Assignment 1 – Part 3: GEMM

Introduction

Matrix-matrix multiplication is computationally intensive because it involves a large number of arithmetic operations, often scaling as O(n^3) for naive implementations, and requires efficient memory access patterns to optimize performance. It is widely used in scientific computing, machine learning, computer graphics, and simulations, where large datasets and complex calculations demand high efficiency. To alleviate the processing burden, researchers have developed optimized algorithms, parallel computing techniques using multi-core CPUs and GPUs, and highly tuned libraries like BLAS, which exploit hardware-specific optimizations to accelerate computations while reducing memory bottlenecks.

To perform matrix-matrix multiplication for floating point numbers, most people use BLAS's GEMM (General Matrix-Matrix Multiplication), which has 4 levels of precision – truncated 16-bit, half, single and double.

In this assignment, we will attempt to parallelize and optimize the matrix-matrix multiplication, by using threads and comparing our results with sequential execution of the same as well as with the benchmarked standard libraries of BLAS for GEMM.

Machine Specifications

To attempt this task, it is required to know the hardware of the system on which the optimization will occur in depth, to able to make informed decisions in the later stages. The specifications of my system are given as follows:

Manufacturer	HP
Operating System	Windows 11 (24H2)
Processor	Intel(R) Core (TM) i7-14650HX
GPU	Nvidia GeForce RTX 4060
Clock Speed (Base Speed)	2.2 GHz
Cores	16
Logical Cores	24
L1 Cache Size	1.4 MB
L2 Cache Size	24 MB
L3 Cache Size	30 MB
IDE	Visual Studio 2022
Programming Language	C++
Peak GFLOPs	784[1]

Peak GFLOPs is theoretical, as the most recent documentation for processor efficiency did not include the Processor I have. For this reason, I have chosen the most similar processor I could find, which was Intel® Core™ i5-13600KF Processor (24M Cache, up to 5.10 GHz).

Program

Pseudocode

For Naïve Implementation

To start this assignment, we first ready a pseudocode which can be implemented for the sequential GEMM operations:

```
for (int i = 0; i<size; i++) {
    for (int j = 0; j<size; j++) {
        temp = 0;
        for (int k = 0; k < size; k++) {
            temp += matrix1[i][k] * matrix2[k][j];
        }
        result[i][j] = temp;
    }
}
```

For Optimized Implementation

For parallel implementation of GEMM operations, I have tried optimizing the code by implementing loop unrolling. The pseudocode for the same can be found below:

```
for (i = 0; i<size; i += stepSize) {
    for (j = 0; j<size; j++) {
        temp1 = 0;
        temp2 = 0;
        .
        .
        .
        tempn = 0;
        for (k = 0; k < size; k++) {
            temp1 += matrix1[i][k] * matrix2[k][j];
            temp2 += matrix1[i+1][k] * matrix[k][j];
            .
            .
            .
            tempn += matrix[i+stepSize-1][k] * matrix[k][j];
        }
        result[i][j] = temp1;
        result[i+1][j] = temp2;
        .
        .
        .
        result[i+stepSize-1][j] = tempn;
    }
}
```

Pre-Requisites

In order to start implementing matrix-matrix multiplications, in both sequential and parallel methods, we first need to establish a few things which will help with the analysis of HPC algorithms:

Theoretical Peak Performance

Theoretical peak performance that can be achieved by calculating this formula:

Peak Performance = Q * Clock Speed * NumberOfCores

Where, $Q = f/m$

where Q = theoretical peak performance/cycle; f = number of floating-point operations; m = number of memory references.

By the above pseudocode, it is clear that:

$$f = 2n^3$$
$$m = n^3 + 3n^2$$

Therefore, $\mathbf{Q = 2n^3/(n^3 + 3n^2)}$

Flops

Flops or floating-point instructions per second are the number of floating-point instructions the computer is performing per second. This can be calculated by:

FLOPs = total number of floating-point instructions/total time taken

We can calculate from the pseudo code that the total number of floating-point instructions executed for any size n, is $2n^2$.

So, the calculation can be modified as:

FLOPs = $\mathbf{2n^3}$**/Total Time Taken**

Residual

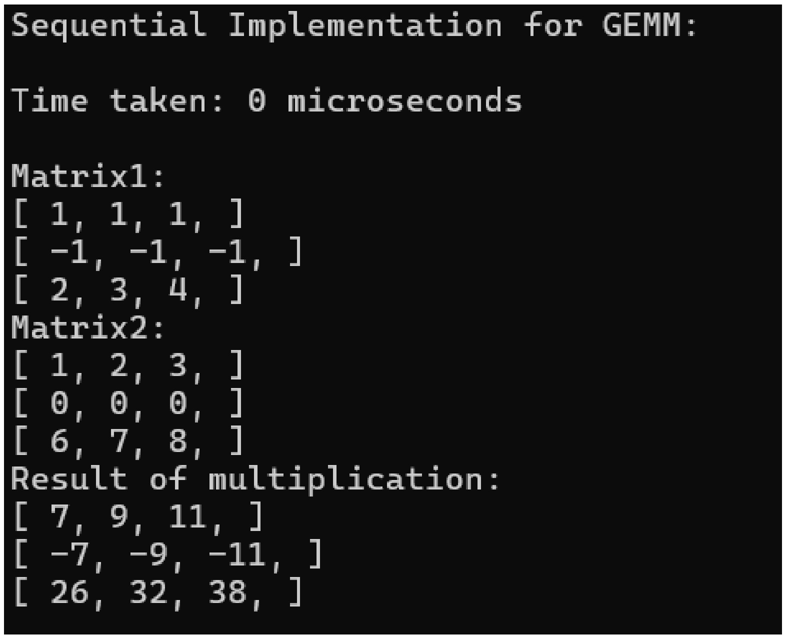
This is the difference from your solution to the known solution. In this assignment, I have calculated the residual for the resulting vector with the following formula:

$$\textbf{Residual} = \frac{\sum\left|\left|\boldsymbol{My\ Solution}[\boldsymbol{i}][\boldsymbol{j}] - \boldsymbol{BLAS\ Solution}[\boldsymbol{i}][\boldsymbol{j}]\right|\right|}{\boldsymbol{SizeOfMatrix} * \boldsymbol{SizeOfMatrix}}$$

Running Code in Different Environments

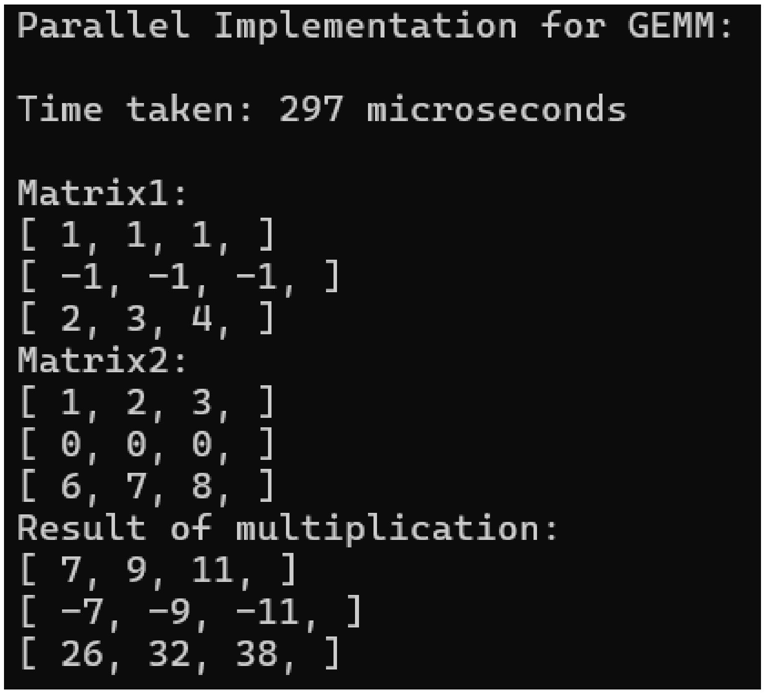
We first check the correctness of the code by using known matrices and checking if the result matches.

Sequential Double Precision Data Type



```
Sequential Implementation for GEMM:

Time taken: 0 microseconds

Matrix1:
[ 1, 1, 1, ]
[ -1, -1, -1, ]
[ 2, 3, 4, ]
Matrix2:
[ 1, 2, 3, ]
[ 0, 0, 0, ]
[ 6, 7, 8, ]
Result of multiplication:
[ 7, 9, 11, ]
[ -7, -9, -11, ]
[ 26, 32, 38, ]
```

Parallel Double Precision Data Type

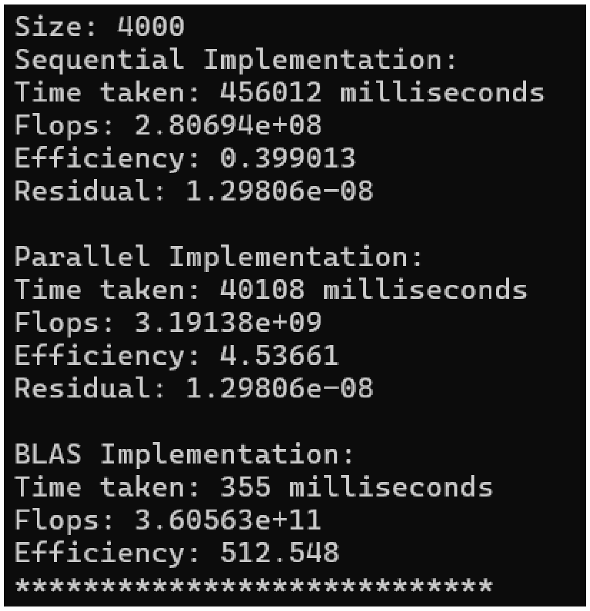


```
Parallel Implementation for GEMM:

Time taken: 297 microseconds

Matrix1:
[ 1, 1, 1, ]
[ -1, -1, -1, ]
[ 2, 3, 4, ]
Matrix2:
[ 1, 2, 3, ]
[ 0, 0, 0, ]
[ 6, 7, 8, ]
Result of multiplication:
[ 7, 9, 11, ]
[ -7, -9, -11, ]
[ 26, 32, 38, ]
```

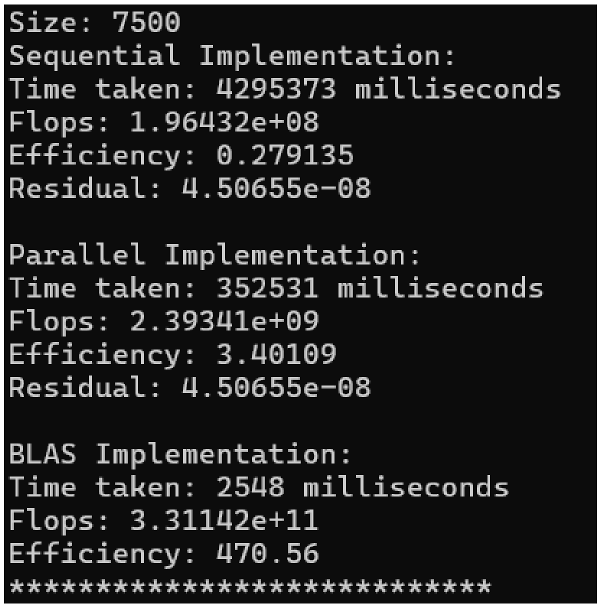
Implementation on Actual Data

Now that the correctness is checked, we can implement the code on a randomly generated matrix and a vector of differing sizes.

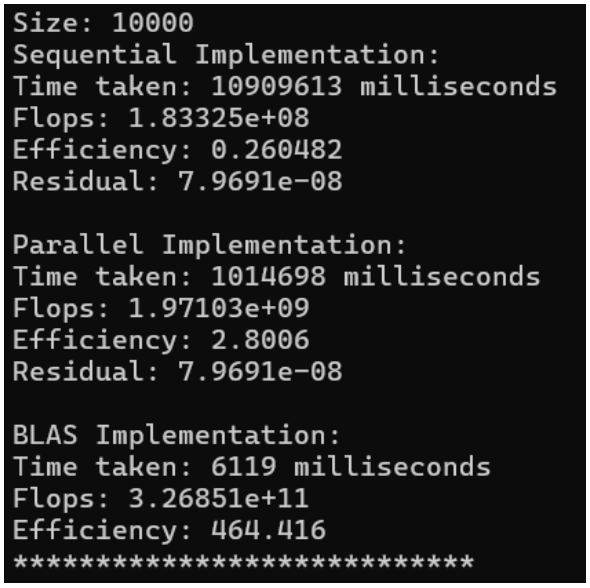
Some implementations are shown below:



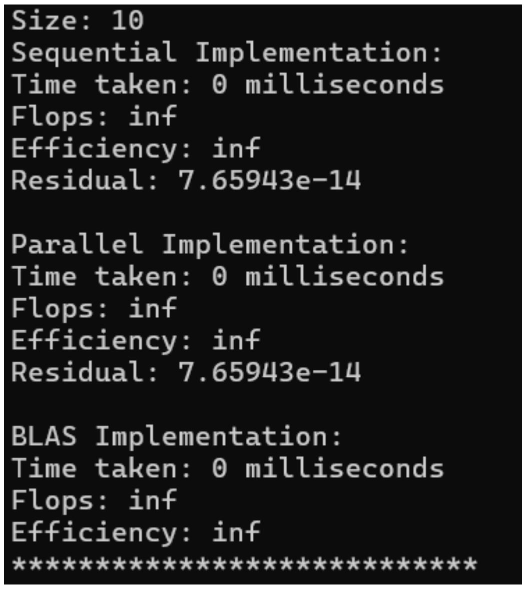
```
Size: 4000
Sequential Implementation:
Time taken: 456012 milliseconds
Flops: 2.80694e+08
Efficiency: 0.399013
Residual: 1.29806e-08

Parallel Implementation:
Time taken: 40108 milliseconds
Flops: 3.19138e+09
Efficiency: 4.53661
Residual: 1.29806e-08

BLAS Implementation:
Time taken: 355 milliseconds
Flops: 3.60563e+11
Efficiency: 512.548
****************************
```



```
Size: 7500
Sequential Implementation:
Time taken: 4295373 milliseconds
Flops: 1.96432e+08
Efficiency: 0.279135
Residual: 4.50655e-08

Parallel Implementation:
Time taken: 352531 milliseconds
Flops: 2.39341e+09
Efficiency: 3.40109
Residual: 4.50655e-08

BLAS Implementation:
Time taken: 2548 milliseconds
Flops: 3.31142e+11
Efficiency: 470.56
****************************
```



```
Size: 10000
Sequential Implementation:
Time taken: 10909613 milliseconds
Flops: 1.83325e+08
Efficiency: 0.260482
Residual: 7.9691e-08

Parallel Implementation:
Time taken: 1014698 milliseconds
Flops: 1.97103e+09
Efficiency: 2.8006
Residual: 7.9691e-08

BLAS Implementation:
Time taken: 6119 milliseconds
Flops: 3.26851e+11
Efficiency: 464.416
****************************
```



```
Size: 10
Sequential Implementation:
Time taken: 0 milliseconds
Flops: inf
Efficiency: inf
Residual: 7.65943e-14

Parallel Implementation:
Time taken: 0 milliseconds
Flops: inf
Efficiency: inf
Residual: 7.65943e-14

BLAS Implementation:
Time taken: 0 milliseconds
Flops: inf
Efficiency: inf
****************************
```

Comparison between Different Implementations

A more granular comparison of different implementations can be seen in the attached excel workbook. Here, we will talk about general trends and interesting points we can find from the data.

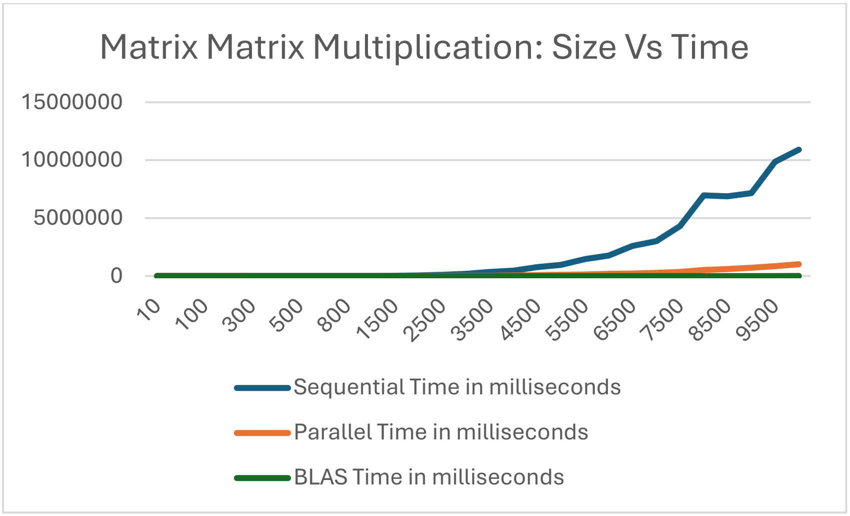
Also, some of the data points had to be ignored, as their values were nearing infinity, and that would have made the plot increasing unstable, and no real patterns could have been determined.

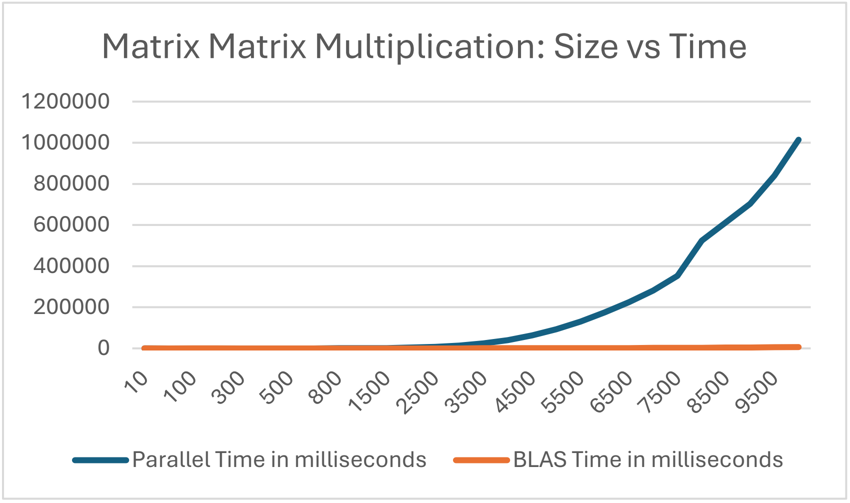
All time comparisons are in milliseconds.

Time Comparison

We can clearly see that sequential implementation has performed the worst out of all three methods of execution, followed by parallel and then BLAS.

As it seemed that the parallel implementation was almost as good as the BLAS implementation, a new graph was plotted between the two. This made it clear that the parallel implementation, although better than the sequential implementation, performed much worse than the BLAS method.

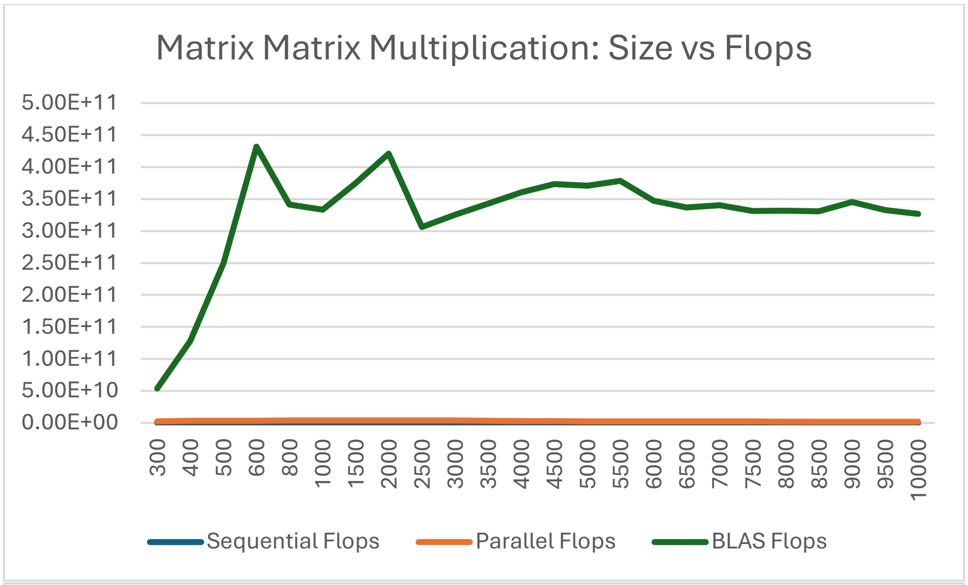


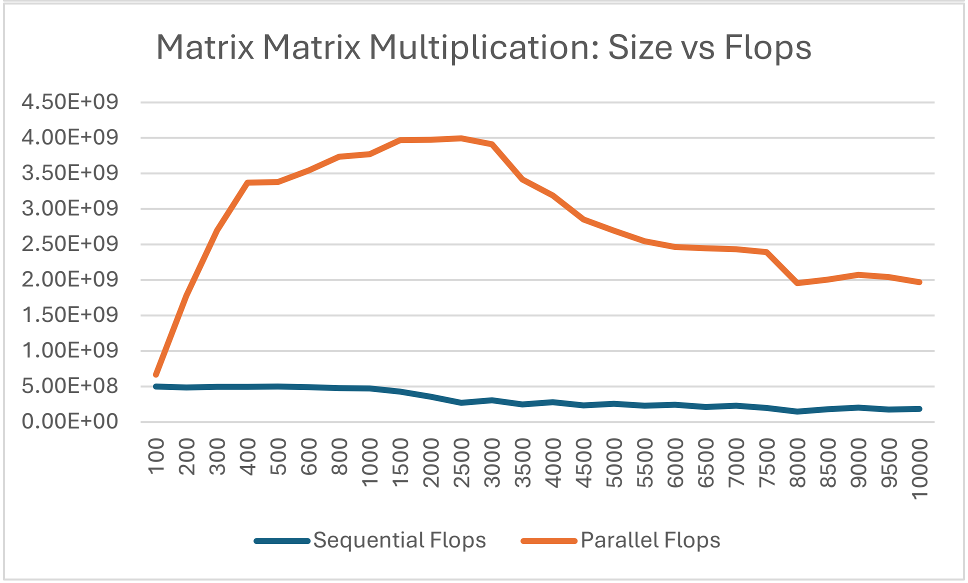


Flops Comparison

BLAS implementation was clearly the best in all three methods of performing GEMM, followed by parallel implementation and then sequential.

Although this is as expected, it is interesting to note that the while the graph for BLAS stabilizes after a while, the graph for the other two gradually decreases as the size of the matrix increases. This is probably due to the fact that, as BLAS implementation optimizes on the basis of the underlying hardware, as the size of matrix increases, a peak is reached in terms of BLAS's algorithm variables with respect to the machine's hardware.

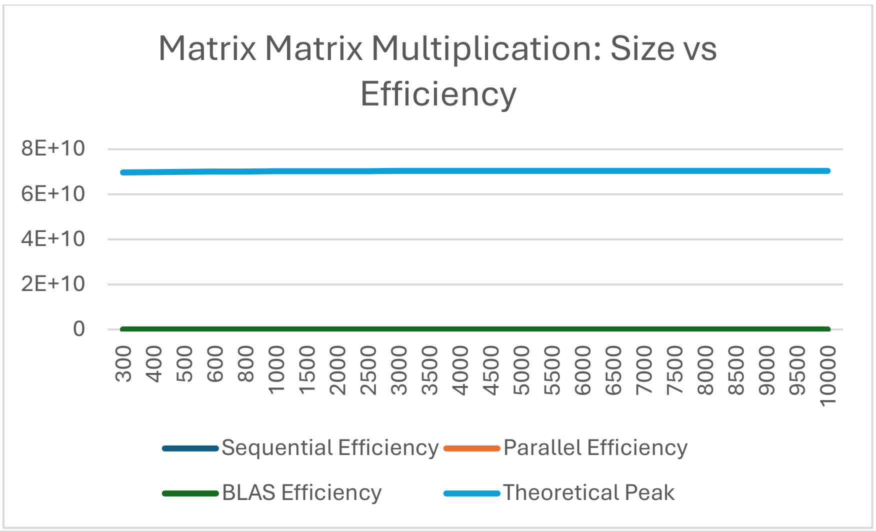


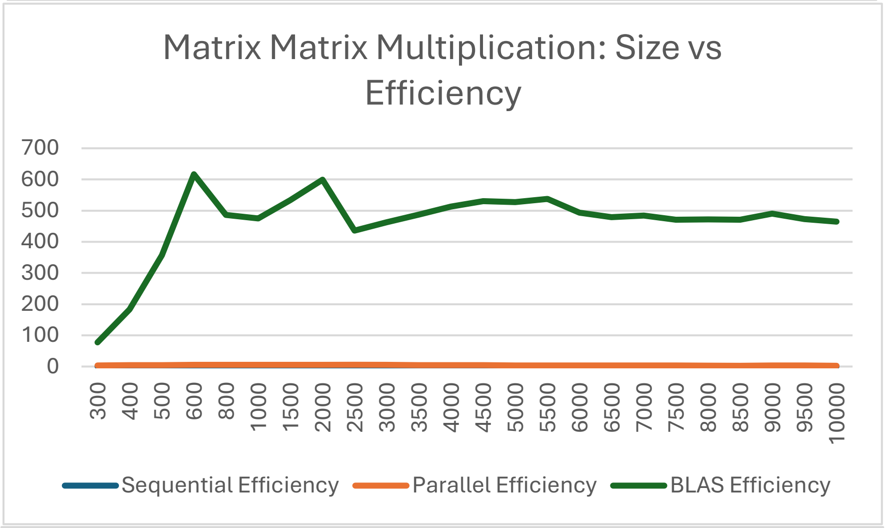


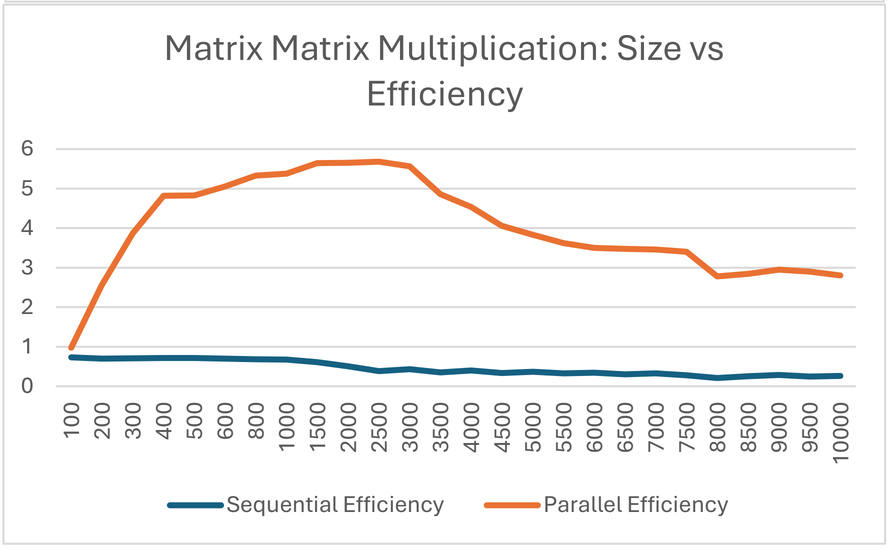
Efficiency Comparison

It is interesting to note that even BLAS is much worse than the theoretical peak performance for GEMM, even more so than in the case of GEMV.

The graphs between BLAS, parallel and sequential implementations follow the same trends as the Flops graphs.



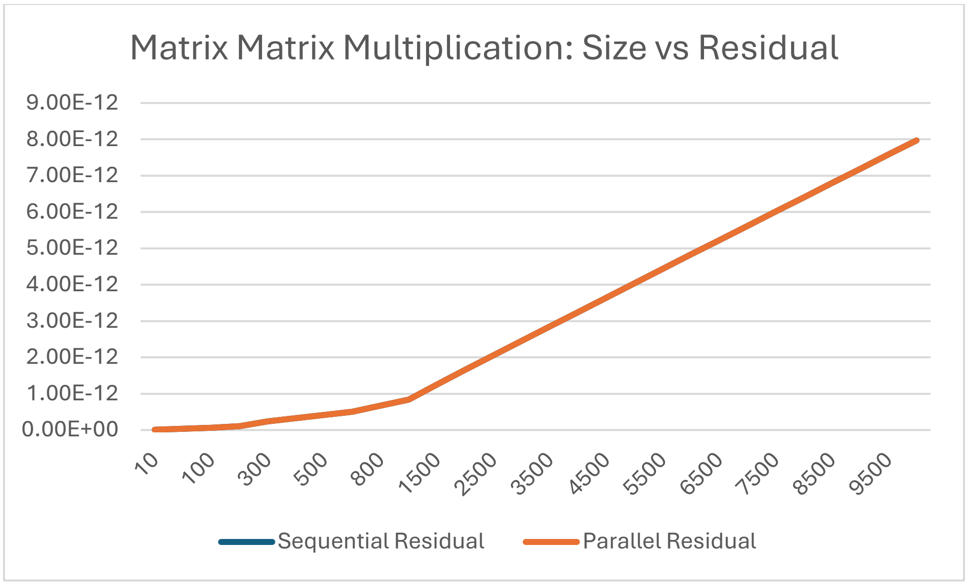




Residual Comparison

As seen in the case of GEMV, in GEMM as well the residual increases as the size of the matrix increases, as expected.

Also, as we can see the values for the sequential and parallel implementation are the same, pointing to the correctness of the implementation.



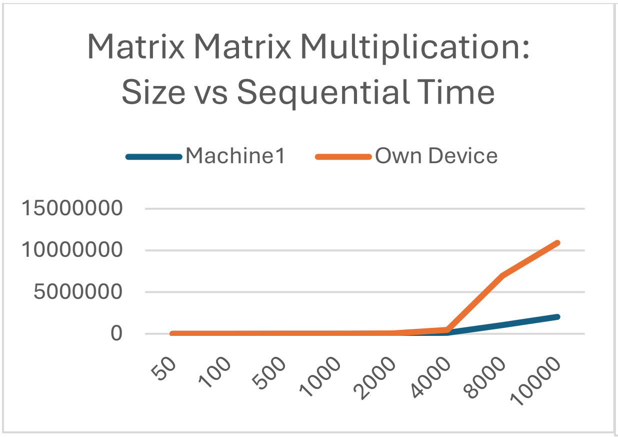
Comparison of Implementation Against Other Machines

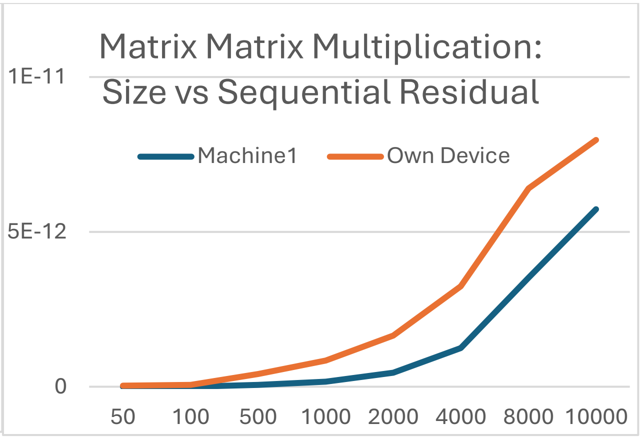
For all these comparisons, I have only talked about the graphs. For granular data points, please refer to the attached Excel workbook.

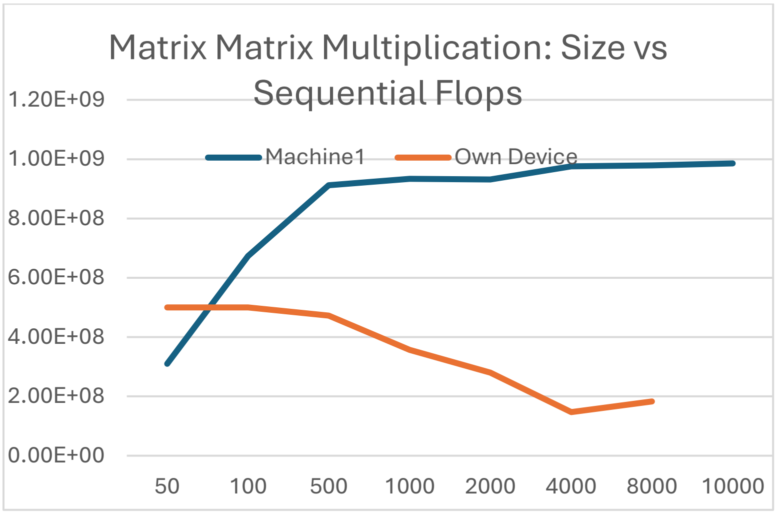
Machine 1

Manufacturer	Apple
Operating System	MacOS
Processor	M3 Pro
Clock Speed (Base Speed)	3.5 GHz
Cores	11 (5 Performance Cores + 6 Efficiency Cores)
Logical Cores	24
IDE	Visual Studio Code
Programming Language	C++ - Version 11
Peak GFLOPs	376

Sequential Implementation Comparison





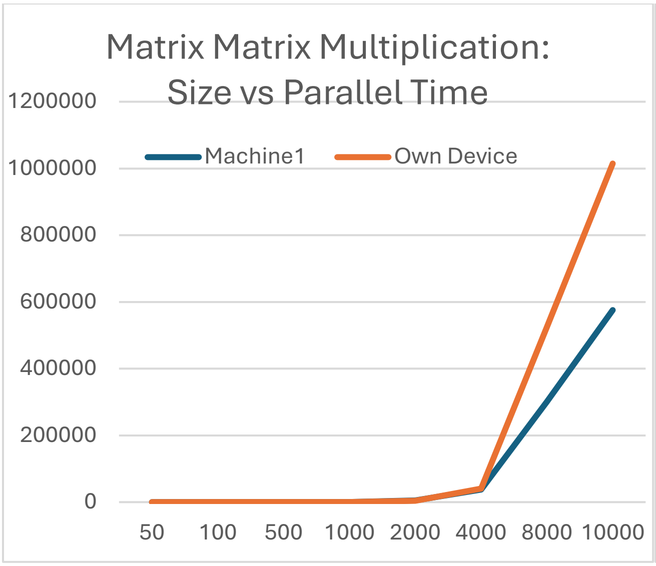


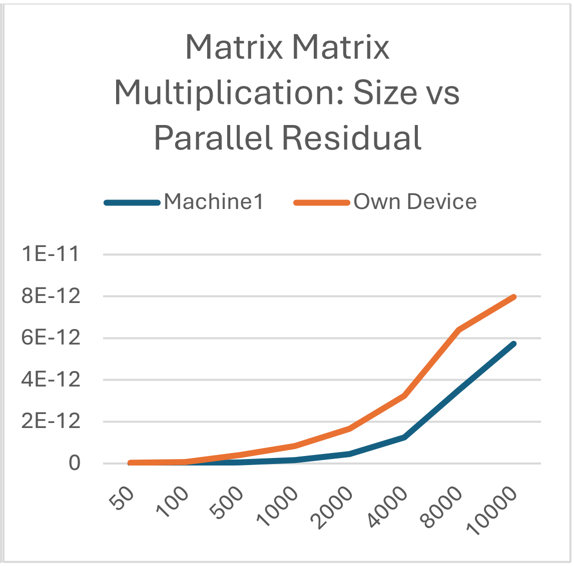
Some interesting points to note:

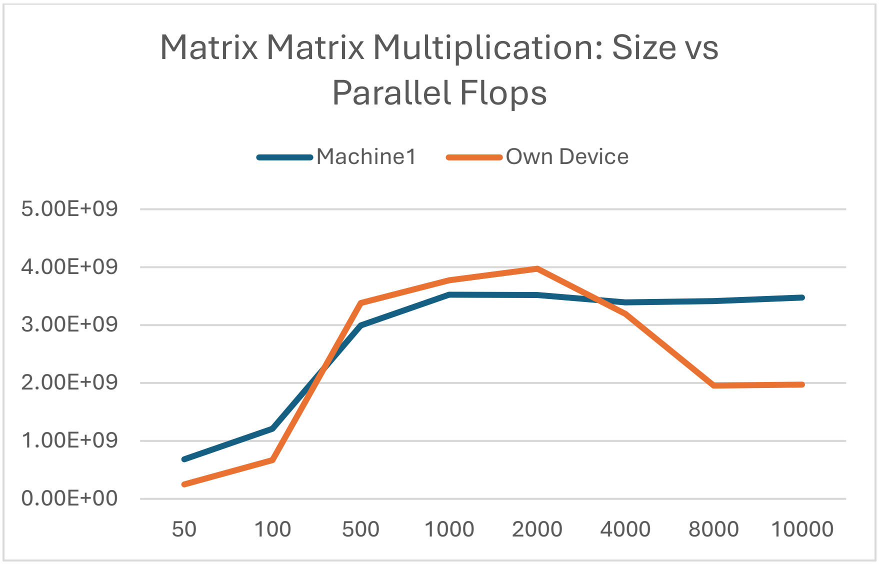
1. Machine 1 takes more time to execute especially as the size of the matrix increases, while the residual is constantly less than my own device. It is possible that one of the reasons why the sequential execution for Machine1 is more, because it is more accurate than my own device's execution.
2. Surprisingly, flops for Machine 1 is higher than my device, even when Machine 1 takes longer to execute. This may point to differences in calculation of the value of flops.

Parallel Implementation Comparison

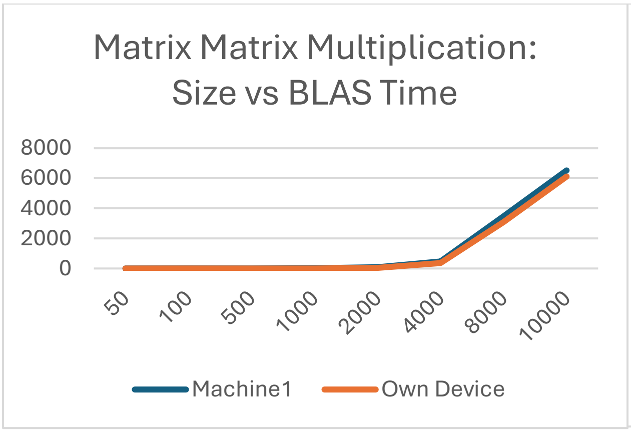
Looking at the plots, we see that the time taken graph, and the residual graph are similar as was the case for Sequential implementation. However, interestingly, while my device's Flops calculation decreased with increase in size of matrix, Machine 1's Flops mostly stabilizes after 1500.

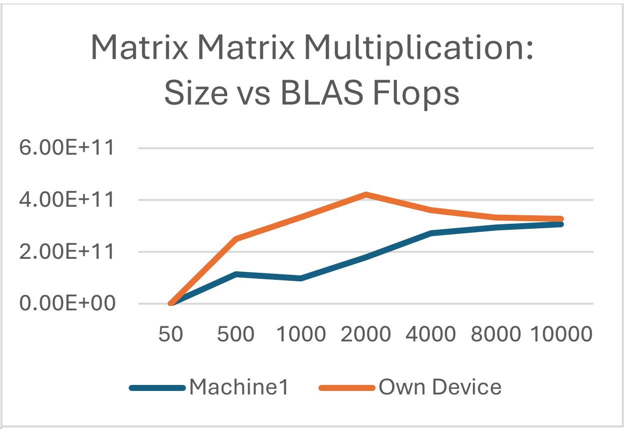






BLAS Implementation Comparison





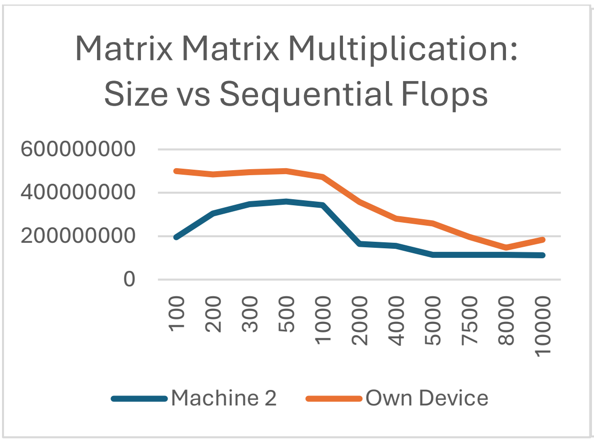
Similar to in case of GEMV, BLAS performs better in Machine1 than in my own device in terms of time taken. However, once again, my device's Flops performance is better than Machine1's, indicating differences in calculating Flops.

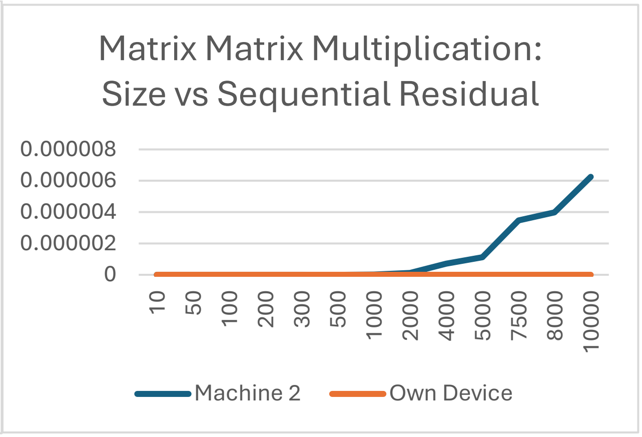
Machine 2

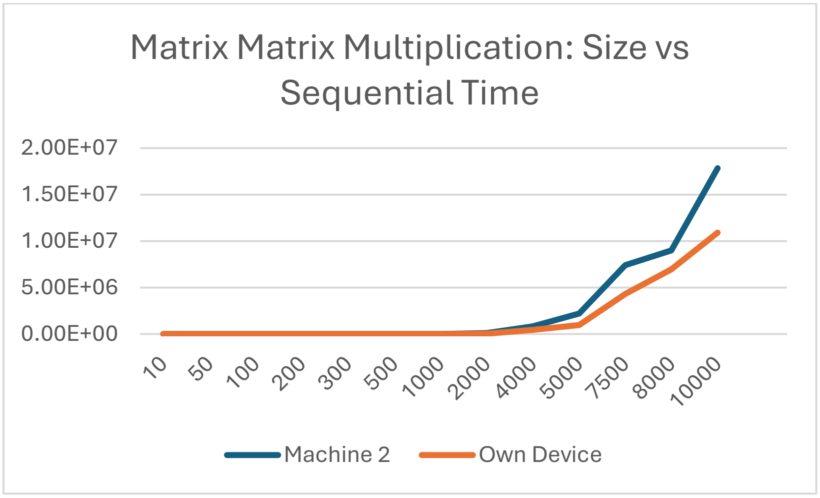
Manufacturer	Apple
Operating System	MacOS
Processor	M1
Cores	8
Programming Language	C++

Sequential Implementation Comparison

From the graphs, we can clearly see that sequential implementation for my device trumped Machine2 in all three graphs – time taken by my machine was less than Machine2, therefore, flops was higher in my machine, and lastly, the residual calculated was much less in my machine than Machine2.



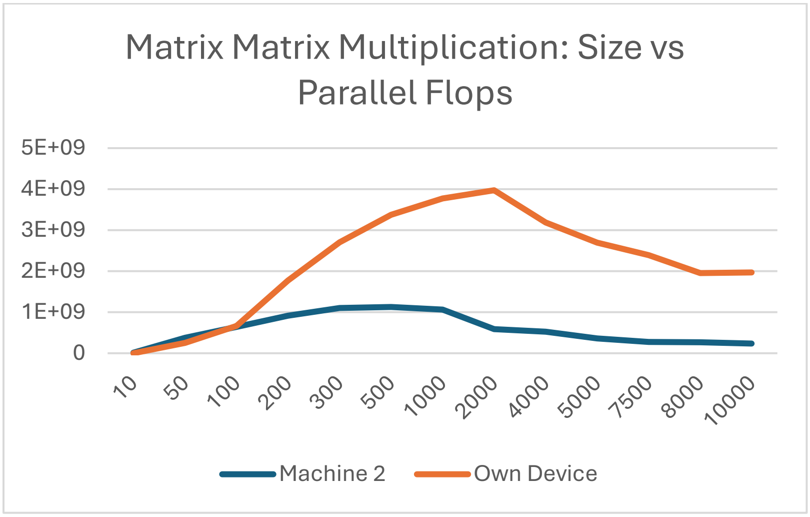


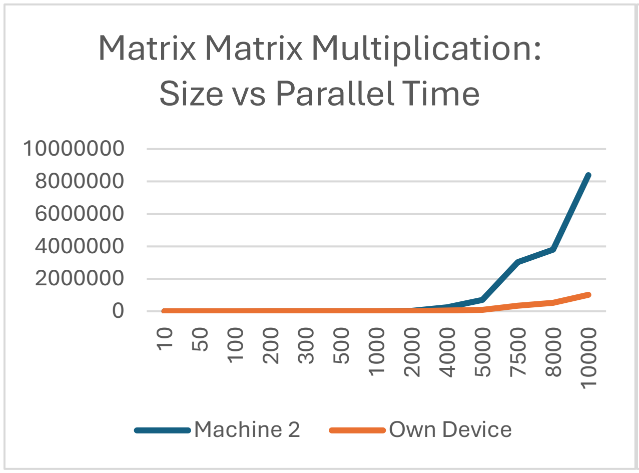


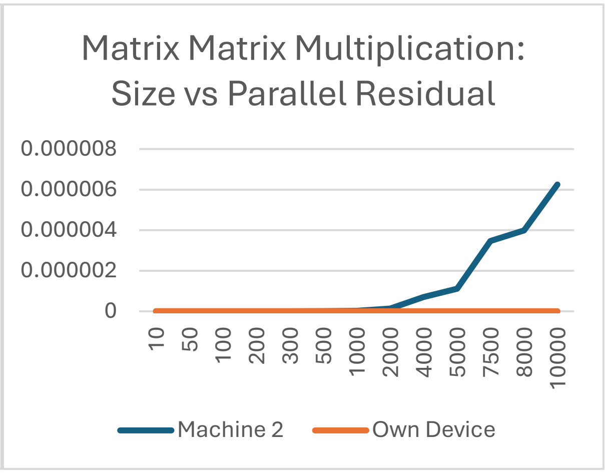
Parallel Implementation Comparison

Comparison of the parallel implementation of GEMM reveals similar trends as sequential implementation, only my parallel implementation performs better than Machine2's parallel implementation. This is likely due to the loop unrolling optimization added to the parallel code.

One interesting thing to note is that Machine2's residual slope and time taken slope look very similar, even when the units may be different. We can say that there is a chance for correlation between residual and time taken, although small.

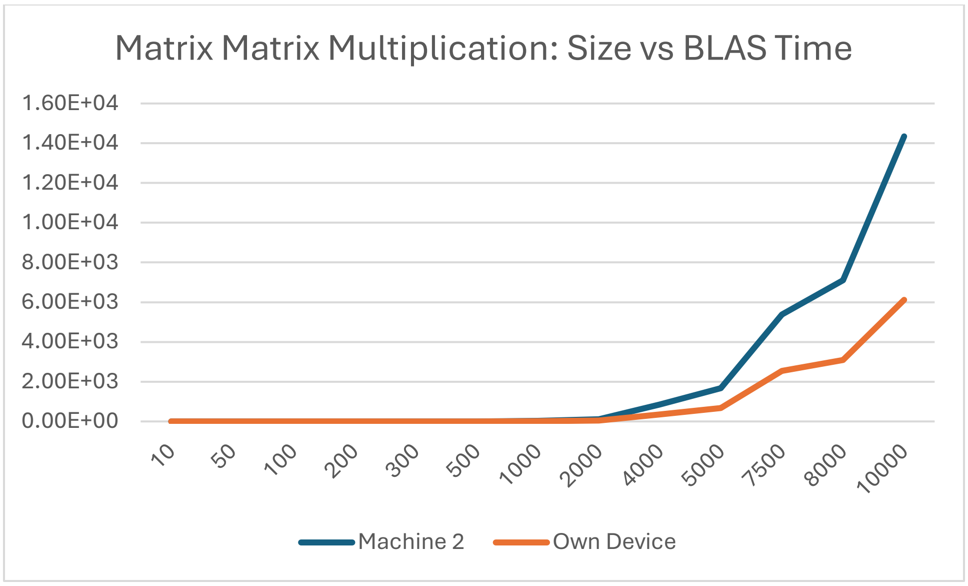


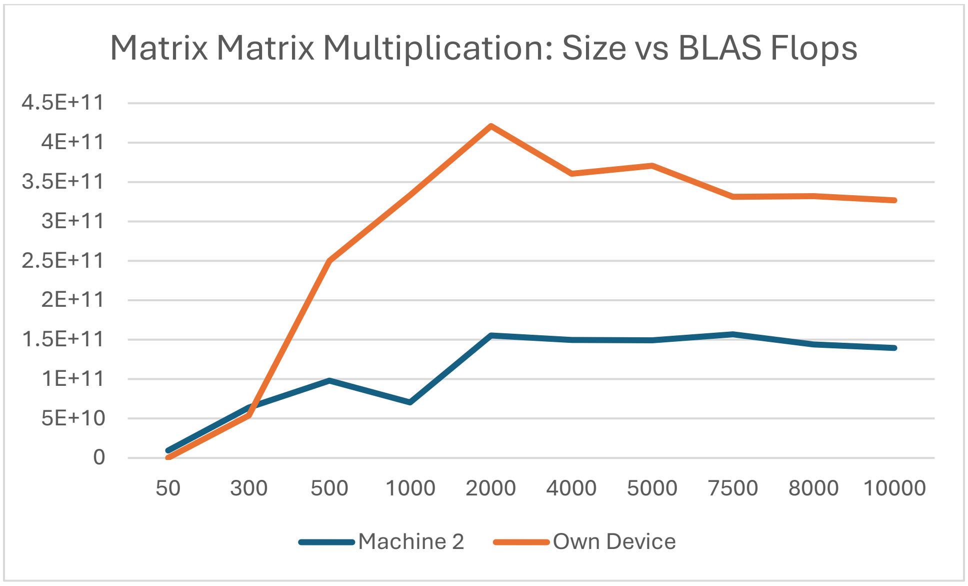




BLAS Implementation Comparison

As seen in above comparisons, BLAS implementation for my device performs better than Machine2, in terms of both time taken and flops. These graphs indicate that my device is more powerful than Machine2, for performing GEMM operations.





References

[1] "APP Metrics for Intel® Microprocessors - Intel® Core^{TM} Processor," Intel. Accessed: Feb. 04, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/841556/app-metrics-for-intel-microprocessors-intel-core-processor.html