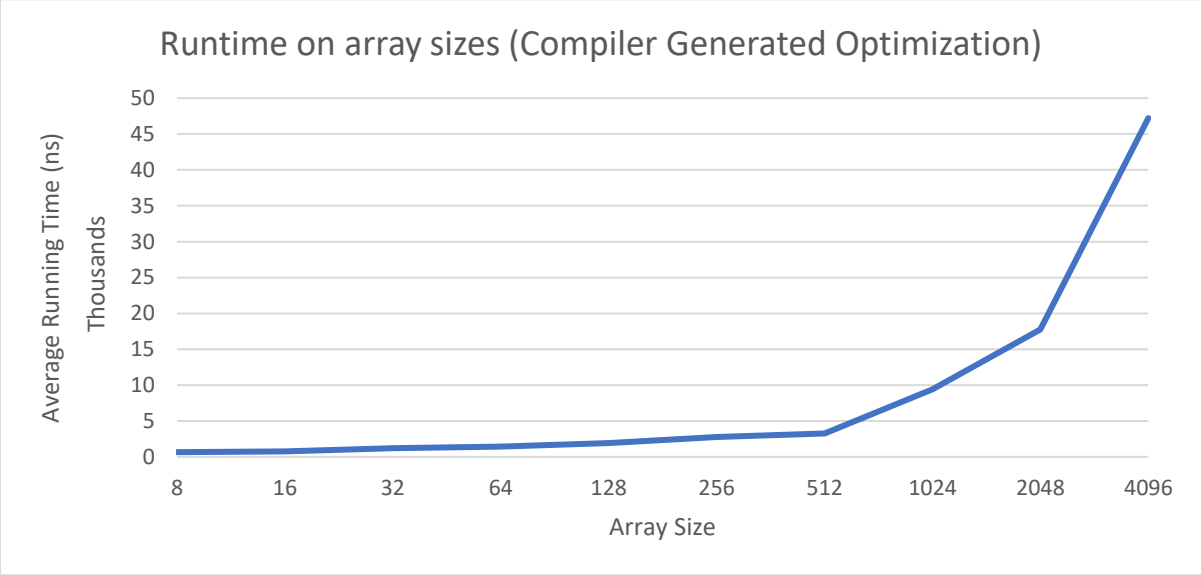


Figure 79: Runtime graph (Compiler Generated Optimization)

Dot Product [Manually Optimized]:

In figure , we can see the average time for array size {8,16,32,64,128,256,512,1024,2048,4096}. We did the execution in order. We ran **g++ -o2 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out** to compile the files. It generated an a.out output file. Then we did **./a.out** to execute the file.

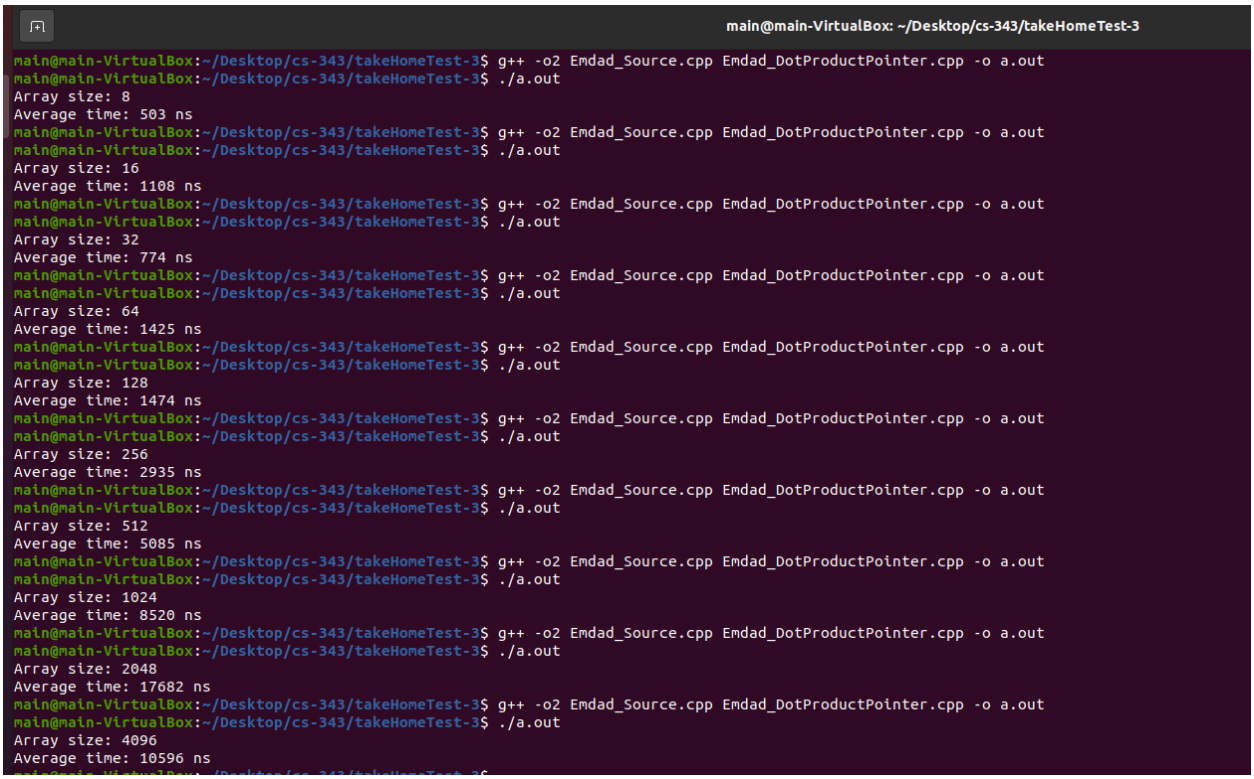


Figure 80: Runtime on array sizes (Manually Optimized)

In figure 81, we can see that I made a table for the average runtime based on the array sizes.

Array Size	Average Running Time (ns)
8	503
16	1108
32	774
64	1425
128	1474
256	2935
512	5085
1024	8520
2048	10682
4096	12596

Figure 81: Average Runtime Table (Manually Optimized)

In figure 82, we can see that I made a graph for the average runtime based on the array sizes. We can how the runtime went as the array size increased.

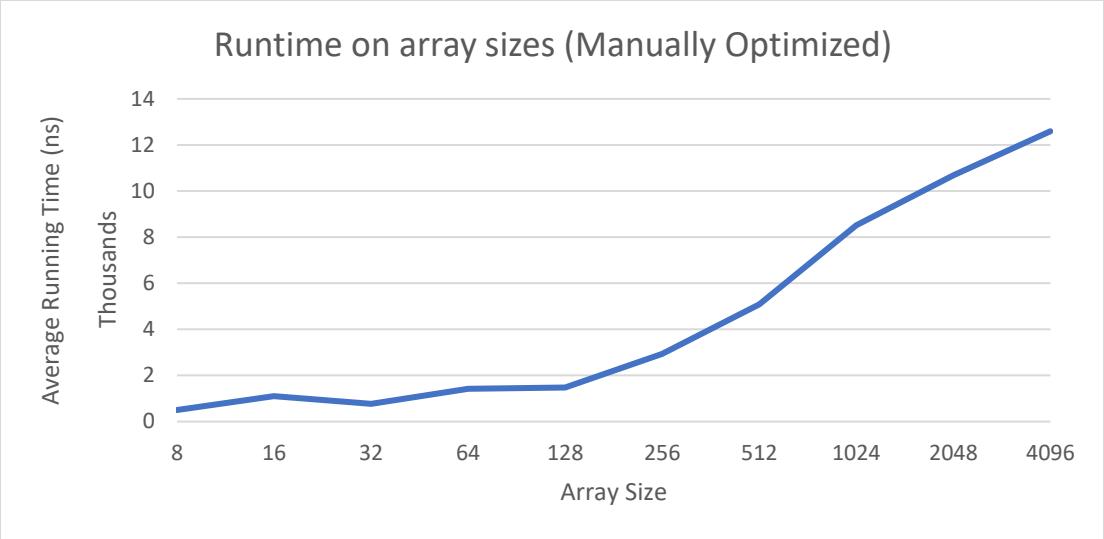


Figure 82: Runtime graph (Manually Optimized)

VDPPS:

In figure , we can see the average time for array size {8,16,32,64,128,256,512,1024,2048,4096}. We did the execution in order. We ran **g++ -o3 Emdad_Source.cpp Emdad_DotProductPointer.cpp -o a.out** to compile the files. It generated an a.out output file. Then we did **./a.out** to execute the file.

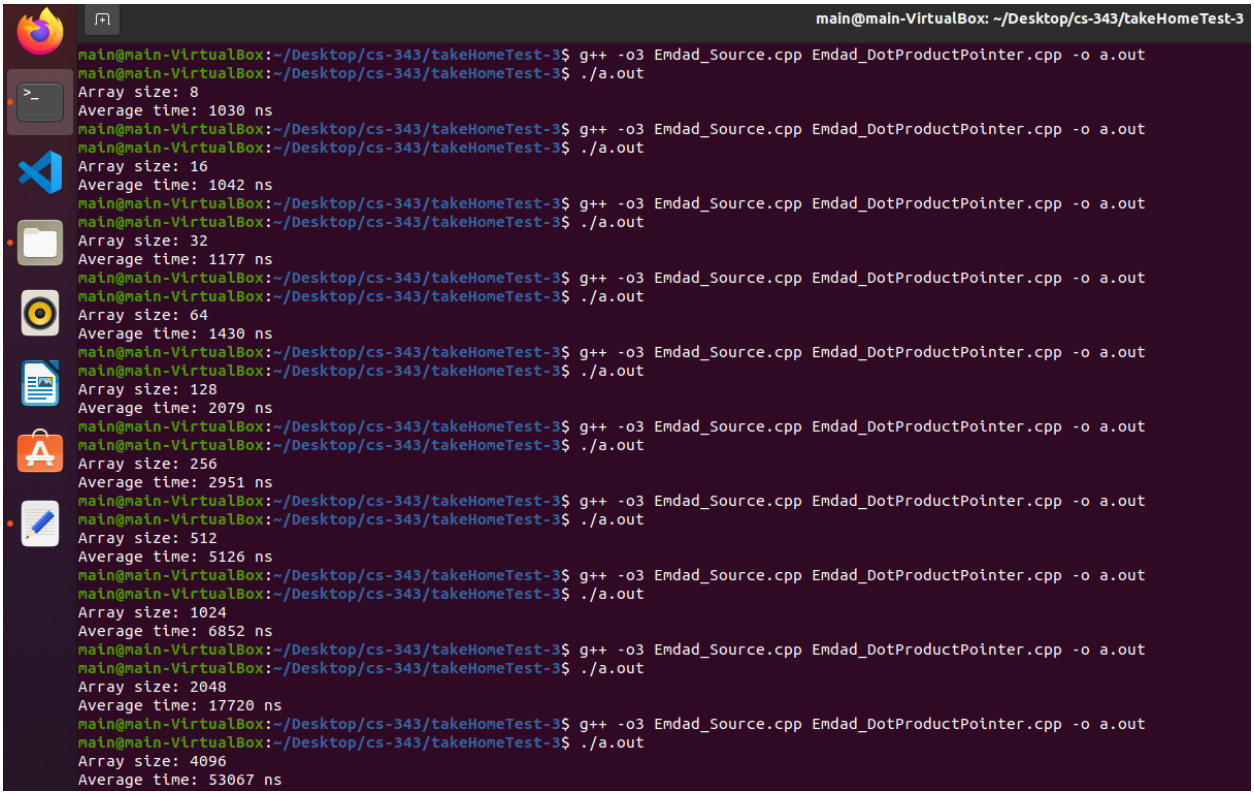


Figure 83: Runtime on array sizes (VDPPS)

In figure 84, we can see that I made a table for the average runtime based on the array sizes.

Array Size	Average Running Time (ns)
8	1030
16	1042
32	1177
64	1430
128	2079
256	2951
512	5126
1024	6852
2048	17720
4096	53067

Figure 84: Average Runtime Table (VDPPS)

In figure 85, we can see that I made a graph for the average runtime based on the array sizes. We can how the runtime went as the array size increased.

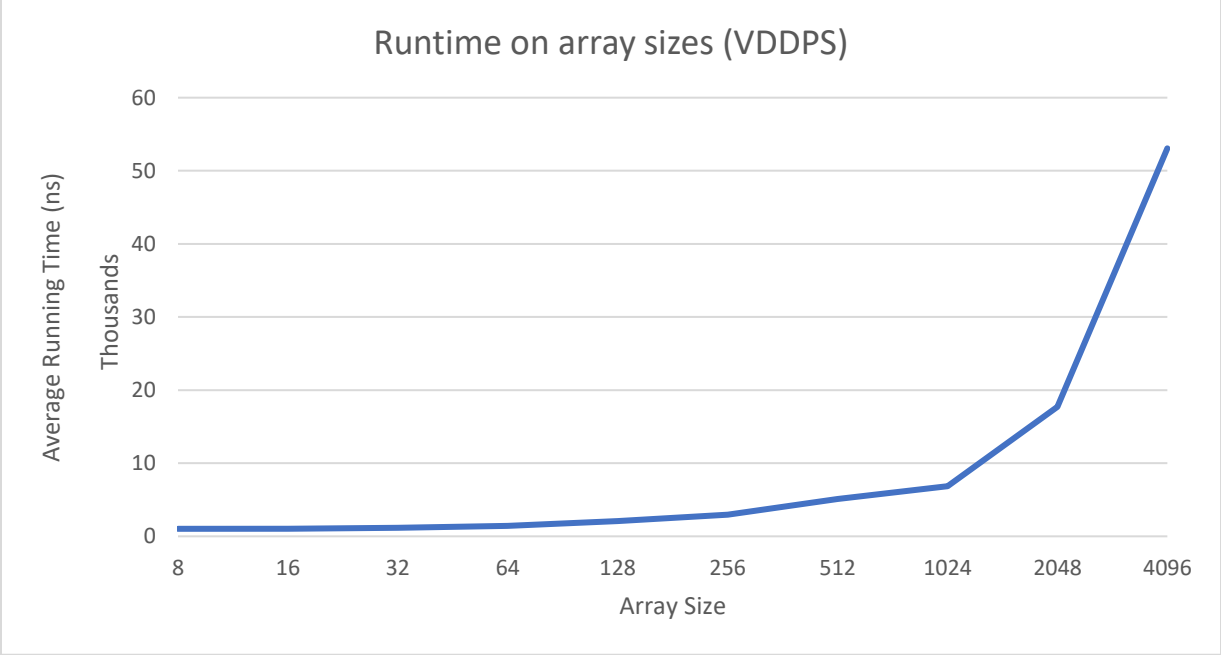


Figure 85: Runtime graph (VDPPS)

Explanation:

Intel x86 (Visual Studio Code):

In figure 86, we can see that I took the average time from all the methods to run the program for each vector array size and put them in a single table. I am going to make a graph out of this table to visualize to check which method is faster than others.

Array Size	Non-Optimized (seconds)	Compiler Generated Optimization (seconds)	Manually Optimized (seconds)	VDDPS (seconds)
8	2.60E-07	4.50E-07	2.20E-07	3.10E-07
16	2.70E-07	5.00E-07	2.90E-07	2.20E-07
32	4.00E-07	4.90E-07	3.10E-07	3.20E-07
64	5.70E-07	8.20E-07	4.70E-07	4.30E-07
128	6.90E-07	1.46E-06	5.90E-07	6.40E-07
256	9.80E-07	1.13E-06	1.24E-06	9.00E-07
512	2.36E-06	2.70E-06	2.46E-06	1.94E-06
1024	3.43E-06	3.86E-06	3.03E-06	3.42E-06
2048	7.58E-06	9.03E-06	5.96E-06	8.18E-06
4096	1.60E-05	1.85E-05	1.16E-05	1.09E-05

Figure 86: Average Runtime Table (all methods)

In figure 87, we can see the generated graph from the table(figure 86).

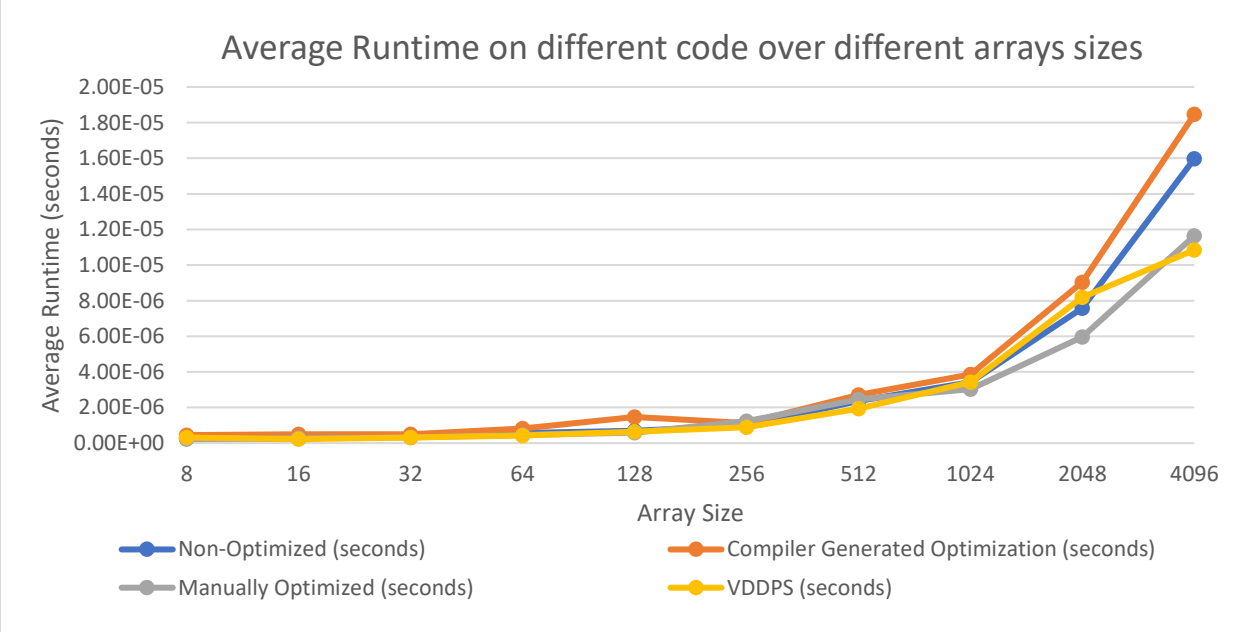


Figure 87: Average Runtime graph (all methods)

The more we optimized the code the more runtime decreased. We can see that VDPPS instruction has the lowest runtime meaning it ran faster than other methods. It is because VDPPS instruction used vectorization and parallelization which is more efficient for the larger array sizes and can be noticeable.

Intel x64 (Linux g++):

In figure 88, we can see that I took the average time from all the methods to run the program for each vector array size and put them in a single table. I am going to make a graph out of this table to visualize to check which method is faster than others.

Array Size	Non-Optimized (ns)	Compiler Generated Optimization (ns)	Manually Optimized (ns)	VDDPS (ns)
8	995	654	503	1030
16	1023	744	1108	1042
32	1111	1202	774	1177
64	669	1425	1425	1430
128	1981	1943	1474	2079
256	3003	2778	2935	2951
512	5162	3256	5085	5126
1024	9408	9433	8520	6852
2048	28550	17757	10682	17720
4096	34675	47222	12596	53067

Figure 88: Average Runtime Table (all methods)

In figure 89, we can see the generated graph from the table(figure 88).

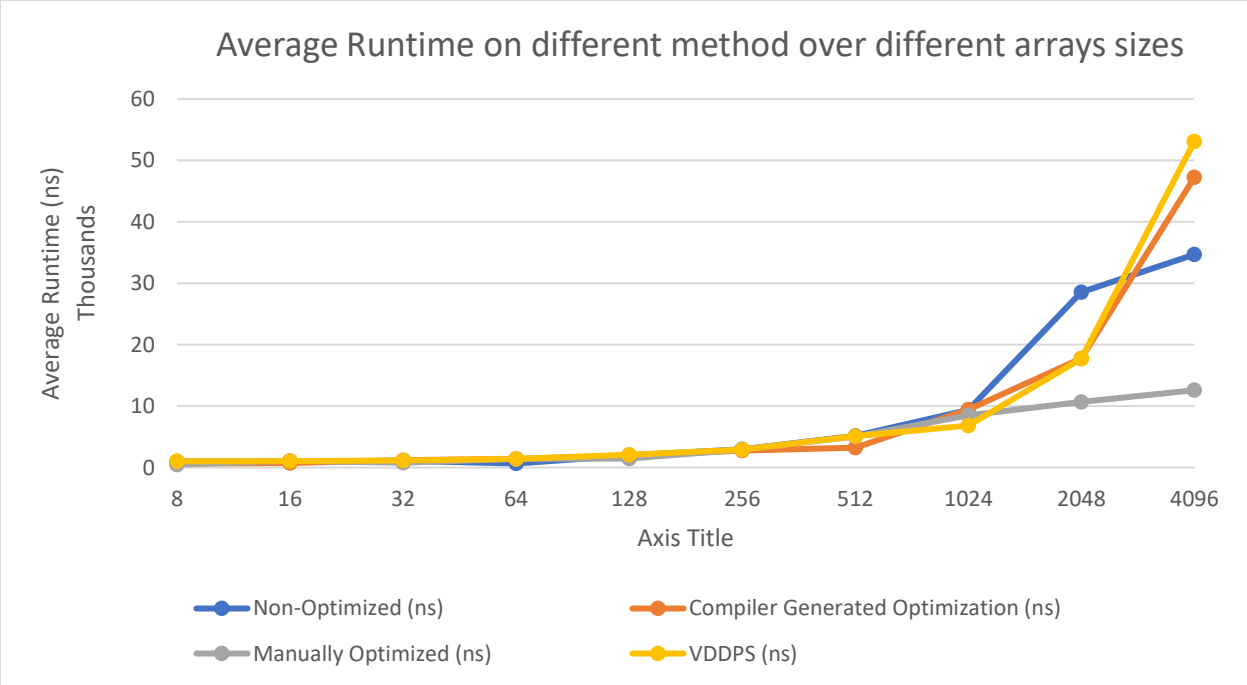


Figure 89: Average Runtime graph (all methods)

The more we optimized the code the more runtime decreased. We can see that our manually optimized code is executing faster than any other methods. Interestingly, VDPPS instruction is taking longer here than any other method. I am assuming because I used virtual box which has limited processor and depend on the parent computer, there is a drastic difference runtime.

If we compare both graphs from both environments how drastically optimizing code can reduce the runtime. It is more noticeable while working with larger array sizes. The manual optimized runtime changed more than other methods. The vector instruction, VDPPS made the program running faster.

Conclusion:

This take-home test taught me how to compute and demonstrate dot products using vector instructions and optimize the code by compiler generate in both Intel x86(MS VS) and Intel x64(Linux) environment. I also learned to use parallelization, vectorization to optimize the code. I also manually optimized the code. I learned to see how much optimal code matter while working with a large data set. I will remember than for my future code projects.