fkamal000@citymail.cuny.edu

[Email address]

Professor Gersten

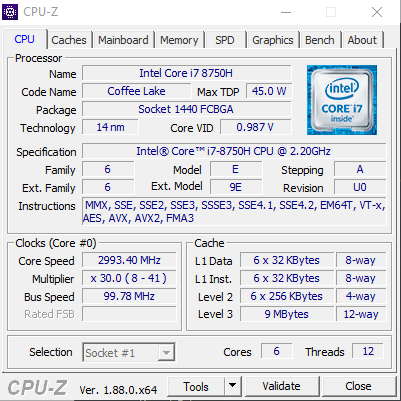
cS342: take Home Test Final

Faheem Kamal

**Objective**

Today, developers can write intuitive code at high-level languages and have them executed in a reasonably fast way. Modern compilers prove to be extremely efficient at translating these programs written in these high-level languages such as C, C++, Java and Python to Assembly code. To understand the capabilities that these compilers have, our goal is to test and run a dot product function, using pointers and arrays with and without optimization features. The time needed to run is measured with various array sizes based on the power of 2. Afterwards, we will compare the runtime between each. Afterwards, we will use methods like parallelization or vectorization to optimize the code when the arrays become large.

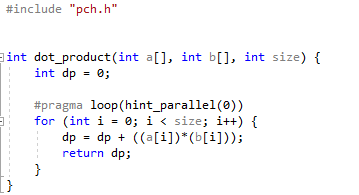
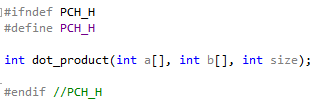
**CPU Specs**

****

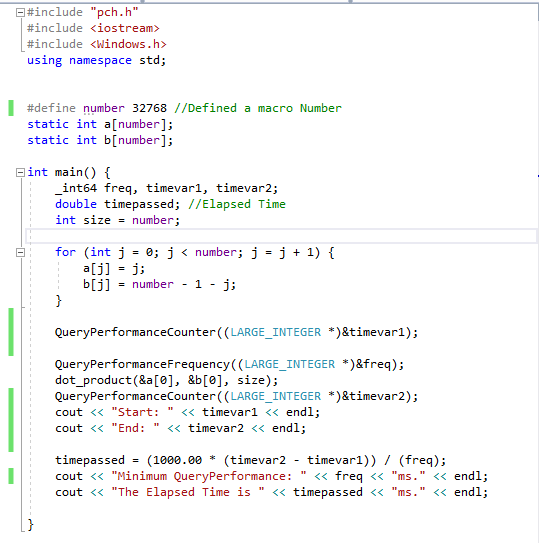
This CPU supports Vector Processing. This contains AVX (Advanced Vector Extensions) and AVX2 (Advanced Vector Extensions 2) Instruction Sets. AVX are a set of High-Performance Instructions introduced in 2008. They perform many of the same operations as SSE but on larger chunks of data at higher speed. With AVX2, we have the following new additions:

1. Expansion of Most Vector Integer SSE and AVX Instructions to 256 Bits
2. Three Operand General-Purpose Bit Manipulation and Multiplications
3. Vector Elements can be loaded from Non-Contiguous Memory Locations
4. DWORD and QWORD granularity any-to-any permutes
5. Vector Shifts

**Visual Studios on Intel**



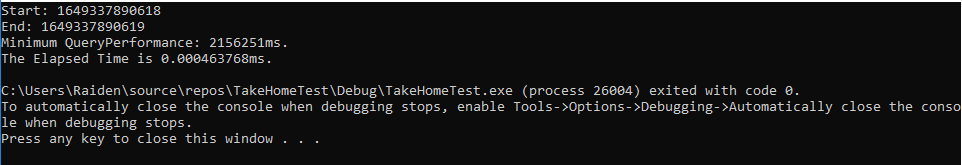
Dot Product Calculation is implemented using an array index notation.



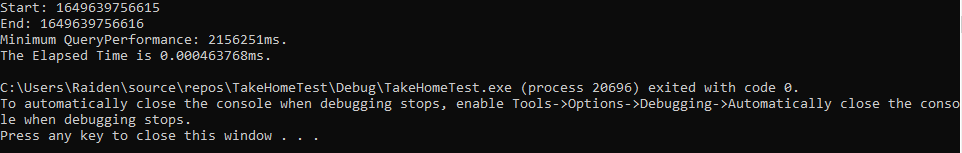
The main function is shown above here. We declare a macro number which in this case is 32768. It will be changed several times to measure different times at various powers of 2. We will explore that in this section. Furthermore, two static arrays have been declared of the same size. The values are arbitrary numbers within each array and are irrelevant to the purpose of this program. QueryPerformanceCounter will retrieve the current value of the performance counter of the high-resolution timer and is used to measure the run time of both dot product functions. QueryPerformanceFrequency will return the current performance-counter frequency, in counts per millisecond. Both functions provide value information of high-resolution performance counters in the system.

Non-Optimized

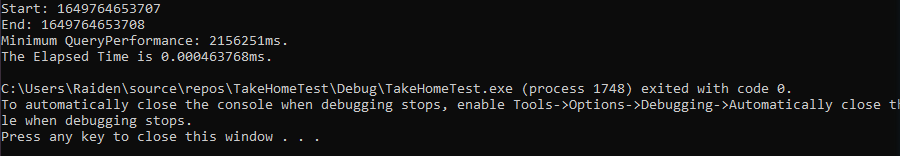
Here, Optimization, Auto-Vectorization and Auto-Parallelization are disabled in Visual Studios. We measure the time for both dot product functions at array sizes number = 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 and 32768. For each array size, I ran the function at least 5 times.



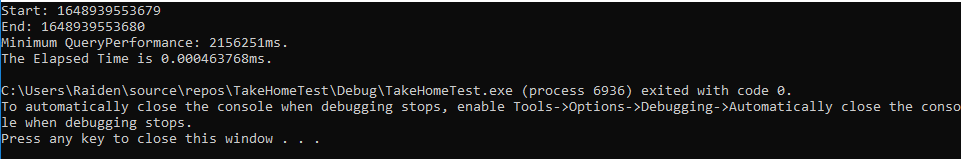
This is for size number = 2. The average came out to be 0.0004637 ms.



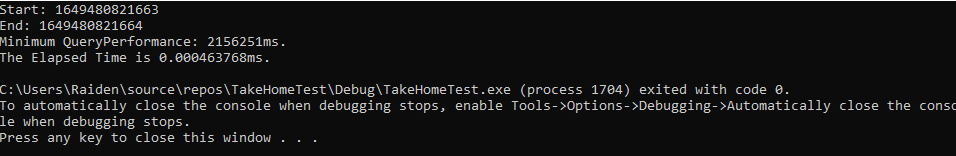
This is for size number = 4. The average came out to be 0.0004637 ms.



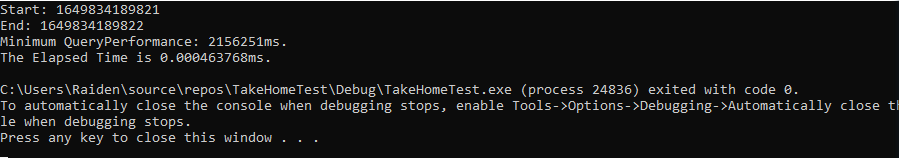
This is for size number = 8. The average came out to be 0.0004637 ms.



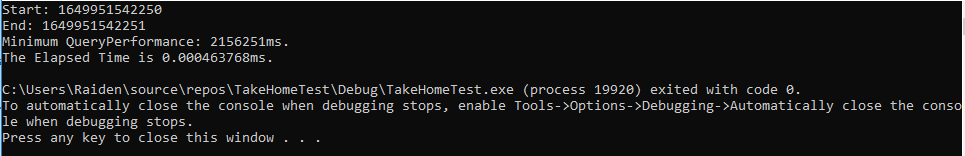
This is for size number = 16. The average came out to be 0.0004637 ms.



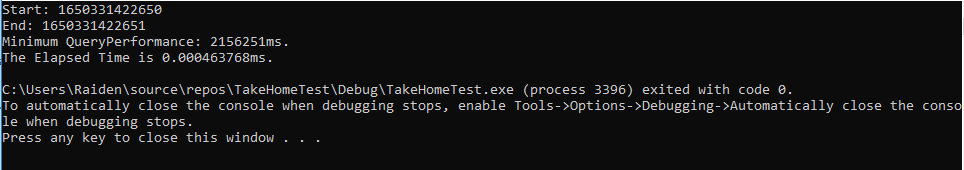
This is for size number = 32. The average came out to be 0.0004637 ms.



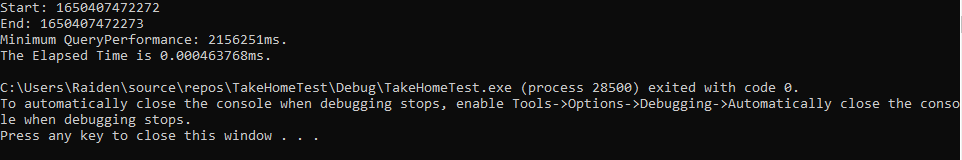
This is for size number = 64. The average came out to be 0.0004637 ms.



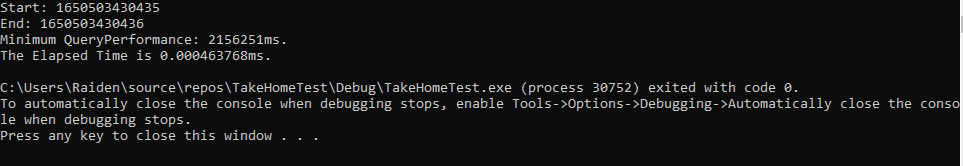
This is for size number = 128. The average came out to be 0.0004637 ms.



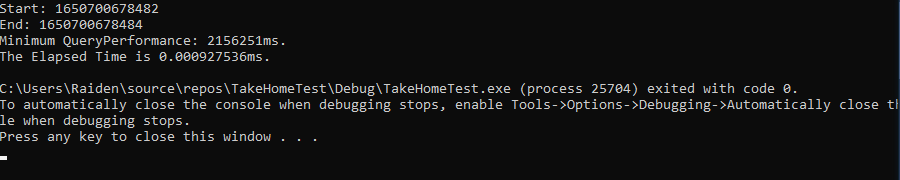
This is for size number = 256. The average came out to be 0.0004637 ms.



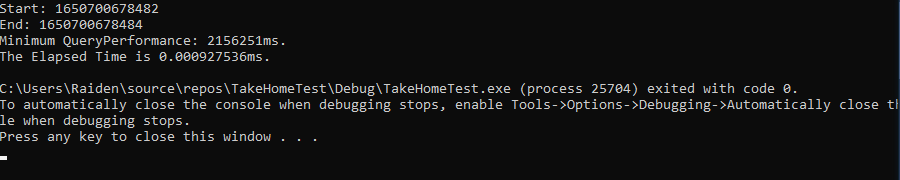
This is for size number = 512. The average came out to be 0.0004637 ms.



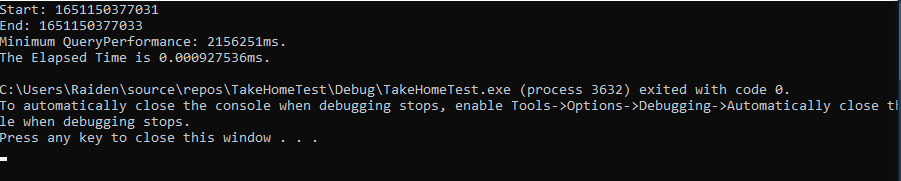
This is for size number = 1024. The average came out to be 0.0004637 ms.



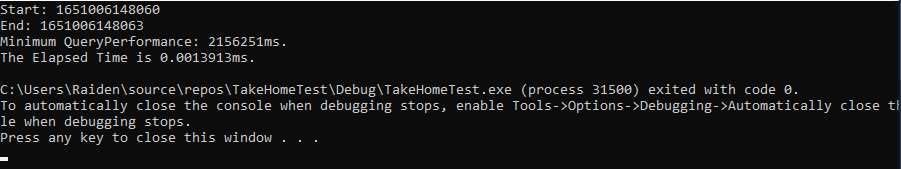
This is for size number = 2048. The average came out to be 0.00092753ms.



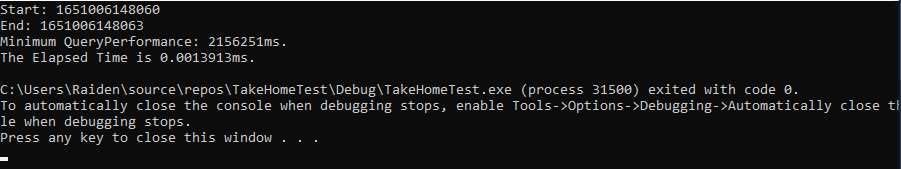
This is for size number = 4096. The average came out to be 0.00092753 ms.



This is for size number = 8192. The average came out to be 0.00092753 ms.

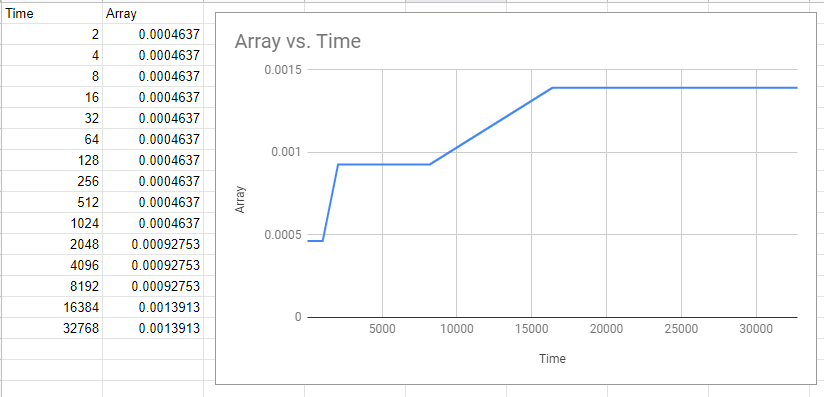


This is for size number = 16384. The average came out to be 0.0013913 ms.



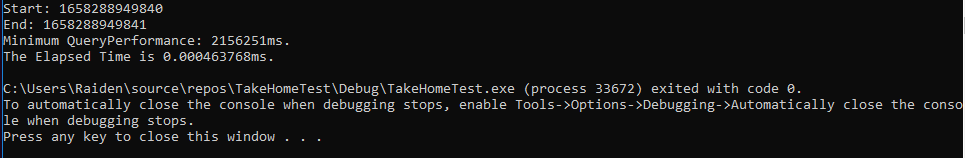
This is for size number = 32768. The average came out to be 0.0013913 ms.

**Graph for Non-Optimized**

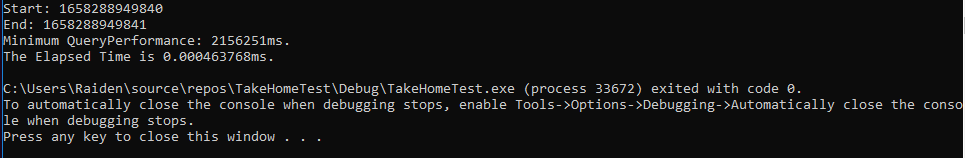


**Optimized**

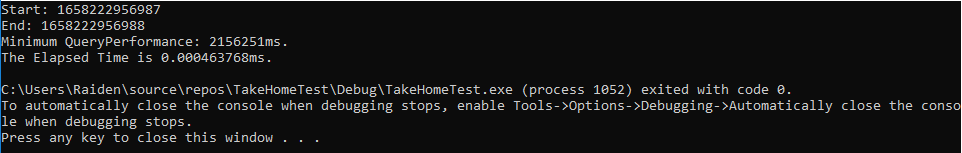
Next, we will have vectorization and parallelization is enabled. We measure the time for both dot product functions at array sizes number = 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 and 32768. For each array size, I ran the function at least 5 times.



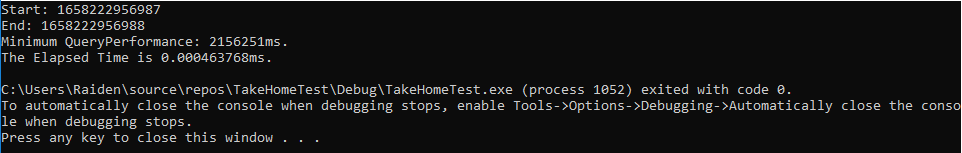
This is for size number = 2. The average came out to be 0.0004637 ms.



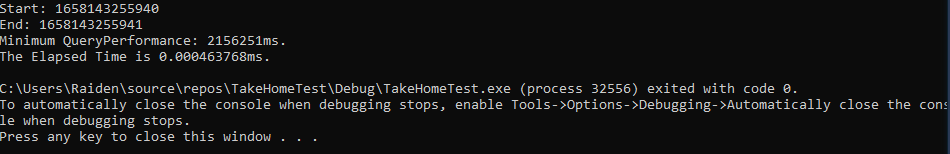
This is for size number = 4. The average came out to be 0.0004637 ms.



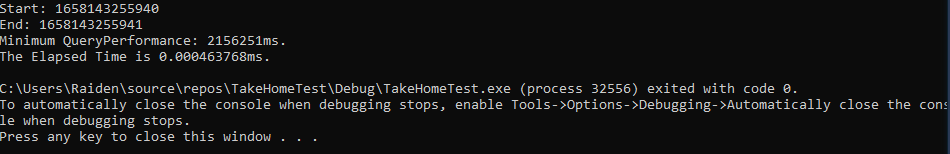
This is for size number = 8. The average came out to be 0.0004637 ms.



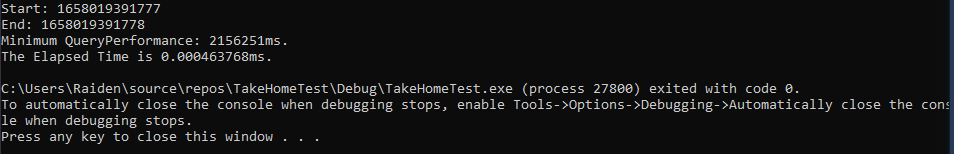
This is for size number = 16. The average came out to be 0.0004637 ms.



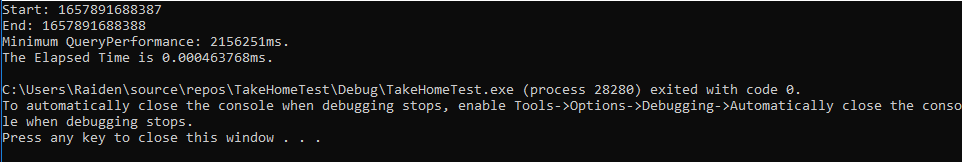
This is for size number = 32. The average came out to be 0.0004637 ms.



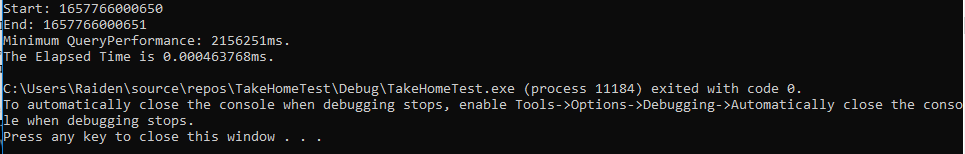
This is for size number = 64. The average came out to be 0.0004637 ms.



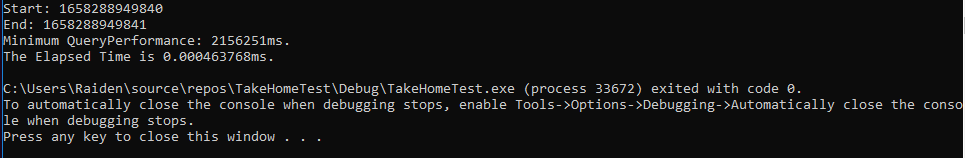
This is for size number = 128. The average came out to be 0.0004637 ms.



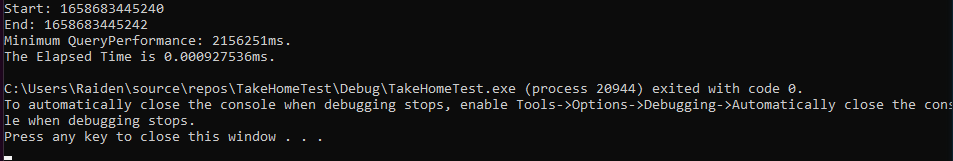
This is for size number = 256. The average came out to be 0.0004637 ms.



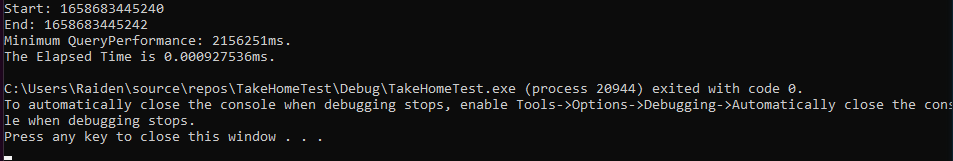
This is for size number = 512. The average came out to be 0.0004637 ms.



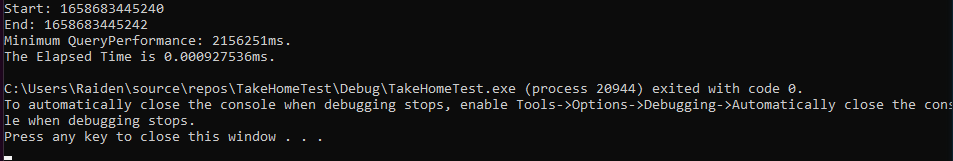
This is for size number = 1024. The average came out to be 0.0004637 ms.



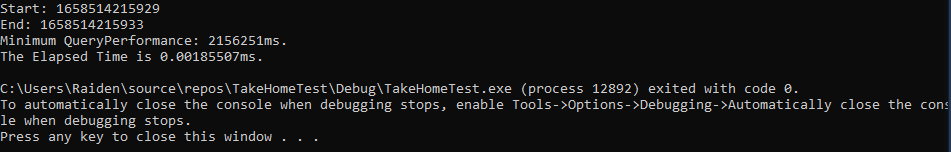
This is for size number = 2048. The average came out to be 0.00092753 ms.



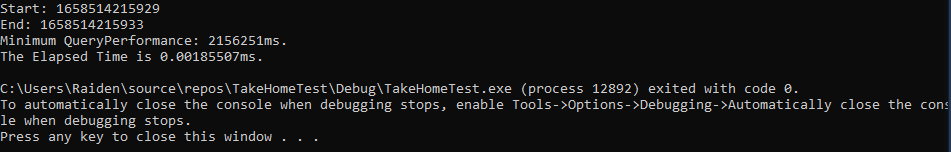
This is for size number = 4096. The average came out to be 0.00092753 ms.



This is for size number = 8192. The average came out to be 0.00092753 ms.

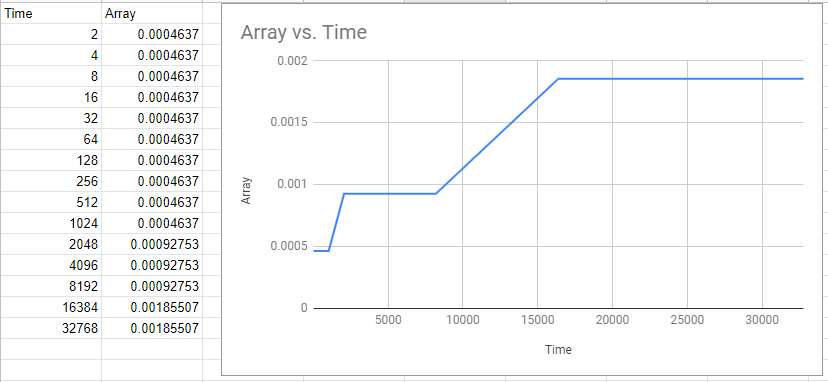


This is for size number = 16384. The average came out to be 0.00185507 ms.

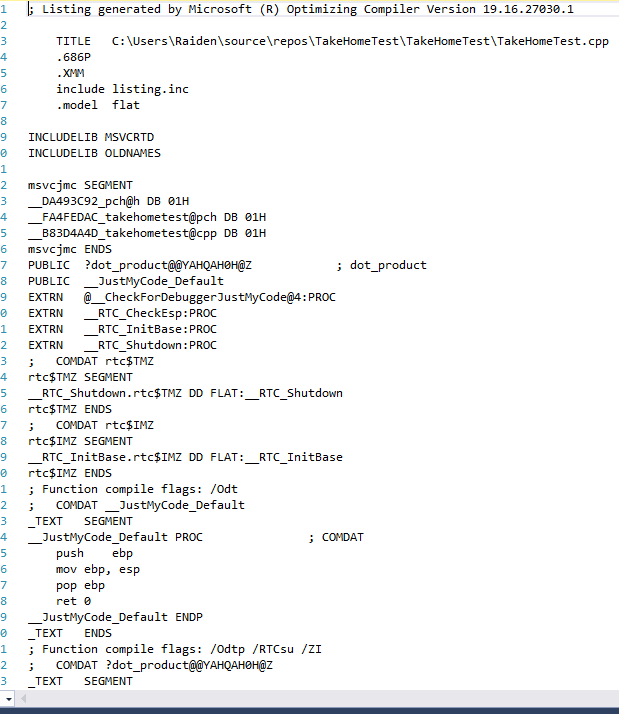


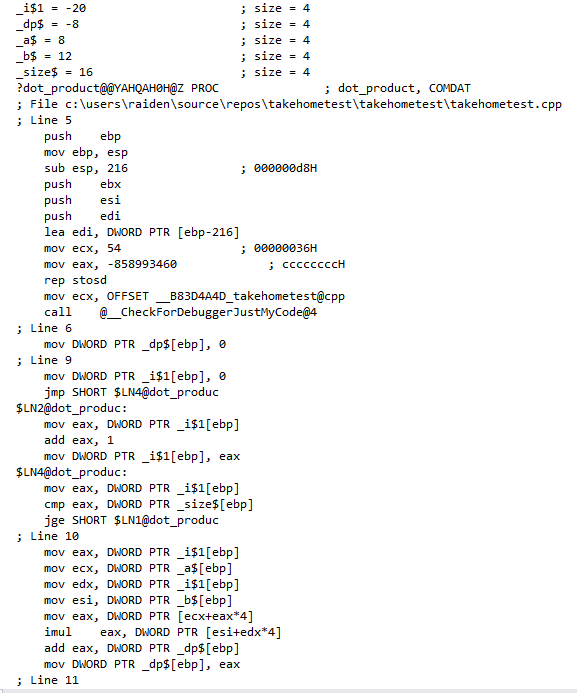
This is for size number = 32768. The average came out to be 0.00185507 ms.

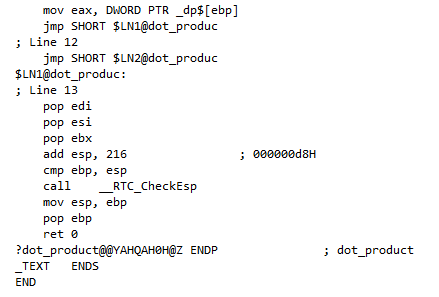
**Graph for Optimized**



**Assembler Output of Dot Product Code Generated by Visual Studios (Optimized)**



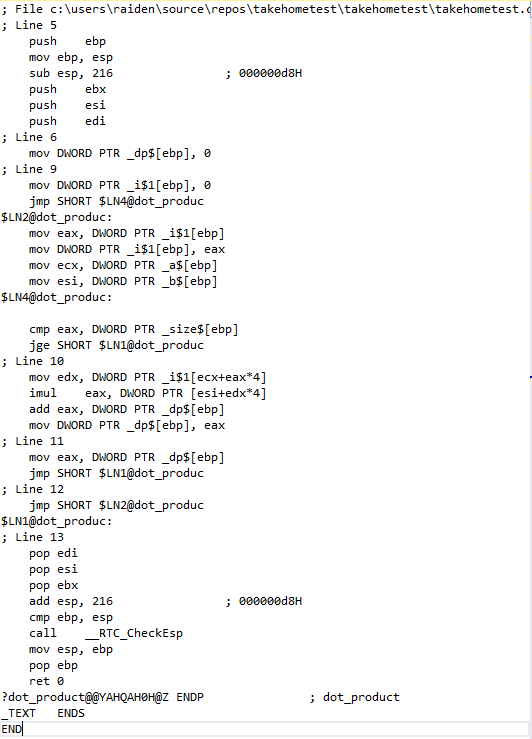




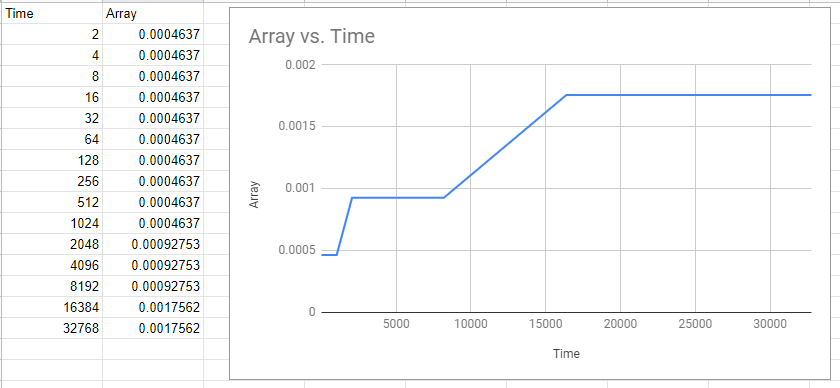
This is the assembly file generated by the compiler with no optimization.

**Manual Optimization of Assembly File**

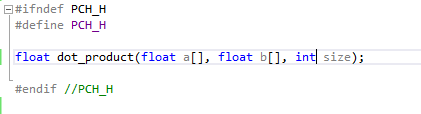
For the dot product implementation, we changed the code to this:

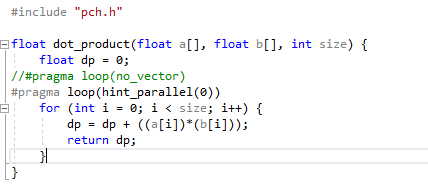


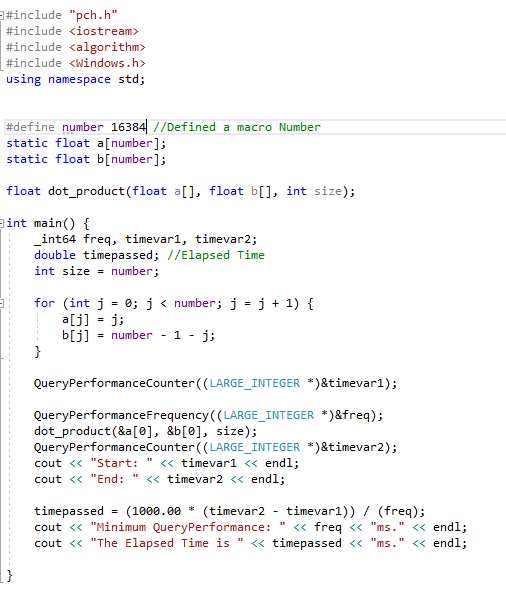
Here, we removed several inefficient code and information. Most of which were repetition that were removed. By removing unnecessary code, the program will compute dot products much faster.



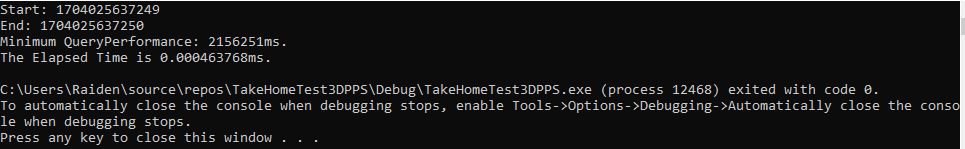
**Optimization with DPPS**



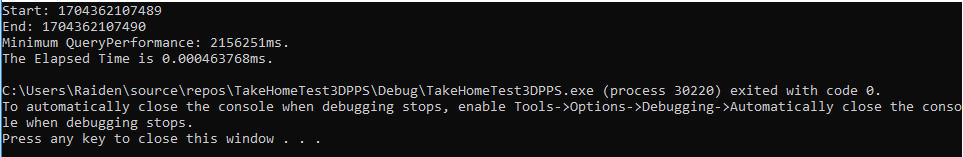




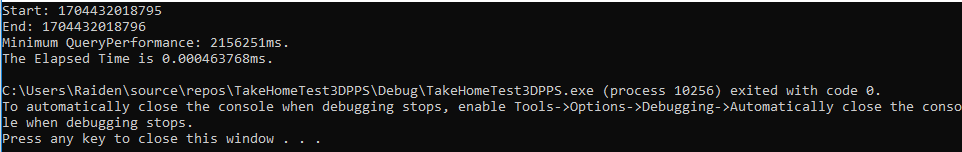
Here, we optimize using DPPS (Dot Product of Packed Single Precision Floating Point values). Multiply packed SP floating point values from xmm1 with packed SP floating point values from xmm2/mem selectively add and store to xmm1. We then measure the time for both dot product functions at array sizes number = 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 and 32768. For each array size, I ran the function at least 5 times.



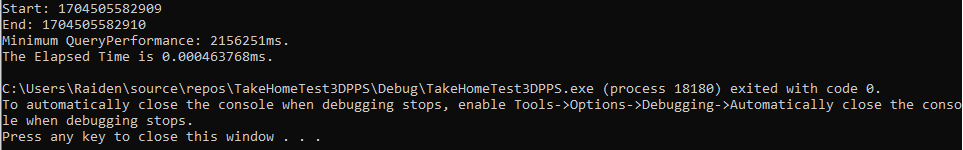
This is for size number = 2. The average came out to be 0.0004637 ms.



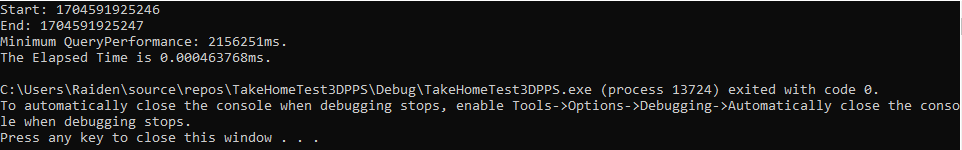
This is for size number = 4. The average came out to be 0.0004637 ms.

****

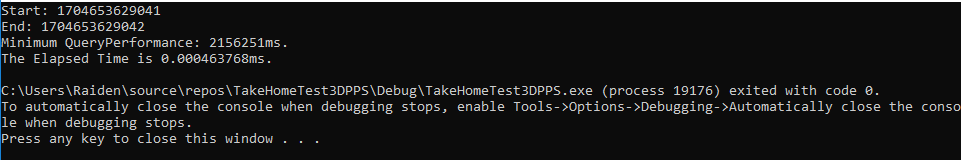
This is for size number = 8. The average came out to be 0.0004637 ms.

****

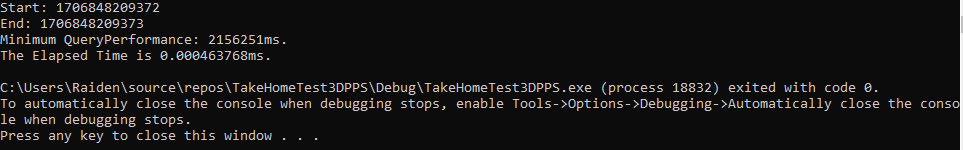
This is for size number = 16. The average came out to be 0.0004637 ms.

****

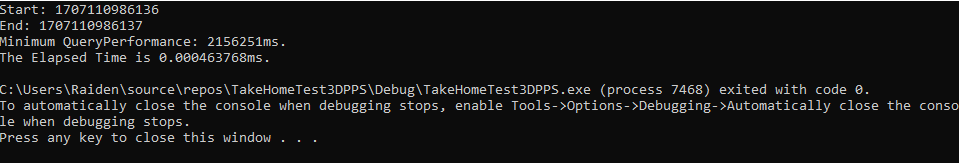
This is for size number = 32. The average came out to be 0.0004637 ms.

****

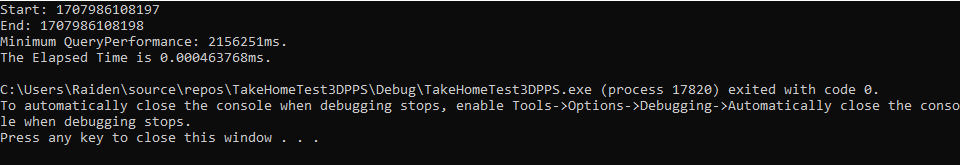
This is for size number = 64. The average came out to be 0.0004637 ms.

****

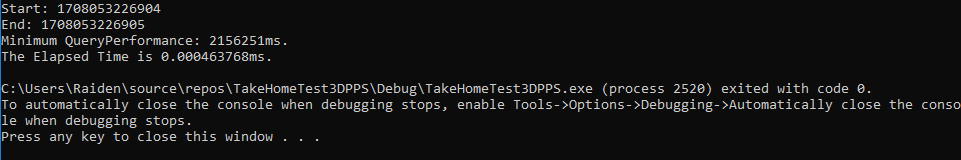
This is for size number = 128. The average came out to be 0.0004637 ms.

****

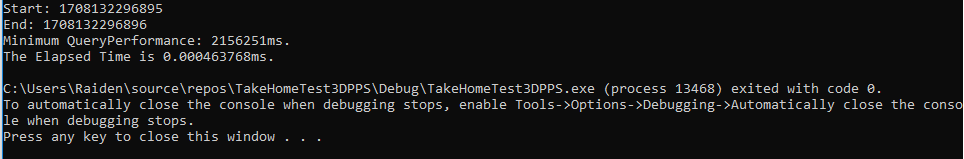
This is for size number = 256. The average came out to be 0.0004637 ms.

****

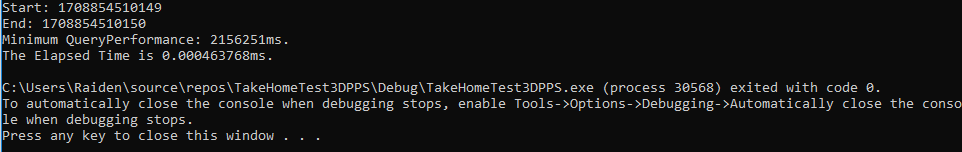
This is for size number = 512. The average came out to be 0.0004637 ms.

****

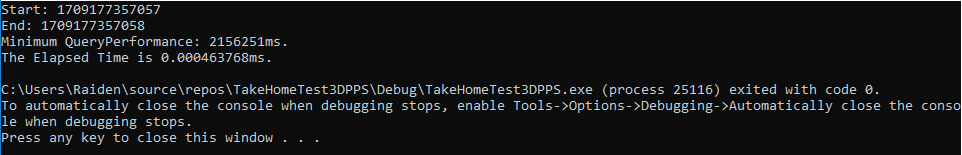
This is for size number = 1024. The average came out to be 0.0004637 ms.

****

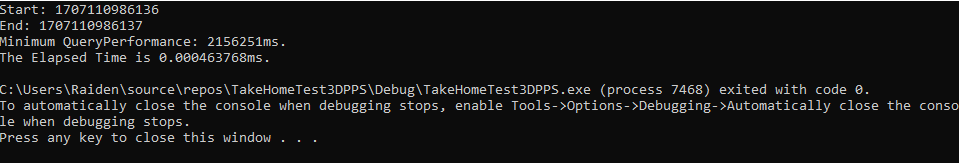
This is for size number = 2048. The average came out to be 0.0004637 ms.

****

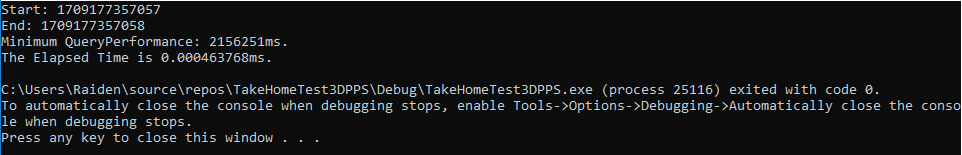
This is for size number = 4096. The average came out to be 0.0004637 ms.

****

This is for size number = 8192. The average came out to be 0.0004637 ms.

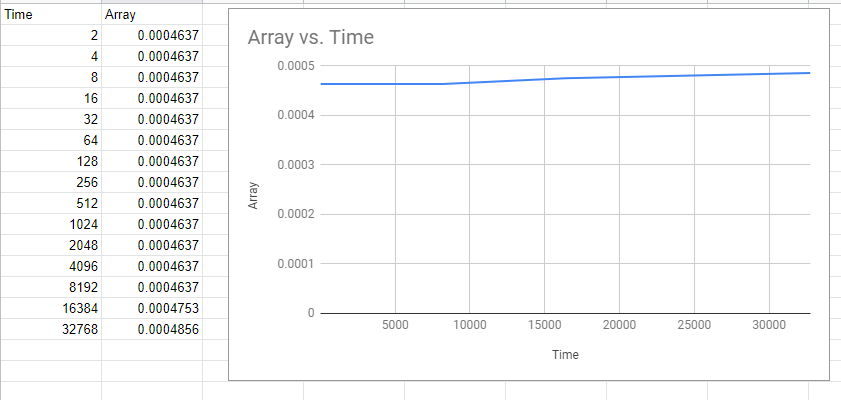
****

This is for size number = 16384. The average came out to be 0.0004753 ms.

****

This is for size number = 32768. The average came out to be 0.0004856 ms.

**Graph for DPPS**



**Comparison of all methods**

