Danielle Nnorom
Dennise Kwarteng

Deliverable 2- Experimental results:

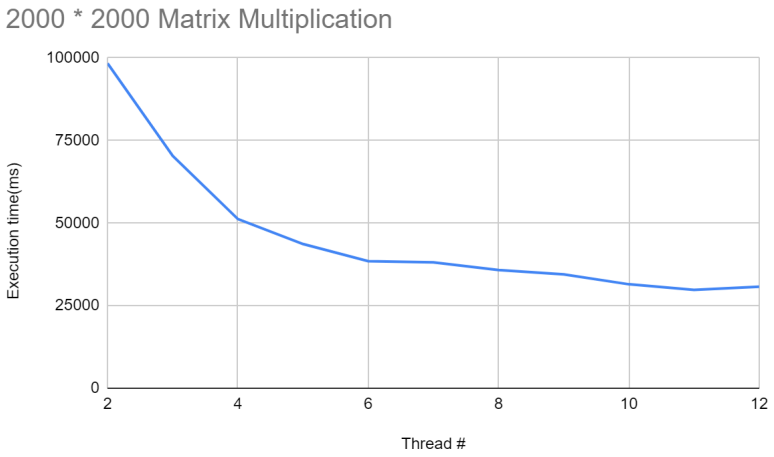
This test was first performed with 2 matrices of size 1000 * 1000.

Native Float	Native Fixed	Thread Float	Thread Fixed	Blocking Float	Blocking Fixed
16717	16236	16060	11986	23528	21691

Threading

A separate test was done on the thread method. The hypothesis was that as the number of threads increased, the time it took for the multiplication to be done would decrease. For this test, we chose the 2000x2000 matrix, and incremented the number of threads from 2 to 7. As can be seen from the graph, the computation time was greatly reduced when the number of threads went from 2 to 3. When we got to 6 to 9 threads, the effectiveness became less steep than from 2 to 4 threads, however it is still there. This goes to show that the more threads that exist, the better the performance. From the output of the C++ command, “std::thread::hardware_concurrency()”, the machine this test was being run on has a maximum of 12 threads. The test was done up to the 12th thread, where the computation time went up. While the time is not significantly higher, it does indicate that there is a limit to the trend. The reason for this is not apparent. It could be possible that the overhead of managing all the threads becomes significant at some point which in this case is the maximum number of threads the system supports. Nevertheless, the performance of using threads is much better than using the normal multiplication operation. We also performed the test on a 3000 x 3000 matrix. While it took a while for the operation to be completed, the time was far better than that of the Blocking method as well as the normal matrix multiplication.

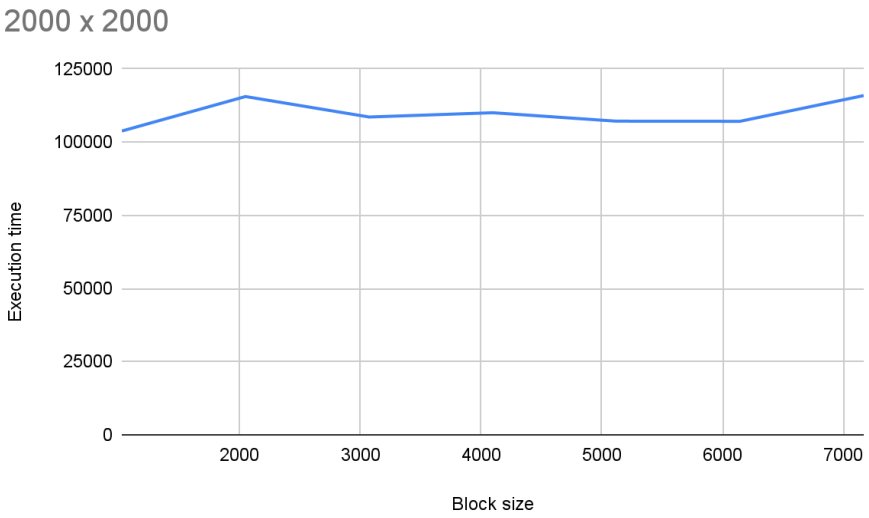
Danielle Nnorom
Dennise Kwarteng



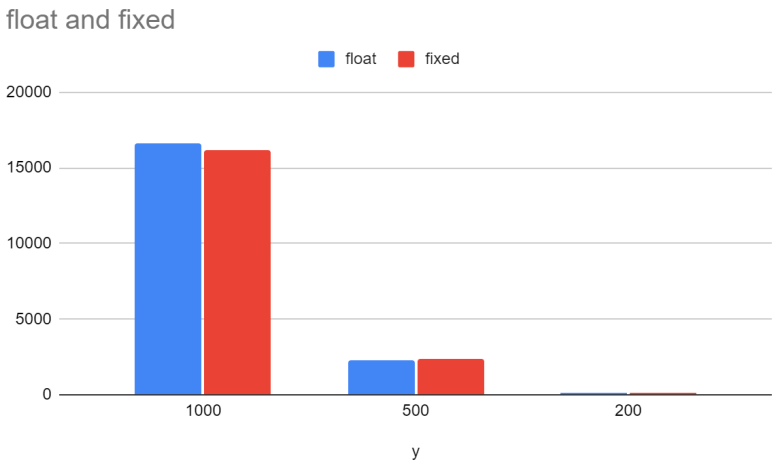
Blocking

When the experiment was performed with the Blocking method, the results were worse than expected. The purpose of the Blocking algorithm is to reduce cache miss as much as possible. It accomplishes this by dividing the matrices into separate blocks and computing the multiplication algorithm on those parts. By dividing the matrices enough, it is able to load an acceptable amount into memory to reduce cache misses. The results of the experiment seem to be that there isn't any actual correlation between the number of divisions and the performance of the algorithm. It is possible that the code that was written doesn't capture what the algorithm is supposed to do. Nevertheless, these are the results of using the Blocking algorithm on a 2000 x 2000 sized matrix multiplication. This algorithm was slightly better than when the test was performed with the normal matrix operation. This, again, is worse than expected. We expected it to be significantly better than without any optimization. The test was not performed with matrices of higher dimensions, because the system being used to test it is not too powerful, and hangs up.

Danielle Nnorom
Dennise Kwarteng



Native



The native implementation performed about the same regardless of whether the floating point or the fixed type was being used. As is expected, when the matrix sizes are 1000x1000, the time is quite high, but drops significantly once the dimensions are reduced.

Danielle Nnorom
Dennise Kwarteng

Deliverable 3 & 4- Performance comparison & Conclusion:

SIMD: So SIMD, single instruction multi data mode for the matrix class works as follows. The way this is done is that the row and columns of the two matrices were mapped to a dynamic array of the set length. For every row of the matrix the row and column of interest are adhered by the arr1 & arr2 variables to hold the data of interest. Then the data is transformed into the correct data type for the intrinsic functions using the masking function and then the column and rows of each grouping is multiplied with its respective counterpart in the other list and then adds the resulting values generated in order to create the values in the newly generated matrix. Then because the arrays were using dynamic memory they are freed before the program comes to a close.

As seen in the Deliverables 2 section the multi threading option performs very well in fact it is more effective than just plain matrix multiplication and the SIMD implementation also helps to improve the overall performance of the program. Using them together is more effective than using them apart, but the benefits of the three techniques do not add linearly.

Conclusion:

The best outcome for the larger programs is to utilize parallelism via multi threading and SMID instructions when possible. This is due to bulky programs needing

to constantly get data from the hard drive will take a considerable amount of time, especially if the program is stalled by requiring the information before moving on. By combining the three techniques it is possible to greatly reduce the amount of time needed to perform such operations, however there is a limit to their effectiveness. Multi-threading loses effectiveness once the number of threads becomes so great that the amount of data needed to be transferred is so small that additional threads actually start to hurt the performance of the system. A similar limit is seen in the SMID messages as they also improve the efficiency of the program, however there is a point of efficacy where the program is no longer limited by the amount of time it takes to process the data using these messages, but by the sheer ability of the machine to process and execute instructions. Although low cache miss rate practices can help improve performance when it comes to running big programs like these ones it is sometimes hurtful as there is an immense amount of data that is being moved, required to do all the operations leading to this type of optimization sometimes hurts especially when combined with the other two methods.