Application of Multithreading on BigInteger Matrices

Tamer Bannourah

January 2020

Table of Contents

Abstract

This paper goes over an analysis of applying multithreading to see performance improvements on matrices of Java BigIntegers. The multithreading used was Java's newWorkStealingPool which was added in v1.8. The matrices were squares and consisted of sizes between 100 and 2000 elements that were random values of 8, 16, 32, 64, 512- and 1024-bit numbers. Multiplication and addition operations were tested on the matrices in single thread and multithreaded instances. The average percent change from single thread to multithread time is used to measure performance. The results for multiplication produced an average percent change of -74% over the single threaded runs. The results for addition produced less favorable results of an average of 214% change over the single threaded runs. This seemed strange until a modified version of the multithreaded addition was tested and produced an average -2% change. This led to multiple insights on how to properly use multithreading and when to, or not, to use it.

Introduction

While trying to figure out what to do over winter break, there was a comment on a stackoverflow.com post that talked about multithreading matrix multiplication. This idea was taken and expand upon it by making the matrices of type BigInteger. The BigInteger class was used to add more meat to the processing time to help observe differences. The bit sizes that were used were 8, 16, 32, 64, 512- and 1024-bit numbers. The concept was expanded even more when it was decided to test the four basic math operations on matrices as well. Multithreading is one of many techniques in computer science that allows scientist increase productivity and efficiency of computer systems. Most central processing units (CPU) today have multiple cores and some CPUs allow for the ability of simultaneous multithreading (SMT), where one core allows for two logical processors.

Regarding the operations tested, division of two matrices is just multiplication of two matrices with the second matrix being the inverse of itself. Doing subtraction would have produced similar results to addition so matrix subtraction was omitted. This left multiplication and addition in the testing process. To do the actual multithreading, the ExecutorService interface was used, along with the newWorkStealingPool object. The newWorkStealingPool was added in Java v1.8 to allow for parallelism at the level which the system allows for. Once these things were decided, the process could begin.

Procedure

PC Specs

The entire project was executed on a PC that was running Windows 10 version 20H2, OS build 19041.685, with Eclipse IDE Version 2020-09 for Java. The computer specs are as follows:

Intel Core i7-3770K 4-cores 8-threads @ 3.50 GHz base clock and 3.90 GHz turbo clock

ASUS P8H77-V motherboard

16GB DDR3 RAM @ 1333MHz

Samsung Evo 250GB SSD (Boot Drive)

Seagate 2TB HDD @ 7.2K RPM

Java Classes

MatrixMult.java

The first class that was written is ‘MatrixMult.java’. This housed the code to conduct the matrix multiplication. The object’s fields contain three integers for the height and width, for the matrices, and the bit length of the BigIntegers that they will be filled with. The fields also included three 2D BigInteger arrays. These are, ‘matrix1’, ‘matrix2’ and ‘answer’.

When the constructor is called. It takes only one set of integers for the dimensions of both matrices. It will input the dimensions as n*m for matrix1 and m*n for matrix2. This results in an answer matrix of size n*n.

The next function that is called is the ‘fillMatrcies’ function. This fills the matrices with random BigInteger values with a size of bitLength. The bitLength’th bit of the value will always be one while the lesser significant bits will be random.

The next two functions conduct the multiplication. ‘multiplySingleThread’ does exactly what it says. It does a single threaded multiplication operation on matrix1 and matrix2 and places the answers in the ‘answer’ matrix. The ‘multiplyMultithread’ function uses an inner class above it called ‘dotProduct’ which implements the Runnable interface. The multithreaded fuction call starts off by creating an executor service pool with the newWorkStealingPool call. This creates 8 threads for the multiplication to since the CPU has 8 logical cores. Every element in the answer matrix is pushed as a task to the pool to do the calculation and place the answer in that space. Once the main thread pushes all the tasks to the pool, the pool shutdown is called, and the main thread is blocked awaiting termination of the pool before it can exit the entire function call.

There are extra function that were used during the debugging process which allowed for printing the matrices, filling them will defined numbers and clearing the answer matrix.

MatrixAdd.java

The next class that was written supported the ability to add two matrices. MatrixAdd.java and MatrixMult.java are nearly identical except for a few differences. When the constructor is called, both matrix1 and matrix2 are given dimensions of n*m since matrix addition requires them to be. The answer matrix also has those dimensions.

The addSingleThread function goes through the matrices and adds all the elements and puts the answer in the answer matrix. The addMultithread function call does the multithreaded version of addition. It also uses a newWorkStealing pool and an inner class that extends the runnable interface to have each thread do one addition calculation. The main thread shutsdown the pool when the tasks are submitted and then awaits termination. The addMultiThreadModi function does the same thing but instead a task per element, it submits a task per row. The same debug functions exist in the MatrixAdd.java file.

mainMatrix.java

The last class contains the main function for the project. It also contains five functions that call and time their respective operation/thread case. These are multST, multMT, addST, addMT, and addMTModi. The addMTModi function is part of a modification that was made after the results for the addition were unexpected. These functions return the time it took to execute the operation in milliseconds. The actual main function creates an output file to push the results to, sets the output to the file and then runs a loop of how ever many executions are wanted.

Output

Both operations were done on square matrices. Multiplication was done with units of 100 to 1000 incrementing by 100. Addition was done with units of 100 to 2000 incrementing by 100. In each loop iteration, a new object instance is called for the operation in question, the matrices are filled, and the operation is executed. The time is converted to seconds and is pushed to the output file with the size of the matrices. The name of the output file must be changed manually. Once all the data was acquired, it was compiled into a useful format using Excel.

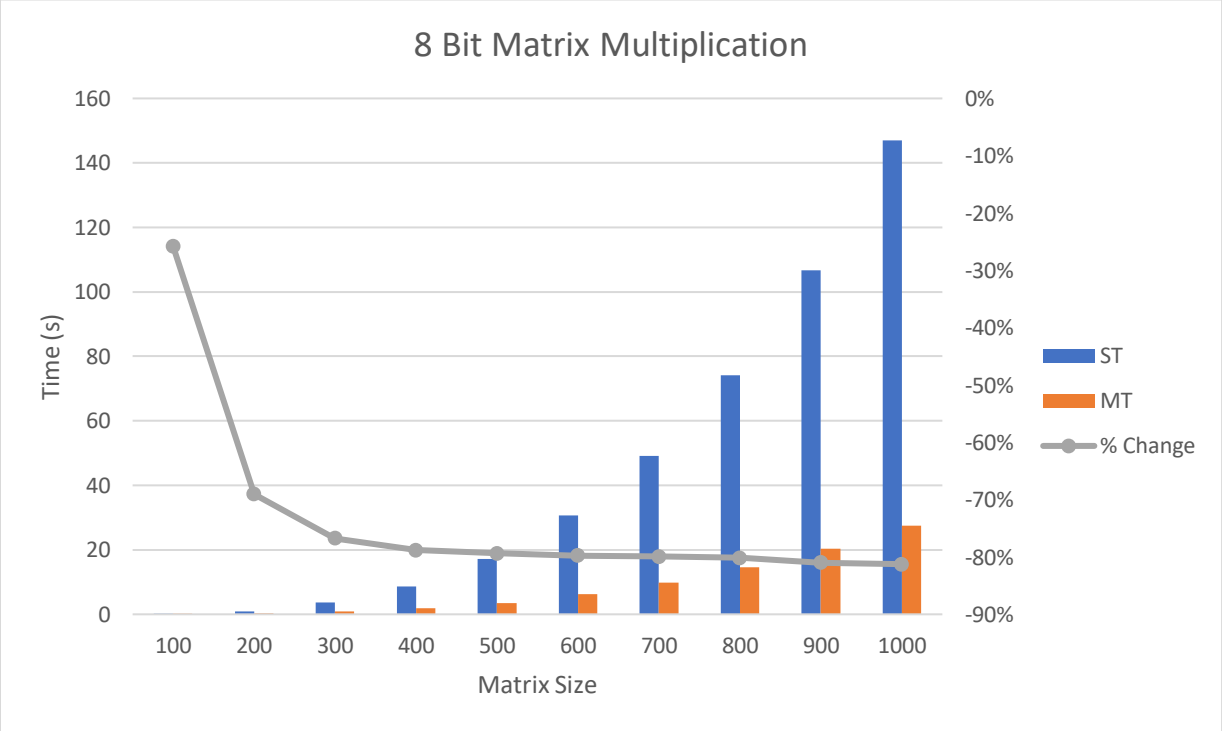
Results

The results are formatted with single threaded data in blue, multithreaded data in orange and percent change in gray. The percent change is measured from single thread time to multithread time.

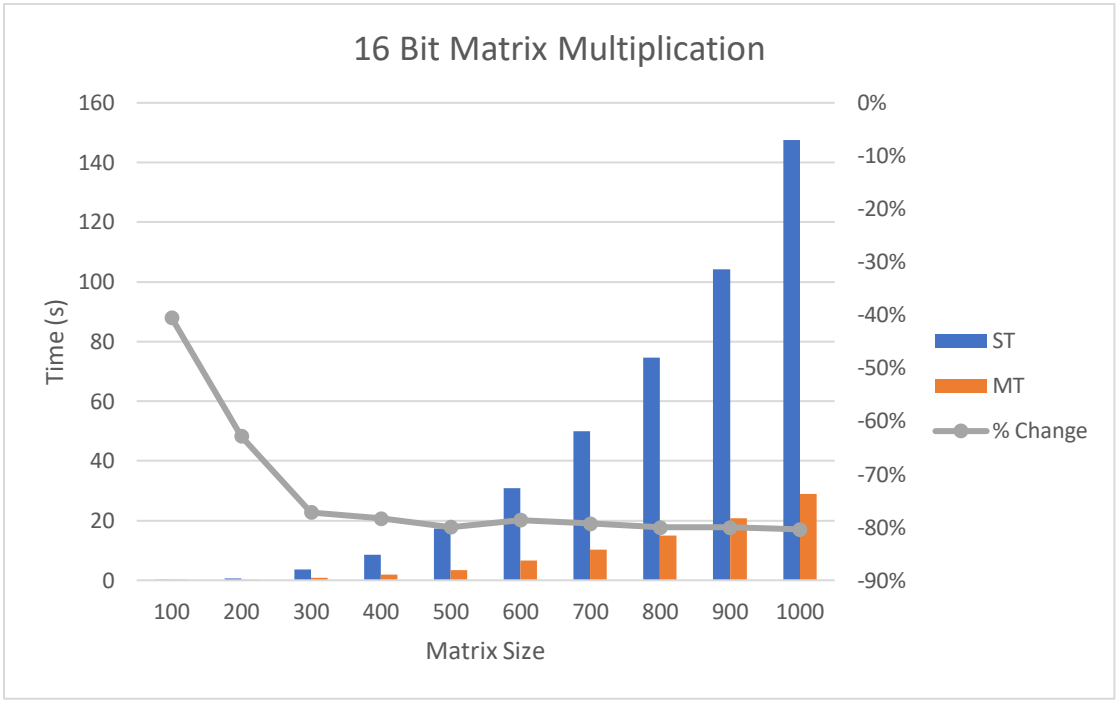
Multiplication

The results of the multiplication comparison produced very favorable results. A specific trend is noticed among every data set in this case which is that when the size of the matrices reached 300 or 400, the percent change levels out to about -80%. The total average percent change was -74%. The 1024-bit tests had the largest average percent change with -75% while the 32-bit tests had the smallest with -72%.

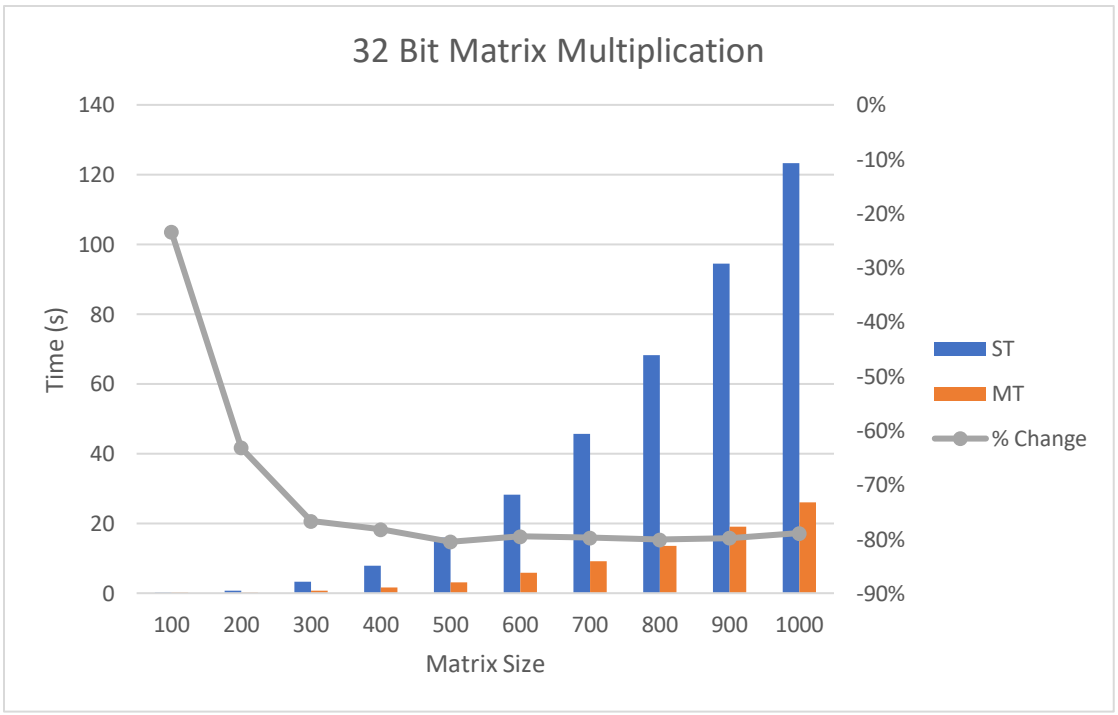
8-bit Average Percent Change: -74%



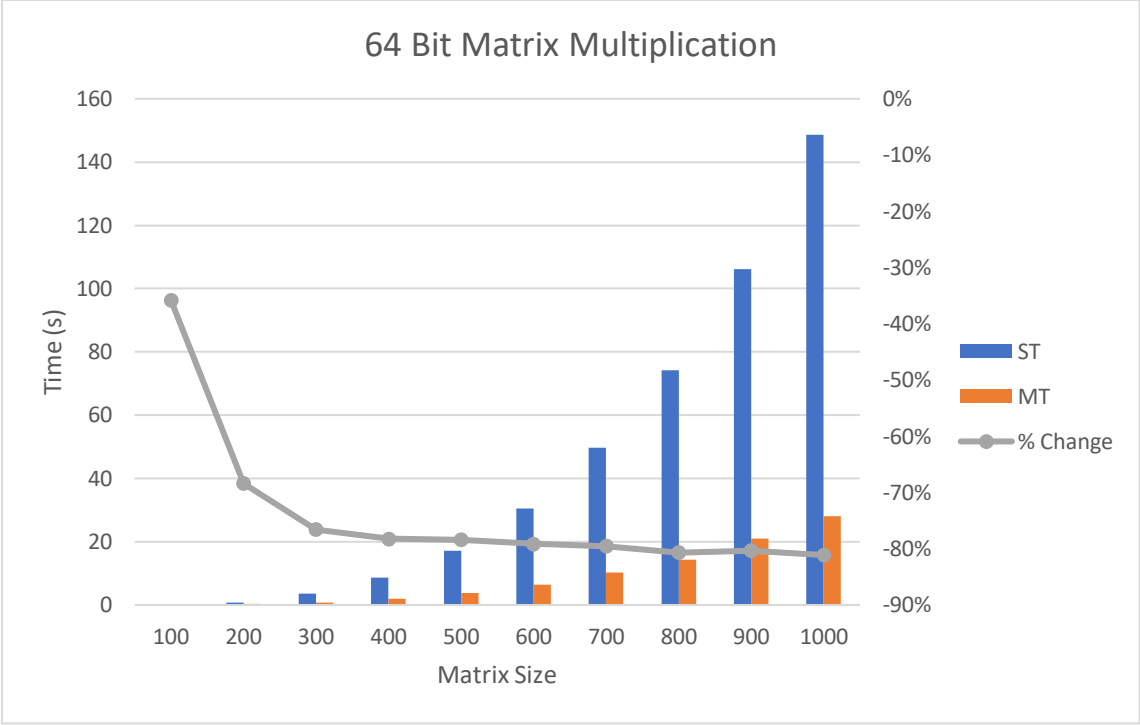
16-bit Average Percent Change: -74%



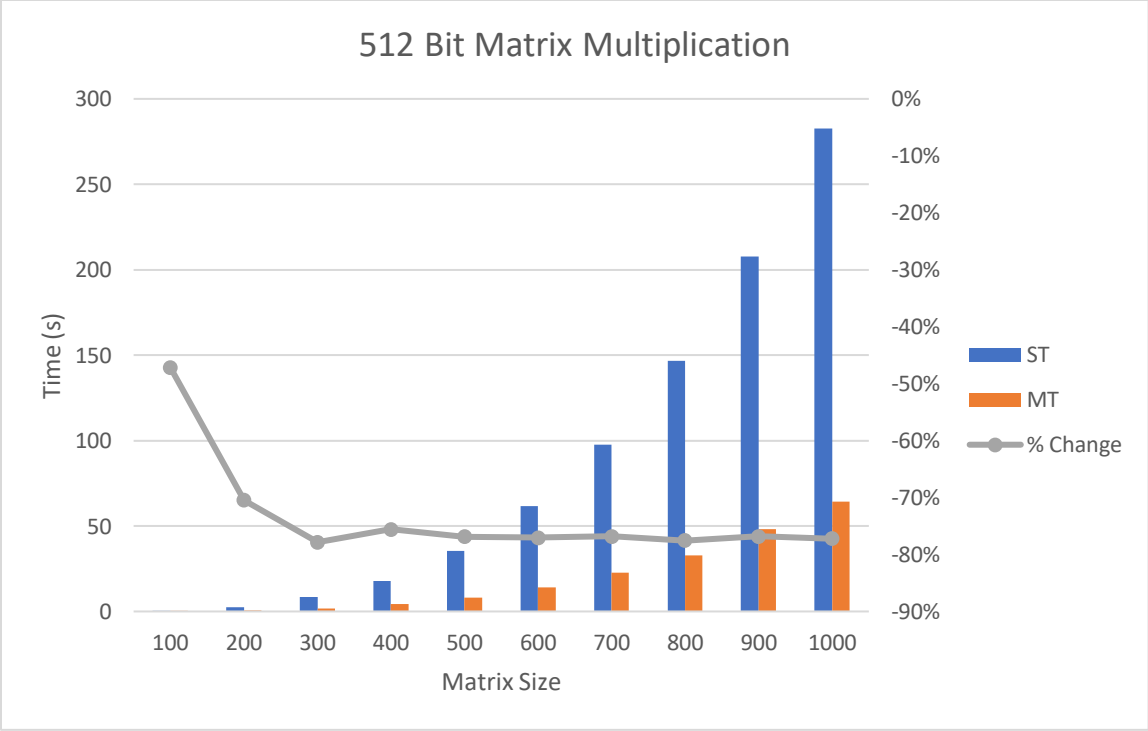
32-bit Average Percent Change: -72%



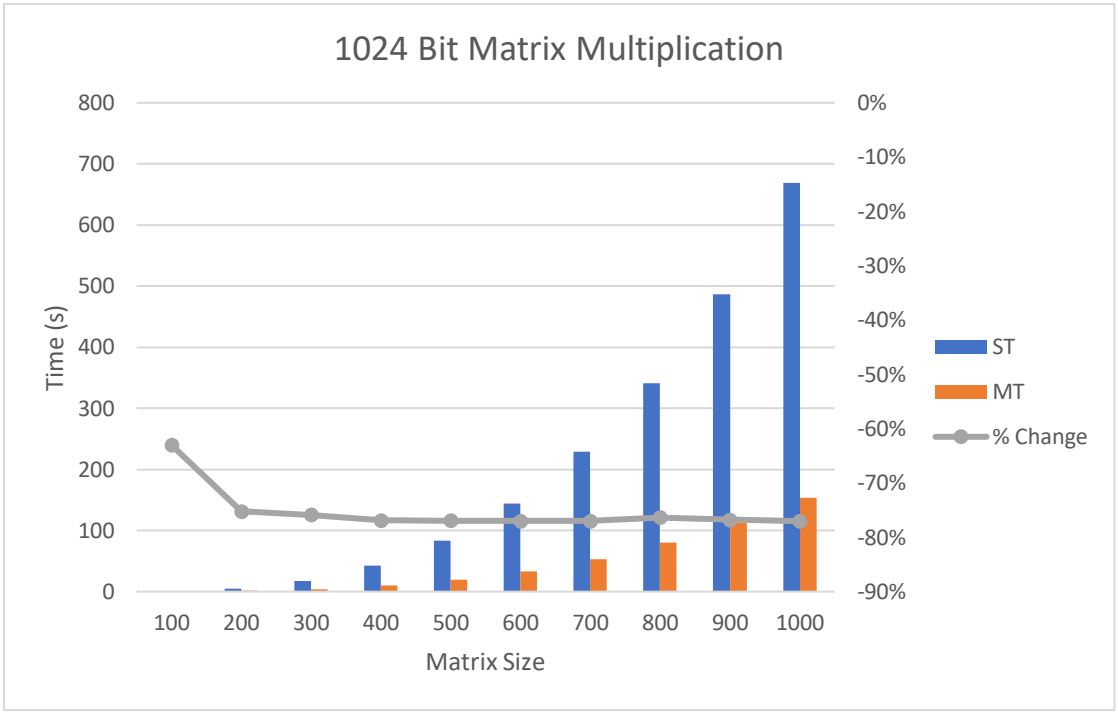
64-bit Average Percent Change: -74%

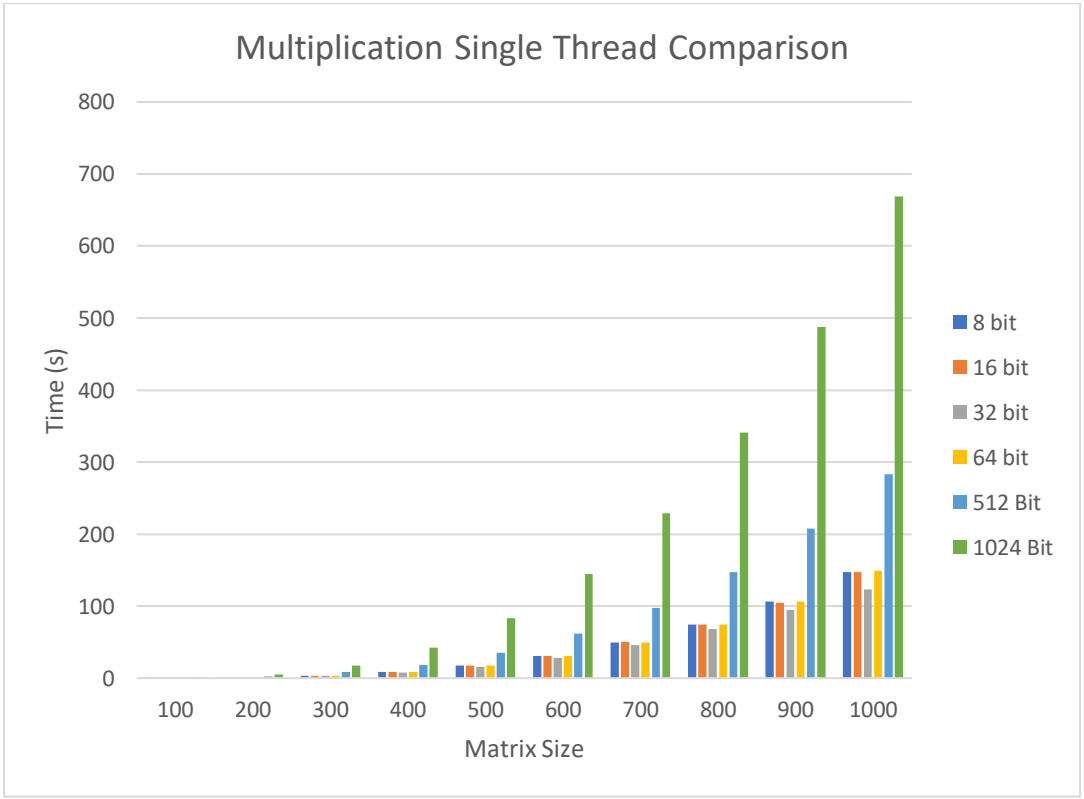


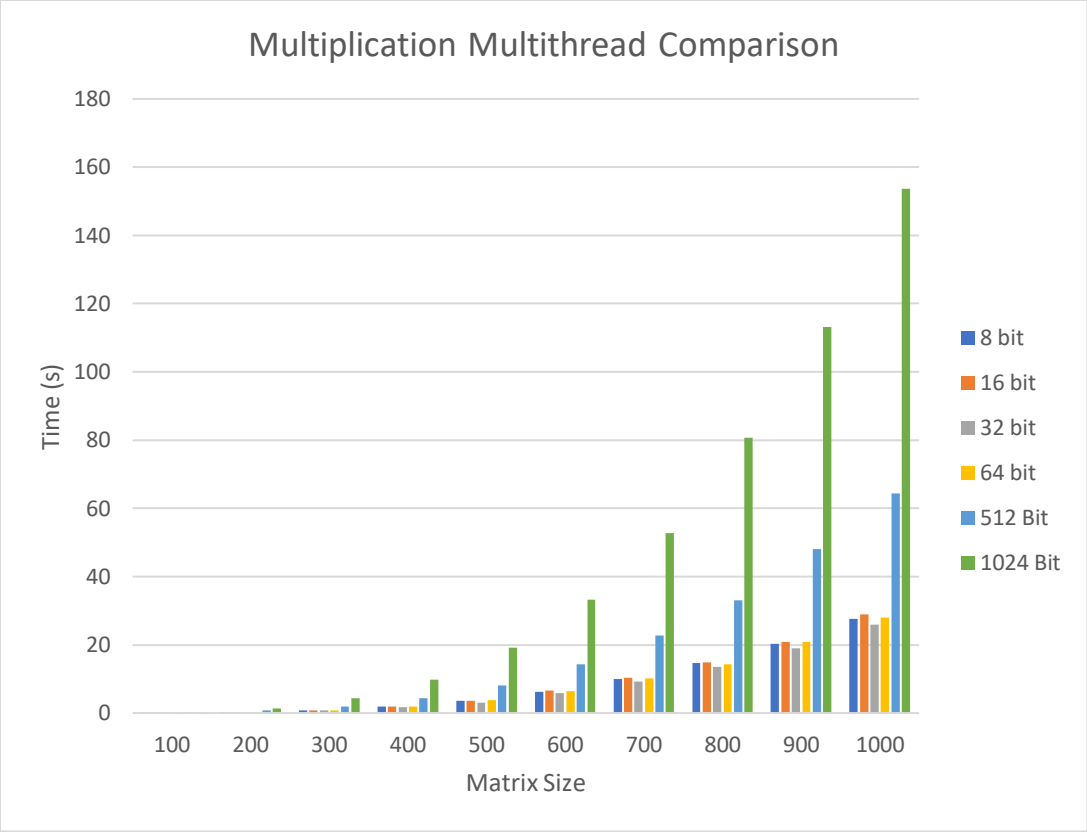
512-bit Average Percent Change: -73%



1024-bit Average Percent Change: -75%



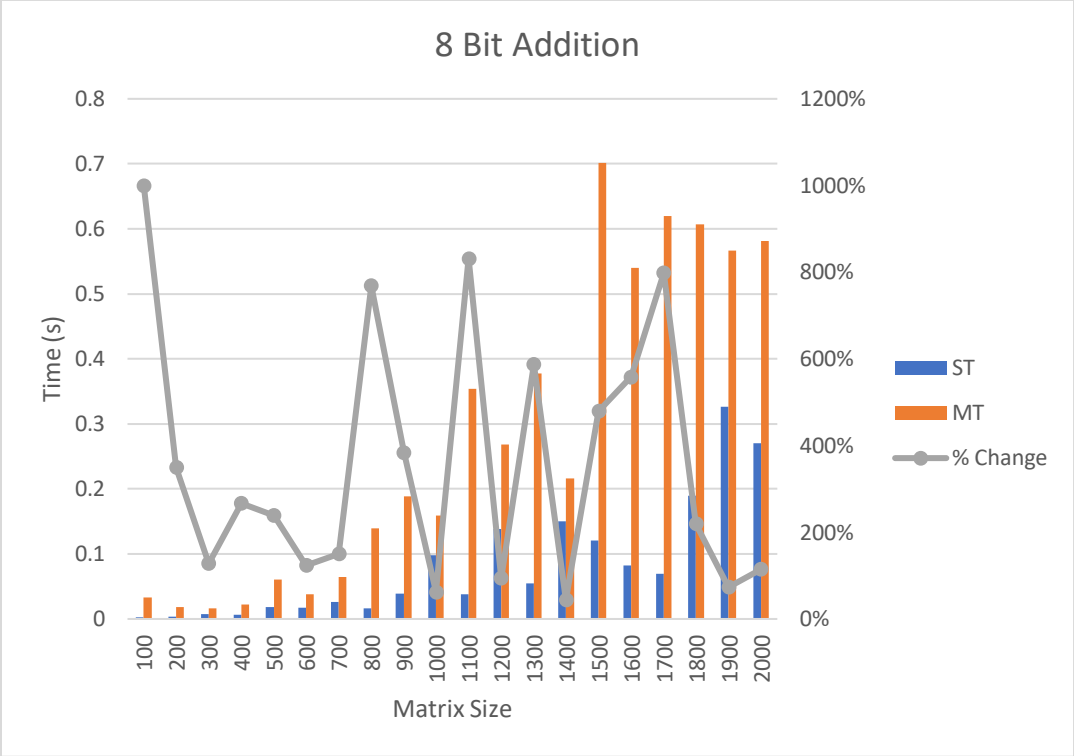




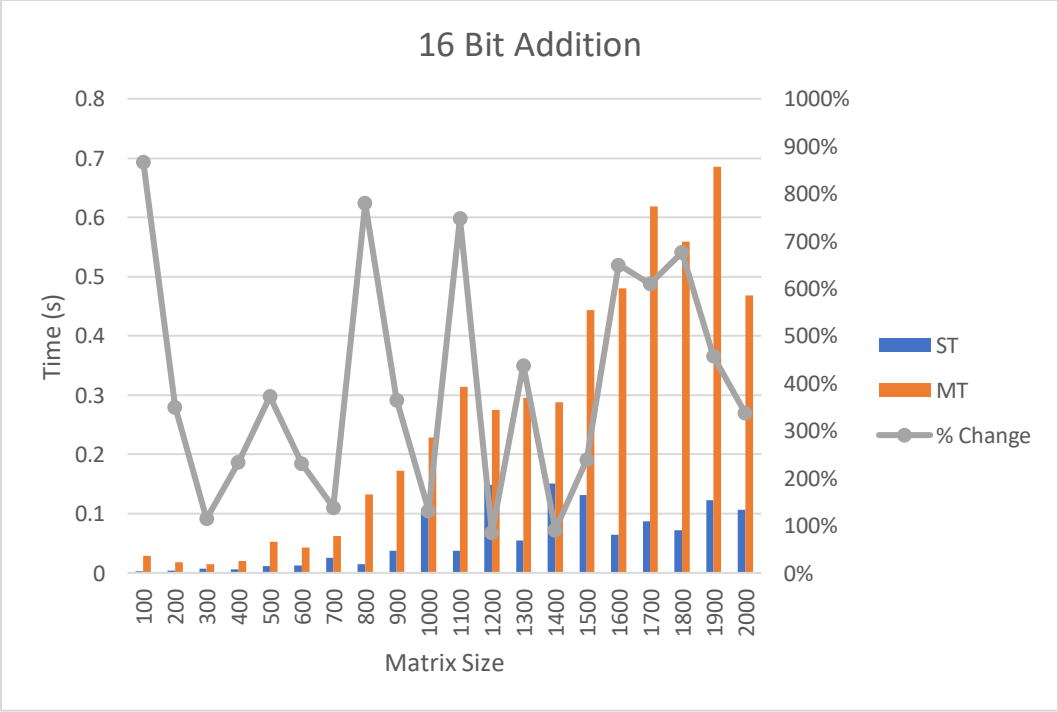
Addition

The tests for addition produced results that were opposite to the multiplication. The multithreaded tests ran much slower than their single thread counterparts. The tests also varied wildly as they were conducted, unlike the multiplication tests. The total average percent change for addition was 214%. The multithreaded tests ran over three times slower than the single threaded counterparts. The 16-bit test had the largest difference with 396% while the 1024-bit tests had the least with 45%.

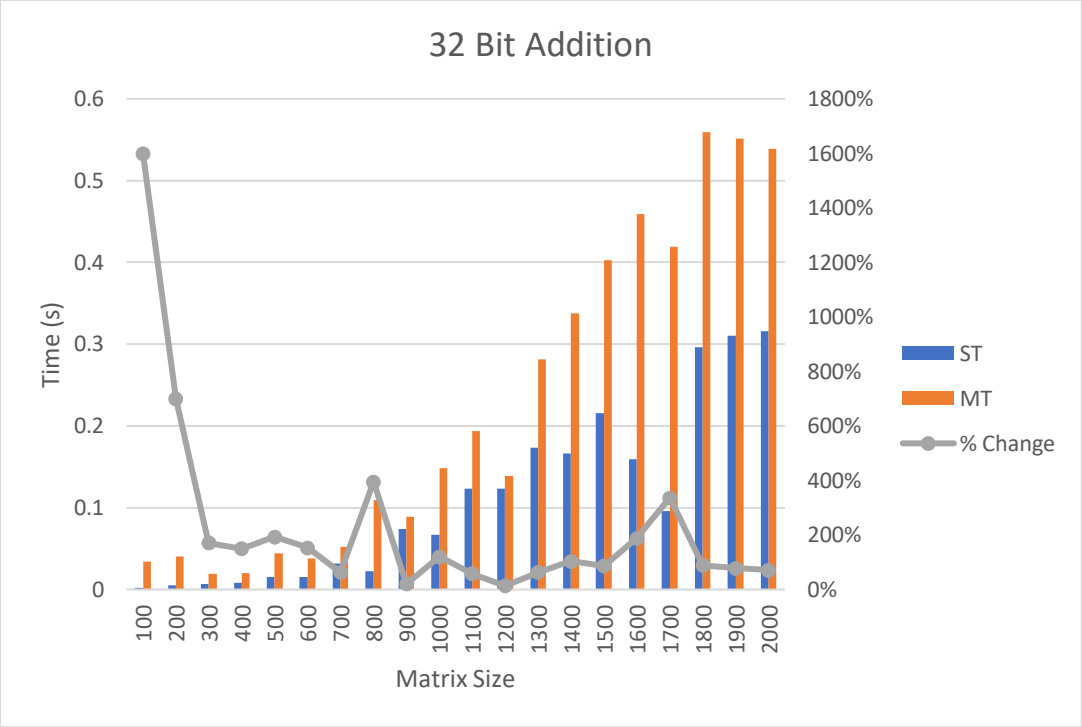
8-bit Average Percent Change: 364%



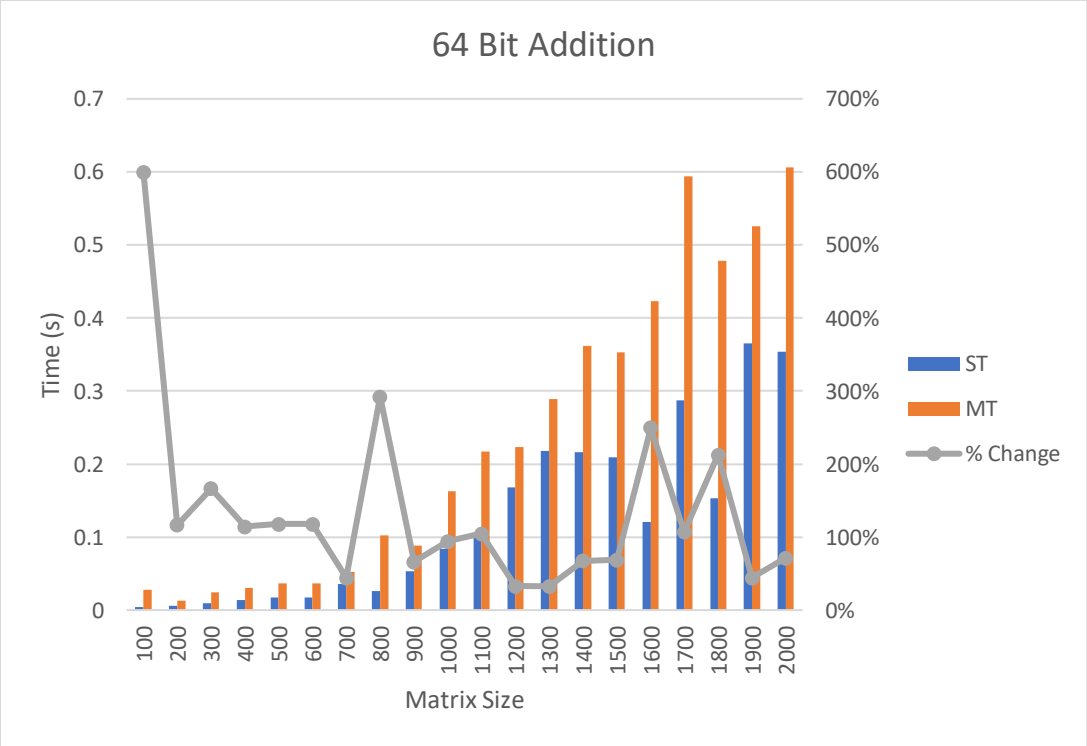
16-bit Average Percent Change: 396%



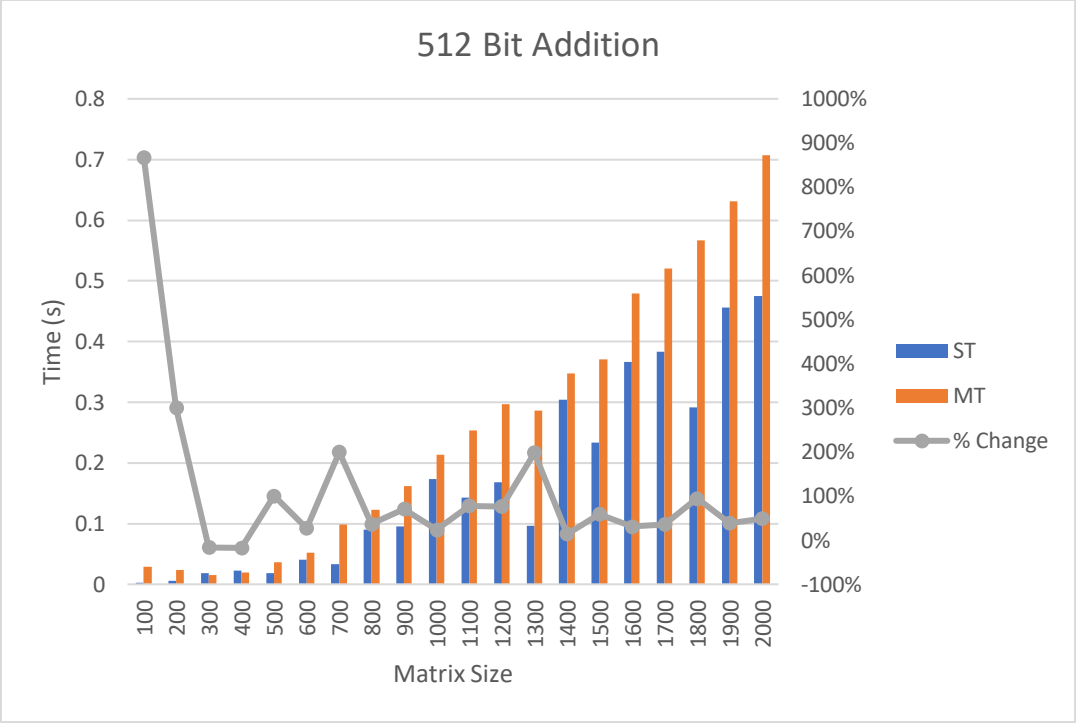
32-bit Average Percent Change: 233%



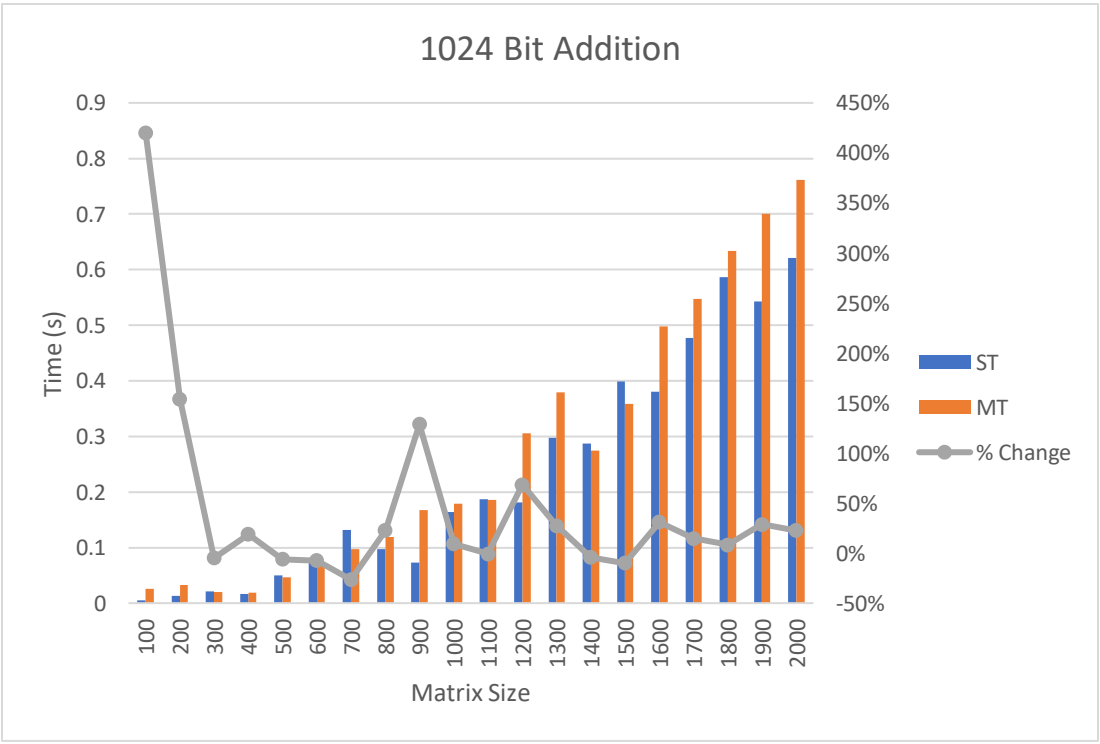
64-bit Average Percent Change: 136%

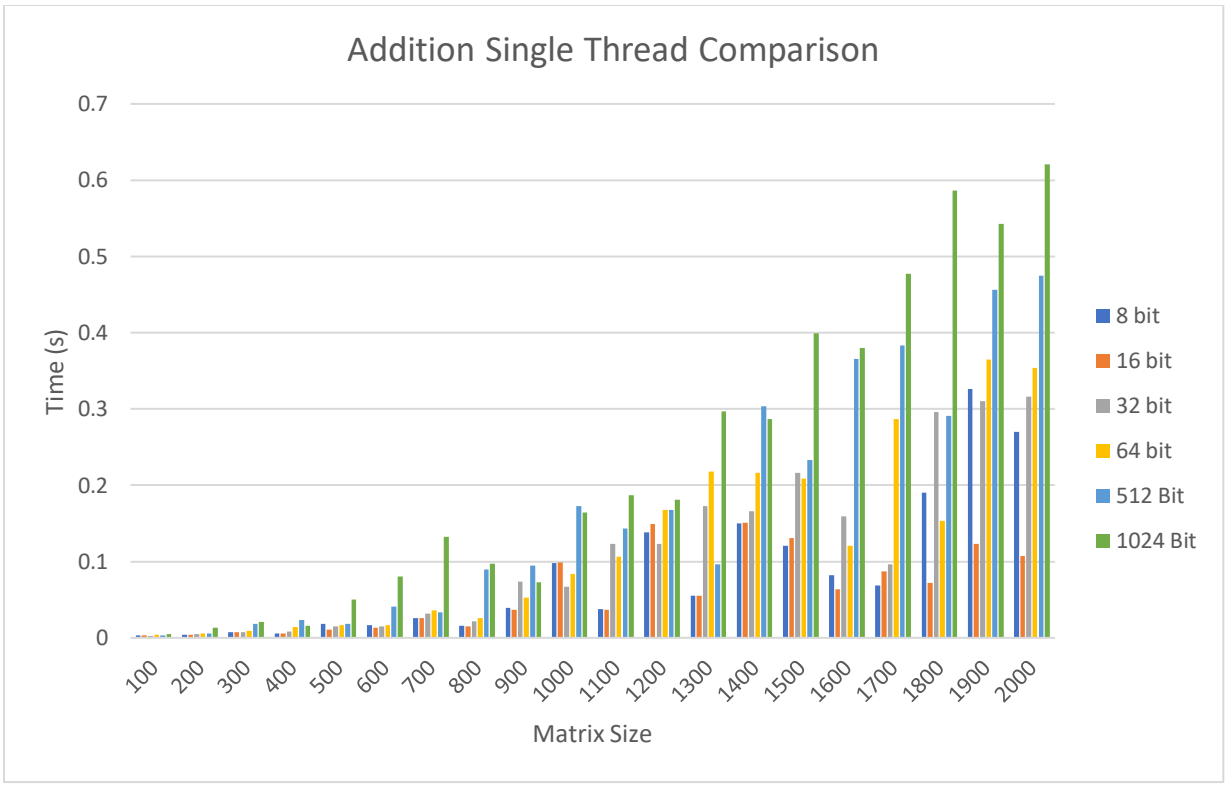


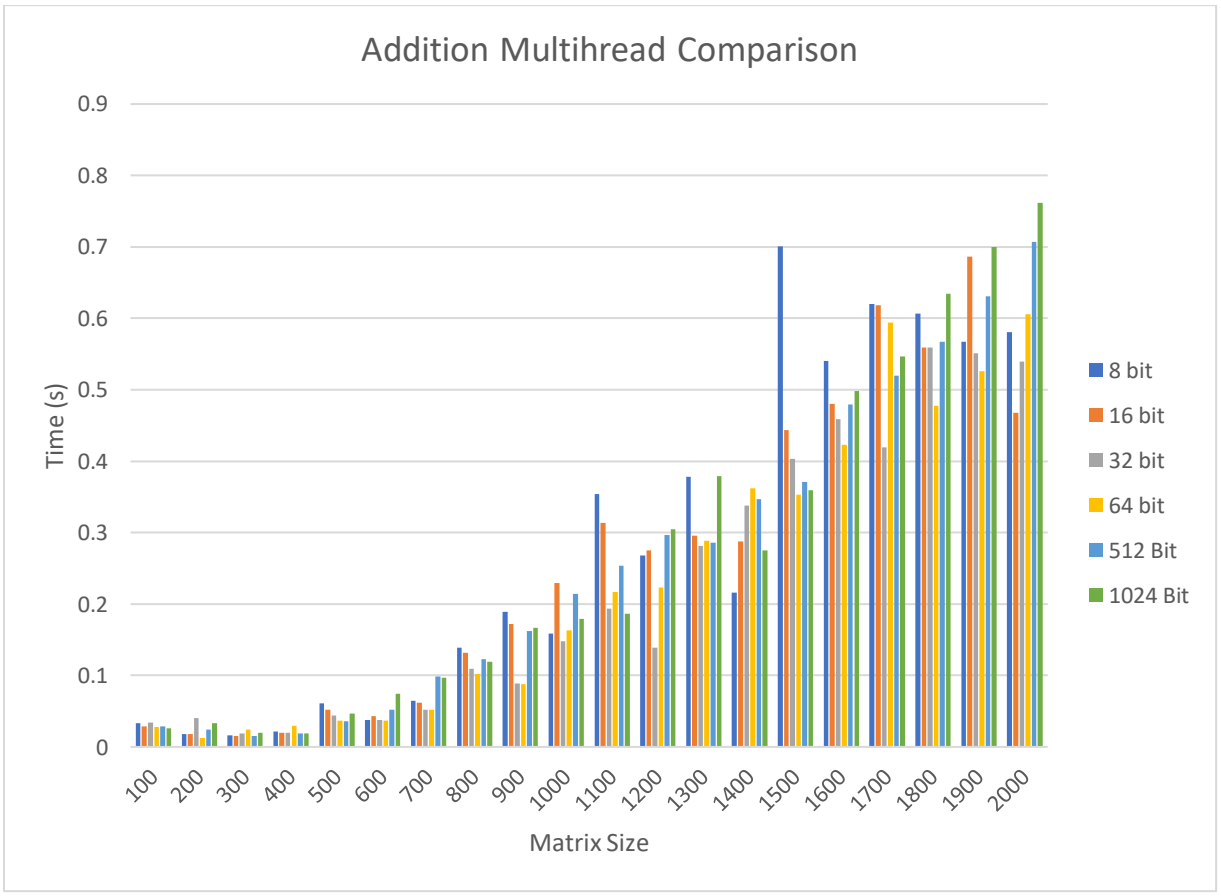
512-bit Average Percent Change: 113%



1024-bit Average Percent Change: 45%




Addition Single Thread Comparison
Time (s)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
Matrix Size
8 bit
16 bit
32 bit
64 bit
512 Bit
1024 Bit


Addition Multihread Comparison
Time (s)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
Matrix Size
8 bit
16 bit
32 bit
64 bit
512 Bit
1024 Bit

Modified Addition

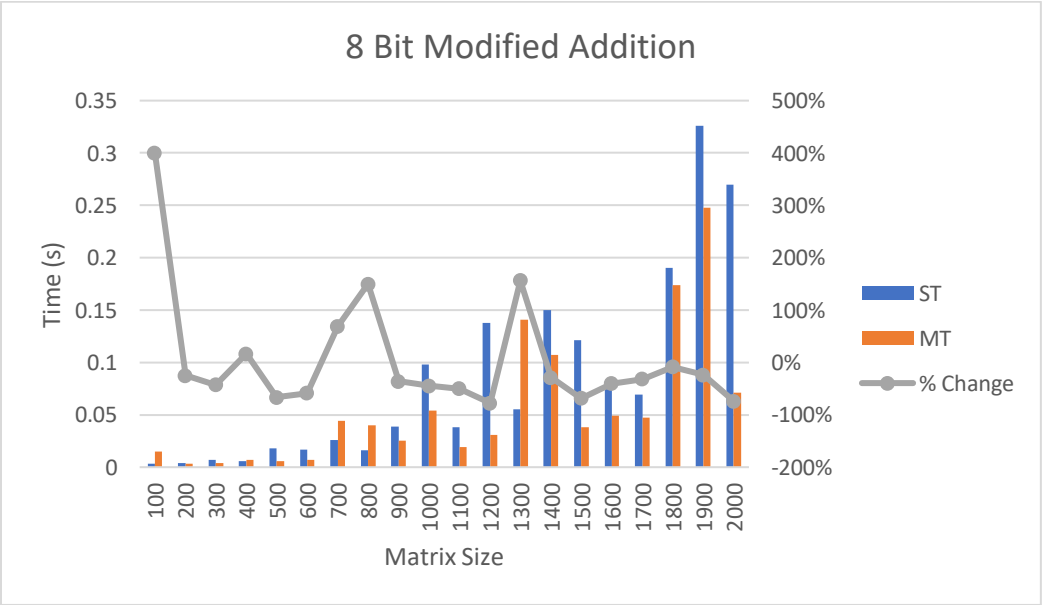
The massive discrepancy between the addition and multiplication tests led to another run of tests with a modified version of the multithreaded addition. In the original test, each multithreading task was just one element in the sum matrix, as shown below.

$$\begin{bmatrix} t0 & t1 & t2 \\ t3 & t4 & t5 \\ t6 & t7 & t0 \end{bmatrix}$$

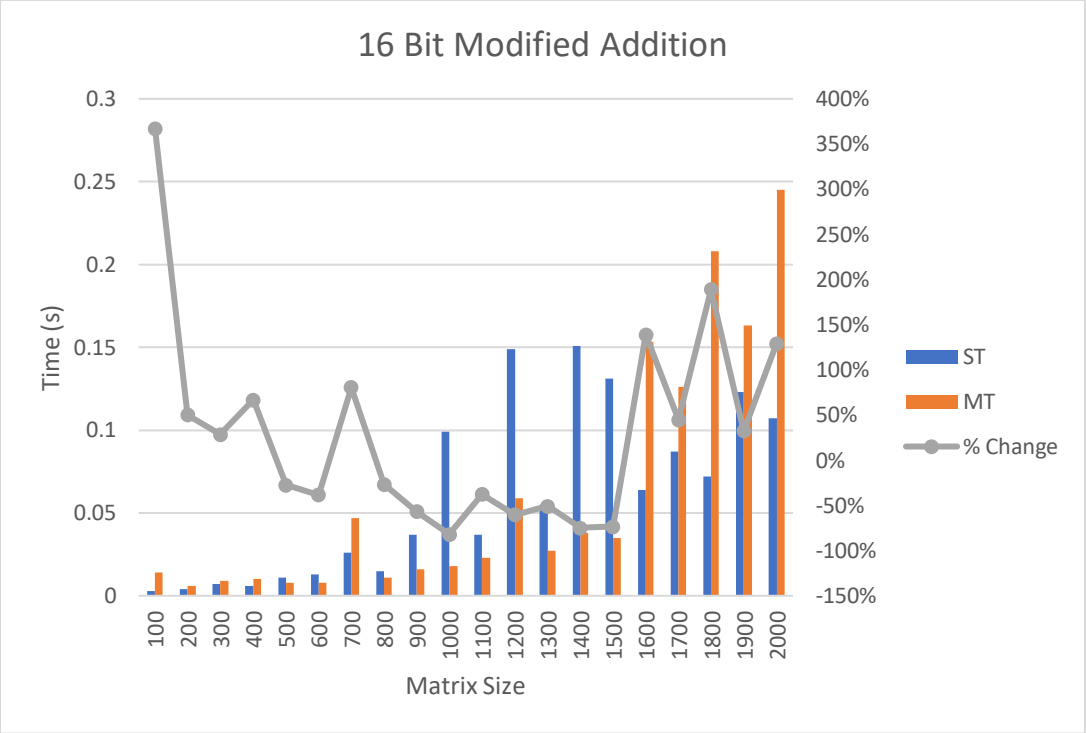
The modified addition had each task arraigned to process each *row* of the matrices instead of each element. This resulted in much shorter execution times but not entirely faster than single threaded execution in every case. The total average percent change observed was -2%. The 16-bit tests had the largest average percent change of 30% while the 64-bit had the smallest with -25%.

$$\begin{bmatrix} t0 & t0 & t0 \\ \vdots & \vdots & \vdots \\ t7 & t7 & t7 \end{bmatrix}$$

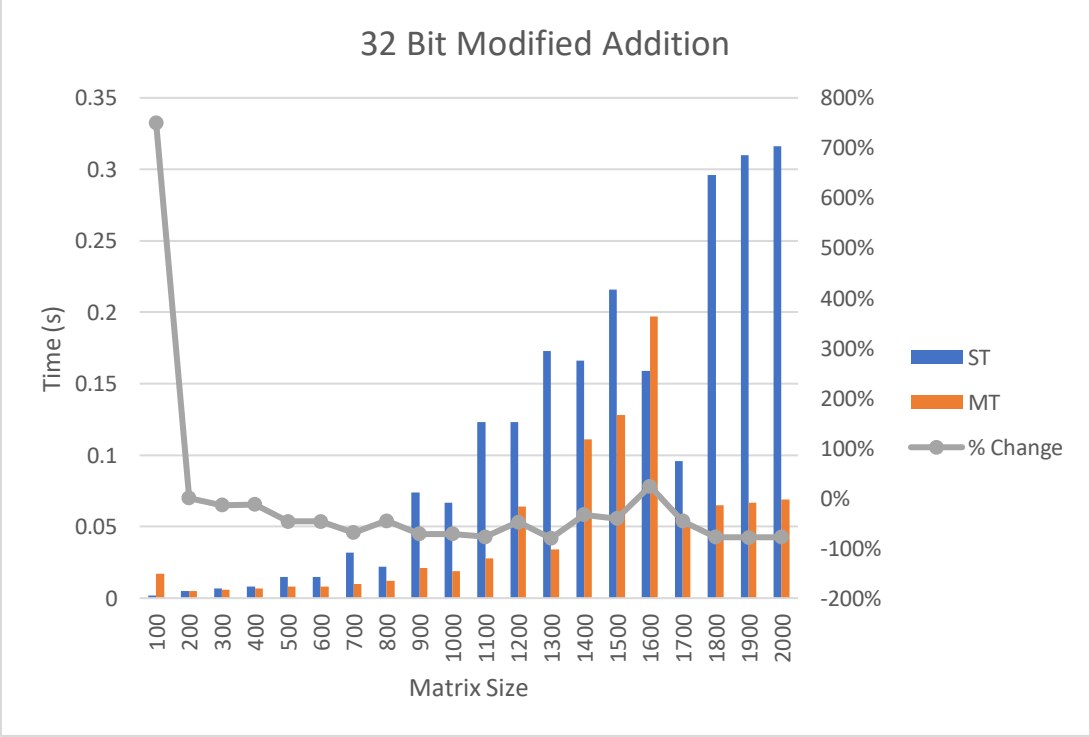
8-bit Average Percent Change: 6%



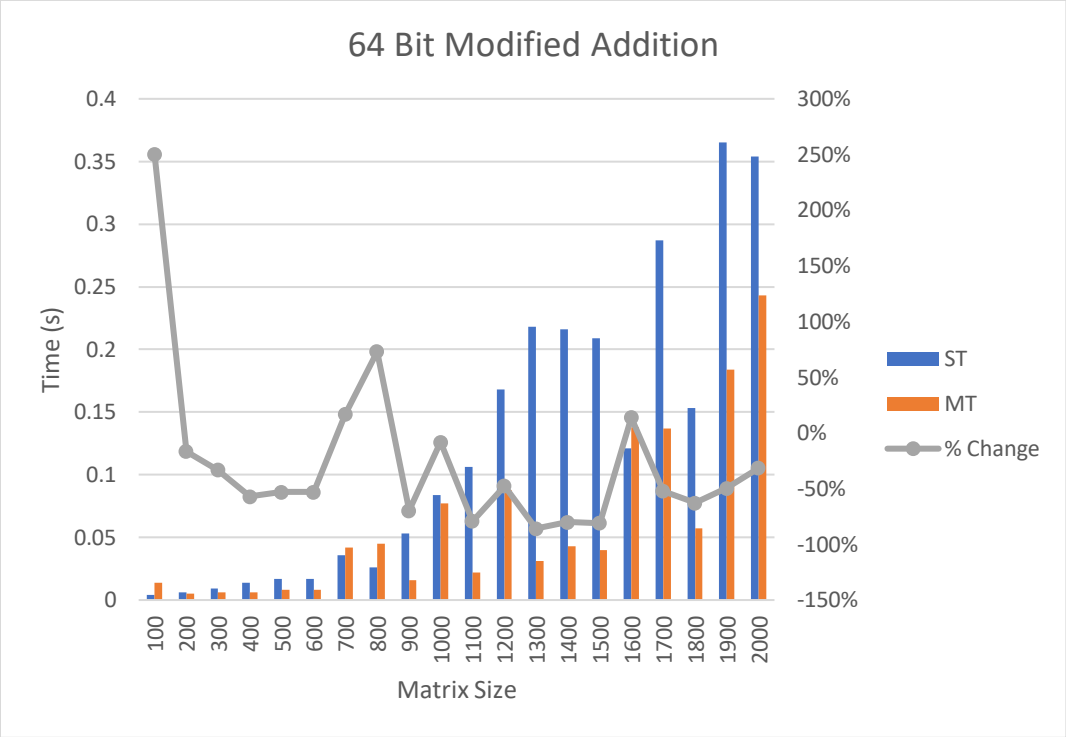
16-bit Average Percent Change: 30%



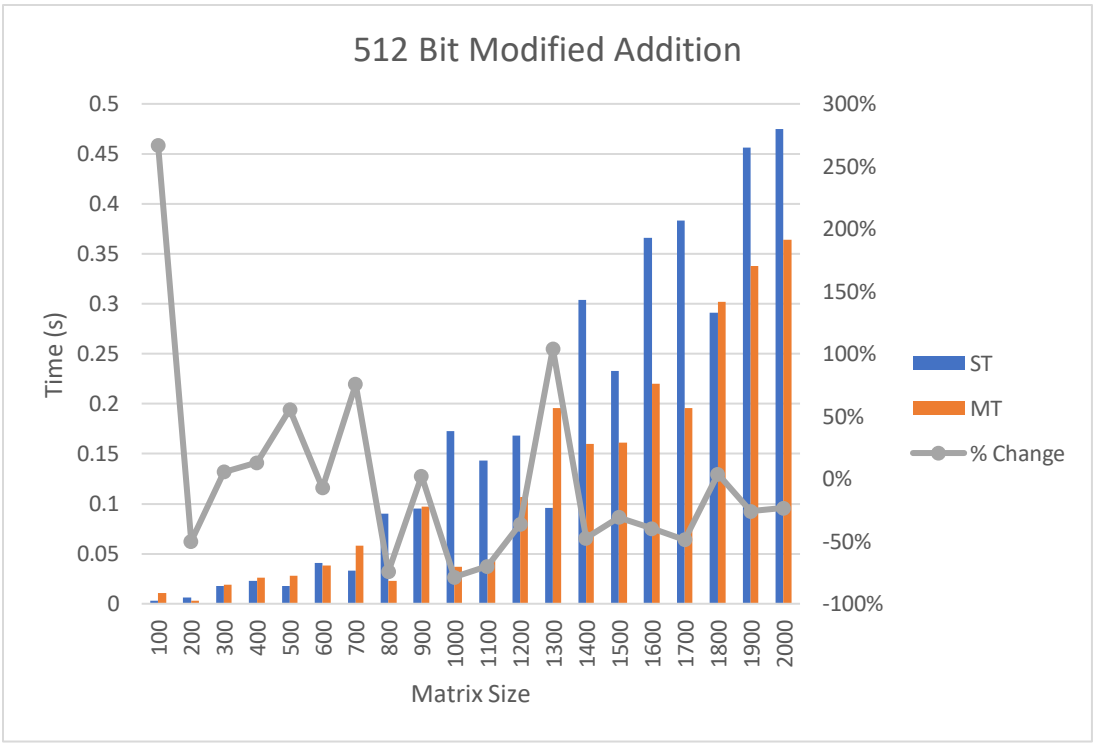
32-bit Average Percent Change: -8%



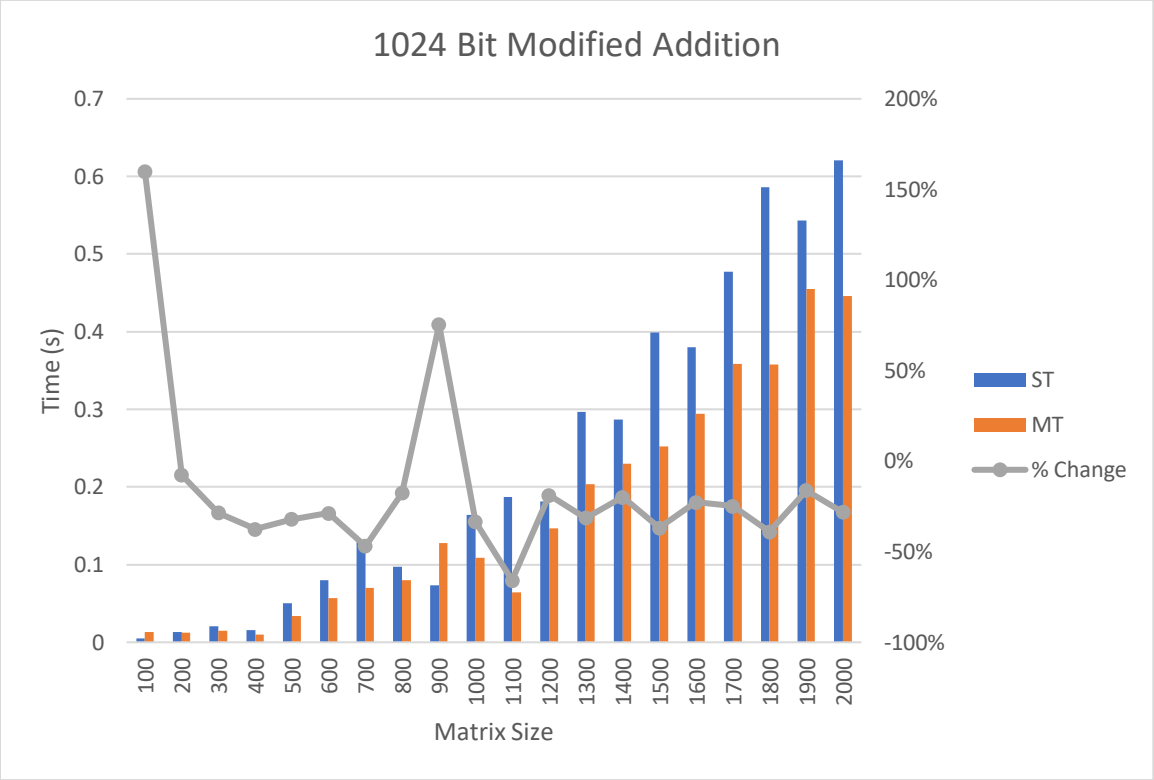
64-bit Average Percent Change: -25%

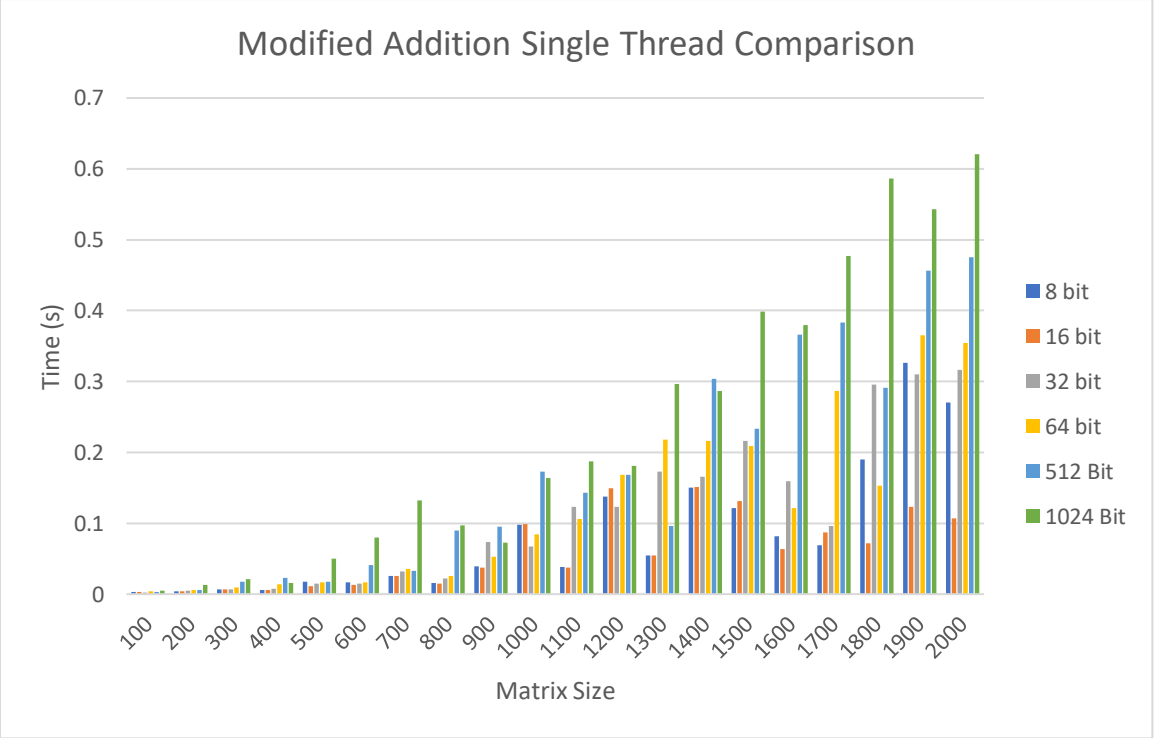


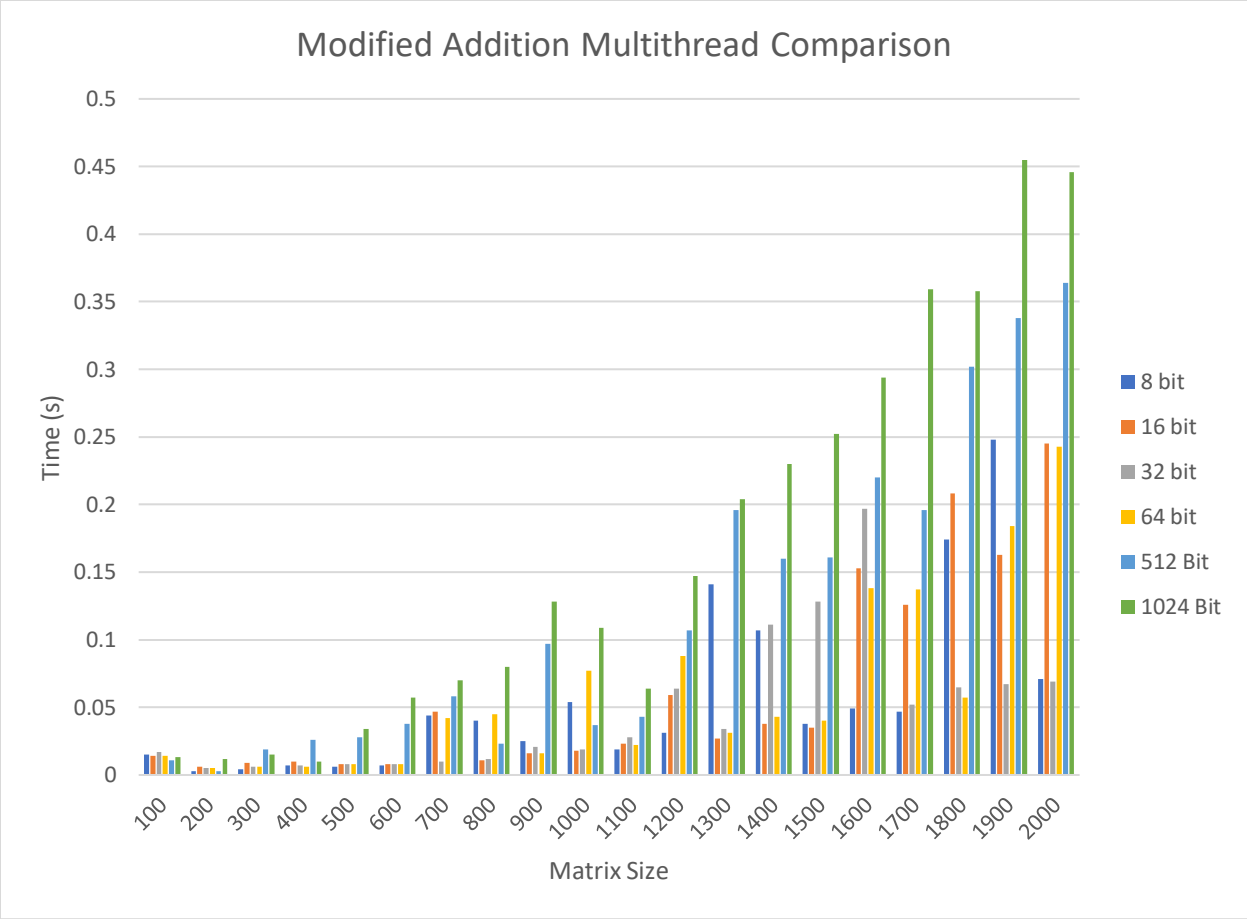
512-bit Average Percent Change: 0%



1024-bit Average Percent Change: -15%







Conclusion

Multithreading can be a very useful tool but only when it is used correctly. Just because a process can be parallelized does not mean it should be. Matrix multiplication is a very good task that can be parallelized due to the amount of work required. Since each element in the product matrix needs to be calculated by doing the dot product, the worker threads spend enough time working on those dot products to have the multithreading be useful. On the other hand, matrix addition does not contain enough work to do the same.

Each element in a sum matrix is just a simple addition. This can be executed much faster than calculating a dot product. Only when modifying the multithreaded addition to have each thread work a row instead of one element, does the multithreading sometimes produce shorter

execution times. The size of the matrix also matters. The 100 size matrices benefited much less from the multithreading that the larger sizes did. The percent change line on most of the graphs follow an exponential decay. The only difference that the change in bit size made was close the gap between the smallest and largest bit size matrix between the larger and smaller bit sizes.

To benefit from multithreading as much as possible, these conditions should be met:

1. The task in question should be able to be broken into subtasks.
2. The subtasks should still be long enough that it still takes a significant amount of time to complete. If the subtask is only a simple math problem that goes through the ALU of a CPU a handful of times, then it is probably not enough.
3. The subtasks should be independent of each other, otherwise synchronization needs to be implemented which could result in slower execution times.

If these conditions are met, then applying parallelism should greatly benefit the task at hand.