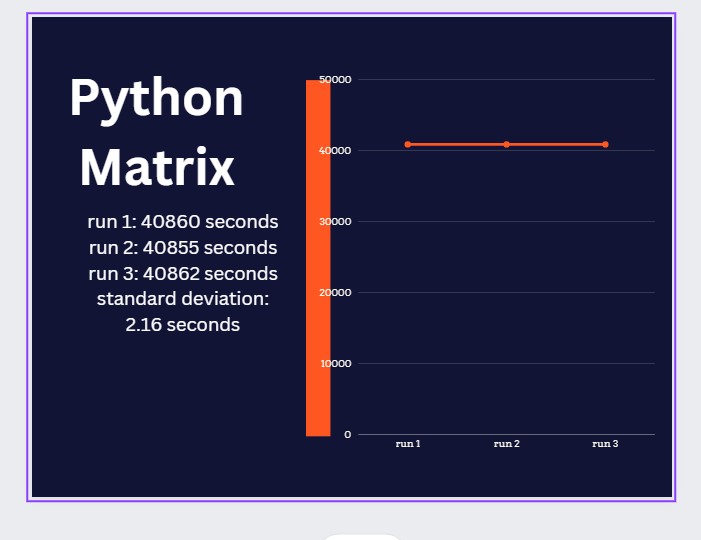
**My Journal**

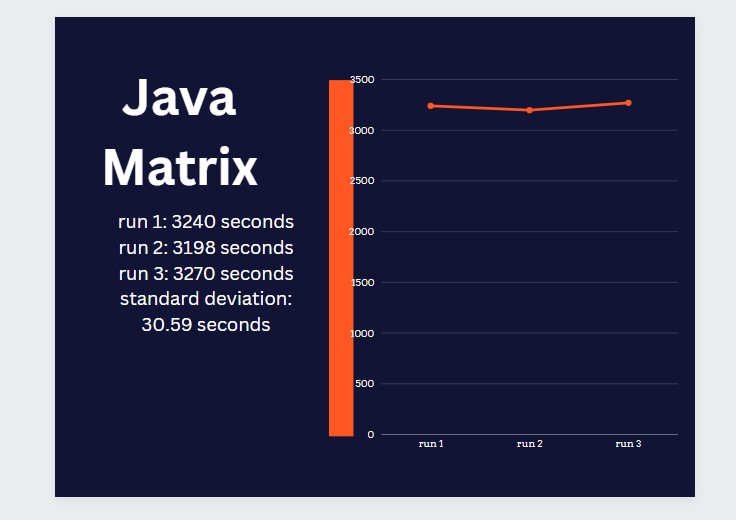
**Name: Saad Hussain Kazmi**

**PART 1**



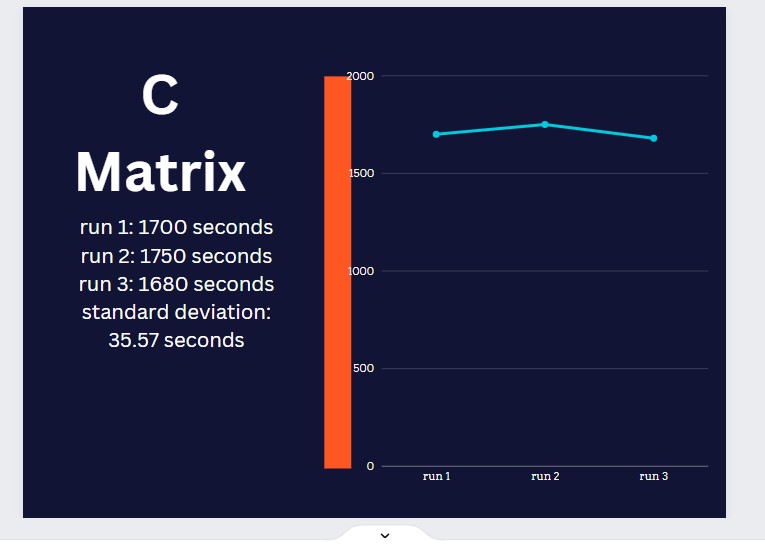
This Python code was written on VSCODE since my VMWARE was an older version. The first step was to import the necessary libraries, which include time for measuring execution time, sys for system-specific functionality, and NumPy for effective array operations. Next, the script uses the command-line options to retrieve the input and output file paths. NumPy data types that equal to matrix types were then used. Information regarding the matrix and multiply handling was received from files and multiplying matrix data, respectively. The execution time and result matrix are written to an output file by the output function. The code scans the input file, creates random matrices using the data supplied, and only performs matrix multiplication in the event that the given opcode is 1. Although doing a 4096 x 4096 multiplication in python takes up a lot of time, which can be seen in the graph, thus maybe it is not so suitable for such a large computation.

Also it took a lot of time to make the necessary computations and I felt as though the laptop was utilizing all available resources just for this computation, quite often.



The javamatrix calculation was done by building classes like MatrixDimensions and Matrix to contain relevant information about our matrices. The enumerator MatrixType was also used to make collecting of info about our types of matrices easier. I used a random number generator to create matrices of specific types and sizes. The matrixMultiply function actually is the one that computes, and prints multiplication progress updates to let me know how the operation is proceeding. The main function includes all the managing file operations, input parameters, and producing random numbers.

This is then followed by writing of the data that is required to the output file. From the graph results, while running the Java code I saw that the computation was occurring quicker than Java at-least, but still it took quite a bit of time as can be seen from the above graph. Although in hindsight it is still better than python in this case.

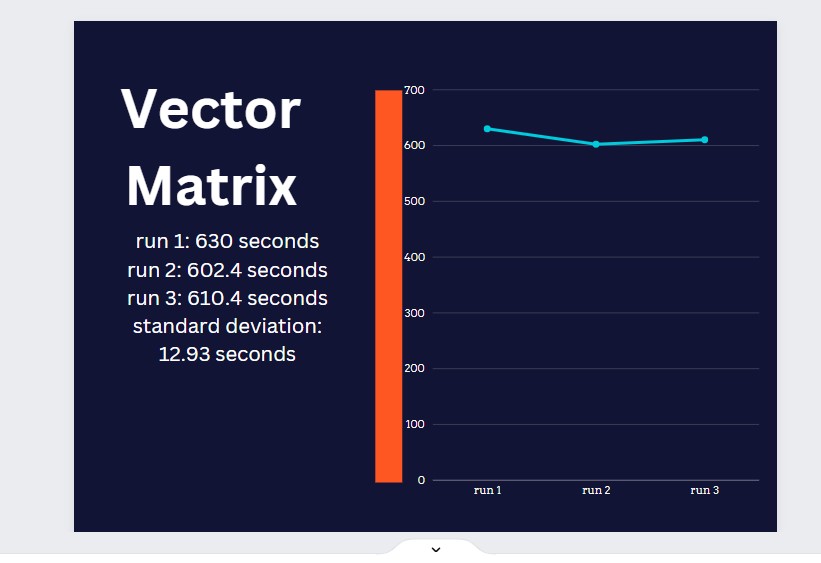


The C code was made to work with matrices of various data types.To represent matrix dimensions and the matrix itself, the code used two structures: M\_dimensions and Matrix. To differentiate between the different data types I used enumerator called Matrixdtype. The create\_matrix method allocates memory and fills it with random values based on the supplied data type, initializing a matrix with the rows, columns, and data type specified.

The multiply function, which performs matrix multiplication, is the central component of the program. It first initializes the result matrix, sets aside memory for its contents, and then multiplies the given data types. The result matrix is formatted and written using the write\_output function.

I then saw that the C program was working better than both the Java and Python program. It was doing computations and giving results more fast in comparison to the both of them. Even in serial form. This would suggest that C would be a faster language I guess.

**PART 2**

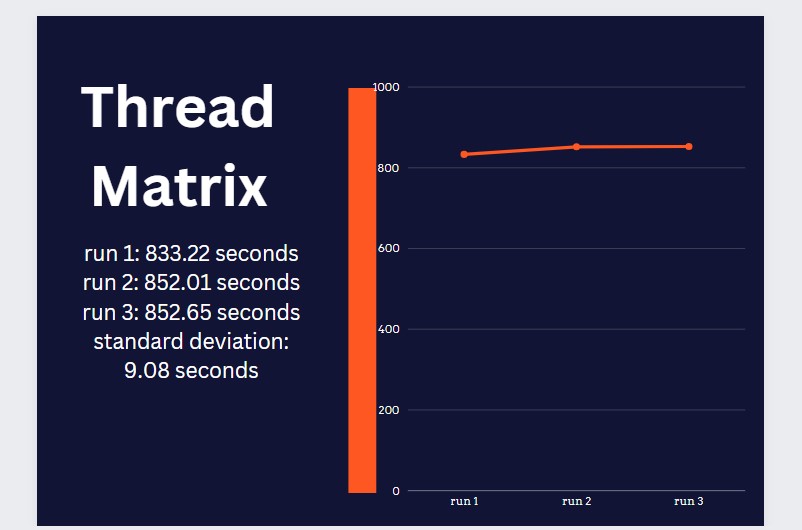


In terms of the C code with the vector instructions I used uses two structures: M\_dimensions and Matrix to represent matrix dimensions and the matrix itself. To differentiate between the different data types, an enumerator used was Matrixdtype. The create\_matrix method allocates memory and fills it with random values based on the supplied data type, initializing a matrix with the rows, columns, and data type specified. The multiply function, performs matrix multiplication by first initializing the result matrix, sets aside memory for its contents, and then multiplies the given data types. The result matrix is formatted and written using the write\_output function.

The multiply function utilizes AVX intrinsics to enhance the performance of the matrix multiplication. AVX allows for parallelization of arithmetic operations on multiple data elements, significantly speeding up matrix multiplication.

Inside the AVX loop of the multiply function, two sets of SIMD (Single Instruction, Multiple Data) instructions (\_\_m128d) are used to perform parallelized multiplication and addition operations, leading to more efficient computation. The resulting matrix is then stored in memory.

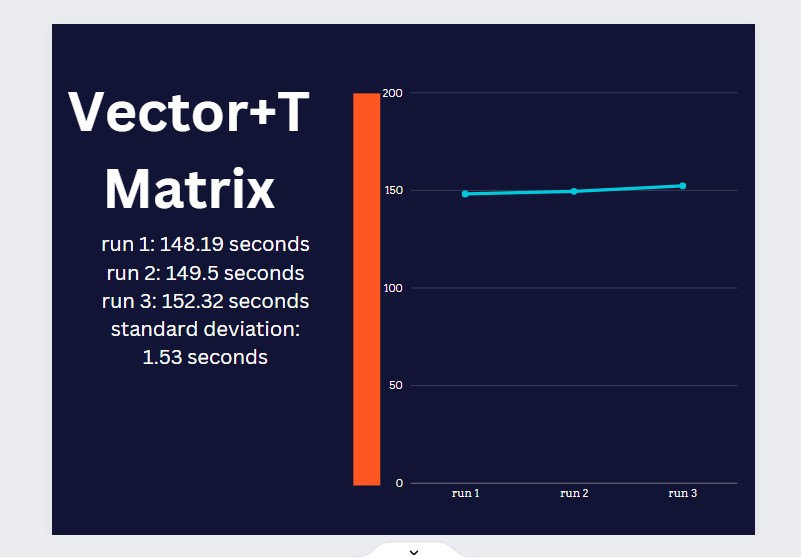
**PART 3**



**16 Threads were used**

To enable multithreading, I used Threadinfo struct to hold all the thread-related information, such as the thread ID, total number of threads, pointers to matrices, and the final result matrix. The multiply\_with\_threads function launches a specified number of threads, each responsible for a portion of the matrix multiplication. The main function reads input matrices, initializes them using the creation functions, and records the execution time using the clock\_gettime function. It then calls the multithreaded matrix multiplication function and writes the result to an output file along with the execution time. The allocated memory for matrices is freed at the end to prevent memory leaks. Through the use of multithreading I saw that matrix multiplication overall performance improved by distributing the workload among multiple threads.

**PART 4**



The key optimization in this code lies in the multiply\_with\_threads characteristic, wherein matrix multiplication is carried out in parallel the use of more than one threads. Each thread is liable for a part of the ensuing matrix, and inside every thread, the computation is in addition elevated the use of SIMD commands. Specifically, the code employs SSE (Streaming SIMD Extensions) commands for int32 matrix multiplication, enhancing mathematics operations efficiency. The multiply characteristic encapsulates the SIMD-primarily based totally computation, making use of (\_mm\_loadu\_si128, \_mm\_mullo\_epi32, \_mm\_add\_epi32, \_mm\_storeu\_si128) for parallel processing. This aggregate of multithreading and SIMD commands complements the general overall performance of matrix multiplication. This method resulted in the overall best speedup of computations as I saw. The computations were done extremely quickly and then written to the file as output.

**PART 5**

In relative terms, Vector Instruction (In C) supplied the most speedup in comparison to different strategies. It indicates that optimizing the matrix multiplication set of rules to take benefit of vector processing units, consisting of AVX, can substantially enhance performance., the general computation time reduced substantially, in particular whilst handling big matrices. However, the combination of vector commands and multithreading (the use of SSE and multithreading together) may offer even better speedups, especially when the workload may be in addition optimized for vectorization.

Therefore, primarily based totally in this analysis, the first-rate of speedup finished through vector commands seems to be high, contributing substantially to overall performance development in matrix multiplication. Additionally, integrating vectorization with multithreading should probably cause even higher results, despite the fact that cautious optimization and tuning are required to make sure that the speedup scales efficiently with increasing processor count.