Lab 2: Parallel Matrix Multiplication

Kollin Trujillo
May 15, 2024

CS 530: High Performance Computing
Dr. Szilárd Vajda

Department of Computer Science
Central Washington University
Ellensburg, WA, USA
Spring Quarter, 2024

Technical Report: Parallel Matrix Multiplication

Kollin Trujillo[1]

Central Washington University
trujilloK@cwu.edu

Abstract. Matrix multiplication is ubiquitously used across all of modern scientific computing. The matrix, or array, is one of the fundamental data structures utilized in numerical programming alongside with being a common data structure implemented to represent data. This data can be represented by a variety of number types, in our case we used doubles, which are 8 byte representations of data. How the matrix operations are parallelized and on what can have profound effects on runtime. On top of the data storage methods being static or dynamic, there exists parallel matrix processing methods to speed up computations. These include the use threads via the POSIX threads (pthreads) API, and additionally the usage of CUDA, NVIDIA's language for GPGPU computing. The purpose of this technical report is ascertain how various programming languages and parallel approaches affect the computational time of MxM arrays with differing level of N matrix power calls.

Keywords: Matrix Multiplication · C++ · Python · CUDA · Parallel · pthreads

Table of Contents

1 Introduction 5
  1.1 Timing Functions 5
  1.2 Multithreaded code models 6
2 Methods 8
  2.1 Algorithms Used 8
3 Experiments 11
  3.1 PC Specs 11
  3.2 Implementation Details 13
4 Results 14
5 Discussion 14
6 Conclusion 20

List of Figures

1 Figure outlining the structure of a peer worker model (taken from [3]). 7
2 Figure outlining the structure of a boss model (taken from [3]). 7
3 Figure outlining the structure of a pipeline model (taken from [3]). . . . 8
4 Time scaling for $M = 1$ 14
5 Time scaling for $M = 2$ 15
6 Time scaling for $M = 5$ 15
7 Time scaling for $M = 10$ 16
8 Time scaling for $M = 25$ 16
9 Time scaling for $M = 50$ 17
10 Time scaling for $M = 100$ 17
11 Time scaling for $M = 250$ 18
12 Time scaling for $M = 500$ 18
13 Time scaling for $M = 1000$ 19

List of Tables

1 PC Specifications 12

List of Algorithms

1 Matrix Multiplication 9
2 Matrix Exponentiation by Squaring 9
3 Multithreaded Matrix Multiplication (Row-wise) 10
4 Multithreaded Matrix Multiplication (Column-wise) 10
5 Multithreaded Matrix Multiplication (Element-wise) 11
6 Multithreaded Matrix Exponentiation by Squaring 11

1 Introduction

Matrix multiplication is a simple yet potential more involved topic. In a lower-level language such as C or C++, there is more performance to be gained by performant code compared to Python or other higher-level language.

On top of matrix multiplication being impacted by language choice, whether it be compiled, interpreted, or semi-compiled language like Java, all have impacts on performance of the code. As the systems under study increase in `M`, the dimensionality of the matrix, we quickly start to necessitate utilization of more powerful computers. Parallel Programming alleviates the bigger data problem.

Pthreads are an API in C that stands for POSIX threads. These pthreads are fairly portable and allows for computation on unix-like systems. This is opposed to CUDA which is more hardware-oriented and the accompanying software and compiler are accessible across more common operating systems.

1.1 Timing Functions

For C/C++, there are a few common timing functions. These include `clock`, `time`, among others. There are also linux OS timing functions that will not be discussed much at depth here.

There are multiple different time functions for the use of benchmarking. They vary based on the the resolution of the timing function (s, ms, etc), and whether wall, system, CPU, or clock time is the one being measured.

Wall time Wall time is a clock measurement where it measures the real-world time progressed since invocation of the start of clock. This can be useful and generally is what is measured. However, if something is shuffled off of the primary task of the job scheduling part of the OS.

`gettimeofday()` is part of a system call. It can be utilized with C as well and returns the number of seconds and microseconds that have passed since the epoch, which is the start of January 1st, 1970 UTC. This is affected by the precision/granularity of the system's tick schedule. This would be an

CPU Time CPU time differs from wall time in that for serial programs wall time $\geq$ cpu time in serial (single CPU) times. However, if there is parallel elements via utilization of more than one CPU then CPU time would be greater than wall time. However, CPU time also differs from wall time when I/O operations or thread/process shuffling occurs, as mentioned briefly under the wall time section above.

`time()` vs `clock()` differ in a similar aspect as wall time and CPU time. Time will measure the real time whereas clock will measure the time of processes, similar to CPU time.

`time.time()` was utilized as the class and then multiplied by 1000 to get milliseconds (ms) for the Python timings. This introduces some potential issues especially for misrepresenting data on the lower end of the time scale for quicker

runs rounding up or down. This is slightly noticed in the Python performance times shown below.

1.2 Multithreaded code models

Multithreaded code and the need for concurrency is everywhere. Data is constantly being generated and there is a need for more efficient utilization of hardware required to process this ever increasing amount of data. From this, multithreaded/parallel computation approaches pop up and some below are briefly mentioned, illustrated, and detailed. However, it should be known that thread models are not limited to this and there exists almost limitless thread models for computation [3].

With parallel and multithreaded code, there arises the issues of data races. A data race is formally described in the context of parallel programming [4]

> A race condition occurs in a parallel program execution when two or more threads access a common resource, e.g., a variable in shared memory, and the order of the accesses depends on the timing, i.e., the progress of individual threads.

this requires a little bit more complexity required in order to properly manage these race conditions.

Controlling parallel programming requires the use of locks in the code. These locks come in the form of mutexes (mutual exclusions), which are the simplest form of synchronizations and work by restricting write access to data that is to be protected and can introduce issues such as spinlocking [6]. Then with pthreads that includes the use of barriers which are a forced wait synchronization meaning that the threads created all need to wait until they "check in" before releasing the barrier [3]. Synchronization can lead to downtime on CPUs and lead to not fully utilizing all of the possible clock cycles but helps to alleviate potential threads finishing early starting before the rest of the prior rounds have finished.

When discussing pthreads, and threaded models, for parallel programming, there are a few typical approaches utilized but not at all exhaustive. These are further described in the follow subsections.

Peer/workcrew model In this model of threads (see figure 1) we can see division of duties with threads. In this model, one thread makes all the other ones, then they run and they are suspended after completion until all the peers finish [3]. This is also simplified in logic since each peer knows what data it deals with. This is useful for matrix multiplication [3], and is the approach utilized here.

Boss model In this model of threads, (see figure 2) we can see how there is a thread referred to as the boss thread that accepts and distributes work for the rest of the worker threads [3]. Usually the threads will run then wait to sync up with the boss who will decide how to process the output data [3].

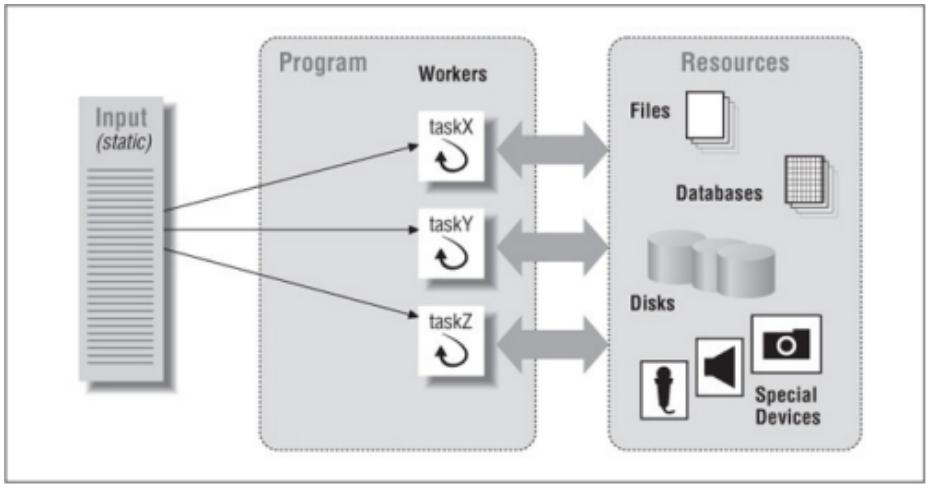

Fig. 1: Figure outlining the structure of a peer worker model (taken from [3]).

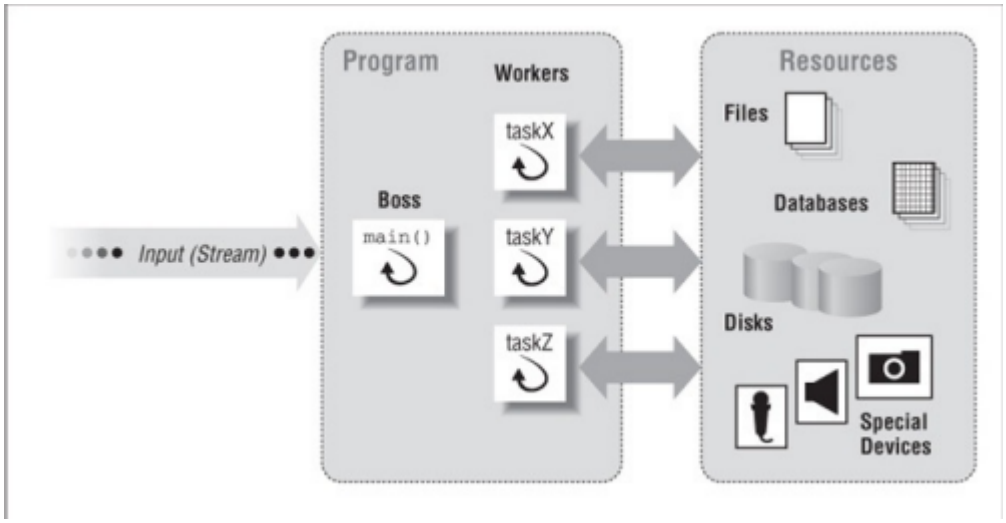

Fig. 2: Figure outlining the structure of a boss model (taken from [3]).

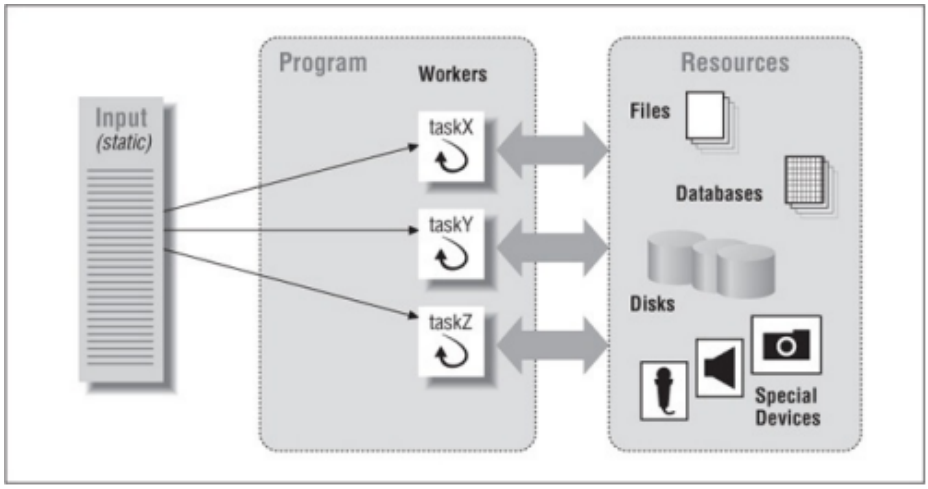

Fig. 3: Figure outlining the structure of a pipeline model (taken from [3]).

Pipeline model In this model of threads, (see figure 3) we can how the input stream affects the creation of these threads in a linear matter [3]. From here a single thread takes the input, processes it, spits out the output which the next stage filter threads will take care and often times there will be a final stage that has the specialty use of processing the final output stage [3].

2 Methods

2.1 Algorithms Used

The program is carried out utilizing the C/C++, CUDA, and Python programming languages. The algorithms utilized are illustrated in algorithm 1 and 2 along with their time complexities.

The time complexity of multithreaded matrix multiplication algorithms 5, 4, and 5 are still $O(M^3)$, which arises from the three nested loops, each running from 0 to $M-1$. This complexity indicates that the time required to compute the matrix multiplication scales cubically with the increase in the dimension M of the square matrices. However, with multithreaded code, we can in theory have it be $\frac{O(M^3}{T}$ for T threads for the MxM size matrix. This is in comparison to algorithm 1 which is $O(M^3)$ and doesn't have the $\frac{1}{T}$ factor due to only using one thread.

The time complexity of the matrix power raise algorithm is $O(log_2 N * M^3)$, where the M^3 portion could be reduced to a lower scaling exponent if Strassen's [7] or the Coppersmith–Winograd [1] algorithm which have a cap on the first portion as $O(n^{log_2 7}) \approx O(n^{2.807})$ (or $O(n^{2.479})$ for the newer revision) and $O(n^{2.3755})$ respectively. So, these represent an obvious improvement in computational com-

Algorithm 1 Matrix Multiplication

Require: Two square matrices A and B of size $M \times M$
Ensure: Matrix C as the product of matrices A and B, $C = AB$
  Initialize matrix C of size $M \times M$ with all zeros
  for $i = 0$ to $M - 1$ **do** ▷ Iterate over rows of A
    for $j = 0$ to $M - 1$ **do** ▷ Iterate over columns of B
      $C[i][j] \leftarrow 0$
      for $k = 0$ to $M - 1$ **do** ▷ Compute dot product
        $C[i][j] \leftarrow C[i][j] + (A[i][k] \cdot B[k][j])$
      end for
    end for
  end for
  Complexity: The time complexity of this algorithm is $O(M^3)$.

Algorithm 2 Matrix Exponentiation by Squaring

Require: A square matrix A of size $M \times M$ and a non-negative integer N
Ensure: Matrix $Result$ as the matrix A raised to the power N, $Result = A^N$
  Initialize matrix $Result$ of size $M \times M$ to the identity matrix
  Copy matrix A into temporary matrix $Temp$
  while $N > 0$ **do**
    if $N \mod 2 = 1$ **then**
      Create a temporary matrix $TempResult$
      $TempResult \leftarrow Result \times Temp$ ▷ Multiply matrices
      Copy $TempResult$ into $Result$
    end if
    Create a new matrix $TempSquared$
    $TempSquared \leftarrow Temp \times Temp$ ▷ Square the matrix
    Copy $TempSquared$ into $Temp$
    $N \leftarrow \frac{N}{2}$ ▷ Halve the power
  end while
  Complexity: $O(log_2 N * M^3)$ where M^3 is from the algorithm 1.

plexity from the $O(n^3)$ approach utilized. These are multiplied by a factor of $\frac{1}{T}$ where $O(M^3)$ is at due to utilization of multithreaded code.

I utilized `std::chrono::high_resolution_clock` for the timing of my dynamic arrays code. This is a wall clock and it seemed suitable. For Python I utilized, `time.time()` which is also a wall clock. To account for issues for with possible I/O delays or processes taking priority, I carried out 5 trials for each run to get an average and standard deviation for each configuration of M and N (can see code snippet in 1.3).

The presence of threads and more computational bandwidth does not majorly change the time complexity of the algorithms utilized. All this does is just increases the amount of power that we throw at the problem. If the problem was small enough that we could throw all k threads available by the system at once, then it could theoretically be $\frac{O(runtime)}{k\ threads}$ but due to Amdahl's law (see equation 1) we can calculate the theoretical speedup based on the sequential and

Algorithm 3 Multithreaded Matrix Multiplication (Row-wise)

Require: Two square matrices A and B of size $M \times M$, number of threads T
Ensure: Matrix C as the product of matrices A and B, $C = AB$
  Initialize matrix C of size $M \times M$ with all zeros
  Divide rows of C among T threads
  function ThreadWorker(id)
    $rows \leftarrow$ rows assigned to thread id
    for each $i \in rows$ **do**
      for $j = 0$ to $M - 1$ **do**
        $C[i][j] \leftarrow 0$
        for $k = 0$ to $M - 1$ **do**
          $C[i][j] \leftarrow C[i][j] + (A[i][k] \cdot B[k][j])$
        end for
      end for
    end for
  end function
  Create T threads, each running ThreadWorker with a unique id
  Wait for all threads to finish
  Complexity: The time complexity of this algorithm is $O(M^3/T)$, with T threads.

Algorithm 4 Multithreaded Matrix Multiplication (Column-wise)

Require: Two square matrices A and B of size $M \times M$, number of threads T
Ensure: Matrix C as the product of matrices A and B, $C = AB$
  Initialize matrix C of size $M \times M$ with all zeros
  Divide columns of C among T threads
  function ThreadWorker(id)
    $cols \leftarrow$ columns assigned to thread id
    for each $j \in cols$ **do**
      for $i = 0$ to $M - 1$ **do**
        $C[i][j] \leftarrow 0$
        for $k = 0$ to $M - 1$ **do**
          $C[i][j] \leftarrow C[i][j] + (A[i][k] \cdot B[k][j])$
        end for
      end for
    end for
  end function
  Create T threads, each running ThreadWorker with a unique id
  Wait for all threads to finish
  Complexity: The time complexity of this algorithm is $O(M^3/T)$, with T threads.

the paralellizable portions but Amdahl's law is only for fixed problem sizes.

$$S_{latency}(s) = \frac{1}{1 - p + \frac{p}{s}} \tag{1}$$

where S represents the speedup, p is the parallelizable portion, s is the speedup ability of the code [5].

Algorithm 5 Multithreaded Matrix Multiplication (Element-wise)

Require: Two square matrices A and B of size $M \times M$, number of threads T
Ensure: Matrix C as the product of matrices A and B, $C = AB$
  Initialize matrix C of size $M \times M$ with all zeros
  Divide elements of C among T threads
  function ThreadWorker(id)
    $elements \leftarrow$ elements assigned to thread id
    for each $(i, j) \in elements$ **do**
      $C[i][j] \leftarrow 0$
      for $k = 0$ to $M - 1$ **do**
        $C[i][j] \leftarrow C[i][j] + (A[i][k] \cdot B[k][j])$
      end for
    end for
  end function
  Create T threads, each running ThreadWorker with a unique id
  Wait for all threads to finish
  Complexity: The time complexity of this algorithm is $O(M^3/T)$, with T threads.

Algorithm 6 Multithreaded Matrix Exponentiation by Squaring

Require: A square matrix A of size $M \times M$, a non-negative integer N, number of threads T
Ensure: Matrix $Result$ as the matrix A raised to the power N, $Result = A^N$
  Initialize matrix $Result$ of size $M \times M$ to the identity matrix
  Copy matrix A into temporary matrix $Temp$
  while $N > 0$ **do**
    if $N \mod 2 = 1$ **then**
      Create a temporary matrix $TempResult$
      Multithreaded Matrix Multiplication($Result$, $Temp$, $TempResult$, T)
      Copy $TempResult$ into $Result$
    end if
    Create a new matrix $TempSquared$
    Multithreaded Matrix Multiplication($Temp$, $Temp$, $TempSquared$, T)
    Copy $TempSquared$ into $Temp$
    $N \leftarrow \frac{N}{2}$ ▷ Halve the power
  end while
  Complexity: $O(\log N \cdot M^3/T)$ where M^3/T is from the algorithm 3, 4, or 5 depending on the chosen method.

3 Experiments

3.1 PC Specs

All experiments were carried out utilizing the PC specified in table 1. The C++ script was automated with the following bash script.

```
#!/bin/bash

# Compile the C++ program
```

Table 1: PC Specifications

Component	Specification
CPU	Intel i9-13900K
Memory	48GB 4800MHz DDR5
GPU	RTX 4090 24GB
Storage	SSD
Operating System	Ubuntu 22.04 LTS

```
cmake ..
make

# Declare arrays of M and N values
declare -a Ms=(1 2 5 10 25 50 100 150 180 250 500 1000) # threads/size
declare -a Ns=(1 2 5 10 25 50 100 250 500 1000 2000 3000 5000 7500 10000) # powers


# Check if the results file exists, if not, create and add the header
RESULTS_FILE="matrix_benchmark_results.csv"
if [ ! -f "$RESULTS_FILE" ]; then
    echo "Method, Matrix Size, Power,Time (milliseconds)" > "$RESULTS_FILE"
fi

# Run the program with each combination of M and N
for M in "${Ms[@]}"
do
    for N in "${Ns[@]}"
    do
        echo "Running matrixOps for M=${M}, N=${N}"
        ./Lab2 $M $N
    done
done

echo "All operations completed."
```

Listing 1.1: Bash script for compiling and running a C++ program with various matrix sizes and powers.

with the Python script being nearly identical:

```
#!/bin/bash

PYTHON_SCRIPT="Lab1PyWork.py"

declare -a Ms=(1 2 5 10 25 50 100 150 180 250 500 1000) # size
```

```
declare -a Ns=(1 2 5 10 25 50 100 250 500 1000 2000 3000 5000 7500 10000) # powers

RESULTS_FILE="matrix_benchmark_results_python3.csv"
if [ ! -f "$RESULTS_FILE" ] || [ ! -s "$RESULTS_FILE" ]; then
    echo "Method, Matrix Size, Power, Time (milliseconds)" > "$RESULTS_FILE"
fi

for M in "${Ms[@]}"
do
    for N in "${Ns[@]}"
    do
        echo "Running ${PYTHON_SCRIPT} for M=${M}, N=${N}"
        python3 ${PYTHON_SCRIPT} $M $N | grep "Time" | while read -r line; do
            TIME=$(echo "$line" | awk '{print $5}')
            echo "Python Array, $M, $N, $TIME" >> "$RESULTS_FILE"
        done
    done
done

echo "All operations completed."
```

Listing 1.2: Bash script for compiling and running a Python program with various matrix sizes and powers.

which greatly eased the execution and subsequent analysis of the data.

3.2 Implementation Details

Algorithms 1 and 2 were implemented in both CUDA/C++ and Python in order to facilitate comparisons. The code was then run on the same machine in order to minimize differences in hardware and execution times.

4 Results

The results are illustrated in the figures below and the table of data is shown in the supplementary information section. Each configuration was ran for 5 times and the mean along with the standard deviation of each `Method`, `Matrix Size`, and `Power` grouped in Python using Pandas code blurb shown in listing 1.3.

```
grouped = df.groupby(['Method', 'Matrix Size', 'Power'])['
    Time (milliseconds)'].agg(['mean', 'std']).reset_index()
```

Listing 1.3: Pandas code blurb detailing the bundling for the plots created.

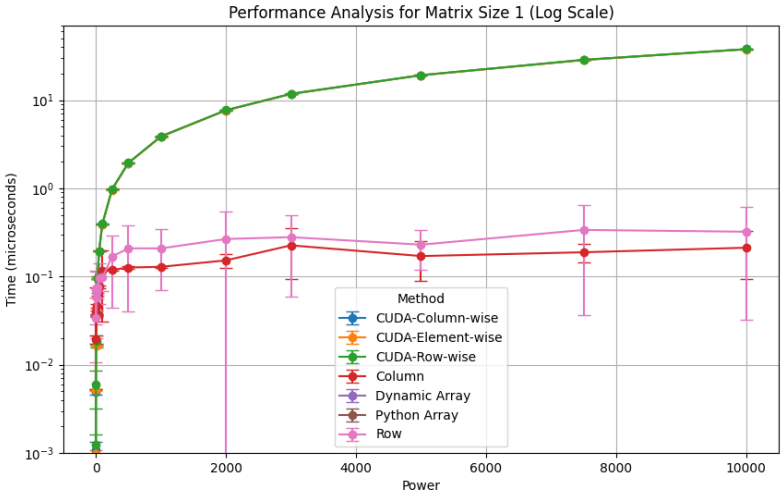

Fig. 4: Time scaling for $M = 1$

5 Discussion

Looking at figure 4, it becomes clear that from the get go that the implementation of element-wise multiplication with CUDA is significantly faster than the other choices. As opposed to the prior report, Lab 1, there is no readily apparent overhead with static arrays. The large overhead falls to the `mult_element` with the initialization of `N*N` threads.

Data has been collected over a wide multitude of `N` and `M` values, which are detailed above in listings 1.1 1.2. The compilation script is ran for powers 1,2,5,10,25,50,100,150,180,250,500,and 1000 whereas the size, and for the C++ and CUDA script, the threads, 1,2,5,10,25,50,100,250,500,1000,2000,3000,5000,7500,

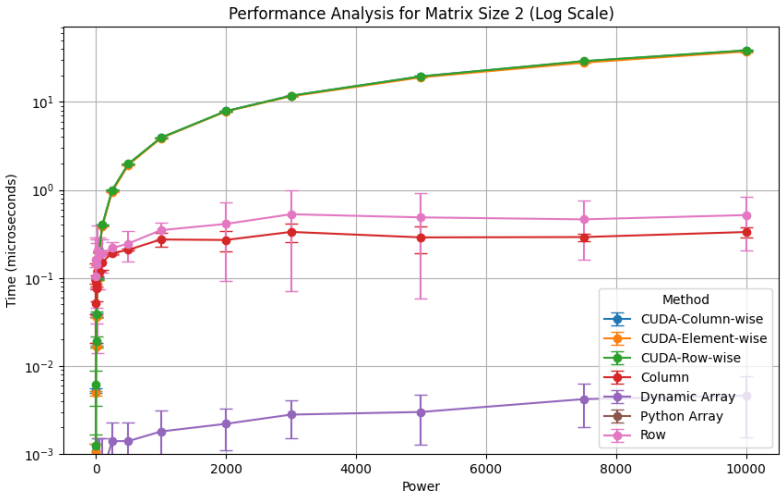

Fig. 5: Time scaling for $M = 2$

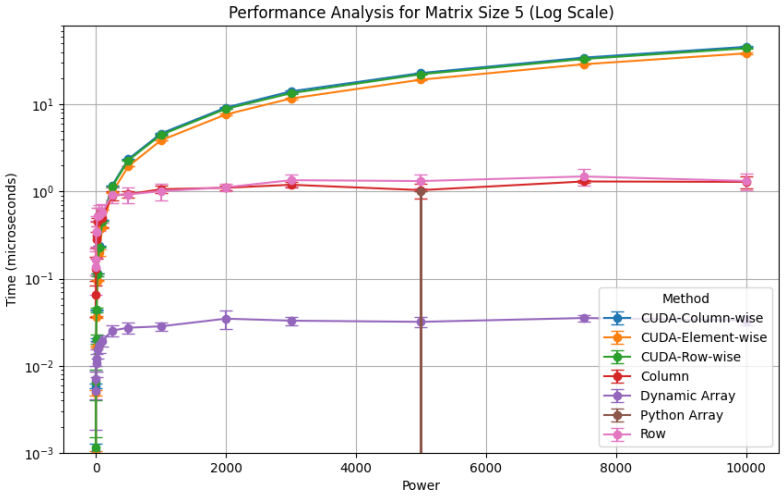

Fig. 6: Time scaling for $M = 5$

and 10000. These are recorded in milliseconds. All runs were carried out on the system outlined in table 1. Looking at the results, it appears that some thermal throttling may have occurred causing some of the GPU runs to be a little slower than usual along with some of the multithreading code, especially after

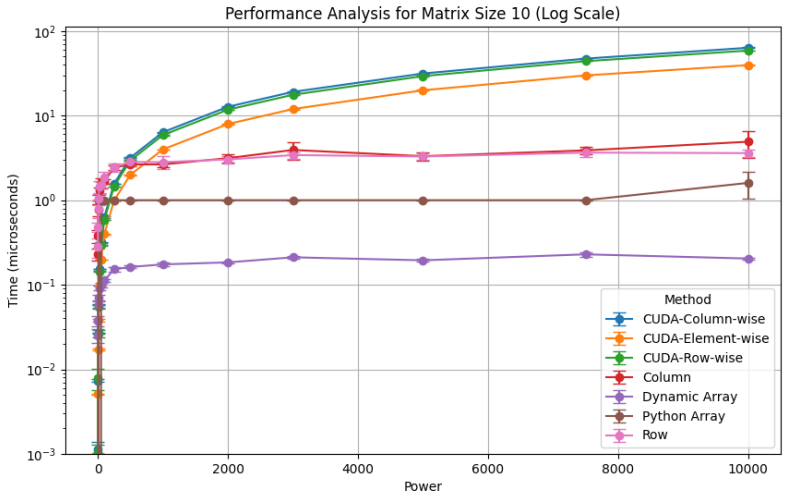

Fig. 7: Time scaling for $M = 10$

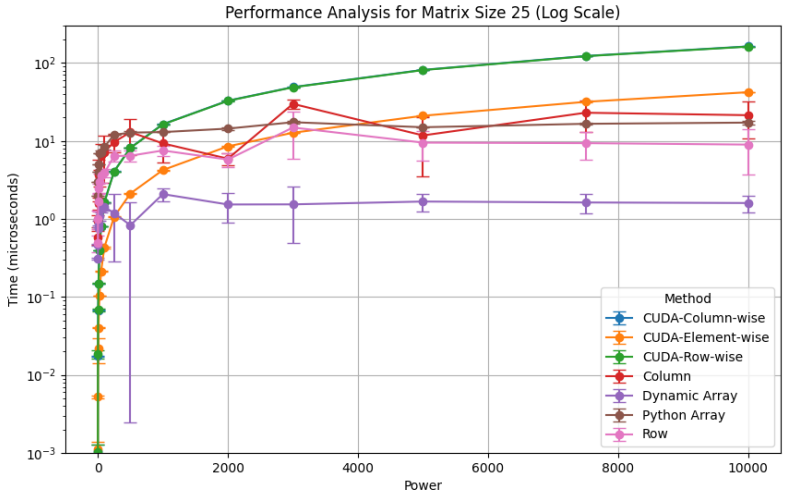

Fig. 8: Time scaling for $M = 25$

running for about 2 or 3 days straight. This is especially noticeable that we see the CUDA-Element-Wise lines start to greatly converge with multiple other methods, such as dynamic, column, and row for multithreading, which logically makes no sense compared to massively parallel programming architectures such

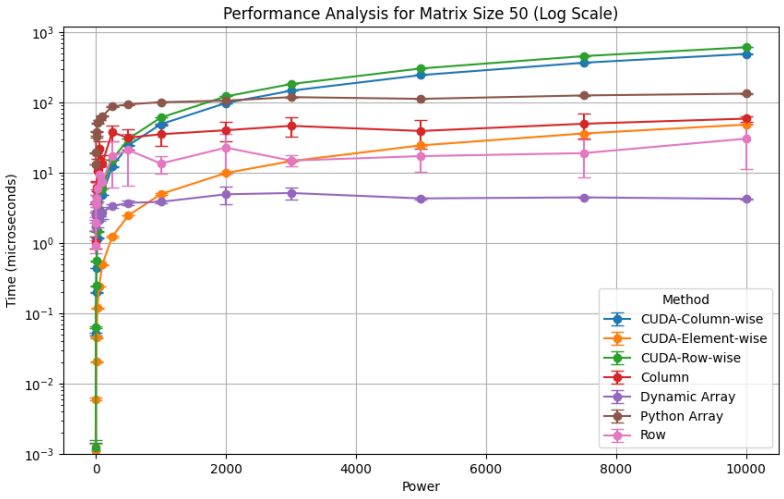

Fig. 9: Time scaling for $M = 50$

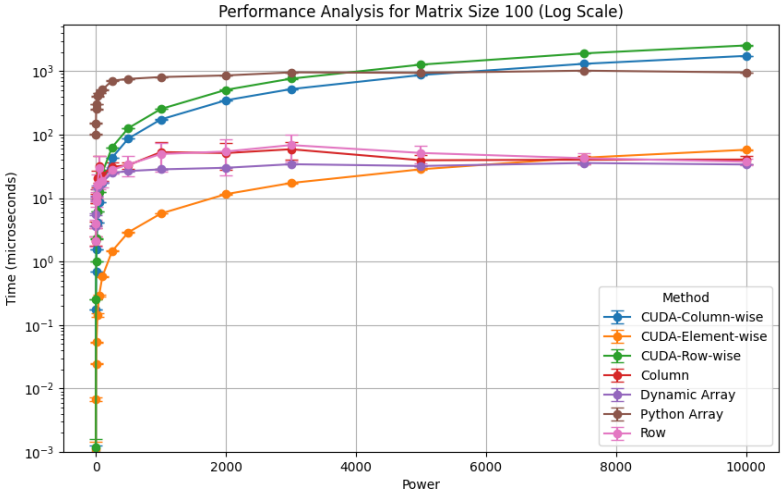

Fig. 10: Time scaling for $M = 100$

as GPUs for GPGPUs computing. However, when the Dynamic Array portion of the code was running, it was the only task specified in the bash script at the time so there was no potential conflict, however unlikely, that occurred.

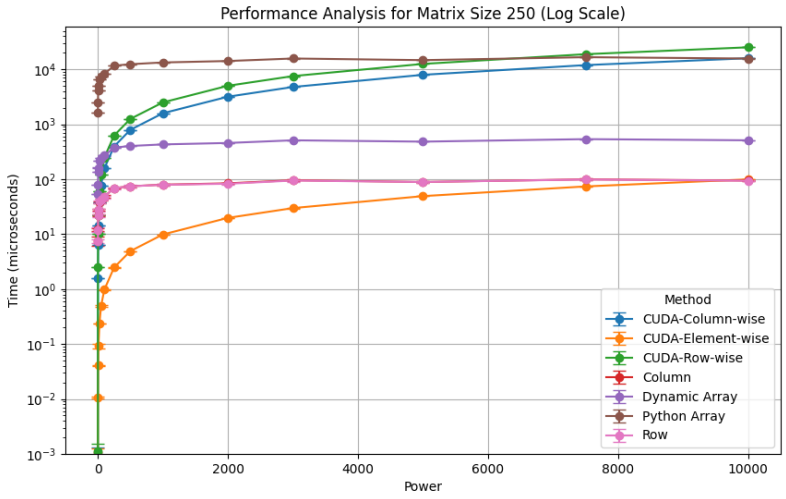

Fig. 11: Time scaling for $M = 250$

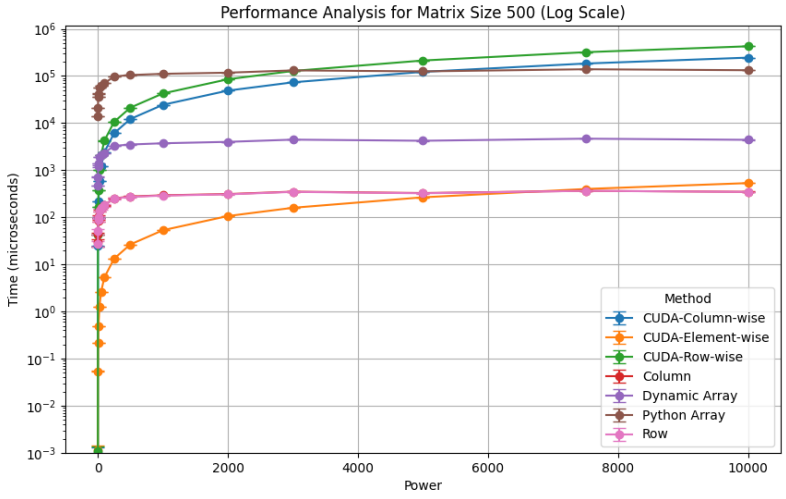

Fig. 12: Time scaling for $M = 500$

Some interesting occurrences are that for figure 4, CUDA-Row-Wise was so much higher than the other ones whereas Row multithreaded has a rather large standard deviation. Not sure of the reason why, other than potential time related to kernel, block, thread initialization specific to CUDA. CUDA also has

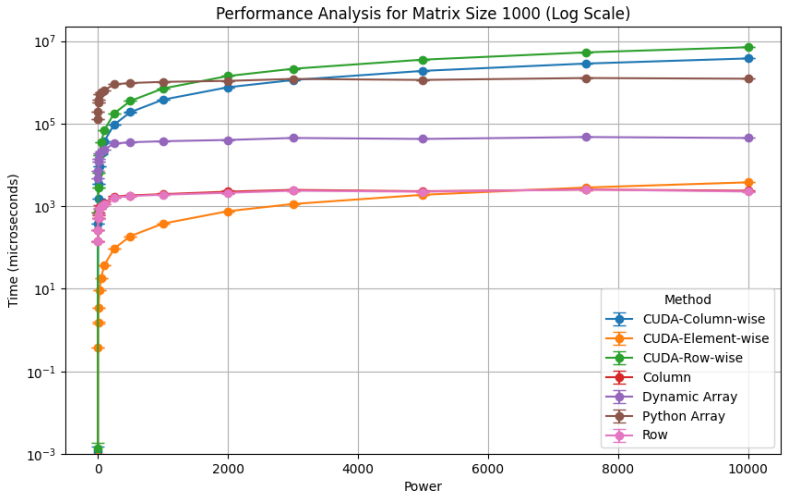

Fig. 13: Time scaling for $M = 1000$

to deal with memory shuffling between the host and device and that adds time to the total runtime. I am sure if more modern code was written and I had more proficiency with CUDA and memory layout, that it could improve than the basic threads and block configuration specified in listing 1.4.

```
1     dim3 threadsPerBlock(16, 16);
2     dim3 numBlocks((numBColumns + threadsPerBlock.x - 1) /
    threadsPerBlock.x, (numARows + threadsPerBlock. y - 1) /
    threadsPerBlock.y);
```

Listing 1.4: C++/CUDA code script detailing grid hierarchy.

Looking at figure 5, CUDA-Row-Wise is also quite high. However, the biggest confusing aspect is that sequential Dynamic arrays were still quite efficient and unsure why.

Figure 6, threaded Row and Column were quite efficient and oddly 10x less than CUDA code, however, these are all still sub 20ms. Python array had quite a bit of variation and cannot ascertain why.

Figure 7 followed a lot of the same similar trends as figure 6.

Figure 8 had a bit more variation in the results but the general order of the magnitudes of time were the same.

Figure 9 marked a slow shift of CUDA-Element-Wise starting to outperform the other CUDA methods, but Dynamic still shined.

Figure 10 showed the CUDA-Element-Wise code was one of the best along with Dynamic sequential arrays and Column and Row multithreaded code.However, it is also weird that a few different plots show that some increasing powers of

code were actually faster than the smaller powers utilized. This could probably be ameliorated if more trials than 5 were ran per instance.

Figure 11 shows CUDA-Element-Wise performing on par with Row and Column multithreaded but a bit better than Dynamic sequential arrays. Also, as expected, Python arrays were some of the longest code to run.

Figure 12 shows that CUDA-Element-Wise performed on par with Row wise and better than Dynamic sequential arrays. Python, CUDA-Column-Wise and CUDA-Row-Wise were the three longest portions of the code to run. This is a little off as I would have thought that CUDA would have been faster.

Figure 13 shows that CUDA-Element-Wise performed on par with Row multithreaded arrays and better than Dynamic arrays. However, it it also highlights that Python is not the best but is on comparison with some slower-performing CUDA code.

Attempts were made to run all code under the same conditions of M and N, but for element-wise multithreaded it will not go above 180^2 threads because it goes above system requirements specified by `ulimit -u` system calls. Adjustments to the stack size via `ulimit -s` changed nothing in its ability to create more threads. So, what will be noticed in the graphs is the element-wise multithreaded code runs up to the M = 180, then does not run past that.

Lastly, some design choices were employed with the CUDA kernel and host functions. A choice was made to not reinitialize the matrix in-between runs of the same method and configuration of power and matrices. This was done to not have to reinitialize the matrices and shuffle it to the GPU. This may have inadvertently led to slower running code.

6 Conclusion

Overall, there is some inconsistencies with the code ran. There exists issues operating system scheduling when utilizing wall clocks for timing of code. Context switching with CPUs can cause latency from swapping processes and threads switching and can cause an increase in performance times that can add up [2], this can vary based on processor architecture and RAM/Cache amounts and this was not under study for this report.

Writing effective parallel code is difficult. Dealing with race conditions, synchronization, and other issues that plague multithreaded and GPGPU computing is difficult. The solution is to code efficiently in a way that alleviates race conditions but does not overload your code with waits, syncs, barriers, mutexes, and the such. GPGPU computing with each thread utilizing an element seems to be hopeful as being the most effective route to calculate large and computational intensive routines such as matrix multiplications. Further work would go into learning how to better write CUDA code alongside how to utilize MPI alongside with CUDA to distribute computation across more than one GPU in a way that utilizes all resources effectively.

References

1. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 1–6 (1987)
2. Kwak, H., Lee, B., Hurson, A.R., Yoon, S.H., Hahn, W.J.: Effects of multithreading on cache performance. IEEE Transactions on Computers **48**(2), 176–184 (1999)
3. Nichols, B., Buttlar, D., Farrell, J.: Pthreads programming: A POSIX standard for better multiprocessing. " O'Reilly Media, Inc." (1996)
4. von Praun, C.: Race Conditions, pp. 1691–1697. Springer US, Boston, MA (2011). https://doi.org/10.1007/978-0-387-09766-4_36, https://doi.org/10.1007/978-0-387-09766-4_36
5. Schmidt, B., Gonzalez-Martinez, J., Hundt, C., Schlarb, M.: Parallel programming: concepts and practice. Morgan Kaufmann (2017)
6. Silberschatz, A., Galvin, P., Gagne, G.: Operating systems concepts. isbn 978-1118063330 (2012)
7. Strassen, V.: The asymptotic spectrum of tensors and the exponent of matrix multiplication. In: 27th Annual Symposium on Foundations of Computer Science (sfcs 1986). pp. 49–54. IEEE (1986)