Assignment 2: GEMV

Introduction

Matrix calculations are an integral part of scientific computing and even in high level processing of data in fields such as research, artificial intelligence and even medicine. However, matrix calculations are time and CPU intensive tasks, especially in the case of highly precise, floating-point operations. To alleviate the stress on the machines, many people have designed numerous algorithms and ways to maximize the use of the cache available in our own machine, and to optimize floating point operations by parallelizing it with respect to the number of cores that is present on the hardware. This led to the advent of a new subject in Computer Science called High Performance Computing.

To perform matrix-vector multiplication for floating point numbers, most people use BLAS's GEMV (General Matrix Vector 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-vector 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 GEMV.

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

To start this assignment, we first ready a pseudocode which can be implemented for both sequential and parallel GEMV operations:

```
for row in range(1, sizeOfMatrix):
        temp = 0
        for column in range(1, sizeOfMatrix):
                temp += vector[column] * matrix[row][column]
        result[row] = temp
```

Pre-Requisites

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

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:

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

Therefore, $Q = 2n^2/(3 + n^2)$

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 = $2n^2$/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:

$$\textbf{Residual} = \frac{||\boldsymbol{My\ Solution - BLAS\ Solution}||}{\boldsymbol{SizeOfMatrix}}$$

Running Code in Different Environments

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

Sequential Single Precision Data Type (Float)

```
PS D:\College\CSS 535 - High Performance Computing\Homework\Assignment2> ./NaiveGemv_Float_Correctness.exe
Time taken: 0 microseconds
Matrix:
[ 1, 2, 3, ]
[ 0, 0, 0, ]
[ 6, 7, 8, ]
Vector:
[ 1, 2, 3, ]
Result of multiplication:
[ 14, 0, 44, ]
```

Sequential Double Precision Data Type

```
PS D:\College\CSS 535 - High Performance Computing\Homework> ./NaiveGemv_Double_Correctness.exe
Time taken: 0 microseconds
Matrix:
[ 1, 2, 3, ]
[ 0, 0, 0, ]
[ 6, 7, 8, ]
Vector:
[ 1, 2, 3, ]
Result of multiplication:
[ 14, 0, 44, ]
```

Parallel Double Precision Data Type

```
Size: 3
Time taken: 0 milliseconds
Residual: 0
****************************

Matrix:
[ 1.0000000000000000, 2.0000000000000000, 3.0000000000000000, ]
[ 0.0000000000000000, 0.0000000000000000, 0.0000000000000000, ]
[ 6.0000000000000000, 7.0000000000000000, 8.0000000000000000, ]
Vector:
[ 1.0000000000000000, 2.0000000000000000, 3.0000000000000000, ]
Blas Vector:
[ 14.0000000000000000, 0.0000000000000000, 44.0000000000000000, ]
Result of multiplication:
[ 14.0000000000000000, 0.0000000000000000, 44.0000000000000000, ]
```

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 (all these are double precision implementations):

```
Size: 4000
Sequential Implementation:
Time taken: 64118 microseconds
Flops: 4.9908e+08
Residual: 3.47499e-12

Parallel Implementation:
Time taken: 27992 microseconds
Flops: 1.14318e+09
Residual: 3.47499e-12

BLAS Implementation:
Time taken: 1970 microseconds
Flops: 1.62437e+10
**************************
```

```
Size: 7500
Sequential Implementation:
Time taken: 228314 microseconds
Flops: 4.92742e+08
Residual: 6.62436e-12

Parallel Implementation:
Time taken: 59241 microseconds
Flops: 1.89902e+09
Residual: 6.62436e-12

BLAS Implementation:
Time taken: 7500 microseconds
Flops: 1.5e+10
**************************
```

```
Size: 10000
Sequential Implementation:
Time taken: 402310 microseconds
Flops: 4.97129e+08
Residual: 8.85016e-12

Parallel Implementation:
Time taken: 84581 microseconds
Flops: 2.3646e+09
Residual: 8.85016e-12

BLAS Implementation:
Time taken: 11031 microseconds
Flops: 1.81307e+10
**************************
```

```
Size: 50
Sequential Implementation:
Time taken: 11 microseconds
Flops: 4.54545e+08
Residual: 4.44977e-14

Parallel Implementation:
Time taken: 501 microseconds
Flops: 9.98004e+06
Residual: 4.44977e-14

BLAS Implementation:
Time taken: 4 microseconds
Flops: 1.25e+09
**************************
```

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.

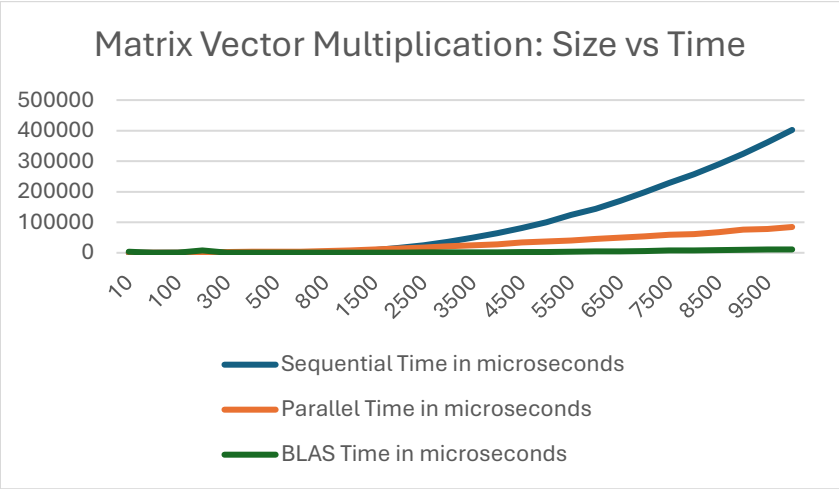
Time Comparison

As we can see from the below graph, as expected, sequential implementation has an exponential graph when compared to the parallel or BLAS implementation.

Some interesting things to note:

1. When n was small, sequential implementation of GEMV was faster than the parallel implementation, sometimes even beating the BLAS implementation, but as the size of the matrix grew, sequential implementation started taking more time.

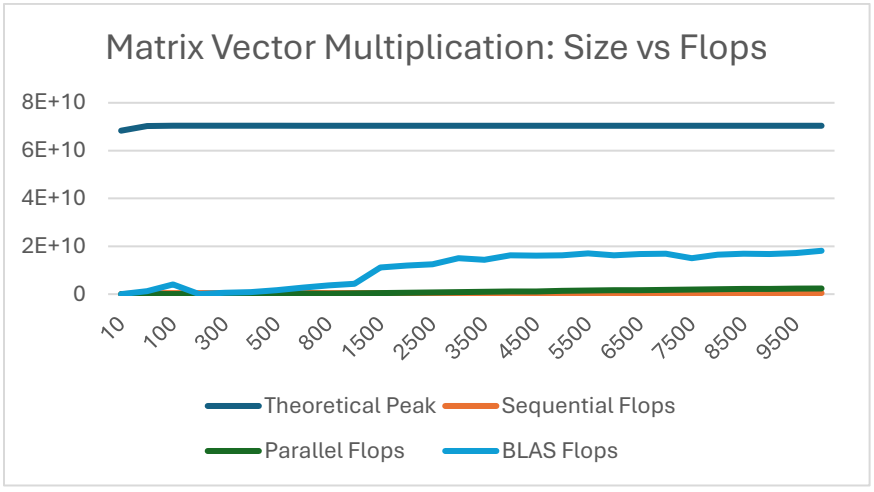
2. Even after multiple runs, BLAS implementation for sizes between 10 and 200 takes more time than expected. This may be because BLAS is more commonly optimized for larger sizes of matrices, and the optimization done by BLAS may be counterintuitive for smaller sizes.

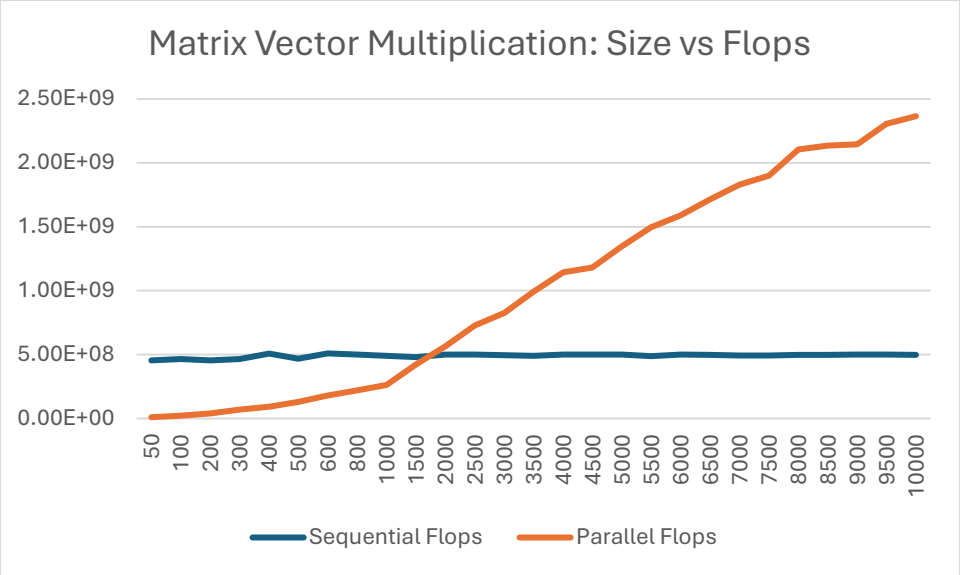


Flops Comparison

As per the graph below, we find that sequential implementation performed the worst, followed by parallel and then BLAS. Some interesting things to note about this graph are:

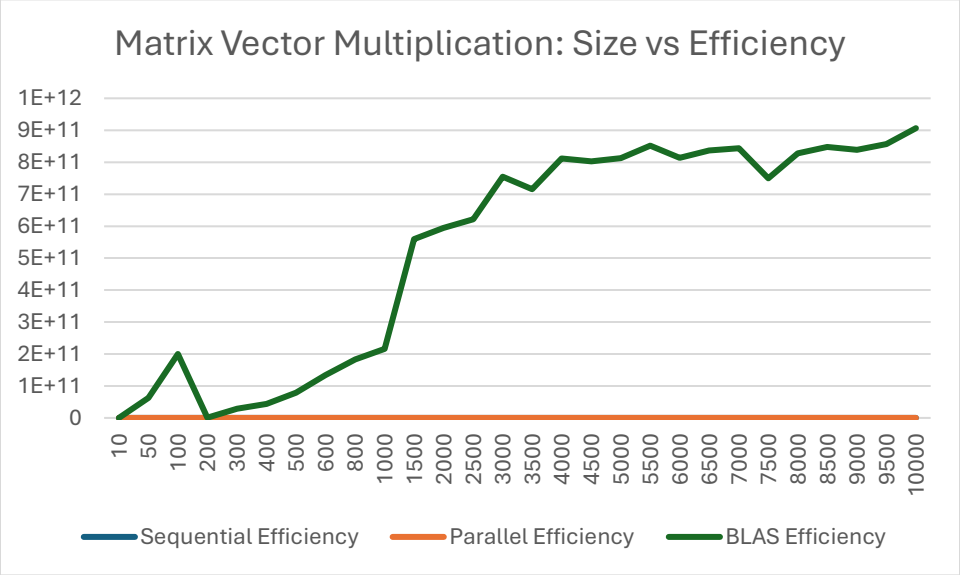
1. We see that even though BLAS is the industry standard in performing matrix multiplication, it is still much worse than the theoretical peak performance which can be achieved by the CPU. This is probably due to the memory latency that comes with doing any computational task.
2. We can also observe the points stated in the earlier section.
3. On removing the BLAS implementation from the graph, we can clearly see that sequential implementation has the same Flops no matter the size of the matrix, while the Flops for the parallel implementation increases with size.



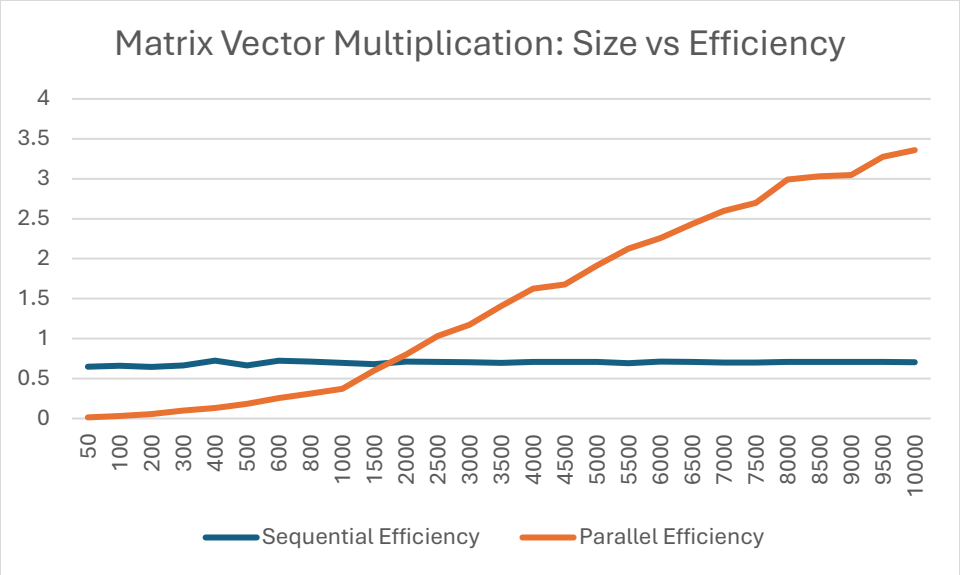


Efficiency Comparison

As we can see from the graph, even though our parallel implementation performed almost the same as the BLAS implementation in terms of time taken, the efficiency for the parallel implementation is much worse, similar to the sequential implementation.



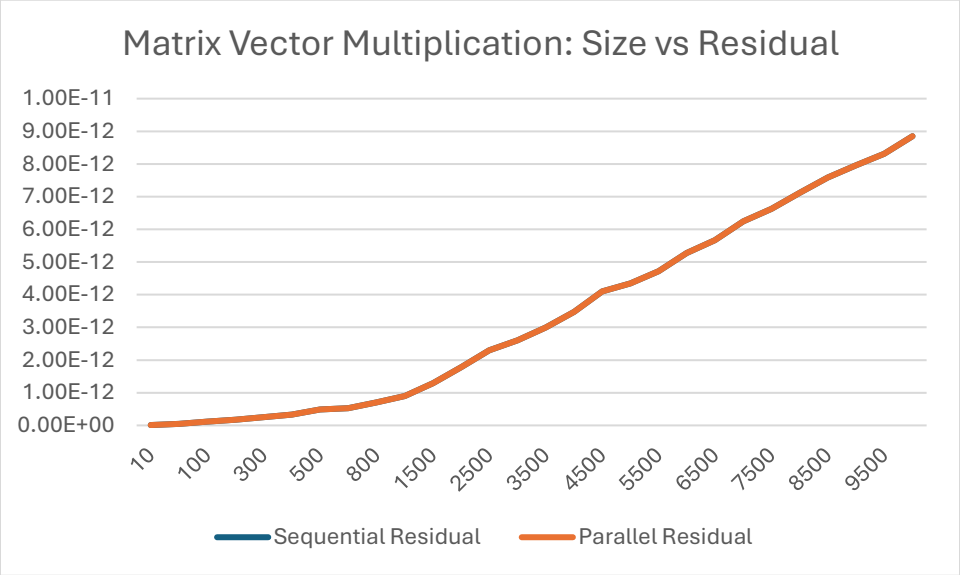
Removing the BLAS efficiency, we can better compare the efficiency of our sequential and parallel implementation, as shown below. We see that while the efficiency of the sequential implementation remains the same, no matter the size of the matrix, the efficiency of the parallel implementation increases with size.



Residual Comparison

The residual generated from both sequential and parallel implementations are identical. This can be considered another sign of correctness between the two implementations.

Additionally, we observe that the residual gradually increases as the size of matrix increases.



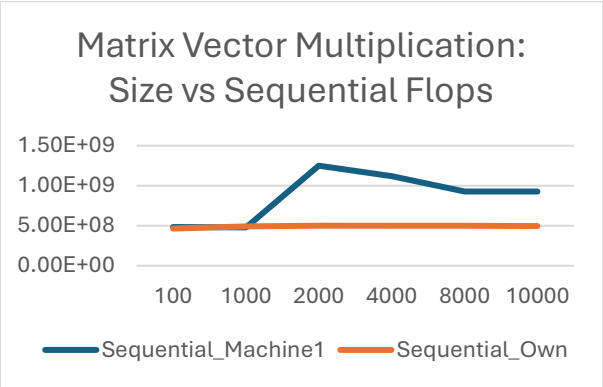
Comparison of Implementation Against Other Machines

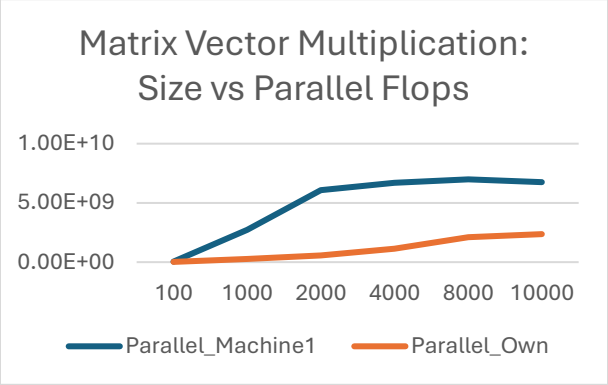
For all these comparisons, I have only talked about the graphs. For granular data points, please refer to the attached Excel workbook.

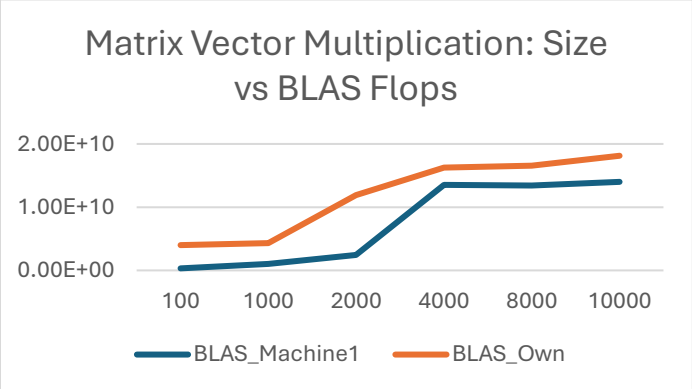
Machine 1

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

The comparison with the data received from Machine 1 can be graphically seen as follows:







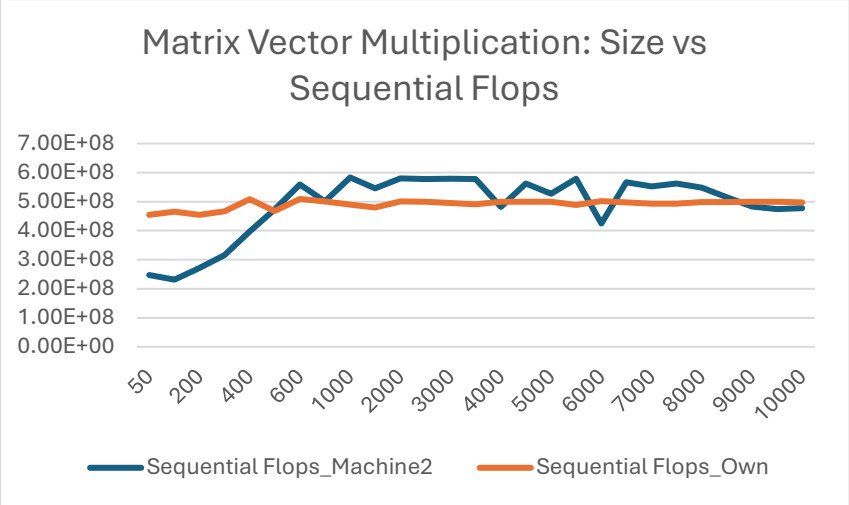
From the above data, we can see that on average, Machine 1 performed better than my own machine. A lot of factors can contribute to this, including the implementation of the code on this machine, the architecture of the processor, the internal implementation of code, etc.

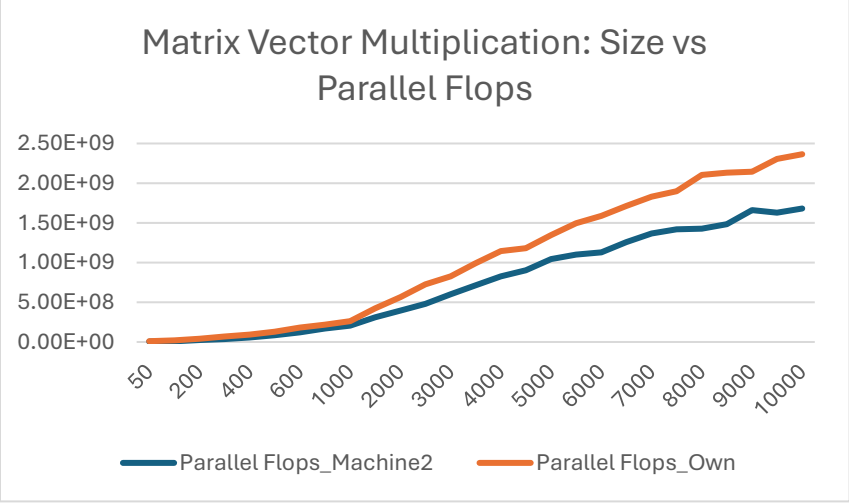
One interesting thing to note is that even though my own implementation of code performed worse than Machine1's code, BLAS implementation of my machine performed better than Machine1's. This may be due to the fact that BLAS may not be well supported in Apple systems.

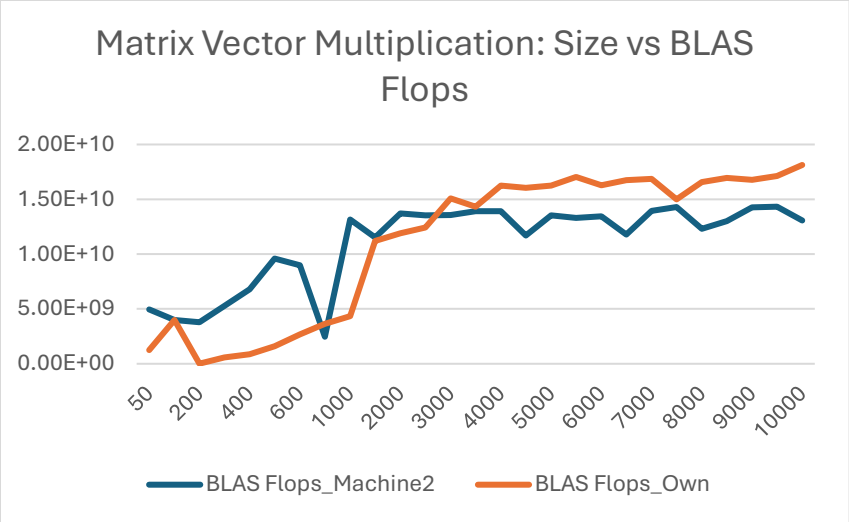
Machine 2

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

The comparison with the data received from Machine 2 can be graphically seen as follows:







In this case, the code run on Machine 2 is the same as my machine. From these graphs, we can note the following:

1. While my machine's implementation had an overall static flops value for sequential implementation, Machine 2's flops changed as per size. This might be due to the internal execution of instructions via the processor.
2. My parallel execution of GEMV fared better than Machine 2's parallel execution.
3. In case of BLAS execution, my execution for GEMV fared worse when the size of the matrix was small, but increased after a while, as size increased. This could be due to the fact that BLAS implementation is more optimized for larger size of the matrix and it can more efficiently use the memory.