Matrix Study

Gabriel Bello
301430169

This research focuses on the multiplication of the Two matrices that are of 2x2 which will use a specialized algorithm and pattern. The goal is to explore how this operation is performed in both high-level and low-level programming languages, more specifically, C and Assembly.

In this study, I will be implementing 4 different ways to do the two 2x2 matrices multiplication algorithms, inspired by a video I came across while looking for different matrix operations. The video is titled "*Multiplying Matrices 2x2 by 2x2*" found on youtube under the Corbett Maths channel. This is a reliable resource if you'd like to have a more detailed explanation on the math operations being done. My objective is to evaluate the performance of four distinct implementations of this algorithm, each employing different techniques and optimizations.

Two implementations will be in C. The first C implementation will use direct array access for the matrix. By doing so the matrix will be able to manipulate data in a more straightforward manner in hopes that the compiler will have an easier time compiling and understanding our human written code. The second implementation will also use direct array access but will be compiled with an unsafe math optimization. This optimization, while in hindsight should improve performance, may also introduce inaccuracies, due to its nature of cutting corners to get the results faster. More particularly given that our matrices are filled with floating-point values also known as floats and doubles, where precision is more crucial.

The other two implementations will be using the Assembly language using the x86 syntax. The first Assembly implementation will work directly with memory addresses using registers, allowing for more control over data access and manipulation at byte level. One thing to note about this implementation is that I had to personally access the addresses of each value meaning that, we had no assistance of remembering where we currently are in memory. Which also proved challenging at certain points because oftentimes we would end up outside the scope of an array causing many inaccuracies. The second Assembly implementation will utilize a stack and pointers to manage memory addresses. This will offer a different approach to memory management in hopes of impacting performance positively as we can easily push and pop memory addresses as needed.

By comparing these implementations, I aim to gain more insight into how different coding practices and optimization levels affect performance and accuracy with data, more specifically, with the matrix multiplication. This study aims to provide a memory detailed look of how high leveled features and low leveled features impact efficiency and precision. Any and all detailed statistics recorded can be accessed through my GitHub under the repository "Matrix_Study".

In regards to performance, the first evaluation I decided to conduct was with the actual run-times. These implementations were evaluated under the different optimization flags of GCC with them being the '-O1', '-O2', '-O3' and no optimization flags. Upon running them at different times with both large and small values, our data reveals that the different compiler optimizations are more noticeable with larger numbers, especially for the functions involving floating-point operations as seen with the 'matrices_double' (first Assembly implementation). For large numbers, the '-O3' provided the best results, reducing the cycle count significantly. Meanwhile, smaller numbers showed less variations in execution time across different optimization levels, with consistent improvements and less dramatic changes as compared to their counterpart. The 'matrices' (C v1) function generally performed better with higher optimization levels, while the 'matrices_Optmized' (C v2) function showed a consistent efficiency level across the different optimization flags. The 'matrices_double' function, implemented in Assembly, shows the most improvement with optimizations. This is likely due to its initial approach of brute forcing memory addresses in registers, and while it is straightforward, it is not the most efficient. When subjected to higher optimization levels, the compiler likely moved away from this less efficient method instead, opting for more advanced techniques to handle memory addresses thus resulting in the major changes in its run time. Meanwhile,'matrices_double_stack' was the function that was more prone to have changes with the different optimization levels. So in regards with run times, higher optimization levels tend to improve performance, particularly when given large numbers and more complex instruction.

Memory locality, as evident by the performance statistics, show variations of the efficiency of caches being used. In this study we will be taking a look at the L1 & LLC data as we go through the four implementations. For the 'matrices' implementation, L1 data cache load misses was 9037 (3.09% of all access), and the LLC load misses were 313 (9.21% of L1 instructions) which indicate some inefficiencies in the cache use. 'Matrices_Optmized' showed some improvement to its counterpart with the L1 cache load misses being reduced to 8280 (2.81%) and LLC load misses decreasing to 148 (6.42% of L1 instruction cache access), which shows much better cache efficiency. The 'matrices_double' implementation, L1 data cache load misses was 6839 (2.34% of all access) but it also increased LLC load misses to 1056 (63.37% of L1-instruction cache access). This indicates a much improved L1 cache performance but we do now have a big reliance on the LLC. This again, I think is evident when it comes to brute forcing memory not being as efficient as we may think. Finally the 'matrices_double_stack' function shows the L1 data cache load misses was at 8239 (2.78%) and LLC load misses was at 303 (14.25% of the L1 instruction cache access). With this in mind we can see the variations between the performance and LLC usage, demonstrating the complexity of optimizing memory access patterns with regards to the compiler. The biggest take away would be direct brute forcing of memory showed a bigger reliance with LLC and an increase with its misses.

Now let's analyze branch predictability by tracking branch-misses data that provides valuable insights into the efficiency of the different matrix multiplication implementations. For the 'matrices' branch-misses were recorded at 6593, accounting for all 3.01% of all branches. The 'matrices_Optmized' ,contrary to past tests, actually reports a higher percentage of branch misses which was 6887 or 3.15%. I think this is due to the unsafe math operations causing for more unpredictable branches when compiled. Meanwhile, the 'matrices_dobule' achieved much fewer branch-misses at 5883 or 2.70% which suggests much better branch prediction! Out of all the functions this one netted having the best branch prediction which I presume may be the cause of the direct access to memory addresses when brute forced and since the branches can pick up on a pattern with the arrays being accessed it leads to a much easier time in predicting. When looking at the 'matrices_double_stack' it reports 6568 branch-misses which corresponded to 2.97% of all branches showing a slight downgrade to 'matrices_double'. These results demonstrate that while predictability is generally better in assembly implementations, the optimization on C code leads to greater branch misses.

Finally, I want to touch upon seeing how the base implementation, which is the ‘matrices’ function, and its changes to Assembly code when there are optimizations compared to having no optimization. When it has no optimization it relies on a manual calculation process, where each element of the matrices is seen being individually loaded, multiplied, and added, while being stored onto a stack before being moved to a %xmm0 (%rax) register. Compared to our optimized version which is using the -O1 optimization since we want to work as closely to the machine. The main thing that we can note in the second version is the usage of ‘vfmadd213sd’ instruction, which combines multiplication and addition into a single instruction which is why we see in the run time it not only performed much faster but it enhanced performance and precision! This approach reduces the number of instructions and memory accessing, which results in a more compact and efficient function! Between the two codes, it is safe to assume that using SIMD functions helps processing and efficiency as we are able to combine multiple instructions into one! This also helps as less instructions can lead to less confusion with data which in turn plays a huge role in curating accurate data.

After conducting this study on the 2x2 matrix multiplication, it highlights the significant performance improvement achieved through higher optimization levels and the use of advanced techniques. Optimized Assembly implementation, particularly with SIMD instruction like ‘vfmadd213sd’, demonstrates superior performance and precision compared to both base C implementations and less optimized Assembly code. Also, higher optimization levels, especially ‘-O3’, marked enhanced runtime efficiency for larger datasets, while improved cache utilization was observed in optimized C functions. Direct memory access in Assembly also resulted in better branch prediction and reduced branch misses. Overall, the findings understand the substantial benefits of low-level optimizations and SIMD instructions enhancing computing efficiency and accuracy.