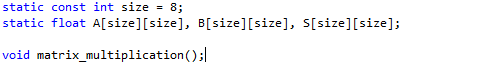
|  |
| --- |
|  |
| **Matrix Multiplication, Compiler Generated Code Optimization** |
| Brandon Chin |
|  |
| **CSC342 - Instructor: Prof. Izidor Gertner** |
| **4/23/2015** |

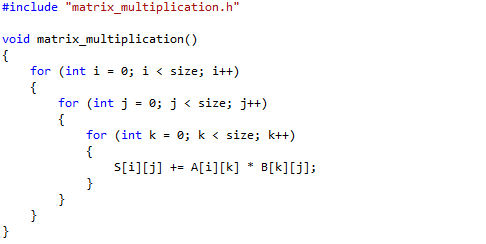
|  |
| --- |
|  |

**Objective:**

We are going to create compiler generated assembly code for a matrix multiplication function. Then we will modify this compiler generated assembly code using three different optimization techniques. First without any optimization, then with automatic parallelization and vectorization, and finally with DPPS instruction. In the end, we will organize and compare our results after execution of all methods.

**Matrix Multiplication Function**

*matrix\_multiplication.h*

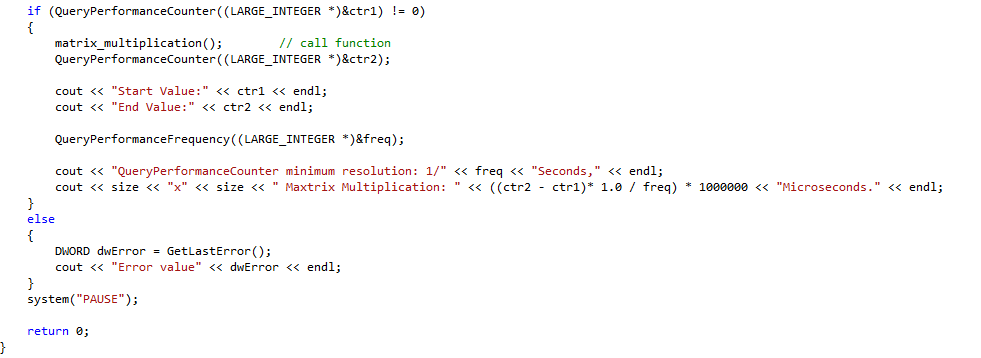
**

*matrix\_multiplication.cpp*

Here, we can see the matrix multiplication function written C++. The function computes the product of two square matrices. We will start with two 8x8 matrices, where the size of the matrices will go up by powers of 2, for example, 8x8, 16x16, 32x32, 64x64, etc. Each element of the product matrix S is equal to the summation of the corresponding row of A multiplied by the corresponding column of B.

**Un-optimized Compilation**

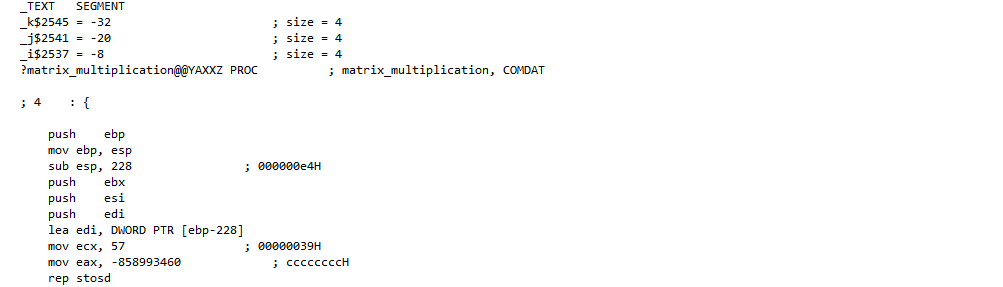


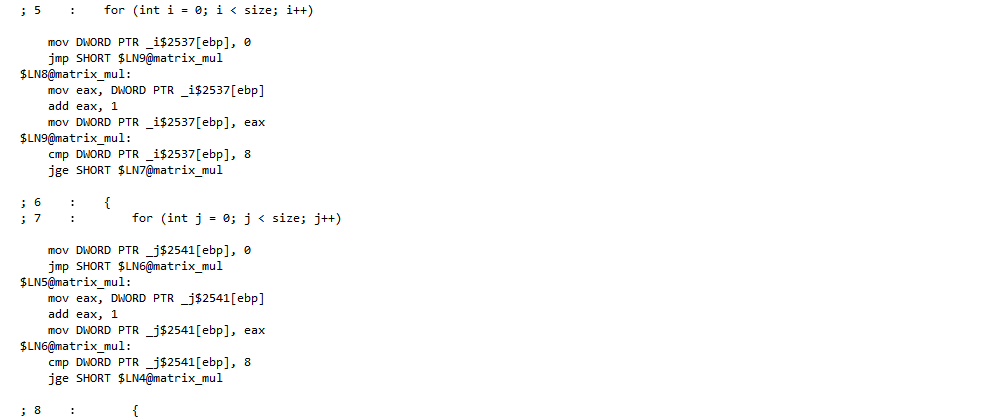


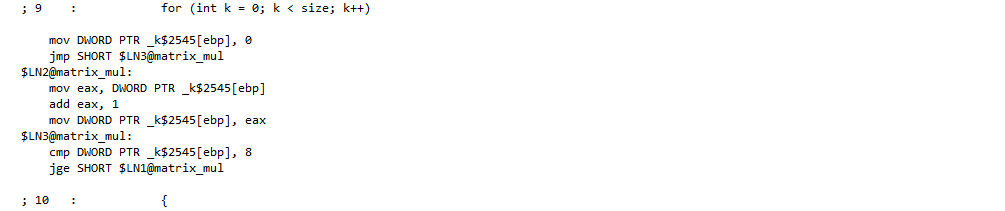
*main.cpp*

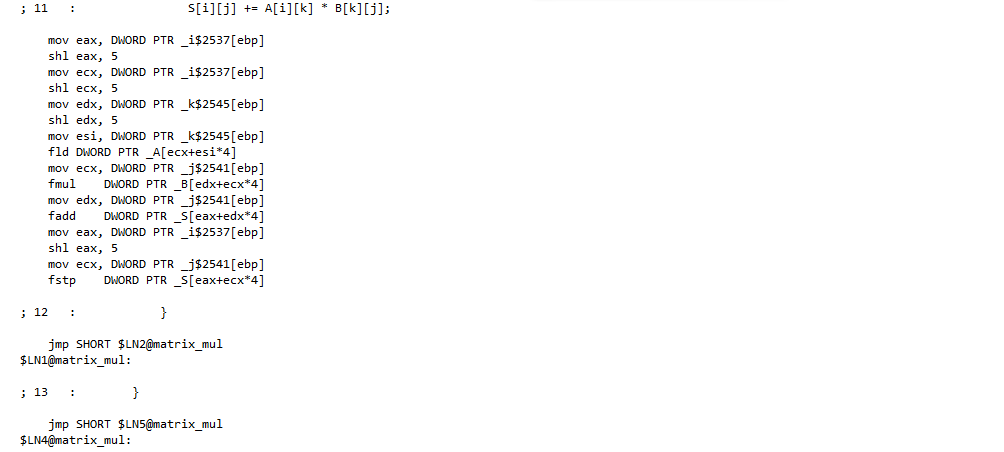
The main function initializes all of the arrays and calls the matrix multiplication function. We are using the same "QueryPerformanceCounter" function that we used before to time the duration of the multiplication function.

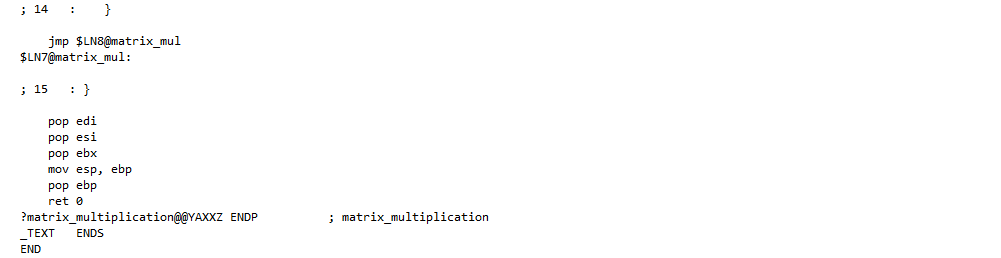
Now let's link the compiler generated assembly code shown below:







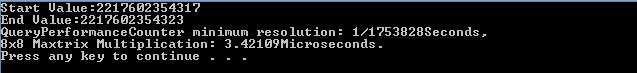




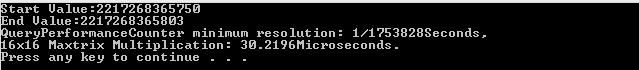
*matrix\_multiplication.asm*

After executing this code, we get the following results shown below:

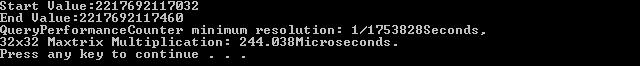
***8x8 Matrices:***



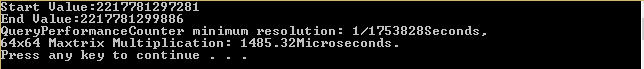
***16x16 Matrices***



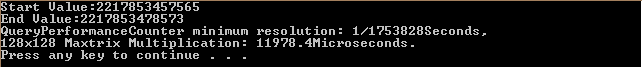
***32x32 Matrices***

******

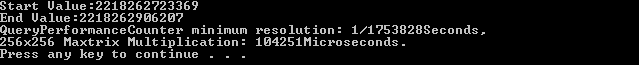
***64x64 Matrices***



***128x128 Matrices***

******

***256x256 Matrices***

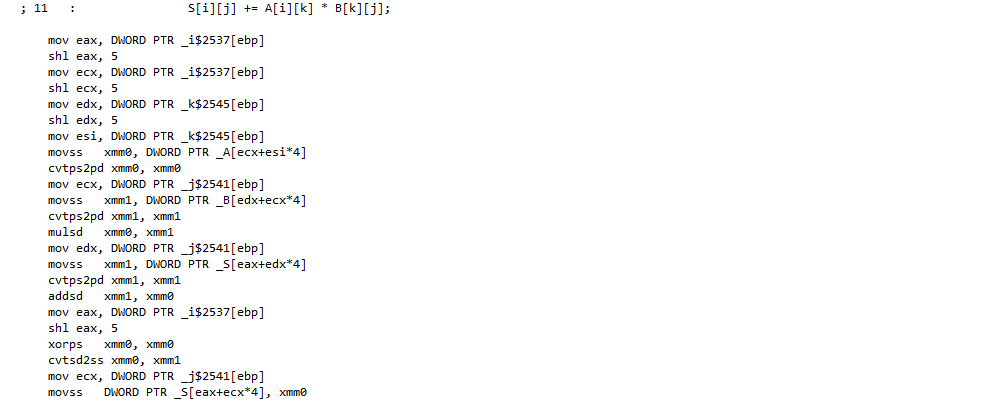
******

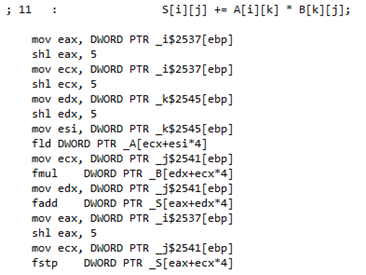
***512x512 Matrices***



**Optimized with Automatic Parallelization and Vectorization**

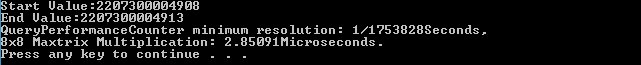
To enable these instructions, we must go to Properties -> C/C++ -> Code Generation. Next to "Enable Parallel Code Generation" click "**Yes (/Qpar)**" and next to "Enable Enhanced Instruction Set" click "**Streaming SIMD Extension 2 (/arch:SSE2)**". Now let's recompile the code and notice the xmm registers being used instead:



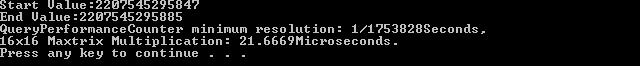


*matrix\_multiplication.asm -- line 11 (before) matrix\_multiplication.asm -- line 11(after)*

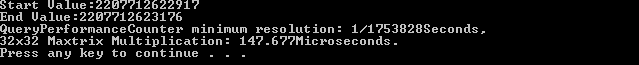
***8x8 Matrices:***



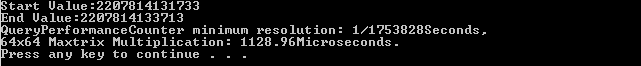
***16x16 Matrices***



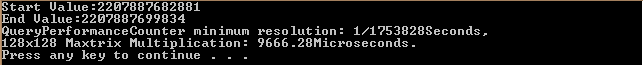
***32x32 Matrices***

******

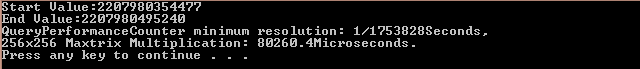
***64x64 Matrices***



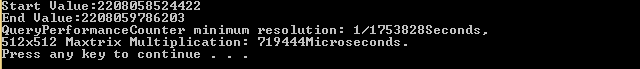
***128x128 Matrices***

******

***256x256 Matrices***

******

***512x512 Matrices***



We can manually optimize our asm file slightly by modifying the loop executions. The compiler automatically reads from memory and moves the value into the eax register. Then it increment the value in the register by one, then moves it back into memory. This is done every time each loop iterates. Instead we can optimize this by keeping the value inside memory and incrementing the value right from memory. That way we never need to use the eax register for the loops.



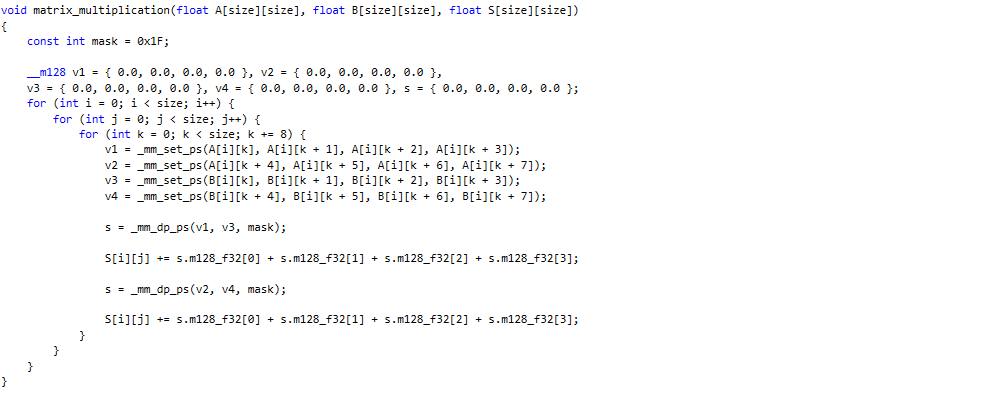




*matrix\_multiplication.asm -- lines 7-12*

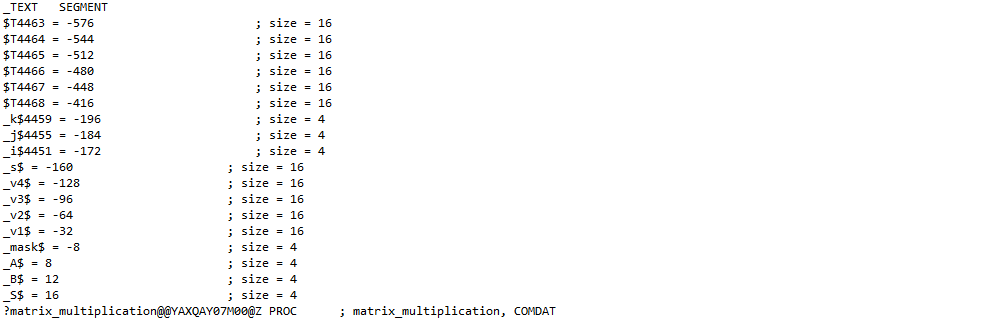
**DPPS Vector Instruction Optimization**

Using the smmintrin.h library, we can call the m\_dp\_ps function. We then can create four \_m128 type vectors and load the matrix values using the \_mm\_set\_ps function.

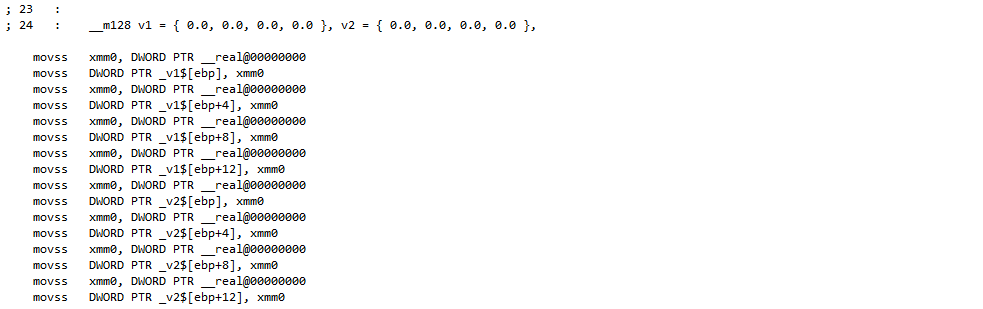


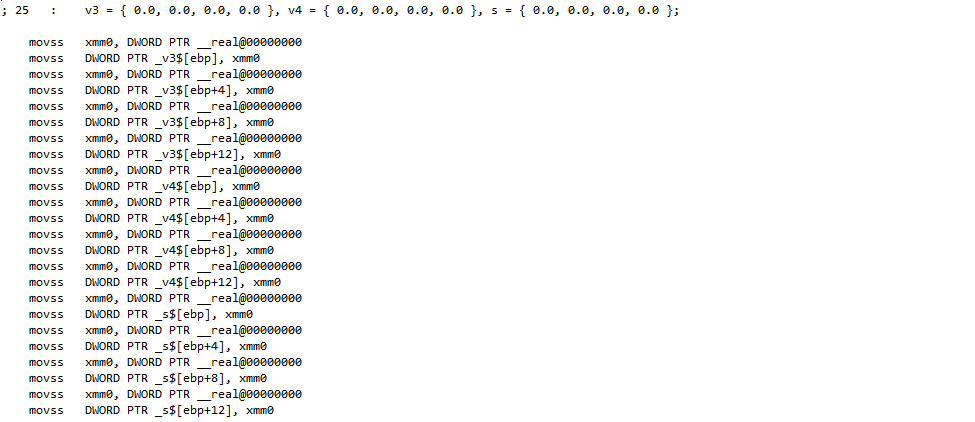
*matrix\_multiplication.cpp*

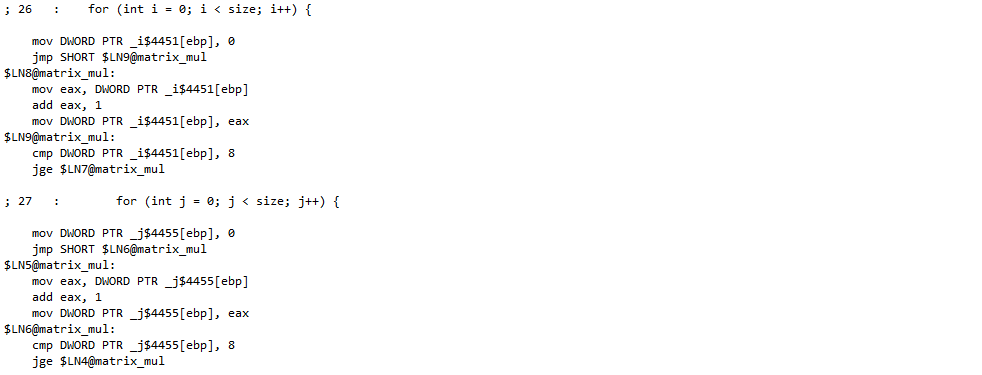
Below is the compiler generated assembly code for the matrix multiplication function using dpps:



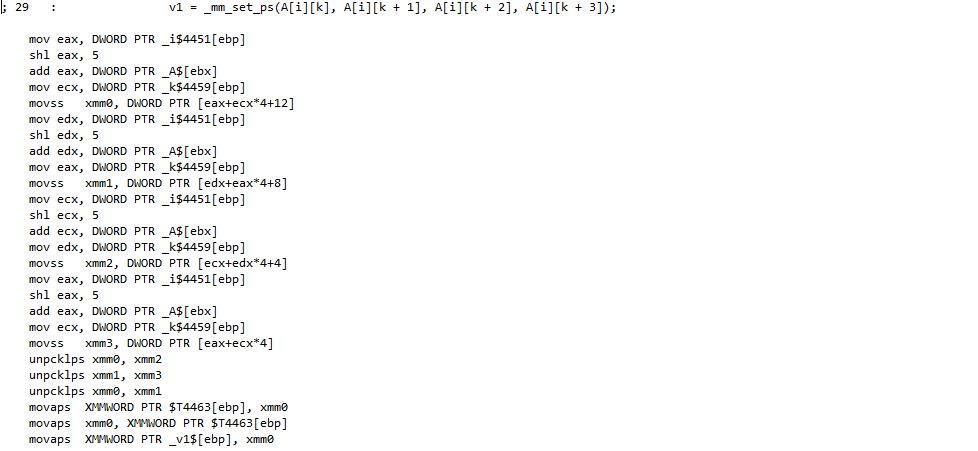


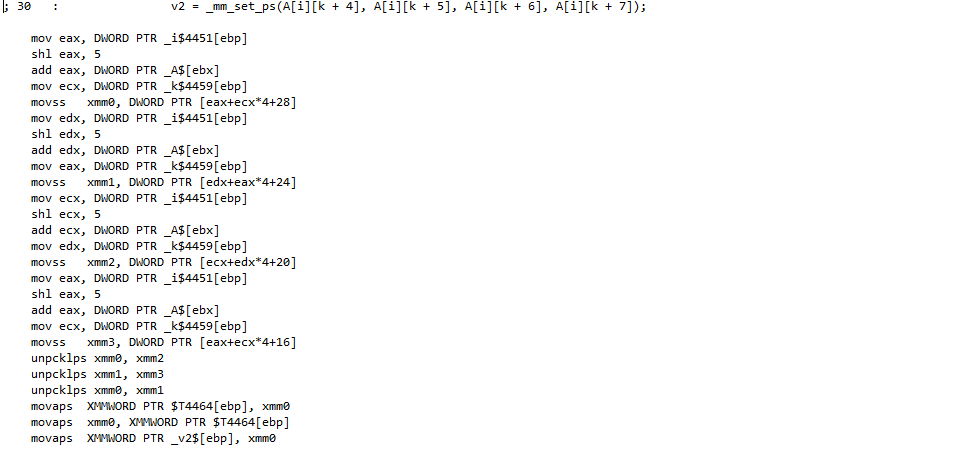


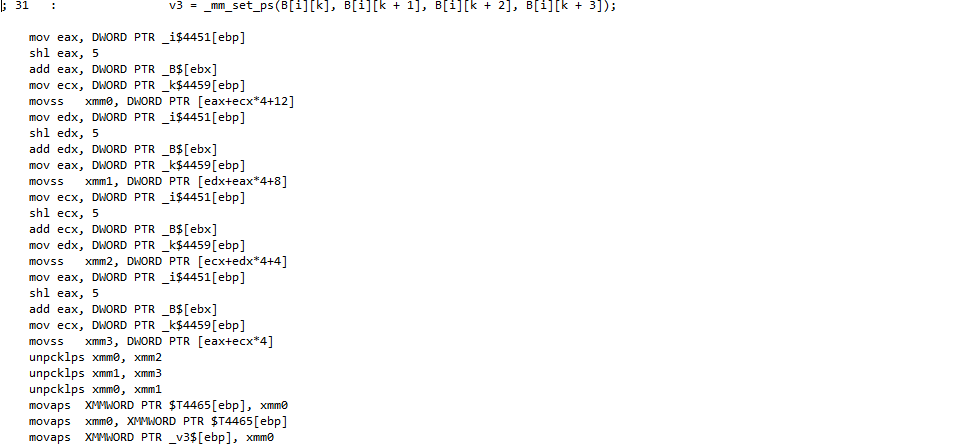


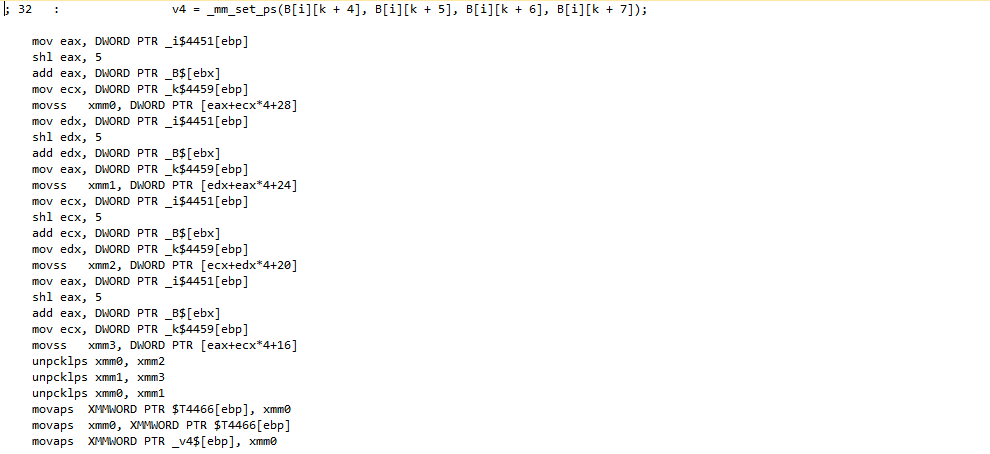


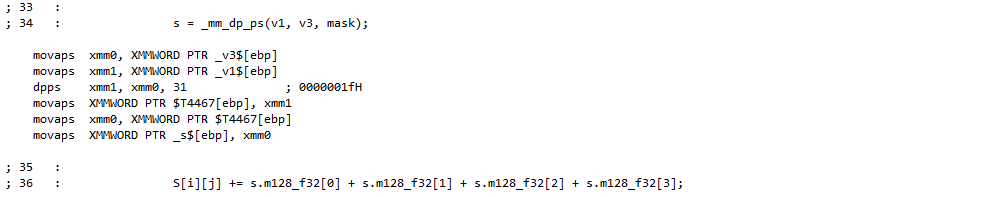


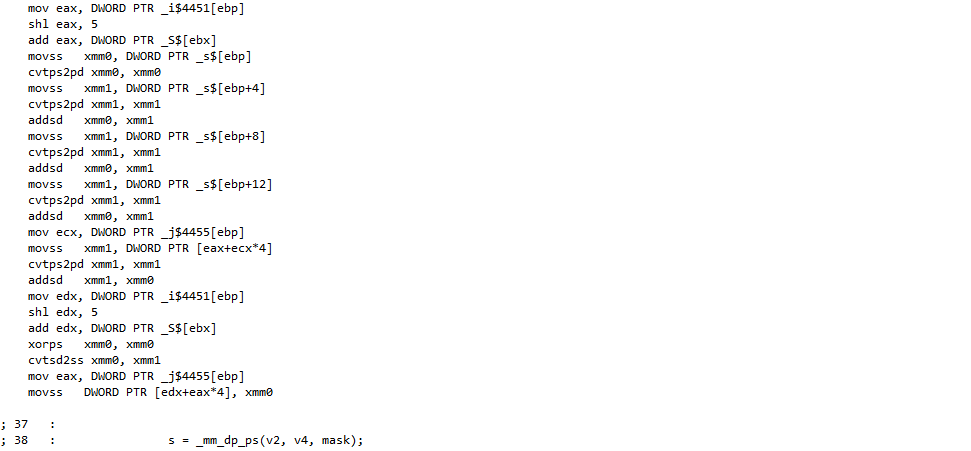


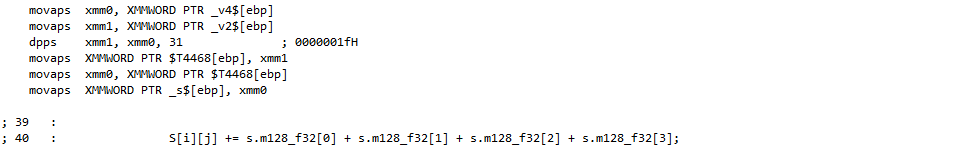




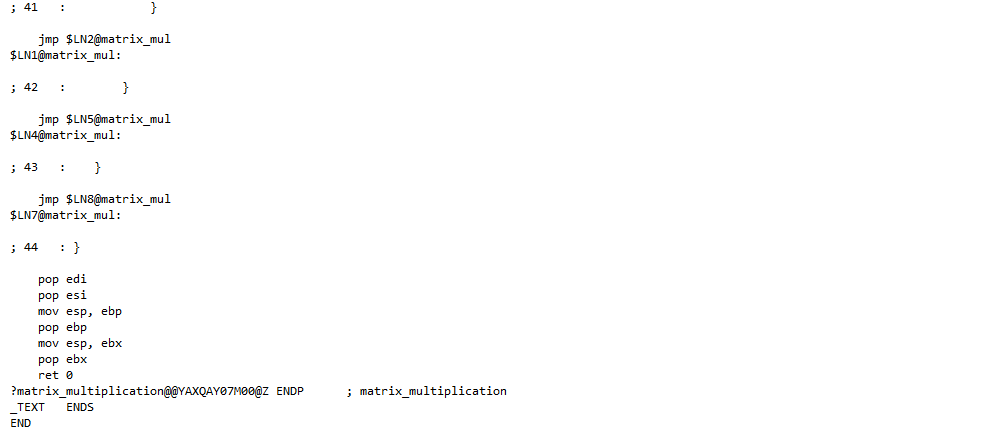












**Results**

Now let's compare all of our results with matrices ranging from 8x8 to 512x512:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Size N** | | | | | | |
| **8x8** | **16x16** | **32x32** | **64x64** | **128x128** | **256x256** | **512x512** |
| **Before Optimization** | 3.42 | 30.22 | 244.04 | 1485.32 | 11978.40 | 104251.00 | 850372.00 |
| **/Qpar and /arch:SSE2** | 2.85 | 21.67 | 147.68 | 1128.96 | 9666.28 | 80260.40 | 719444.00 |
| **/Qpar and /arch:SSE2 (Optimized)** | 2.85 | 21.67 | 139.78 | 1017.18 | 8821.59 | 77767.20 | 746109.00 |
| **DPPS Instruct** | 2.28 | 12.59 | 128.88 | 1013.98 | 7891.66 | 60142.20 | 492825.00 |

*(Table #1)*

*(Graph #1)*

**Conclusion**

We tested the performance of various optimization methods on square matrix multiplication such as enabling automatic parallelization and vectorization, manipulating compiler generated assembly code, and implementing dpps functions. In the end, we timed, organized and compared the results of each of these optimization methods using the QueryPerformanceCounter. We then charted and graphed these results in order to easily analyze and compare the performance between them.