**TAKE HOME TEST 3 OPTIMIZATION**

**CSC 343 FALL 2017**

**DECEMBER 4, 2017**

**Jeter Gutierrez**

**TABLE OF CONTENTS**

**1 OBJECTIVE pg. 2**

**2. DOT PRODUCT ON LINUX 64 BITS pg. 2-13**

**2.1 DOT PRODUCT USING INDEXES ON LINUX 64 BIT. pg. 3-5**

**2.2 DOT PRODUCT USING OPTIMIZED INDEX ON LINUX 64 BIT pg. 5-8**

**2.3 DOT PRODUCT USING POINTERS ON LINUX 64 BIT pg. 8-10**

**2.4 DOT PRODUCT USING OPTIMIZED POINTER ON LINUX 64 BIT pg. 10-13**

**3. DOT PRODUCT ON LINUX 32 BIT pg. 13-25**

**3.1 DOT PRODUCT USING INDEXES ON LINUX 32 BIT. pg. 13-16**

**3.2 DOT PRODUCT USING OPTIMIZED INDEX ON LINUX 32 BIT pg. 16-19**

**3.3 DOT PRODUCT USING POINTERS ON LINUX 32 BIT pg. 19-21**

**3.4 DOT PRODUCT USING OPTIMIZED POINTER ON LINUX 32 BIT pg. 21-25**

**4. DOT PRODUCT ON WINDOWS 32 BIT pg. 25-38**

**4.1 DOT PRODUCT USING INDEXES ON WINDOWS 32 BIT. pg. 25-29**

**4.2 DOT PRODUCT USING OPTIMIZED INDEX ON WINDOWS 32 BIT pg. 29-33**

**4.3 DOT PRODUCT USING POINTERS ON WINDOWS 32 BIT pg. 33-35**

**4.4 DOT PRODUCT USING OPTIMIZED POINTER ON WINDOWS 32 BIT pg. 35-38**

**5. GRAPHS pg. 38-42**

**5.1 INDEX FOR DIFFERENT VALUES NORMAL AND OPTIMIZED ON ALL 3 PLATFORMS pg. 38-40**

**5.2 POINTER FOR DIFFERENT VALUES NORMAL AND OPTIMZED ON ALL 3 PLATFORMS pg. 40-42**

**6. CONCLUSION pg. 42**

**1 OBJECTIVE**: The purpose of this assignment is to test how a Windows 32-bit compiler, Linux 64-bit compiler, and Linux 32-bit compiler handle performing dot product using pointers and indexing. We will be using Visual studio to run these examinations on Windows 32-bit compiler, we will be using GDB and GCC to test the both Linux versions. In both the case of running the examination using array indexing or pointers we will be performing Dot product. When we call the Dot product function the function will be written in assembly while the function call will be made using a CPP file. The procedure for every case will be to write the code for performing Dot product in C++ on each platform, after that we will then generate the assembly code for the function. Once we have generated the assembly code for the function we will then call the from the main file and measure its performance, we will do this for different lengths of arrays. After we have measured the performance for both indexing and pointer Dot product computations we will then optimize the code on our own and measure performance again, doing this for all 3 compiling platforms.

In this report we will be performing the following examinations:

1. DOT PRODUCT USING INDEX ON LINUX 64 BITS
2. DOT PRODUCT USING POINTERS ON LINUX 64 BIT
3. DOT PRODUCT USING INDEX ON LINUX 32 BIT
4. DOT PRODUCT USING POINTER ON LINUX 32 BIT
5. DOT PRODUCT USING INDEX ON WINDOWS 32 BIT
6. DOT PRODUCT USING POINTER ON WINDOWS 32 BIT

**2. DOT PRODUCT ON LINUX 64 BITS**

In this section we will be performing the dot product using index arithmetic for the arrays. The process is as follows, we will write the code that will perform the dot product using indexes in C++, then we will generate the compiler assembly code for the instruction, after we have done that we will optimize the compiles assembly code. With the assembly code not optimized and with it optimized we as well will be timing it and comparing the values that we get. With the values that we get for optimized times and not optimized times we will see the difference. We will also be testing this for different sized arrays.

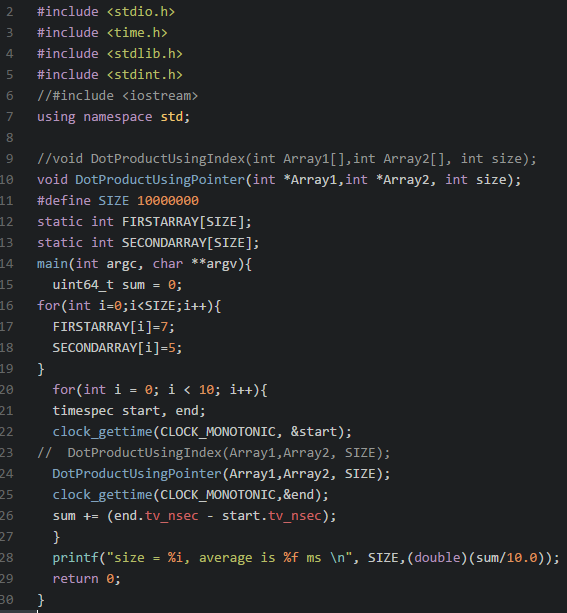


Figure 1: C++ code that will be used to measure the time that it takes to calculate the dot product using index normally, pointers normally, index optimized, and pointers optimized.

*2.1 DOT PRODUCT USING INDEXES ON LINUX 64 BIT.*

In this section I will be computing the dot product using GCC generated assembly code on the Linux 64-bit compiler on Ubuntu. To generate the code what I need to do is use the flag -S with the GCC compiler. After it is loaded. I will link the assembly code to the main file that will be used to time how long it takes to perform the dot product using indexes on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to its optimized code.

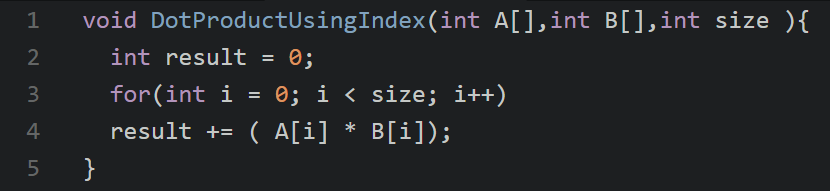


Figure 2: C++ code that will be used to calculate the Dot Product on the Linux 64-bit compiler using indexes.

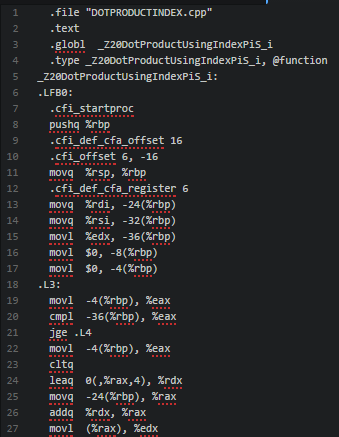


Figure 3: Compiler generated assembly code for running dot product using indexes on Linux 64-bit compiler. We can see that the file is being compiled into assembly code from the file Dotproduct.cpp which has the instructions written in C++ that were generated into assembly.

These are the instructions that use the registers to look at the values that will be added to multiply and add into the compiler. Some of the instructions are redundant because they are repetitive and some of them are unnecessary because they make the process take even longer then they should take if the code were written more efficiently.

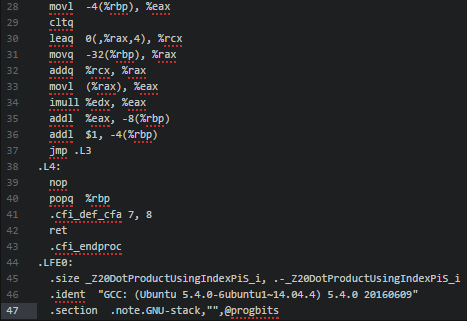


Figure 4: Compiler generated assembly code that is also used for calculating dot product using indexes on Linux 64-bit compiler. The final instructions of deallocating the memory from the stack pointer and from the stack completely.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LINUX 64 BIT TIME VALUES FOR INDEX | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 0.244 | 1.005 | 8.369 | 82.227 | 825.2497 | 6254.3498 | 61412.5957 |
| TRIAL | 0.256 | 0.989 | 8.326 | 81.788 | 819.9741 | 6268.9691 | 61373.9987 |
| TRIAL | 0.251 | 0.986 | 8.357 | 81.891 | 827.4812 | 6251.973 | 61272.2451 |
| TRIAL | 0.247 | 0.986 | 8.357 | 81.772 | 711.6758 | 6221.9977 | 61109.8625 |
| TRIAL | 0.243 | 0.993 | 8.363 | 81.714 | 814.9616 | 6226.9499 | 61408.5084 |
| TRIAL | 0.247 | 1.024 | 8.328 | 81.695 | 801.4223 | 6253.0208 | 61264.6651 |
| TRIAL | 0.244 | 0.985 | 8.294 | 81.71 | 724.8512 | 6252.2141 | 61208.5171 |
| TRIAL | 0.247 | 0.995 | 8.348 | 81.816 | 812.2048 | 6267.7566 | 61215.4541 |
| TRIAL | 0.249 | 0.984 | 8.323 | 81.742 | 834.4072 | 6215.4807 | 61470.7489 |
| TRIAL | 0.252 | 1.012 | 8.326 | 81.891 | 749.5387 | 6264.4722 | 61201.0715 |

Figure 5: Table that was recorded of the data when running the dot product using the assembly code generated by Linux 64-bit compiler for the different values of N for the different sizes of the arrays. This information will later be graphed and compared to other sections of this project.

*2.2 DOT PRODUCT USING OPTIMIZED INDEX ON LINUX 64 BIT*

In this section I will be computing the dot product using GCC generated assembly code that I have optimized on the Linux 32-bit compiler on a Raspberry Pi To generate the code what I need to do is use the flag -S with the GCC compiler. After it is loaded. I will optimize it and link the assembly code to the main file that will be used to time how long it takes to perform the dot product using indexes optimized on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to its normal assembly code.

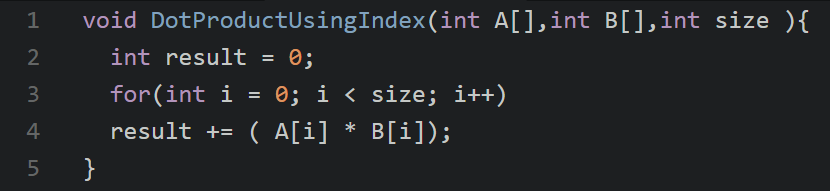


Figure 6: C++ code that will be used to generate the assembly code using the Linux 64-bit compiler on Ubuntu and will then be optimized by me. After I optimize the code I will then link it to the timing file that will be used, the main file that will be used to measure the time that it takes to run the Dot Product using my optimized assembly code. I will be doing this for different values of sized arrays.

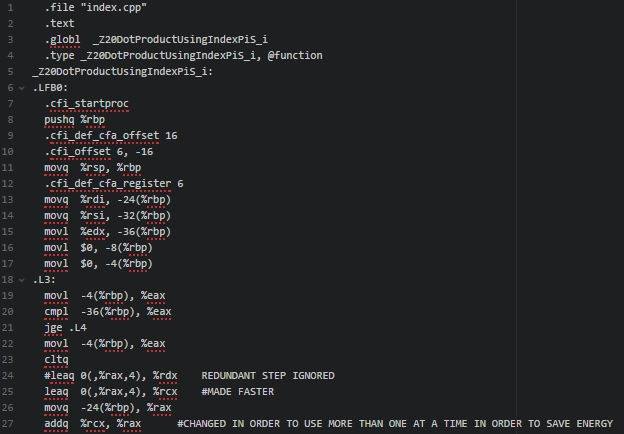


Figure 7: Assembly code for my optimized code for calculating the dot product using Indexes. In this file I have removed line 24 which is redundant because it is looking at a register to store the values more than once, that is not needed. It can continue to look at the same register and update the value that is stored there over time instead of having to continue to switch between the registers to find the total sum. Line 25 was also optimized to make the process go even faster using the same implementation. Line 27 was modified to make the summation process take less energy from the processor.

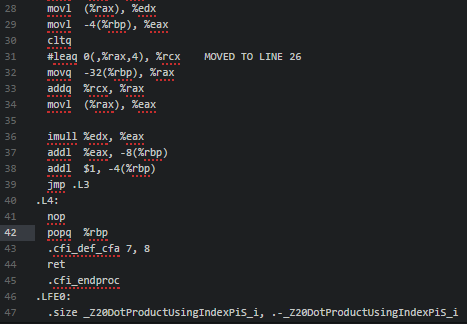


Figure 8: My optimized assembly code for calculating the Dot product using the indexes instead of pointers for the Linux 64-bit compiler, I have moved line 31 to line 26, earlier in the program because I believe that if this condition is checked earlier the program will run quicker. Because it is checking a condition, the condition should be checked beforehand.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LINUX 64 BIT TIME VALUES FOR INDEX OPTIMIZED | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 0.2396 | 0.9095 | 7.8694 | 77.1904 | 783.6297 | 5919.4531 | 57921.714 |
| TRIAL | 0.2509 | 0.8867 | 7.8933 | 77.2364 | 650.3334 | 5931.8516 | 57939.2392 |
| TRIAL | 0.2397 | 0.7516 | 7.9282 | 77.1041 | 682.2527 | 5920.5353 | 58186.8999 |
| TRIAL | 0.2366 | 0.9242 | 7.9068 | 77.1778 | 786.1063 | 5908.4519 | 57830.5958 |
| TRIAL | 0.2395 | 0.801 | 6.937 | 63.6697 | 706.0471 | 5989.7084 | 57984.7236 |
| TRIAL | 0.2476 | 0.9068 | 8.2007 | 77.1659 | 644.2415 | 5880.0702 | 58012.8488 |
| TRIAL | 0.2398 | 0.9234 | 7.9487 | 65.3068 | 688.7589 | 5921.5033 | 58208.8686 |
| TRIAL | 0.2484 | 0.9229 | 7.8318 | 74.459 | 766.4919 | 5879.3991 | 57834.553 |
| TRIAL | 0.2439 | 0.9402 | 7.8435 | 77.6407 | 776.0728 | 5917.1016 | 57912.7809 |
| TRIAL | 0.2346 | 0.8448 | 8.1168 | 77.198 | 734.11 | 5908.7739 | 57965.0948 |

Figure 9: Table that was recorded of the data when running the dot product using my optimized assembly code for Linux 64-bit compiler for the different values of N for the different sizes of the arrays. This information will later be graphed and compared to other sections of this project.

*2.3 DOT PRODUCT USING POINTERS ON LINUX 64 BIT*

In this section I will be computing the dot product using GCC generated assembly code on the Linux 64-bit compiler on Ubuntu using pointers. To­­ generate the code what I need to do is use the flag -S with the GCC compiler. After it is loaded. I will link the assembly code to the main file that will be used to time how long it takes to perform the dot product using pointers on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to the assembly pointer optimized code.

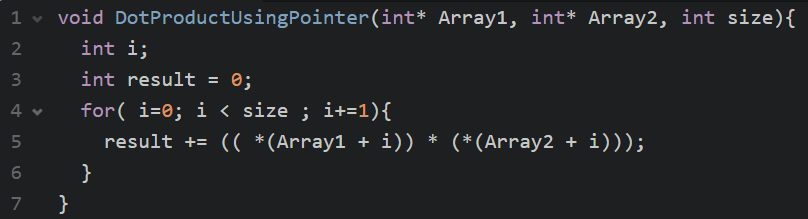


Figure 10: C++ code that will be used to generate the assembly code using the Linux 64-bit compiler on Ubuntu using pointers. I will then link it to the timing file that will be used, the main file that will be used to measure the time that it takes to run the Dot Product using the compiler generated assembly code using pointers. I will be doing this for different values of sized arrays.

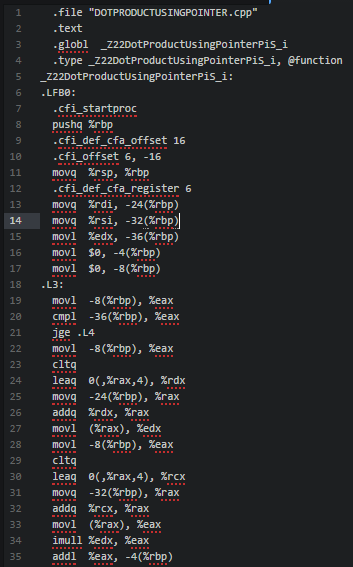


Figure 11: Compiler generated assembly code for running dot product using pointers on Linux 64-bit compiler. We can see that the file is being compiled into assembly code from the file Dotproductusingpointer.cpp which has the instructions written in C++ that were generated into assembly.

These are the instructions that use the registers to look at the values that will be added to multiply and add into the compiler. Some of the instructions are redundant because they are repetitive and some of them are unnecessary because they make the process take even longer then they should take if the code were written more efficiently.

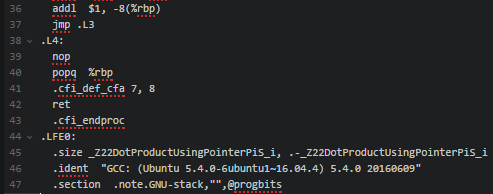


Figure 12: Compiler generated assembly code that is also used for calculating dot product using pointers on Linux 64-bit compiler. The final instructions of deallocating the memory from the stack pointer and from the stack completely and dereferencing.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LINUX 64 BIT TIME VALUES FOR POINTER | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 0.1838 | 0.977 | 7.2893 | 71.5545 | 714.1938 | 6237.9745 | 61342.2397 |
| TRIAL | 0.2646 | 0.9423 | 7.3291 | 86.8798 | 826.8648 | 6241.4384 | 61164.0782 |
| TRIAL | 0.1901 | 0.9842 | 7.3364 | 81.6914 | 817.4812 | 6226.7609 | 61150.8543 |
| TRIAL | 0.2106 | 0.875 | 8.3267 | 71.5269 | 773.8148 | 6268.5479 | 61272.3362 |
| TRIAL | 0.2562 | 0.9901 | 8.3464 | 81.7703 | 773.6742 | 6247.637 | 61267.8586 |
| TRIAL | 0.1888 | 0.7296 | 8.3769 | 60.2667 | 681.0909 | 6262.3244 | 61258.7707 |
| TRIAL | 0.2575 | 0.9973 | 11.6487 | 81.6332 | 705.022 | 6216.3497 | 61284.4935 |
| TRIAL | 0.1976 | 1.0136 | 6.1492 | 71.5555 | 719.1089 | 6247.8569 | 61230.9624 |
| TRIAL | 0.2268 | 0.885 | 8.3284 | 82.3511 | 779.0199 | 6223.1534 | 61212.1054 |
| TRIAL | 0.243 | 0.8691 | 7.3178 | 60.2061 | 751.0914 | 6337.74 | 61279.9792 |

Figure 13:Table that was recorded of the data when running the dot product using the assembly code generated by Linux 64-bit compiler for the different values of N for the different sizes of the arrays for using pointers. This information will later be graphed and compared to other sections of this project.

# *2.4 DOT PRODUCT USING OPTIMIZED POINTER ON LINUX 64 BIT*

In this section I will be computing the dot product using GCC generated assembly code on the Linux 64-bit compiler on Ubuntu that I have optimized. To generate the code what I need to do is use the flag -S with the GCC compiler. After it is loaded. I will link the assembly code to the main file that will be used to time how long it takes to perform the dot product using pointers on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to the compiler generated assembly code for performing the dot product using pointers.

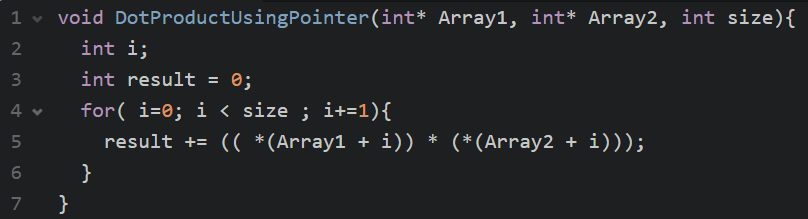


Figure 14: C++ code that will be used to generate the assembly code using the Linux 64-bit compiler on Ubuntu and will then be optimized by me for calculating the dot product of arrays using. After I optimize the code I will then link it to the timing file that will be used, the main file that will be used to measure the time that it takes to run the Dot Product using my optimized assembly code. I will be doing this for different values of sized arrays.

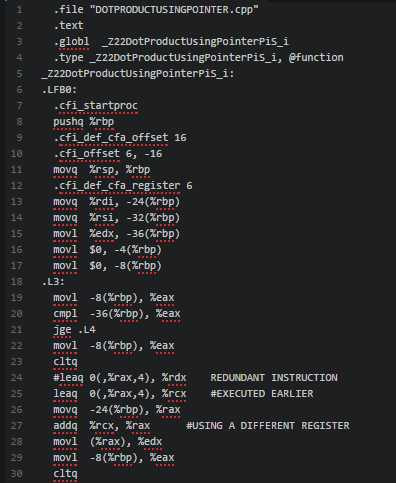


Figure 15: Assembly code for my optimized code for calculating the dot product using pointers. In this file I have removed line 24 which is redundant because it is looking at a register to store the values more than once, that is not needed.

It can continue to look at the same register and update the value that is stored there over time instead of having to continue to switch between the registers to find the total sum. Line 25 was also optimized to make the process go even faster using the same implementation by making it be performed earlier in the code. Line 27 was modified to make the summation process take less energy from the processor by using a different register.

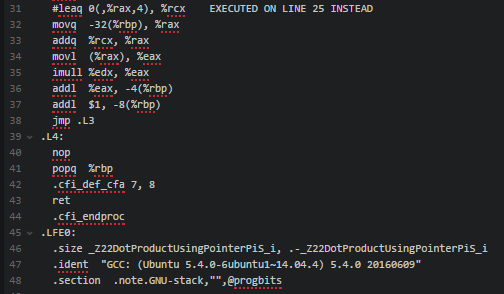


Figure 16: My optimized assembly code for calculating the Dot product using the pointers instead of pointers for the Linux 64-bit compiler, I have moved line 31 to line 27, earlier in the program because I believe that if this condition is checked earlier the program will run quicker. Because it is checking a condition, the condition should be checked beforehand which is going to make the running time significantly decrease.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LINUX 64 BIT TIME VALUES FOR POINTER OPTIMIZED | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 0.2417 | 0.9369 | 7.9269 | 76.9864 | 748.7691 | 5906.7886 | 57699.6842 |
| TRIAL | 0.2501 | 0.8999 | 7.9221 | 77.0992 | 754.3107 | 5909.4796 | 57903.585 |
| TRIAL | 0.2392 | 0.9347 | 7.9153 | 77.1465 | 782.1948 | 5896.2628 | 57842.3739 |
| TRIAL | 0.2458 | 0.9158 | 7.92 | 77.6904 | 742.119 | 5914.5256 | 57839.8326 |
| TRIAL | 0.2402 | 0.9262 | 7.8654 | 77.1256 | 748.7056 | 5895.6659 | 57802.6829 |
| TRIAL | 0.2593 | 0.9065 | 7.8748 | 77.1631 | 787.069 | 5894.6824 | 57882.6436 |
| TRIAL | 0.2472 | 0.9076 | 7.9426 | 77.4981 | 743.5704 | 5906.9981 | 57885.6313 |
| TRIAL | 0.2419 | 0.8961 | 7.867 | 76.9301 | 736.5196 | 5928.273 | 57898.9538 |
| TRIAL | 0.2364 | 0.9358 | 7.8643 | 77.1463 | 744.3211 | 5924.4438 | 57703.2393 |
| TRIAL | 0.2446 | 0.9193 | 7.8952 | 77.8602 | 717.1894 | 5908.8214 | 57940.9472 |

Figure 17: Results for the times of calculating the dot product for different values of N using my optimized code and pointers on Linux 64-bit compiler.

**3. DOT PRODUCT ON LINUX 32 BIT**

In this section I will be calculating dot product using different methods. I will be doing it on Linux 32-bit compiler and I will be using a Raspberry Pi. I will calculate it using compiler generated assembly code, that is generated for calculating using indexes and pointer and I will optimize both to measure more time and efficiency.

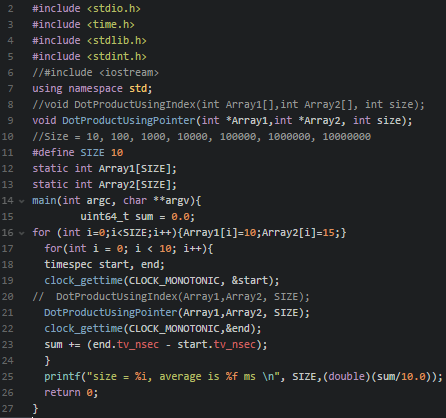


Figure 18: The main file that will be used in each of the following 4 sections in order to measure the time of running each of the dot product methods on a Raspberry pi for Linux 32-bit.

**3.1 DOT PRODUCT USING INDEXES ON LINUX 32 BIT.**

In this section I will be computing the dot product using GCC generated assembly code on the Linux 32-bit compiler on a Raspberry Pi. To generate the code what I need to do is use the flag -S with the GCC compiler as well. After it is loaded, I will link the assembly code to the main file that will be used to time how long it takes to perform the dot product using indexes on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to its optimized code.

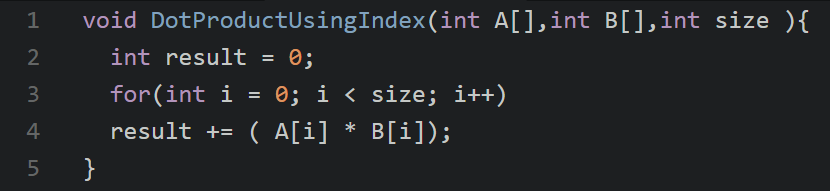


Figure 19: C++ code that will be used in for generating the dot product assembly code that will then be linked to the main file on a Raspberry Pi.

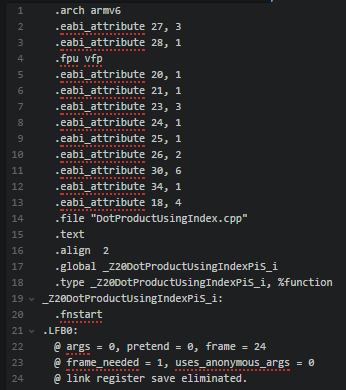


Figure 20: Assembly code that is generated by compiler on Raspberry pi.

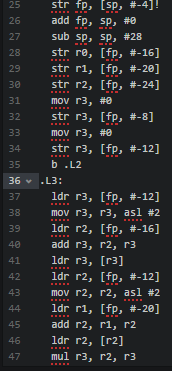


Figure 21: Assembly code for calculating the dot product using indexes on the raspberry pi.

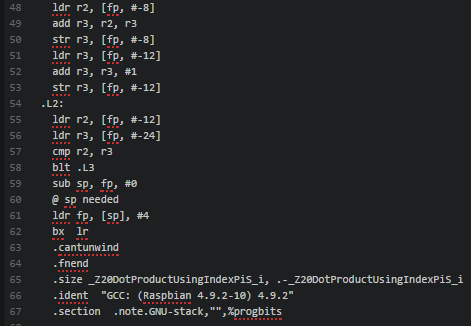


Figure 22: Continued assembly code for calculating the dot product using indexes on the Raspberry pi these are the final steps where the program terminates and deallocates the memory.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LINUX 32 BIT TIME VALUES FOR INDEX | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 1.5729 | 6.3074 | 52.7863 | 545.5257 | 3711.325 | 30968.2505 | 14757395.3 |
| TRIAL | 0.8697 | 6.2187 | 27.3228 | 532.6872 | 5585.355 | 30895.0895 | 14757395.3 |
| TRIAL | 1.5782 | 6.2395 | 52.7865 | 539.0047 | 3506.81 | 32547.6609 | 14757395.3 |
| TRIAL | 0.802 | 6.224 | 53.0573 | 538.1557 | 5488.095 | 29809.533 | 14757395.3 |
| TRIAL | 1.552 | 6.1459 | 26.5885 | 528.5774 | 2855.514 | 30802.1625 | 14757395.3 |
| TRIAL | 1.5574 | 6.2345 | 53.1718 | 262.4059 | 5620.345 | 30306.5379 | 14757395.3 |
| TRIAL | 1.6407 | 6.1927 | 53.1719 | 535.9736 | 2849.769 | 29885.809 | 14757395.3 |
| TRIAL | 1.5782 | 6.2032 | 53.1407 | 261.0623 | 5609.037 | 31123.2299 | 14757395.3 |
| TRIAL | 1.5468 | 3.1198 | 52.828 | 544.4006 | 2845.805 | 28103.9249 | 14757395.3 |
| TRIAL | 1.5885 | 6.3436 | 52.7916 | 262.5206 | 7241.947 | 29037.1169 | 14757395.3 |

Figure 23: Time measurements for calculating the dot product when the compiler generated assembly code is linked to the main file that calculates time on the raspberry pi using different values of N for the sizes of the arrays.

**3.2 DOT PRODUCT USING OPTIMIZED INDEX ON LINUX 32 BIT**

In this section I will be computing the dot product using GCC generated assembly code that I have optimized on the Linux 32-bit compiler on a Raspberry Pi To generate the code what I need to do is use the flag -S with the GCC compiler. After it is loaded. I will optimize it and link the assembly code to the main file that will be used to time how long it takes to perform the dot product using indexes optimized on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to its normal assembly code.

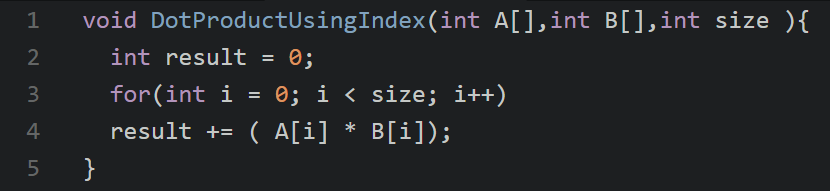


Figure 24: C++ code for calculating the Dot product using indexes on the raspberry pi. This code will be used to generate assembly code with the compiler, after it is generated I will optimize it and then calculate the dot product using indexes for various sizes of arrays.

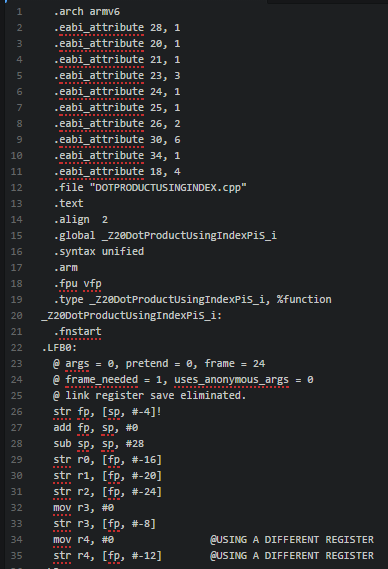


Figure 25: Optimized assembly code by me for calculating the Dot product using indexes on a raspberry pi. What I did on line 34 was use a different register so that I can speed up the process. Instead of using the register that was being used before to copy a temporary value over, I am copying the recursively like using an accumulator so that it computer faster. I then use the same register on line 35 to speed up even more the assembly code for calculating the dot product using indexes on a raspberry pi.

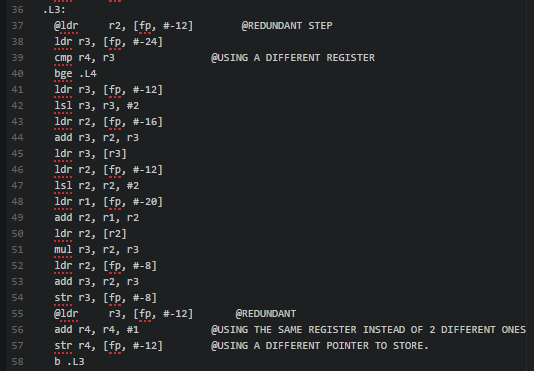


Figure 26: assembly code for calculating the dot product using indexes on the raspberry pi, I have optimized it by removing 2 redundant steps on line 37 and 55, I also used the same register to add on line 56 and a different pointer n line 57. I also used a different register on line 39.

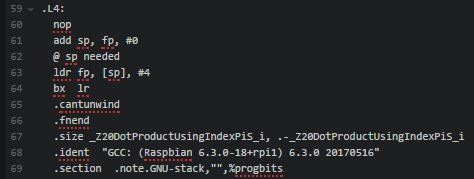


Figure 27: The rest of the assembly code for calculating the dot product using indexes on the raspberry pi that I have optimized, I used the stack pointer instead on line 62 instead of what it was previously using because I was certain that using the stack pointer instead would have an impact on the calculation time.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LINUX 32 BIT TIME VALUES FOR INDEX OPTIMIZED | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 1.4844 | 14.047 | 46.5884 | 455.6138 | 4782.2605 | 27412.4208 | 48552.1957 |
| TRIAL | 0.7864 | 5.5365 | 23.073 | 234.442 | 3569.0911 | 26126.4709 | 46483.8735 |
| TRIAL | 1.4843 | 2.7914 | 46.1147 | 455.8635 | 2454.7499 | 29549.6349 | 49432.5579 |
| TRIAL | 1.4947 | 5.5155 | 25.3335 | 234.3222 | 5084.4007 | 26893.8493 | 48684.3898 |
| TRIAL | 1.5937 | 2.8072 | 46.0834 | 455.7335 | 2937.1501 | 25171.0252 | 48594.6608 |
| TRIAL | 1.4582 | 5.4947 | 46.1561 | 234.333 | 4873.9586 | 29456.3335 | 53501.1047 |
| TRIAL | 1.4844 | 2.7814 | 23.2084 | 227.7706 | 3917.377 | 27633.7959 | 53589.6603 |
| TRIAL | 1.5885 | 5.4948 | 46.3543 | 455.4627 | 5760.3211 | 27513.3849 | 48210.6327 |
| TRIAL | 1.5001 | 5.5104 | 23.0833 | 227.7912 | 3785.8201 | 25914.5242 | 74211.7123 |
| TRIAL | 0.7552 | 5.5157 | 46.5777 | 234.3172 | 4962.1299 | 25675.1653 | 48831.0885 |

Figure 28: Data table for the measured time for calculating the dot product using indexes with my optimized linked assembly code on the Raspberry pi using 32-bit Linux and GCC.

**3.3 DOT PRODUCT USING POINTERS ON LINUX 32 BIT**

In this section I will be computing the dot product using GCC generated assembly code on the Linux 32-bit compiler on a Raspberry Pi using pointers. To generate the code what I need to do is use the flag -S with the GCC compiler. After it is loaded. I will link the assembly code to the main file that will be used to time how long it takes to perform the dot product using pointers on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to the assembly pointer optimized code.

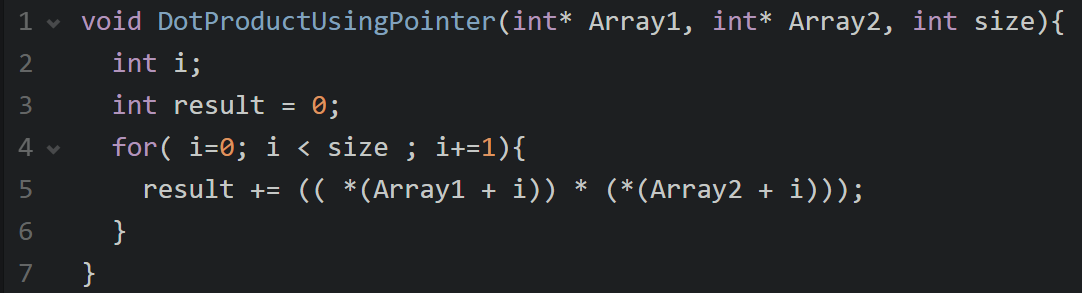


Figure 29: C++ code that will be used to generate the assembly code for calculating the dot product using pointers on the raspberry pi. After I generate the assembly code I will link the code to the source file that I am using as a main file to calculate the time for each run. Then I will take measurements of time to see how quickly or how long it takes to calculate the dot product for each different size, in the next section I will optimize the code.

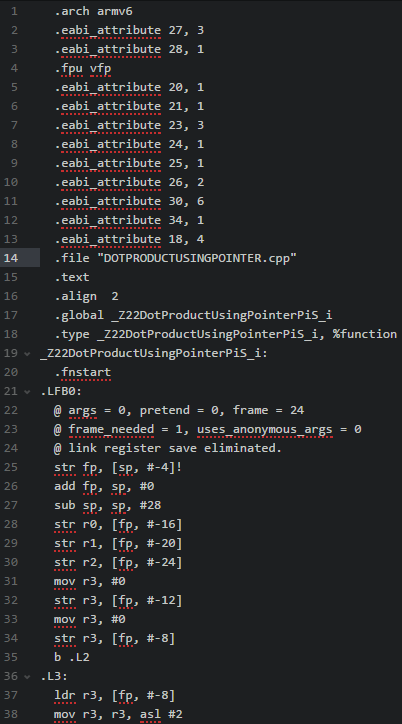


Figure 30: Assembly code for running dot product using pointers on the Linux 32bit compiler. On a raspberry pi that I will later link to the main file to take time measurements.

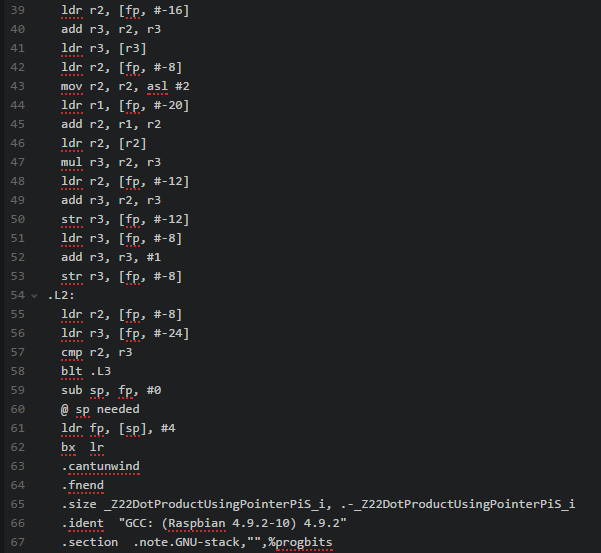


Figure 31: Continuation of the assembly code for calculating the dot product using pointers on the Raspberry pi.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LINUX 32 BIT TIME VALUES FOR POINTER | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 1.4947 | 6.2032 | 53.2031 | 538.1715 | 5584.2298 | 31913.1197 | 12912720.85 |
| TRIAL | 1.5207 | 6.2187 | 53.203 | 532.4422 | 3639.3249 | 30134.6108 | 12912720.85 |
| TRIAL | 1.5314 | 3.1406 | 52.8385 | 532.1662 | 2809.1905 | 28614.5755 | 12912720.85 |
| TRIAL | 0.7916 | 6.1824 | 26.5313 | 268.6142 | 6507.9321 | 30937.4278 | 12912720.85 |
| TRIAL | 1.5 | 4.2448 | 52.7604 | 531.8125 | 4638.2565 | 30770.2404 | 12912720.85 |
| TRIAL | 0.7761 | 6.2084 | 26.3855 | 262.38 | 5474.0995 | 29561.5544 | 12912720.85 |
| TRIAL | 0.7864 | 3.0834 | 52.8072 | 531.4215 | 2803.2841 | 31763.5157 | 12912720.85 |
| TRIAL | 1.5626 | 3.099 | 26.3855 | 262.38 | 5535.8132 | 31015.8394 | 12912720.85 |
| TRIAL | 1.7656 | 6.2031 | 26.3802 | 530.1768 | 4402.3608 | 32091.953 | 12912720.85 |
| TRIAL | 1.5468 | 6.25 | 52.7706 | 530.6664 | 9222.9558 | 30712.4802 | 12912720.85 |

Figure 32: Data segment for the measurements of calculating the dot product using pointers on the Raspberry pi. Some of these measurements will be graphed and compared to the other platforms that are being tested in this project and to the optimized measurements.

**3.4 DOT PRODUCT USING OPTIMIZED POINTER ON LINUX 32 BIT**

In this section I will be computing the dot product using GCC generated assembly code on the Linux 32-bit compiler on Raspberry Pi that I have optimized. To generate the code what I need to do is use the flag -S with the GCC compiler. After it is loaded. I will link the assembly code to the main file that will be used to time how long it takes to perform the dot product using pointers on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to the compiler generated assembly code for performing the dot product using pointers.

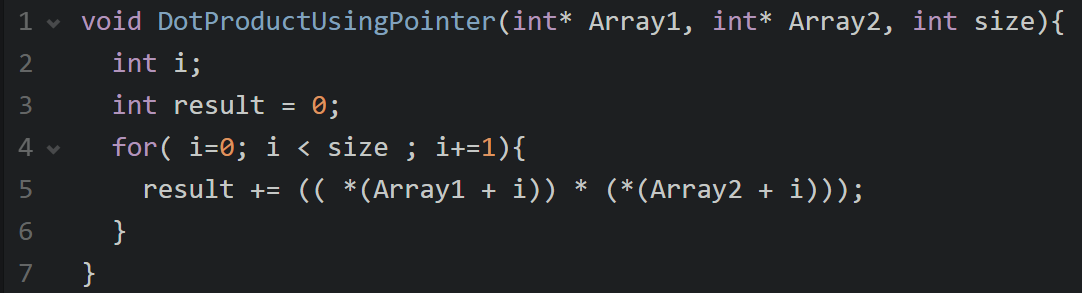


Figure 33: C++ code that will be used to generate the assembly code for calculating the dot product of n size arrays, that I will then optimize and link to the main file that will be used to measure the time that it takes. The original assembly code that will be optimized by me is initially generated by the compiler that is 32 bits on Linux on the raspberry pi, but I will then optimize it.

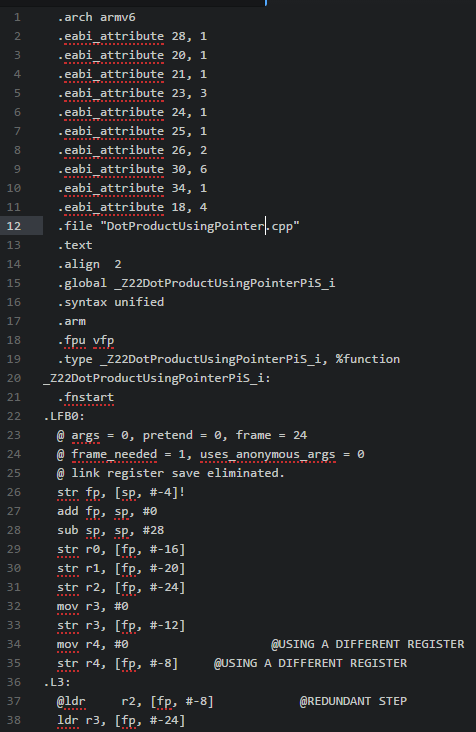


Figure 34: Assembly code for calculating the dot product using pointers that I have optimized. In line 34 I have used a different register like in the previous example. In the line 35 I have also used a different register and on line 37 I have removed the unnecessary step of using register r2 again, it is not needed because I am using a different register in the previous instruction.

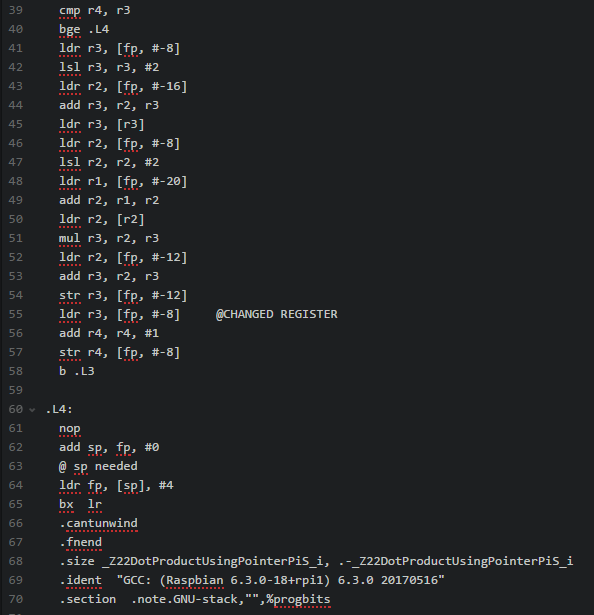


Figure 35: Continued assembly code that I optimized for calculating the dot product using pointers on the raspberry pi. In line 55 I have changed the register that is being used so that instead of storing the value into a temporary register it uses the same register and saves time.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LINUX 32 BIT TIME VALUES FOR POINTER OPTIMIZED | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 1.4739 | 5.6823 | 47.8072 | 486.8433 | 5907.8336 | 28070.5658 | 1660206.967 |
| TRIAL | 1.5314 | 5.7032 | 23.9062 | 478.0361 | 2593.5761 | 25804.2758 | 1660206.967 |
| TRIAL | 0.7501 | 2.8594 | 47.8126 | 236.1719 | 5047.7824 | 28059.0967 | 1660206.967 |
| TRIAL | 1.5105 | 5.6875 | 24.1093 | 477.4583 | 2554.9668 | 29735.7571 | 1660206.967 |
| TRIAL | 0.7344 | 5.8074 | 47.7604 | 236.25 | 7137.5308 | 30443.996 | 1660206.967 |
| TRIAL | 1.5001 | 5.7032 | 23.9688 | 494.7183 | 2589.9927 | 26019.6767 | 1660206.967 |
| TRIAL | 0.7187 | 2.8699 | 47.7552 | 237.3956 | 5042.9435 | 25423.2865 | 1660206.967 |
| TRIAL | 1.5417 | 5.6613 | 47.8645 | 490.859 | 2572.8832 | 27562.9879 | 1660206.967 |
| TRIAL | 1.5 | 2.9895 | 47.8855 | 237.0831 | 5061.121 | 29667.2624 | 1660206.967 |
| TRIAL | 1.4373 | 5.6719 | 23.8801 | 481.5622 | 2572.4666 | 28223.2373 | 1660206.967 |

Figure 36: Data table for the measurements of calculating the dot product on a Linux 32-bit compiler using my optimized assembly code for different values of N or the different values for the sizes of the arrays.

**4. DOT PRODUCT ON WINDOWS 32 BIT**

In this section I will be calculating the dot product and measure the time that it takes to run the dot product on windows 32-bit compiler. I will test it using indexes and pointers, linking using assembly code and I will use the code that is generated but I will also use the assembly code that I will optimize. In the final section I will graph and compare the data to show that my optimizations were effective.

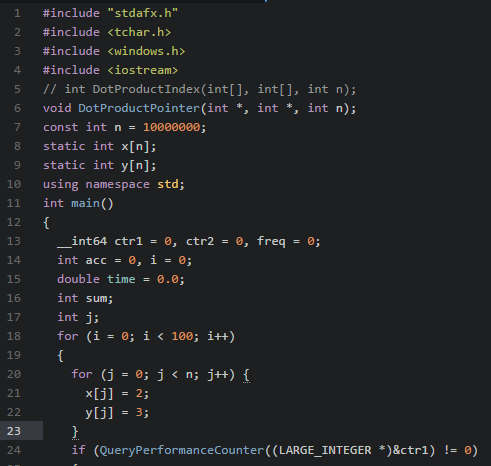


Figure 37: C++ code for measuring the time that it will take to calculate the dot product for each of the 4 incoming sections.

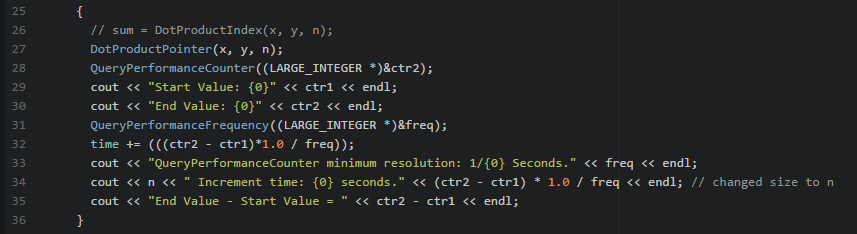


Figure 38: Continued C++ code for measuring time on the windows 32-bit compiler.

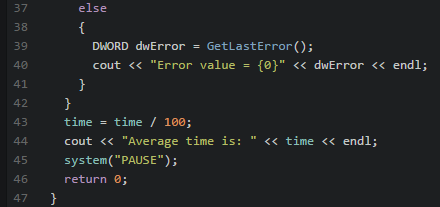


Figure 39: Continued C++ code for measuring the time on windows 32-bit compiler for dot products.

**4.1 DOT PRODUCT USING INDEXES ON WINDOWS 32 BIT.**

In this section I will be computing the dot product using Visual Studio and Windows 32-bit compiler generated assembly code using Indexes. To generate the code what I need to do edit the properties of the Compiler to generate assembly code. This will happen by selecting properties and then selecting output assembler FA to see the assembly instructions that will be stored into the debug folder. After it is loaded, I will link the assembly code to the main file by using the Windows Macro Assembler setting. Then it will be used to time how long it takes to perform the dot product using indexes on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to its optimized code.

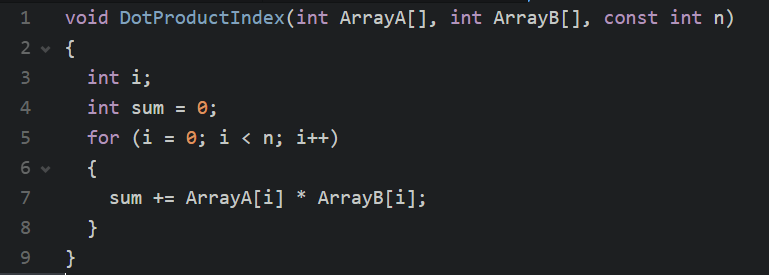


Figure 40: C++ code that will be used to generate the assembly code for running the dot product using indexes on Windows 32-bit compiler using visual studio.

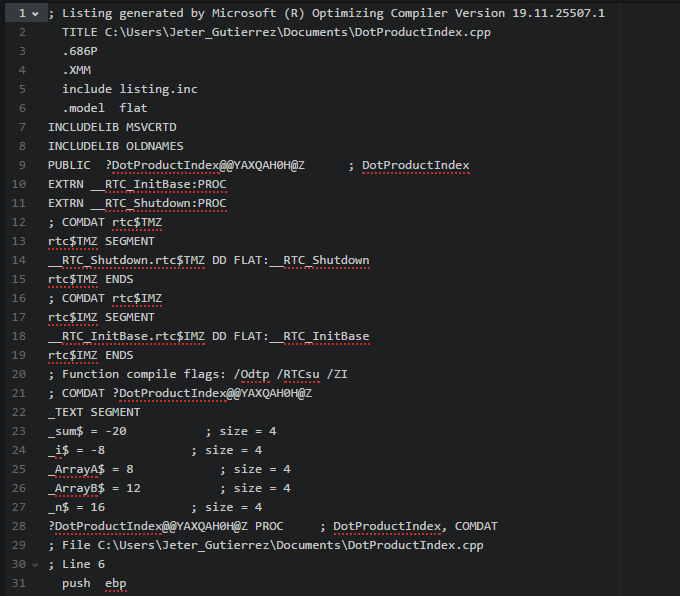


Figure 41: Compiler generated assembly code for running the dot product using indexes on the Windows 32-bit compiler using visual studio.

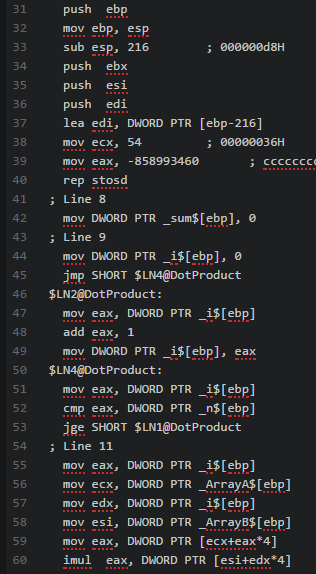


Figure 42: Continued assembly code generated by the visual studio windows 32-bit compiler for calculating the dot product using indexes on the.

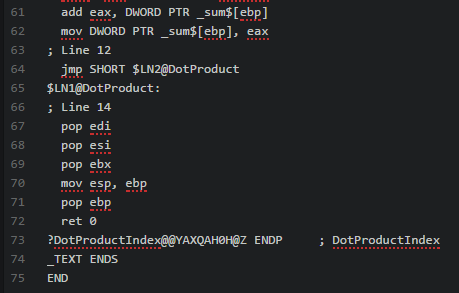


Figure 43: Continued assembly code generated by the visual studio windows 32-bit compiler for calculating the dot product using indexes on the that is now finalized and will be linked to the main file to measure the time of running the dot product using indexes on windows 32-bit compile for various values of N.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| WINDOWS 32 BIT TIME VALUES FOR INDEX | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 1 | 1 | 8 | 79 | 1051 | 6512 | 67008 |
| TRIAL | 1 | 1 | 8 | 80 | 1033 | 6655 | 67509 |
| TRIAL | 1 | 1 | 8 | 80 | 1188 | 6538 | 71105 |
| TRIAL | 1 | 1 | 8 | 79 | 1034 | 9152 | 71239 |
| TRIAL | 1 | 1 | 8 | 77 | 1032 | 6882 | 66201 |
| TRIAL | 1 | 1 | 8 | 79 | 1131 | 6610 | 67026 |
| TRIAL | 1 | 1 | 8 | 81 | 1076 | 6680 | 69400 |
| TRIAL | 1 | 1 | 8 | 80 | 1052 | 6940 | 65863 |
| TRIAL | 1 | 1 | 8 | 79 | 1124 | 6521 | 66176 |
| TRIAL | 1 | 1 | 8 | 81 | 1033 | 6680 | 67351 |

Figure 44: Measured time data for calculating the dot product for different values of N on the Windows 32-bit compiler using visual studio. This data will later be graphed and compared to the optimized code and to the other platforms.

**4.2 DOT PRODUCT USING OPTIMIZED INDEX ON WINDOWS 32 BIT**

In this section I will be computing the dot product using Visual Studio and Windows 32-bit compiler generated assembly code using Indexes. To generate the code what I need to do edit the properties of the Compiler to generate assembly code which I will then optimize. This will happen by selecting properties and then selecting output assembler FA to see the assembly instructions that will be stored into the debug folder. After it is loaded, I will link the assembly code to the main file by using the Windows Macro Assembler setting. Then it will be used to time how long it takes to perform the dot product using indexes on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to its compiler generated assembly code.

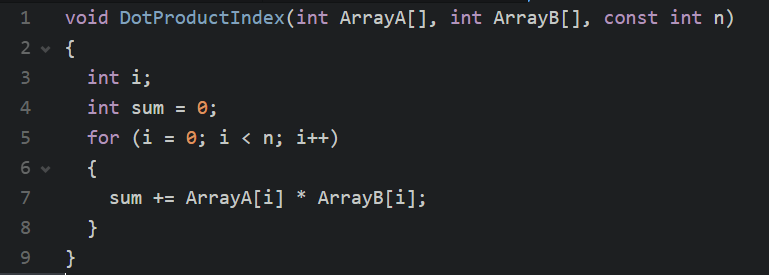


Figure 45: C++ code that will be used to generate the assembly code that will be optimized by me, linked to the main file and used to measure the time it takes to calculate the dot product using indexes with my optimized assembly code.

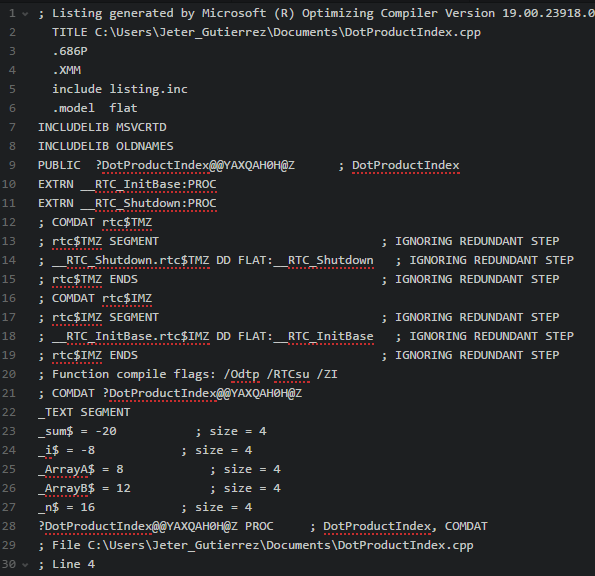


Figure 46: Assembly code that is optimized by me, lines 13-19 are redundant I don’t need them they do not affect the process of us running the code, I removed the steps that were repetitive and redundant on the visual studio assembly code.

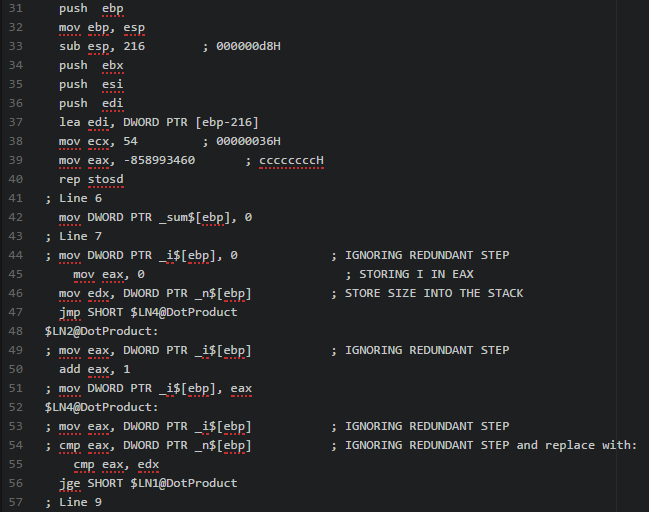


Figure 47: The continued assembly code that I have optimized for calculating the dot product using indexes. I have removed the steps that are redundant, there were several repetitive steps that were happening, I ignored them. There were other values that were stored into different registers than the ones that the compiler had determined them to be because I determined it would be quicker to store them into eax instead of where they were being stored before.

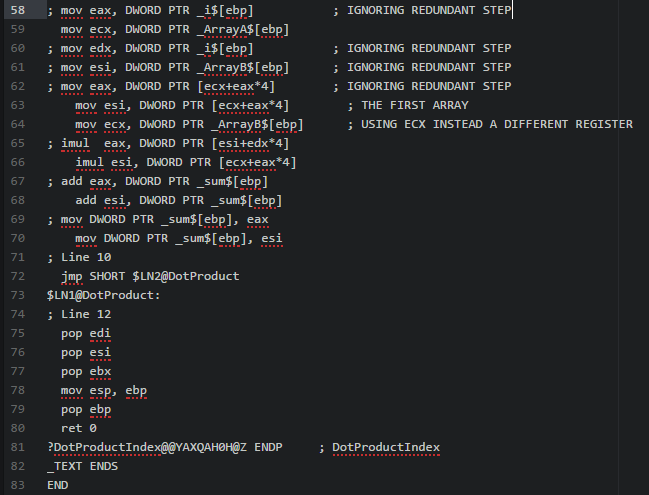


Figure 48: Assembly code for calculating the dot product using indexes that I have optimized. I have removed more redundant steps and used a different register in line 64. This will make the program run faster.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| WINDOWS 32 BIT TIME VALUES FOR INDEX OPTIMIZED | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 1 | 1 | 9 | 82 | 800 | 9174 | 50000 |
| TRIAL | 1 | 1 | 8 | 88 | 850 | 5000 | 51324 |
| TRIAL | 1 | 1 | 10 | 90 | 900 | 5345 | 54312 |
| TRIAL | 1 | 1 | 8 | 100 | 937 | 5565 | 59872 |
| TRIAL | 1 | 1 | 6 | 120 | 956 | 5454 | 53211 |
| TRIAL | 1 | 1 | 9 | 127 | 820 | 5134 | 51234 |
| TRIAL | 1 | 1 | 8 | 130 | 825 | 5222 | 58792 |
| TRIAL | 1 | 1 | 8 | 124 | 872 | 5321 | 58797 |
| TRIAL | 1 | 1 | 6 | 78 | 778 | 5768 | 58972 |
| TRIAL | 1 | 1 | 7 | 80 | 801 | 5673 | 54312 |

Figure 49: measured data after I had run the dot product using my optimized code for windows 32-bit compiler, the data will be graphed later in this report and compared to the original time measurements for using the compiler generated assembly code that I have already linked and measured.

This will make it easier for us to notice that my optimized assembly code is efficient, more efficient than the code that was generated by the windows 32-bit compiler, but this is an improvement, it is more efficient.

**4.3 DOT PRODUCT USING POINTERS ON WINDOWS 32 BIT**

In this section I will be computing the dot product using Visual Studio and Windows 32-bit compiler generated assembly code using pointers. To generate the code what I need to do edit the properties of the Compiler to generate assembly code. This will happen by selecting properties and then selecting output assembler FA to see the assembly instructions that will be stored into the debug folder. After it is loaded, I will link the assembly code to the main file by using the Windows Macro Assembler setting. Then it will be used to time how long it takes to perform the dot product using pointers on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to its optimized code.

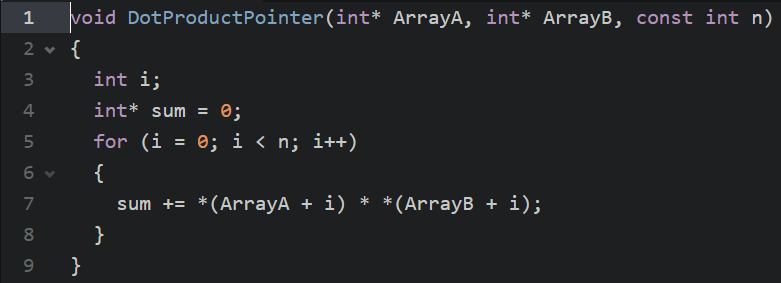


Figure 50: C++ code that will be used to generate the assembly code that will be linked to the main file to measure the time it takes to calculate the dot product on windows 32-bit compiler using pointers.

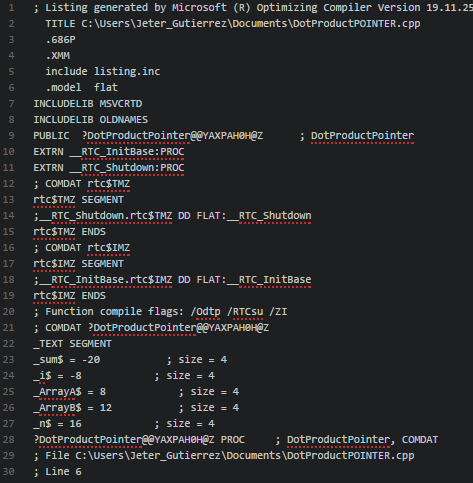


Figure 51: Assembly code that was generated by the visual studio compiler on windows 32-bit compiler to calculate the dot product using pointers that will then be linked and measured.

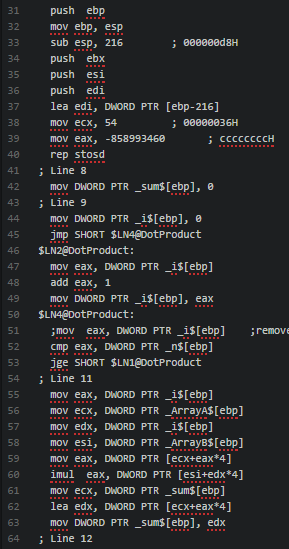


Figure 52: Continued assembly code for generating dot product using pointers on windows 32-bit compiler.

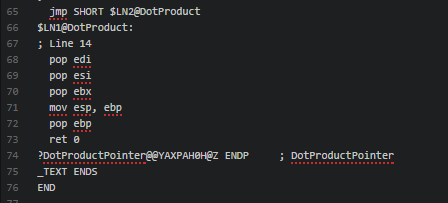


Figure 53: Continued assembly code for generating dot product using pointers on windows 32-bit compiler that is now finalized and will be tested.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| WINDOWS 32 BIT TIME VALUES FOR POINTER | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 1 | 1 | 8 | 80 | 808 | 6601 | 65981 |
| TRIAL | 1 | 1 | 8 | 80 | 803 | 6679 | 98933 |
| TRIAL | 1 | 1 | 8 | 80 | 850 | 7101 | 66392 |
| TRIAL | 1 | 1 | 8 | 80 | 946 | 6602 | 68729 |
| TRIAL | 1 | 1 | 8 | 80 | 815 | 6983 | 67558 |
| TRIAL | 1 | 1 | 8 | 80 | 817 | 6737 | 94812 |
| TRIAL | 1 | 1 | 8 | 80 | 806 | 6602 | 81446 |
| TRIAL | 1 | 1 | 8 | 80 | 803 | 7101 | 68279 |
| TRIAL | 1 | 1 | 8 | 80 | 969 | 6601 | 69072 |
| TRIAL | 1 | 1 | 8 | 80 | 949 | 6595 | 68279 |

Figure 54: Data for calculating the dot product using pointers using the compiler generated assembly code that was linked to the main file to measure the time that it takes to run the operation for different values for the sizes of the arrays.

**4.4 DOT PRODUCT USING OPTIMIZED POINTER ON WINDOWS 32 BIT**

In this section I will be computing the dot product using Visual Studio and Windows 32-bit compiler generated assembly code using pointers. To generate the code what I need to do edit the properties of the Compiler to generate assembly code which I will then optimize. This will happen by selecting properties and then selecting output assembler FA to see the assembly instructions that will be stored into the debug folder. After it is loaded, I will link the assembly code to the main file by using the Windows Macro Assembler setting. Then it will be used to time how long it takes to perform the dot product using pointers on arrays of sizes; 10, 100, 1000, 10000, 100000, 1000000, and 10000000. Then I will measure and record the time it takes and store the data to later use it on Graph to compare to its compiler generated assembly code.

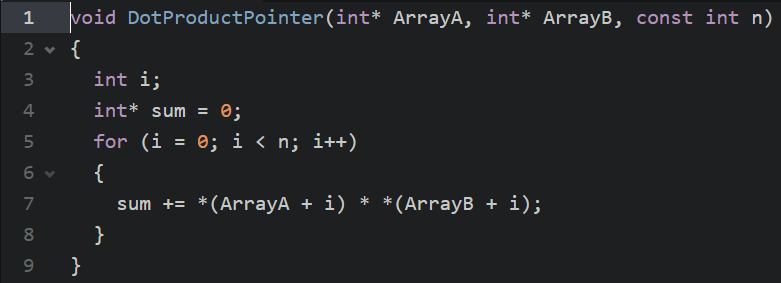


Figure 55: C++ code that will be used to generate the assembly code that I will optimize for calculating the dot product using pointer on windows 32-bit compiler using visual studio.

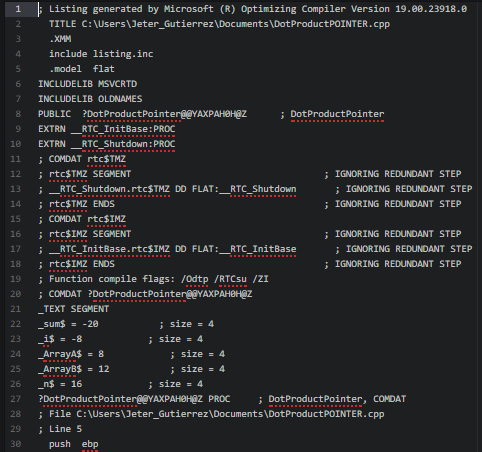


Figure 56: Assembly code where I optimized calculating the dot product on windows 32-bit compiler using pointers. I have removed the repeated steps that are not needed to calculate the dot product.

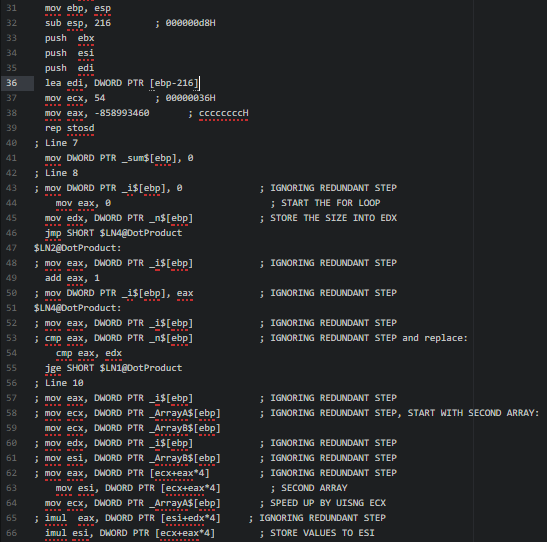


Figure 57: I have removed most repeated steps, they do not need to happen as frequently as they did initially to calculate the dot product of arrays using pointers.

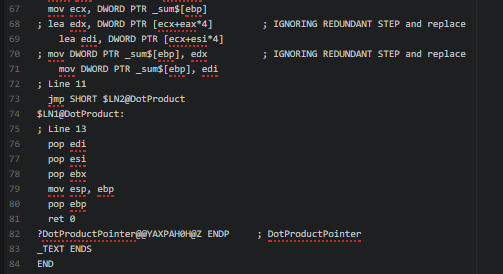


Figure 58: Assembly code for calculating the dot product using pointers that I have optimized for the windows 32-bit compiler. I have removed 2 more redundant steps that repeat a lot.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| WINDOWS 32 BIT TIME VALUES FOR POINTER OPTIMIZED | | | | | | | |
| N= | 10 | 100 | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| TRIAL | 1 | 1 | 6 | 9 | 100 | 619 | 6402.97 |
| TRIAL | 1 | 1 | 5 | 10 | 86 | 760 | 6478.63 |
| TRIAL | 1 | 1 | 6 | 8 | 98 | 820 | 6887.97 |
| TRIAL | 1 | 1 | 6 | 8 | 117 | 600 | 6403.94 |
| TRIAL | 1 | 1 | 5 | 9 | 78 | 1450 | 6773.51 |
| TRIAL | 1 | 1 | 6 | 10 | 79 | 686 | 6534.89 |
| TRIAL | 1 | 1 | 6 | 10 | 94 | 600 | 6403.94 |
| TRIAL | 1 | 1 | 6 | 7 | 94 | 627 | 6887.97 |
| TRIAL | 1 | 1 | 5 | 9 | 78 | 623 | 6402.97 |
| TRIAL | 1 | 1 | 5 | 8 | 79 | 730 | 6397.15 |

Figure 59: Time values recorded for calculating the dot product using my optimized assembly code that was optimized using pointers on Windows 32-bit compiler. This will be graphed and compared to its running time for the original code and to the other 2 platforms that were tested during this project.

**5. GRAPHS**

In this section I will display some graphs for the Index calculated dot product on all 3 platforms, and optimized, and using pointers and optimized for that too. We would have too many graphs if we graphed every column so instead I will only be showing the graphs for N=10, and 10000 in each case.

*5.1 INDEX FOR DIFFERENT VALUES NORMAL AND OPTIMIZED ON ALL 3 PLATFORMS*

In this section I will begraphing the different tables that were used to measure the time that each of the dot product performances take. I will be observing

Figure 60: Time for index on all 3 platforms for N=10;

Figure 61: Time for optimized index on all 3 platforms for N=10.

For a small value of 10 the difference in time based on the optimization is not very noticeable but it exist.

Figure 62: Time for index on all 3 platforms when N=10000.

Figure 63: Graph for time for running index optimized dot product on all 3 platforms for n=1000. Now using a larger value we can see that there is significant difference in the performance of the platforms for higher values of calculating the dot product using indexes, this means that I was able to successfully optimize the assembly code that was linked to the main file in every case to measure the time.

*5.2 POINTER FOR DIFFERENT VALUES NORMAL AND OPTIMZED ON ALL 3 PLATFORMS*

Figure 64: Graph for the times for all 3 platforms for calculating the dot product using pointers and the compiler generated assembly code that was linked to a main file to calculate the time in each case.

Figure 65: Graph for the time that was measured when calculating the dot product on all 3 platforms using my optimized assembly code. The difference for such a small value of N where it is 10 can be see for the raspberry pi in the fluctuation, overall we can see a difference.

Figure 66: Graph for the compiler generated assembly code that I made on all three platforms. This is for N= 10000.

Figure 67: As we can see when N is equal to 10000 the value for using pointer optimized is smaller on all 3 platforms, specifically for windows on all 3 platforms.

*6. CONCLUSION*

Performing this project helped me develop a better understanding for the assembly language. It was helpful to me. Considering that my optimized code generated significant results in calculating the dot product using pointers optimized or indexes optimized, this means that I have a better understanding in the assembly language for 3 different platforms now. In general, it took some time to understand the procedure of what was happening before I could optimize the code but I was successful in doing it.