Massively Parallel Matrix Multiplication

Parallel Programming on High Performance Computers

Andy Zhu
Computer Science
Rensselaer Polytechnic Institute
Troy, New York
zhua6@rpi.edu

Jaisal Patel
Computer Science
Rensselaer Polytechnic Institute
Troy, New York
patelj8@rpi.edu

Patrick Nguyen
Computer Science
Rensselaer Polytechnic Institute
Troy, New York
nguyeq4@rpi.edu

Roger Luo
Computer Science
Rensselaer Polytechnic Institute
Troy, New York
luoz7@rpi.edu

ABSTRACT

CUDA coupled with MPI (Message Passing Interface) is a technique in high-performance computing used to achieve massive speedups in computational performance.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA, which allows developers to harness the computational power of GPUs in general purpose computing. MPI, on the other hand, is a standardized message-passing library interface that facilitates communication between nodes in a distributed computing environment.

Studies have shown that parallel execution can significantly accelerate execution of computationally intensive tasks. However, when the problem size exceeds the memory capacity of a single GPU, the computationally expensive task is limited to a single node, leading to prolonged execution times. MPI addresses this issue by distributing parts of the problem across multiple nodes, enabling faster computation.

The process of matrix multiplication can be computationally intensive, especially for large matrices. Accelerating this operation can significantly impact the overall performance of many applications. In matrix multiplication, where parallelism can be observed to achieve speedups, MPI has the potential to unlock additional speedups. This study aims to demonstrate the effectiveness of combining CUDA and MPI in accelerating matrix multiplication. We demonstrate a parallel algorithm that leverages the MPI framework combined with CUDA to compute matrix multiplication and compare the performance to the serial, or iterative approach. AiMOS (Artificial Intelligence Multiprocessing Optimized System) is a supercomputer located at Rensselaer Polytechnic Institute, serving as the experiment environment for our study. Through testing on a host of large and small matrix sizes, our results are indicative that CUDA combined with MPI fosters significant performance improvements.

CCS CONCEPTS

• Computing methodologies → Parallel computing methodologies; Distributed algorithms • Mathematics of computing → Mathematical software optimization

KEYWORDS

CUDA, MPI, Parallel Processing, GPU

ACM Reference:

Andy Zhu, Jaisal Patel, Patrick Nguyen, and Roger Luo. April 2024. “Massively Parallel Matrix Multiplication.”

1 INTRODUCTION

Matrix multiplication is one of the most fundamental operations in the fields of computer science, engineering, and mathematics. A matrix is a type of multi-dimensional data structure that can hold a

multitude of mathematical objects such as numbers, symbols, and variables. Matrix multiplication is a type of linear transformation that enables the rotation, scaling, or translation of a matrices' vector features with their linear properties preserved. In other words, if a matrix A is multiplied by a matrix B, the resulting matrix C is the result of the compositions of transformations between matrix A and matrix B. Figure 1 demonstrates a simple (2x2) matrix multiplication operation that yields the resulting matrix C.

$$A = \begin{bmatrix} 2 & 1 \\ 3 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}$$

$$C = AB = \begin{bmatrix} (2 \times 1) + (1 \times (-1)) & (2 \times 2) + (1 \times 3) \\ (3 \times 1) + (0 \times (-1)) & (3 \times 2) + (0 \times 3) \end{bmatrix} = \begin{bmatrix} 1 & 7 \\ 3 & 6 \end{bmatrix}$$

Figure 1: Example Matrix Calculation

Matrix multiplications are applied extensively in machine learning [1], artificial intelligence, computer graphics, image recognition, quantum mechanics [2], and algorithms for solving systems of equations.
In education, matrix manipulation is often taught as an introduction to lay groundwork for more in-depth topics covered in computer science and linear algebra.

As a stepping stone for many computational tasks of the modern day, matrices have led researchers in the past decades to investigate ways to optimize the classical operation algorithm - characterized in the last century as the advent of "acceleration of matrix multiplication" [3].

Established algorithms like Strassen's algorithm formulated in the twentieth century are referred to as "fast matrix multiplications" because they reduce the number of required multiplication operations from 8 to 7 steps [4]. However, Strassen's only apply to square matrices with dimensions of $2^n * 2^n$ and extra addition operations are required. Winograd's algorithm is another established algorithm that reduces the number of multiplication operations by first transforming initial matrices into smaller ones before computing partial multiplications and combining the results [5].

A recent paper proposed a deep reinforcement learning procedure involving agents that "search for provably correct matrix multiplication algorithms" for efficient matrix multiplication on specific hardware. The researchers claim that finding the "low-rank decomposition of the matrix multiplication tensor" is a NP-hard problem. Duan and his colleagues from Tsinghua University investigated the optimal time complexity of matrix multiplication using the asymmetric version of the Coppersmith-Winograd's hashing method [6].

With newer system architectures, optimization can also be achieved through a combination of hardware and program parallelization techniques—formally called data parallelism [7]. Workarounds to optimization can be done with only slight modifications to the original matrix multiplication algorithm to account for parallel processing instructions[8].

2 ALGORITHM

$$A = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{21} & \cdots & b_{n1} \\ b_{12} & b_{22} & \cdots & b_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{1p} & b_{2p} & \cdots & b_{np} \end{bmatrix}$$

$$C = AB = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

Figure 2: Input and Output Matrix Dimensions

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

$$c_{23} = a_{21} b_{13} + a_{22} b_{23} + \cdots + a_{2n} b_{n3}$$

Figure 3: Matrix Multiplication Formula

The standard matrix multiplication algorithm requires a matrix of size *m* rows and *n* columns to be multiplied by a matrix of size *n* rows and *p* columns. The result matrix is filled in for each column of the first row, and this procedure is subsequently repeated on all remaining rows. The value at each index in the matrix is obtained by multiplying the first element in the row of matrix A with the first element in the column of matrix B. This number is added to a running sum. This process then repeats for each corresponding element of matrix A and matrix B. Newly computed values are added to the running sum at each index. When all elements of matrix A's currently used row and matrix B's currently used column are used in the running sum calculation, the element in the result matrix is set to the running sum.

The running sum is reset. This process repeats itself until a result matrix of *m* rows and *p* columns is filled.

The parallel matrix multiplication algorithm utilizes calculations in CUDA, writing those calculations to the result matrix in chunks at each run. The chunks of the result matrix are then combined. At the start of the algorithm, empty input matrices are initialized and copied from host (CPU) memory to device (GPU) memory using MPI I/O and the launch of a kernel for the device (GPU). The input matrices are then filled with CUDA's random number generator, cuRAND. These newly initialized matrices are then copied to the host (CPU) through MPI I/O and output into separate files to be stored on the system storage. The grid and block sizes for the CUDA kernel are set based on the matrix dimensions, distributing the workload across multiple thread blocks. Each of these threads computes one element of the result matrix based on its thread and block indices. This process follows closely from the standard matrix multiplication algorithm. That is, each of these threads iterates over the corresponding row of matrix A and column of matrix B, multiplying the elements and accumulating the sum, which is stored in the result matrix. After the GPU work is finished, the result matrix is copied from device (GPU) memory to host (CPU) memory using MPI I/O, which is written to system storage in parallel. The parallel matrix multiplication algorithm is as follows in figure 4.

Algorithm 1 Matrix Multiplication Kernel

```
function MatrixMultKernel(matrixA, matrixB, resultMatrix, m, n, k)
    row ← blockIdx.y * blockDim.y + threadIdx.y
    col ← blockIdx.x * blockDim.x + threadIdx.x
    if row < m and col < k then
        sum ← 0.0
        for i = 0 to n − 1 do
            sum ← sum + matrixA[row * n + i] * matrixB[i * k + col]
        end for
        resultMatrix[row * k + col] ← sum
    end if
end function
```

Figure 4: Parallel Matrix Multiplication Pseudocode

The serial algorithm for matrix multiplication closely follows the standard algorithm.

Algorithm 2 Algorithm 2: Serial Matrix Multiplication

```
procedure MatrixMultSerial(matrixA, matrixB, resultMatrix, m, n, k)
    for i ← 0 to m − 1 do
        for j ← 0 to k − 1 do
            sum ← 0.0
            for l ← 0 to n − 1 do
                sum ← sum + matrixA[i * n + l] * matrixB[l * k + j]
            end for
            resultMatrix[i * k + j] ← sum
        end for
    end for
end procedure
```

Figure 5: Serial Matrix Multiplication Pseudocode

3 Experiments

In order to test the performance for the MPI-CUDA code and serial code, we ran and timed the execution on the AiMOS supercomputer. A majority of the experiments were run on a single node of the supercomputer with the MPI code running on one IBM POWER9 processor and 4 NVIDIA Tesla V100 GPUs.

For our experiments, we tested the performance of the MPI-CUDA code with strong scaling and weak scaling tests in order to identify the effectiveness of our code with parallel I/O. The MPI-CUDA code is then compared with a serial implementation of matrix multiplication on the same processor.

3.1 PERFORMANCE RESULTS

3.1a MPI-CUDA Strong Scaling Experiments

Matrix Size (m x n)	Ranks	MPI I/O Runtime (seconds)	Total Runtime (seconds)
A: 512 x 1024 B: 1024 x 512	**2**	**0.0151**	**0.0159**
A: 512 x 1024 B: 1024 x 512	**4**	**0.02899**	**0.03118**
A: 512 x 1024 B: 1024 x 512	**8**	**0.0572**	**0.0608**
A: 512 x 1024 B: 1024 x 512	**16**	**0.11758**	**0.12248**
A: 512 x 1024 B: 1024 x 512	**32**	**0.21987**	**0.23899**

Figure 6: Strong Scaling Experiment Data

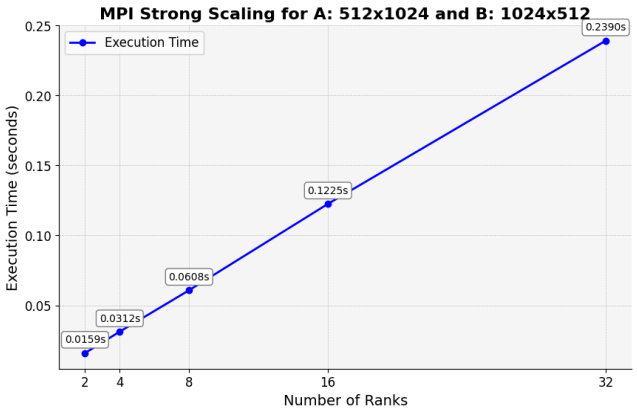

Figure 7: Strong Scaling Visualization

3.1b **Analysis of Strong Scaling**

In our strong scaling experiment, we kept the matrix sizes the same but increased the number of ranks we use to run the matrix multiplication computation. As we increased the number of ranks, the total runtime linearly increased. The reason behind this behavior is that the majority of the total runtime would be dedicated to reading and writing into a file using MPI I/O. As the number of ranks increases, the number of chunks of the result matrix that need to be written to a file using MPI I/O also increases. Roughly 95% of the total runtime is dedicated to reading and writing through MPI I/O. This suggests that there is massive overhead associated with reading and writing into a file as multiple processors would be doing the same operation. For the calculation execution time, the time would stay relatively the same as the rank increases.

3.1c **MPI-CUDA Weak Scaling Experiments**

Matrix Size (m x n)	Ranks	MPI I/O Runtime (seconds)	Total Runtime (seconds)
A: 32 x 64 B: 64 x 32	**1**	**0.006458**	**0.006870**
A: 64 x 128 B: 128 x 64	**2**	**0.012540**	**0.013200**
A: 96 x 192 B: 192 x 96	**3**	**0.019274**	**0.020288**
A: 128 x 256 B: 256 x 128	**4**	**0.026767**	**0.028176**
A: 160 x 320 B: 320 x 160	**5**	**0.032662**	**0.034023**
A: 192 x 384 B: 384 x 192	**6**	**0.039261**	**0.041767**
A: 384 x 768 B: 768 x 384	**12**	**0.082533**	**0.085972**

Figure 8: Weak Scaling Experiment Data

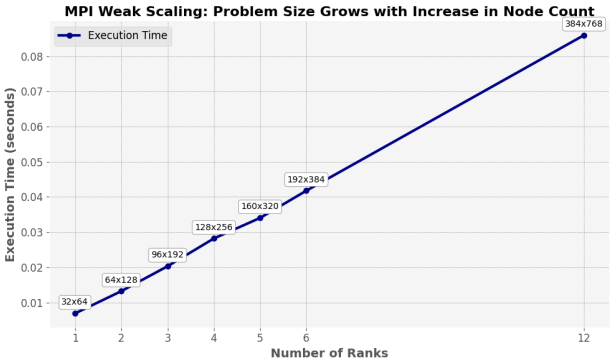

Figure 9: Weak Scaling Visualization

3.1d **Analysis of Weak Scaling**

In our weak scaling experiment, we both increased the matrix sizes and the number of ranks we use to run the matrix multiplication computation. Despite the ever increasing size of the matrices being multiplied, the time taken to calculate the result matrix was, as in the strong scaling experiments, proportional to the number of ranks. This indicates that most of the time was dedicated to using parallel I/O to read and write to files rather than computing the product of the two input matrices. The impact of increasing the two input matrices' size was negligible in comparison.

3.2 **MPI I/O Overhead vs. Computational Work**

Our findings from our strong scaling and weak scaling experiments both suggest that a majority of the time is spent on parallel I/O instead of real computational work. This is due to MPI parallel I/O placing a heavy load upon the drive as multiple processes attempt to read and write to the same file in different places at the same time. As the number of ranks in MPI goes up, more processes attempt to write a significant amount of data to the same file.

This overhead may be significantly reduced if the files only needed to be accessed by one process. This is especially true if every process needs to wait for another process to finish before proceeding. If the reads and writes were all done by a single process that uses scatter and gather, the overhead may be reduced significantly.

3.3a **Serial vs. Parallel Experiments**

Matrix Size (m x n)	Serial Time (seconds)	4 Ranks Parallel Time (seconds)
100 x 100	**0.007682**	**0.028027**
200 x 200	**0.124000**	**0.028025**
300 x 300	**0.408489**	**0.027752**
400 x 400	**0.990019**	**0.027774**
500 x 500	**1.916031**	**0.029182**
600 x 600	**3.373619**	**0.031581**
700 x 700	**3.696828**	**0.032945**
800 x 800	**7.665884**	**0.035124**
900 x 900	**10.849524**	**0.037628**
1000 x 1000	**14.820578**	**0.040141**

Figure 10: Serial vs. Parallel Data

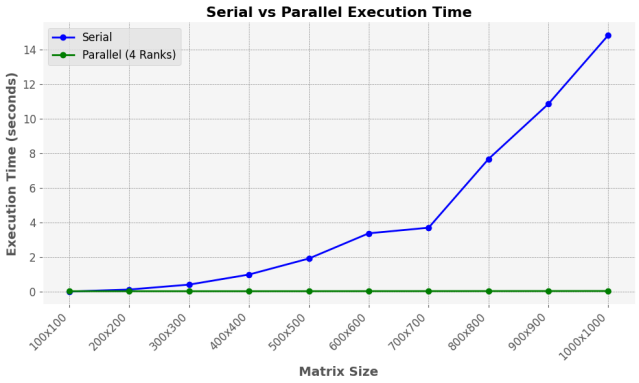

Figure 11: Serial vs. Parallel Visualization

3.3b **Analysis of Serial vs. Parallel Times**

As seen in our previous data, running the MPI program with 4 ranks drastically increased the runtime compared to having less ranks. Due to the I/O overhead in the MPI program, the serial code ran much faster than MPI when matrices of size 100x100 were multiplied. However, as the size of the matrices being multiplied together increased, the gap between serial and MPI runtimes quickly increased with the parallel code taking the lead. The times for serial seemingly grows exponentially.

3.4 **MPI vs. MPI-CUDA**

Our implementation involves the use of MPI and CUDA, which allows the host (CPU) to offload work to the (GPU). This combination allows for faster computation when using parallel techniques. Due to the limits underlying our CUDA code, more MPI ranks meant higher utilization of the GPU. That benefit was overshadowed in our code due to the massive parallel I/O overhead within our implementation.

As MPI ranks each run on a separate core on a CPU, a parallelized algorithm using MPI will still be faster than running in serial. This is due to the fact that having multiple cores at work would be faster than the serial code that has access to only a single core on the CPU. However, MPI-only code would run much slower than MPI and CUDA due to the fact that a GPU can perform much faster calculations than a CPU for parallel programs [9].

4 SUMMARY

4.1 **Instructions for running Code**

The modules required for compilation and running the code are Spectrum MPI and CUDA. In our case, we also used IBM's xl_r compiler. To load the modules, run ```module load xl_r spectrum-mpi cuda```. The serial code can be compiled with a C compiler where we used GNU's gcc. For compilation of the serial code, run ```gcc serial-matrix-mult.c -O3 -Wextra -o serial-matrix-mult.o```. Compilation of the parallel code can be done with our Makefile via ```make```. To run the parallel code, first allocate a node on a computing cluster or a supercomputer. Allocation can be done via ```salloc -N number_of_nodes_to_use -partition=el8 --gres=gpu:number_of_gpus_to_use -t time_in_minutes```. Our code takes 3 arguments that correspond to the sizes of the matrices: m, n, and k. To run the MPI-CUDA parallel code, use``` mpirun -np number_of_ranks ./matrix-mult m n k```. The maximum number of ranks corresponds to the total number of cores the processor has. To run the serial code, run ./serial-matrix-mult.o m n k```. Our code prints the generated input matrices and result matrix when m*k is less than 256.

4.2 **Conclusion**

In this study, we implemented a matrix multiplication algorithm in parallel using MPI and CUDA, as well as in serial. The parallel implementation leveraged the power of CUDA and MPI to distribute the workload across multiple GPUs and nodes, while the serial implementation served as a baseline for comparison. In our experiments, the parallel implementation significantly outperformed the serial approach for larger matrix sizes, highlighting the benefits of parallel

computing in accelerating matrix multiplication. However, parallel I/O proved to be detrimental to the runtime of the algorithm as the overhead grew with more processes. Although our goal was to calculate matrix multiplication in parallel, we ultimately ended up with a program that ran faster with less processes.

5 FUTURE WORK

In our experimentation, parallel MPI I/O took a significant portion of the computation. In the future, we can investigate more efficient reading and writing techniques for the input and result matrices across MPI processes to minimize communication overhead. We could also explore alternative communication patterns, such as non-blocking or collective communication, to further optimize the parallel performance [10].

In the future, we intend to explore using Strassen's algorithm [4], a divide and conquer algorithm, and the Coppersmith-Winograd algorithm [5] for matrix multiplication. The standard matrix multiplication algorithm is $O(n^3)$ in terms of time complexity. Strassen's algorithm, on the other hand, is $O(n^{2.8})$. The Coppersmith-Winograd algorithm is even more efficient at $O(n^{2.3755})$ [11]. The Coppersmith-Winograd algorithm is one of the asymptotically fastest known algorithms for multiplying two square matrices. Combining the power of these asymptotically fast matrix multiplication algorithms with the advantages of parallel computing and high performance computing (HPC) paradigms would introduce extremely large speedups [11].

Matrix multiplication is essential in numerous applications, from large-scale programs to research. Finding a solution that optimizes such operations can greatly benefit application runtime as a whole and make research more productive. Neural spline fields are a research area where networks are trained to map input coordinates to spline control points [12]. The network effectively fuses multiple images together into a single-high resolution picture. This process involves transforming camera coordinates, propagating activation layers, and projecting to higher dimensional space hash encoding, all of which facilitate numerous matrix multiplication operations.

REFERENCES

[1] Fawzi, Balog, Haung, et al. OCt 2022. "Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning." Nature.com Article, pp. 47 – 64.

[2] Changpeng Shao. Jul 2018. "Quantum Algorithms to Matrix Multiplication."

[3] Xiaohan Huang and Victor Y. P. Jan 1997. "Fast Rectangular Matrix Multiplication and Applications". Journal of Complexity 14, pp. 257 – 299.

[4] Paolo D'Alberto. Dec 2023. "Strassen's Matrix Multiplication Algorithm Is Still Faster " J. ACM

[5] Matthew Anderson and Siddharth Barman. Dec 2009. "The Coppersmith-Winograd Matrix Multiplication Algorithm".

[6] Ran Duan, Hongxun Wu, Renfei Zhou. Nov 2023. "Faster Matrix Multiplication via Asymmetric Hashing"

[7] Ghorpade, Parande, Kulkarni, et al. Jan 2012. "GPGPU Processing In CUDA Architecture." Advanced Computing: An International Journal (ACIJ), Vol 3, No.1.

[8] Olow Jimale, Ridzuan, Wan Mohd Nazmee, et al. 2019. "Square Matrix Multiplication Using CUDA on GP-GU." The Fifth Information Systems International Conference, pp. 398 – 405.

[9] Cooper-Baldock, Almirall, Inthavong. Apr 2024. "Speed, power, and cost implications for GPU acceleration of Computational Fluid Dynamics on HPC systems."

[10] Hoefler, Gottschling and Lumsdaine. "Leveraging non-blocking Collective Communication in high-performance Applications."

[11] Ambainis, Filmus, Le Gall. Nov 2014. "Fast Matrix Multiplication: Limitations of the Laser Method."

[12] Chugunov, Shustin, Yan, et al. Dec 2023. "Neural Spline Fields For Burst Image Fusion and Layer Separation."