Ian Romano

Final Project Report

CPEG655

Fall 2021

SSE Matrix Multiplication

# Getting Started

This code base is a .NET solution project build on VS Community edition 2022. You’ll need to import the Solution package into Visual Studio in order to compile it. Also, ensure that the build is set to release mode, not debug mode.



The code base can also be found at <https://github.com/irromano/SSE_MatrixMult>

# Introduction

While Moore’s law has delivered nearly double the transistors into a processor every two years, much of our recent float calculation time has been addressed by changing how the processor handles instructions. One way to increase throughput with vector calculations is through SIMD. SIMD stands for Single Instruction Multiple Data, which describes the logistics of how a processer can calculate multiple but similar calculations simultaneously after receiving a single instruction. SSE stands for Streaming SIMD extensions and is a set of x86 instructions which allow programmers to manually direct a processor to perform a SIMD operation.

The purpose of this project is to compare the performance of a program using SSE vs normal operations when calculating a matrix operation.

# Procedure

A program was developed to calculate the matrix multiplication of two ‘N’ by ‘N’ float matrixes, where ‘N’ is a large number of base 2. The program first makes this calculation normally through serial instructions and then again using SSE instructions. Both functions compute the matrix equation A \* B = C and are timed separately.

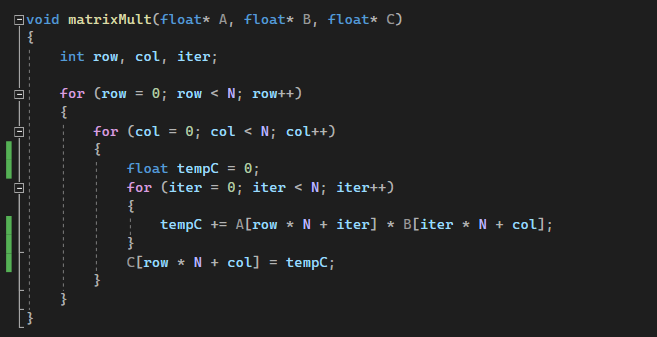


Figure 1 Standard Matrix Multiplication Function

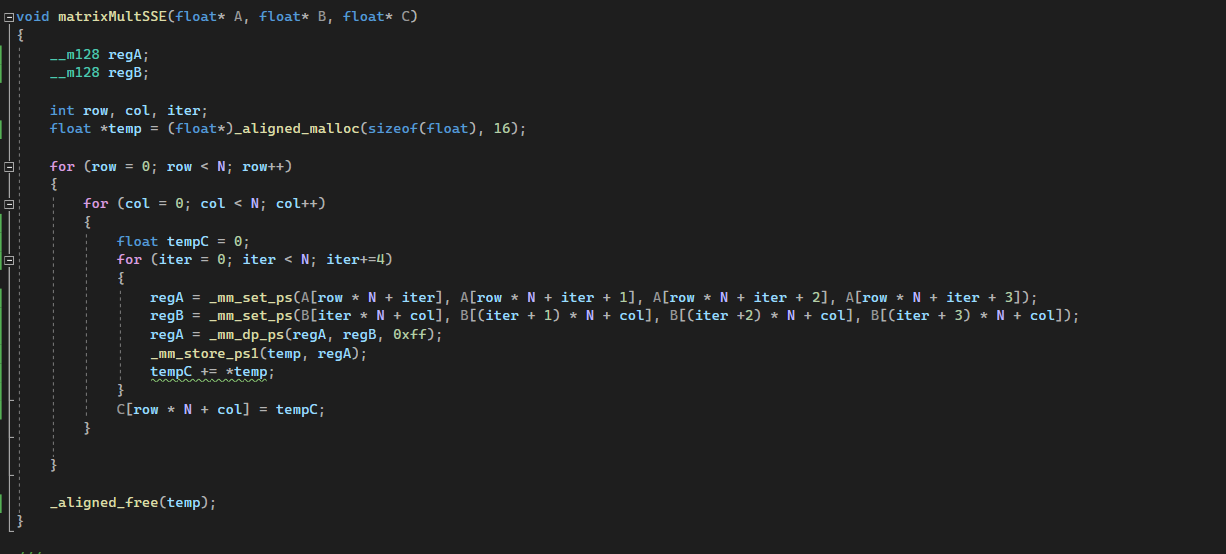


Figure 2 SSE Matrix Multiplication Function

The SSE matrix function uses the \_\_m128 datatype from immintrin.h which represents a 128-bit register used for SIMD operations. 256-bit AVX vector operations were also explored but still appeared to only calculate two 128-bit vectors instead of one large 256-bit vector, which didn’t add any performance increase. The \_mm\_set\_ps() function, which computes a dot product of two float vectors of 4 floats each, was used to compute 4 index calculations for a given index of the ‘C’ matrix simultaneously. Data used with SSE operations should be allocated and aligned by 16 bits otherwise the operations will suffer a large performance overhead.

# Results

The program was running on two separate processors, an AMD Ryzen 5 5600X 6-Core Processor (3.70 GHz 2020) and an Intel i5-7200U Processor (2.50GHz 2016).

On the AMD processor, the SSE operations began to outperform the standard matrix multiplication function when the matrix side length was 512 (29) and higher. Length sizes of 256 (28) and lower produced longer calculation times, most likely due to the overhead of setting data into the registers. But on the Intel processor, the opposite and least expected happened, the standard function began outperforming the SSE function at length sizes 512 (29) and higher. It is my belief that this is due to Intel’s processor can optimize the standard function more so than the AMD can. Perhaps the AMD processor requires programmers to manually optimize their code for matrix operations.

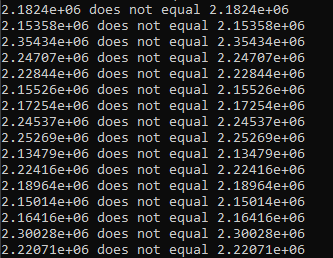
|  |  |  |
| --- | --- | --- |
| Time to compute matrix multiplication given matrix length on Ryzen 5600X | | |
| Operation Time (in µs) | Standard Function | SSE Function |
| N=211 | 42704824.00 | 37990699.50 |
| N=210 | 3495842.33 | 3322157.33 |
| N=29 | 197699.00 | 165360.00 |
| N=28 | 15775.33 | 17387.67 |
| N=27 | 1805.00 | 2034.00 |

Figure 3 Performance of functions on AMD Ryzen 5600X

|  |  |  |
| --- | --- | --- |
| Time to compute matrix multiplication given matrix length on Intel i5-7200U | | |
| Operation Time (in µs) | Standard Function | SSE Function |
| N=211 | 87997598 | 95248261 |
| N=210 | 6919070 | 7361664 |
| N=29 | 297726 | 277956 |
| N=28 | 33336 | 29509 |
| N=27 | 3005 | 2533 |

Figure 4 Performance of functions on Intel i5-7200U

One interesting finding was that a comparator between the result from an index float of the C matrix from the standard function did not always claim to be equal to the same index from the SSE result, even though the number appeared to be equal. This might be due to the slight differences of how float operations are calculated by SSE instructions vs standard x86 instructions.



# Conclusion

While the SSE operations proved to be more efficient than the standard programming on the AMD processor, they weren’t as efficient as I thought they would be given, given how the SSE instructions compute 4 calculations simultaneously. This may be due to the compiler recognizing the potential to vectorize the standard operation and adding SSE instructions into it on its own. This may be especially true for the Intel processor since the x86 architecture as well as SSE is designed by Intel. It’s still noteworthy to recognize that some performance gain can be made by manually calling these operations without relying on the compiler to optimize it for the engineer.