Comparison of Matrix Multiply Parallelization Methodologies: MPI and MPI/CUDA

Shreyas Seethalla
Rensselaer Polytechnic Institute
Troy, New York, USA

Cynthia Gonzalez
Rensselaer Polytechnic Institute
Troy, New York, USA

Weichao Li
Rensselaer Polytechnic Institute
Troy, New York, USA

Jesse Huang
Rensselaer Polytechnic Institute
Troy, New York, USA

ABSTRACT

The study of parallel message passing, CUDA, and their combination is important for understanding how common, but computationally expensive, operations can be parallelized. Matrix multiply is one such operation; as the dimensions of the matrices increases, the computation becomes expensive. In this paper, we research and compare three implementations of matrix multiply with square matrices: a serial version with solely CPU usage and two parallelized versions with usage of the CPU and MPI and a hybrid CPU/GPU MPI implementation. MPI was integrated with OpenMPI to parallelize chunks of the multiplication across processes. Parallelization with the GPU was implemented through the CUDA library. Benchmark data on how efficiently each version performs with matrix multiplication in a strong and weak scaling setup is provided. The performance of MPI I/O with the MPI (non-CUDA) cases was also investigated. Strong and weak scaling experiments were conducted on the Artificial Intelligence Multiprocessing Optimized System (AiMOS) supercomputer with the system's built-in MPI and CUDA modules. Within the benchmark are smaller and larger test cases that are analyzed based on the time for the matrix multiplication and the MPI message passing overhead for the parallelized versions. MPI I/O was executed on the computer's NVMe storage.

1 TEAM AND CONTRIBUTION

The team members for this project are Shreyas, Cynthia, Jesse, and Weichao. As a team, we would meet during class time, which was converted to office hours, to allocate the various tasks of the project among the group members, work together on the code, and figure out portions of the project we were stuck on with help from the professor. We would stay until the end of class to ensure that we have made enough progress to work on portions of the project individually.

Jesse helped with the debugging of the MPI version of the matrix multiply code, wrote the MPI/CUDA hybrid version, added MPI I/O, and helped with the collection of timing data for the MPI and MPI-CUDA versions. He also contributed to the Abstract and Introduction sections, wrote the code and algorithm implementation details section, and assisted with the data collection.

Cynthia contributed by outlining the paper. She worked on a significant part of the paper portion of the project, mainly looking for and summarizing related works that give further understanding to the project. She then worked on the Introduction, and the implementation that describes what the process was behind the project. She also created the works cited section.

Shreyas created the MPI and serial implementations of the matrix multiply code. He also setup the GkeyllZero project but due to later discussed issues had to switch topics. He collected most of the timing data and created all the plots. He wrote the performance results section of the report.

Weichao contributed by adding in information from half of the related papers for the related work/literature review section, wrote part of the introduction, and helped to finish the final report. He also used an ACM proceedings latex template to create the latex document (this report), transferred written and figure content from other documents used by the group to the latex document, and tabularized the data.

2 INTRODUCTION

In this section, a description of the parallel technologies studied and utilized are provided. These technologies are CUDA, MPI, and MPI I/O. An overview of matrix multiplication is also provided.

2.1 CUDA

CUDA [19, 25, 30], also known as Compute Unified Device Architecture, is an architecture created by NVIDIA that allows users to make use of GPUs in parallel. It is also broken up into host and device which in C code is able to be utilized by the "global" variable. The simple process of Cuda is that the hardware connects the GPU to the CPU. The NVIDIA compiler is then able to run the program with or without a device [4]. It allows the user to compile two in one. It is invoked in a GCC like way where there would be a host compiler and Cuda compiler that communicates with the GPU. Along with the host and device, there are also threads, blocks, and synchronizations to be considered while using Cuda.

2.2 MPI

MPI [12, 14, 29], also known as Message Passing Interface, has a goal of passing messages through reading and writing data. This standard library is mainly used within programming languages such as C and C++. Unlike Cuda, there is no memory sharing at this point. Each process and rank within MPI has its own independent address space. Even with differences in Cuda and MPI, they both are able to provide parallelism on a computer. The idea behind MPI

is that it takes read and receives and only returns when it is safe to do so. Once that is the case, send will not return until the matching receive is done with its own process.

2.3 MPI I/O

In addition to its message passing and process communication API, which allows for CPU-core based parallelism, MPI contains an API for input/output (I/O), known as MPI I/O [8, 27]. The provided C I/O does not provide direct support for the reading or writing of data across MPI processes; the built-in C I/O is sequential. Without a system developed and integrated within MPI for I/O, I/O can become a bottleneck in parallel programs [5, 9]. With MPI I/O, processes are able to read to or write from files in parallel with the associated performance benefits. In addition to fundamental file operations, MPI I/O contains a means for I/O to be done in a collective manner across MPI ranks as well as a framework for synchronizing I/O operations [6, 26].

2.4 MPI/CUDA

To garner the full benefits of a computational system, MPI and CUDA can be combined [17, 18, 23]. Both parallel technologies are relatively mutually exclusive in terms of what hardware resources are utilized to parallelize computations. MPI spreads operations across CPU cores while CUDA does so across GPU threads. A system with CPU(s) and GPU(s) can take advantage of parallel programs that employ both MPI and CUDA. Although there is the potential for message passing and synchronization bottlenecks, for computationally intensive problems, it can be surmised that the integration of the two parallelization APIs can further reduce the computational complexity by up to a factor of both the number of available CPU cores and GPU threads [28]. This potential in combined performance advantages is the rationale behind the implementation, study, and analysis of the MPI/CUDA version of matrix multiply.

2.5 Matrix Multiplication

Matrix multiply is an elementary matrix operation whereby two or more matrices are multiplied together. For matrix A with dimension M x K, matrix B with dimension K x N, the matrix multiplication of A and B produces a result matrix C with dimension M x N. Elements c_{ij}, the i^{th} row of C and the j^{th} column of C, are computed with the summed inner product of the ith row of A and the jth column of B, both of which contain K elements each.

$$c_{ij} = \sum_{i=1}^{K} a_{ik} * b_{kj}$$

a_{ik} is the i^{th} row and k^{th} column of matrix A and b_{kj} is the k^{th} row and j^{th} column of matrix B. The second dimension of matrix A and the first dimension of matrix B must match for A and B to be multiplied, in that order.

3 IMPLEMENTATION

Our initial implementation consisted of us transferring GkeyllZero into AiMOS which was originally being used on Frontera. During this process, there were adjustments that needed to be made in order to run tests. One important factor was ensuring that the code was running on CUDA 12.1 as well as ensuring that the other modules were installed onto the system as well. During the module load, there were some issues that came along. One of the modules that needed to be loaded in was hpcx. Some of the students were unable to load that module as it is packaged within another one. After the name of that module was retrieved, hpcx could be loaded in. We then had to figure out why the test runs were working, but there were inconsistencies. One student was processing the test cases quickly, but could not produce the output of any data from the test runs as it would fail at that point. Another student was able to process everything, but it took a very long time to run each process. We then looked for other ways to run GkyellZero, but to no avail.

After spending some time on trying to run GkeyllZero, we then switched over to Matrix Multiply using MPI, CUDA, and MPI I/O. In this circumstance, we then wanted to see if we could work with cuBLAS which is a linear algebra library that would help run matrix multiply a bit faster than us just using CUDA and MPI. There then was a compilation issue which was a merge of spacing errors combined with implementing variable changes. Wherever we used "global" had to be removed because cuBLAS is based on code from the host, the CPU, not the device, the GPU. Another encounter we came across after being able to compile the code was that the outputs were slightly off. After searching through the code there was an inconsistency between the outputs. The cuBLAS was not properly producing the correct outputs as we compared it to a run that was already correct. cuBLAS seems to require the input matrixes to be 2-D. However, in order for the MPI broadcast operations to function correctly with our MPI implementation, we had to allocate memory for the matrices in a 1-D setting. In order to solve this problem with cuBLAS usage, one would have to do a wrap around in the CUDA file to make it a 2-D Matrix. Due to the amount of time for the project, it could not be implemented. We then decided on doing weak and strong scaling using CUDA, MPI, MPI I/O, along with block size tests. It is also to be noted, while considering the clock cycle, we did not use the MPI I/O read operation. There was no data generated externally from files that had to be read in. It then became unnecessary for us to utilize it.

3.1 Implementation Details - Code and Algorithm Description

The implementation consists of three versions of square matrix multiply: a serial version as a baseline comparison, a MPI-only version, and a MPI-CUDA hybrid version. Given the use of N by N square matrices, let A = the first N by N matrix, B = the second N by N matrix, and C = the resulting N by N matrix from multiplying matrix A with matrix B. Memory allocated for the matrices is two-dimensional in the serial version and one-dimensional in the MPI versions for compatibility with MPI collective message-passing features. For simplicity, we reference the contextualization of the matrices as two-dimensional throughout our discussion of the code implementation details. The serial version is comprised of one .c file (matrix.c) that takes as input N; the MPI-only version is comprised of one .c file (matrix-mpi.c) that takes as input N with the number of processes specified with MPI; and the MPI-CUDA hybrid version

is made up of one .c (matrix-mpi-cuda.c) and one .cu (matrix-mpi-cuda.cu) file, which takes as input N and the number of threads per block with the number of processes specified with MPI. The MPI-CUDA hybrid version has an additional .cu file to use to the CUDA device-side syntax, variables, and functions.

The serial version contains a baseline implementation of matrix multiply in which for each cell c_{ij} in the result matrix C, the value is computed by taking the summed inner product as described in section 2.5. This operation yields an asymptotic execution time of $O(N^3)$ for the square matrices. Given the third degree quadratic execution time, without any parallelization, matrix multiplication can be considered a computationally expensive procedure.

For both of the MPI versions, MPI is used to split the computation of the matrix multiplication into chunks. Each MPI rank receives an equal size chunk of matrix A and the entirety of matrix B and computes the resulting chunk matrix. For R ranks r_i, each rank gets rows $r_i, r_i + 1, \ldots, r_i + \frac{N}{R} - 1, r_i + \frac{N}{R}$ from A, a $\frac{N}{R}$ by N matrix. The chunks of matrix A are sent to each process with MPI_Scatter, and matrix B is sent to each process through MPI_Bcast. MPI_Barrier is invoked to synchronize the send and receive of the chunks of matrix A and matrix B across the processes. Each rank computes an $\frac{N}{R}$ by N resulting matrix. In the MPI-only version, this computation is done sequentially in the same manner as with the serial version; in the MPI-CUDA version, this computation is done in parallel with CUDA. A second MPI_Barrier invocation occurs to synchronize the matrix chunk computations before the result matrices are collected into the final result matrix C to process 0 with MPI_Gather. The resulting sequence for the MPI-rank parallelized matrix multiplication is as follows: MPI_Scatter, MPI_Bcast, MPI_Barrier, sequential or CUDA-parallelized matrix multiplication on chunk, MPI_Barrier, MPI_Gather. Without taking into account the overhead from message passing or synchronization as the asymptotic complexity of those two operations is unknown, the parallelization with MPI-only can improve the big-O execution time for matrix multiplication to $O(\frac{N^3}{R})$.

CUDA was employed with the matrix multiplication of the chunk of matrix A per process. The inner product matrix multiplication was parallelized with CUDA for each of the cells of the result matrix. CUDA blocks and threads were utilized such that the inner product computation occurs simultaneously across the GPU threads, a hardcoded CUDA implementation. Each GPU thread is responsible for computing the value of a cell in the result matrix C with the respective inner product. For GPU thread t responsible for cell c_{ij} from C, it takes the summed inner product of the i^{th} row of A from the chunk received by the host process and the j^{th} row of B from the host process. Combining CUDA with the MPI implementation as another standalone version and without accounting for overhead as with the asymptotic analysis with the MPI-only case, the execution time of the matrix multiplication can be further reduced to $O(\frac{N^3}{RTB})$ where T = the number of GPU threads per block and B = the number of GPU blocks.

MPI I/O was used to write the result matrices from each process to a file in parallel. Specifically, each rank r_i writes their $\frac{N}{R}$ by N result matrix to the file at location $r_i \frac{N}{R} N$ with MPI_File_write_at (written to file as a one-dimensional flattened version since the memory allocated for the matrix is one-dimensional for the MPI cases). Before the MPI_File_write_at invocation at each process, atomicity is set to 1 to ensure the correctness of the parallel file writing (not 100% sure about exactly what MPI_File_set_atomicity does). After the write is complete at each process, the MPI_Barrier operation is invoked to synchronize the processes. The resulting sequence for the MPI-rank parallelized I/O is as follows: MPI_File_open, MPI_File_set_atomicity, MPI_File_write_at, MPI_Barrier, MPI_File_close.

Lastly, all the timing data was gathered using inline machine code to get cycle counters available on the POWER9 processors. Using the clock_now() function, we got a cycle count (with a resolution of 512 Mhz that could be converted to seconds. We isolated the matrix multiplication steps and the IO steps when timing them to avoid counting other parts of the code. We also timed the whole real execution time to calculate the communication overhead (time spent not performing the multiplication of the matricies).

4 RELATED WORK/LITERATURE REVIEW

In the project, we attempted to integrate cuBLAS into our GPU-based matrix multiplication implementation; however, KBLAS is a more modern version of this library that has seem to have improvements in performances and functionalities regardless of sizes of the models being used. The performance of KBLAS which is scaled by operational intensity along with the memory bandwidth, is then tested against other libraries that exist which include cuBLAS. They compare through the utilization of the matrix multiply to see how well each library will perform. The operational intensity is what their main focus is on. This takes the number of FLOPs divided by the number of bytes read and written to memory [1]. They also utilize a portion of KBLAS and merge with cuBLAS and test for performances of that as well. There are also different levels of BLAS operations that can be used, and based on those operations, the complexity of the operation can vary. For example, the researchers use Level 2 BLAS which deems more challenging as there can be a "lack of data reuse to compensate for data transfer overhead" [1]. One downfall of an older cuBLAS was how well it could handle symmetry, however, in a current version of the library, it should be able to handle that problem a bit better. They have also determined that KBLAS also works well in every GPU tested as long as there are parameter turnings that take place.

While we have dicussed CuBLAS, other researchers also utilize similar libraries such as Blas. One paper discusses the Design and Performance of Batched BLAS on Modern High-Performance Computing. This paper explores optimizing batched BLAS routines for small matrices on modern parallel architectures [11]. It addresses the trend of decomposing large linear algebra problems into smaller batches for parallel processing. Evaluating existing batched BLAS proposals, it focuses on enhancing performance, particularly for GEMM. The study investigates optimizations, including API designs and memory layouts, to improve batched BLAS performance. Despite support in libraries like Intel MKL and NVIDIA CuBLAS, the lack of a standardized interface poses challenges. The article assesses different APIs, examines associated overheads, and explores a novel data layout to enhance performance on GPUs and Intel KNL architectures. Finally, it concludes with remarks on the findings and future considerations.

Following onto the topic of Cuda, there is a popularly cited paper called Efficient Sparse Matrix-Vector Multiplication on CUDA. This related work discusses certain algorithms of matrix multiply used along with CUDA to test efficiency of Sparse Matrix Vector Multipliers [3]. One of these popular structures being the CSR. They also test a range of different structures from regular to irregular to provide the reader with evidence of how efficient the SpMV will work on a GPU. Their conclusion that they are trying to draw is that they believe that aligning the memory will allow for more efficiency in the system. One of their ideas implemented is utilizing different structures of the matrix multiply and seeing how changes in the threads will affect memory and performance[3]. In a hybrid version of the matrix multiply, which consists of ELL and COO, there seems to be better performance although there is not as much structure. Even with this hybrid model not being the strongest performing architecture, they believe that with some tuning, this model will still prevail as the better performing model. An improvement to their work that they would like to implement would be digging deeper into block formats which they believe can benefit in beginning better performance speed with the correct tuning.

Elaborating on the idea of Cuda a bit further, another work called CUDA-enabled Sparse Matrix–Vector Multiplication on GPUs employs atomic operations [10]. The goal of this work is based on interest in Sliced COO, an implementation of CUDA while being compared against other formats such as CSR, COO and HYB in multiple GPUs that are CUDA-enabled [10]. Some of these comparisons utilize atomic operations which is when "a thread can perform memory transaction without the interference to the memory address from any other thread"[10]. This implementation is tested against the matrix multiplier on GPU as well as a dual-GPU, but it was also stated that they also ran their experiments on CPU as well. Their results show that SCOO does not always outperform in every circumstance which could be because of the double precision used within the matrix. Although it does not always outperform, the overall performance of SCOO has better speedup than the CUSP formats utilized in comparison for single precision performances. Other tests include double precision, memory usage, as well as texture cache hit rate in which SCOO still performs well against other formats even with some of its drawbacks [10]. The hardware also affects the implementation of the program as it has been noted that there is an improvement in performance of SCOO when one is using the hardware called K20c GPU. The overall idea of the paper analyzes that SCOO performs better in single-precision versus double-precision which is something they would like to investigate further.

While using Cuda can be optimal, many researchers have been looking as combining Cuda with MPI to increase performance rates. This next research is about Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters [31]. It aims to take a look at OpenMP, MPI, and CUDA to create better speedups through their algorithm by analyzing the weights of each node being utilized [31]. Their work contains an approach in which there is an interaction dependent on the weight of multi-core nodes. One of their highlights of the work would be their hybrid CUDA clusters which they prove helps their performance rates. In their algorithm, they also state how they resort to the cuda compiler NVCC although it can be more involved than mpicc [31]. This is due to it being able to handle host and device allowing it to be easier for their hybrid model to process. They also provide proof that the performance of the GPU outweighs the CPU even with multiple threads. While comparing MPI and OpenMP against CUDA, it is noted that performance within the GPU consistently excels. Further, they also test MPI and OpenMP which spew similar results.

Observing how Cuda works with MPI and OpenMP it is also important to investigate how MPI and OpenMP work without Cuda as well. In the following paper of Analysis of performance of Matrix Multiplication Algorithms Using MPI and OpenMP [15], it ties together OpenMP and MPI,also known as hybrid programming. It tests on CPUs within a cluster on matrix multiplication. The algorithm used to utilize the hybrid program is called Fox's Algorithm which takes each process and breaks them up into multiple stages. To test their idea, they look to see how well the program will perform as the matrix increases in size. In their results, it is shown how their hybrid experiment outperforms both the sequential code being run as well as sequential that contains openMP. They also take note of the Hybrid against Fox Algorithm which shows that past 500x500, the Fox Algorithm begins to perform a bit better[15]. Their reasoning behind this is because one of the matrices got shared to all the processes [15].

Continuing onto the idea of MPI on matrix multiplication, another work considers a 2.5D Algorithm and One-Sided MPI to increase the Efficiency of Sparse Matrix-Matrix Multiplication [20]. This study extends the DBCSR sparse matrix library to investigate communication-reducing 2.5D algorithms and one-sided MPI communication for linear scaling electronic structure theory. Specifically, the focus is on block-sparse matrix-matrix multiplication crucial for linear scaling electronic structure theory and correlated methods in CP2K. Comparisons are made between the original Cannon's algorithm with MPI point-to-point communication and an approach using MPI one-sided communications (RMA) in both 2D and 2.5D frameworks. Results demonstrate substantial performance improvements, particularly with the RMA-based 2.5D algorithm, with speedups up to 1.80x, increasing with the number of processes involved.

Focusing more on GPU research, one work focus on Efficiency of GPU Algorithms for Matrix-Matrix Multiplication [13]. This study examines GPU performance for numerical computations, focusing on dense matrix-matrix multiplication. Despite parallel computational demands and a regular data access pattern, achieving optimal GPU performance remains challenging. Existing GPU algorithms suffer from bandwidth limitations, hindering their efficiency compared to cache-aware CPU approaches. While efforts have improved GPU algorithms, achieving better performance will require architectural changes.

One work focus on Optimizing sparse matrix-vector multiplication on GPUs [2]. This paper addresses the optimization of SpMV kernels on NVIDIA GPUs using CUDA, focusing on synchronization-free parallelism, optimized thread mapping for optimal memory access patterns, efficient off-chip memory access to mitigate latency, and data locality exploitation. Evaluation on GeForce 8800 GTX and GeForce GTX 280 GPUs demonstrates significant performance gains compared to existing parallel SpMV implementations, outperforming CUDPP and segmented scan implementations by

a factor of 2 to 8 and achieving parity or up to 35% improvement over NVIDIA's SpMV library.

As there are multiple articles focusing on GPUs, another work for GPU computing performance analysis on matrix multiplication focuses on GPU performance analysis in machine learning, particularly in matrix multiplication—a computationally intensive task [16]. This paper aims to elucidate the relationship between GPU performance, matrix scale, and development methods. The experiments reveal insights, showing that in small-scale computations, GPUs may not exhibit significant improvements over central processing units (CPUs). Additionally, they highlight the efficiency disparity between high-level application programming interfaces (APIs) and low-level GPU programming languages like CUDA. This study uncovers factors contributing to GPU performance variations in machine learning, with matrix multiplication as a critical benchmark for assessing high-performance GPU computing.

As we are implementing MPI-CUDA, another related paper discusses MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. This paper focuses on implementing the Preconditioned Conjugate Gradient (PCG) solver and the Approximate Inverse Preconditioner (AIP) for sparse linear systems on hybrid computing systems with multiple CPUs and GPUs interconnected via MPI [22]. Emphasis is placed on optimizing the Sparse Matrix-Vector Multiplication (SpMV) operation, the most time-consuming part of the algorithm. The authors propose a data transfer overlapping approach to minimize communication overhead between GPUs, resulting in significant speedup compared to CPU-only implementations. The paper is organized to cover PCG solver fundamentals, application context, parallelization strategies, CUDA model, multi-GPU implementations, performance results, and concluding remarks, offering insights into enhancing sparse linear system solvers in hybrid computing environments.

Continuing on the topic of solely CUDA, another paper talks about general-purpose sparse matrix building blocks using the NVIDIA CUDA technology platform. The author presents an extensive matrix algorithmic performance study on GPUs using the novel NVIDIA CUDA technology platform to build general-purpose sparse matrix building blocks [7]. This paper uses a GPU as a mathematical co-processor to accelerate sparse direct linear solvers. The stream computing unit is based on the NVIDIA GeForce 8800 which has a scalable ultra-threaded architecture, high performance parallel processing on 128 shader processors and is equipped with 768 MB on-board memory. The aim of this paper is to investigate the performance acceleration of dense and sparse matrix solution kernels. Two fundamental computational kernels have been developed: one is a sparse direct linear factorization method for nonsymmetric and symmetric indefinite matrices based on the PARDISO framework and another is an interior-point optimization solver for large scale non-convex PDE-constrained optimizations.

Another GPU related work talks about automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. The author presents a new storage format for sparse matrices that better employs locality to solve the problem that traditional sparse matrix algorithms are difficult to efficiently parallelized for GPUs due to irregular patterns of memory references [21]. This new framework can do the sparse matrix-vector product on NVIDIA GPUs under no specific assumptions about the structure of the sparse matrix.

In a following related work on GPU and Cuda, this work focuses on gaining performance speed using multi-threading on a GPU, specifically the GeForce 8800. Their objective mainly focuses on ensuring that the the active threads and the amount of usage of the threads does not interfere with their performance speed. [24] Their work is then tested on the matrix multiplier to spew their results. One way they work on resolving resources on a thread is through the usage of tiling which allows for threads to share memory locally in a system. They also had some drawbacks, which consisted of reducing the amount of threads being used by a third. With this drawback came better optimization as the threads would produce better speedup. Although they have some pros and cons to the tuning of the architecture of the threads, they hypothesize within future works that they will have significant optimizations by utilizing other features or compilers as well as experiementing with the code on different processors [24].

5 PERFORMANCE RESULTS AND ANALYSIS

Our baseline test for matrix multiplication is a 4096×4096 matrix size for all 3 matrices in question. In the tables and graphs, N refers to the matrix size; the matrices multiplied and the result matrix are each of dimension N x N. Performing the multiplication in a serial C code is time consuming, so a smaller test case with 1024x1024 matrices was done. The multiplication took around 10.7 seconds. Given that the matrix is 4 times smaller, the multiplication algorithm would take roughly 64 times longer to run, approximately 685.5 seconds. Since the matrix multiplication algorithm is $O(N^3)$, this factor was determined by dividing the matrix sizes cubed by each other. The below table summarizes and compared the serial result to the best parallel results for both MPI and MPI/CUDA.

5.1 Strong Scaling

MPI Strong Scaling: Looking at the scaling of our MPI implementation as we increase the number of ranks while keeping the matrix sizes constant, we can see that with every doubling of ranks, the multiplication time halves. But looking at the MPI NVMe I/O write time, we see an initial drop but after the ranks increase beyond 16, the code takes longer the write the data. This initial drop is likely due to the increase in ranks that allow for more parallelized performance benefits. But as more ranks are being used, more nodes in the cluster are required. The CPU-CPU communication is a lot slower than communicating between CPU threads and cores. We can see this issue exacerbate when jumping to 64 ranks which requires 2 nodes. Communication between nodes is even slower than between CPUs and the write time likely suffers from this. Looking at the communication overhead (disabling parallel IO), we see that the time spent not multiplying drastically increases up to 30% of the total execution time. This is likely due to the decreasing multiplication time meant that a larger portion of the total execution time is the communication overhead, which doesn't decrease nearly as much with increasing ranks. In fact, the overhead increases as more ranks need to communicate during the MPI_Scatter and MPI_Gather calls. Figures 1, 2, 3 and table 1 show these results. In figure 3, the left y-axis, the communication overhead for multiply, is log-scaled.

Table 1: Performance of MPI strong scaling with N = 4096

MPI Ranks	Multiply Time (sec)	NVME IO Write Time (ms)	Communication Overhead for Multiply (sec)	Percentage of Overhead
1	7.77E+02	7.11E+00	1.88E-02	2.42E-03
2	3.92E+02	5.39E+00	3.03E-02	7.74E-03
4	2.37E+02	5.80E+00	5.73E-01	2.71E-01
8	1.01E+02	1.90E+00	7.02E-02	6.98E-02
16	5.16E+01	2.44E+00	1.01E-01	1.96E-01
32	2.78E+01	1.72E+00	1.10E+01	2.84E+01
64	1.32E+01	7.45E+00	6.08E+00	3.15E+01

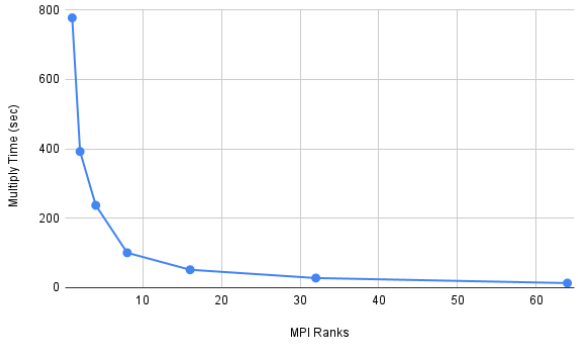

Figure 1: MPI Matrix Multiply Time (Strong Scaling, N = 4096)

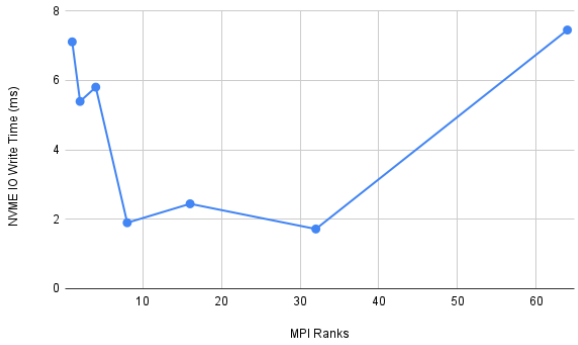

Figure 2: Matrix Multiply MPI NVMe I/O Write Time (Strong Scaling, N = 4096)

MPI-CUDA Strong Scaling: Looking at our MPI-CUDA implementation, we see similar results with the execution time decreasing steadily as we increase the number of MPI Ranks and GPUs to match (1 GPU per MPU rank). The rate of the decrease isn't as much as with the pure MPI implementation but using a GPU is vastly faster than just a CPU regardless. Looking at the communication overhead, it increases much slower and predictably compared to the pure MPI case. One exception being the 2 rank/GPU configuration

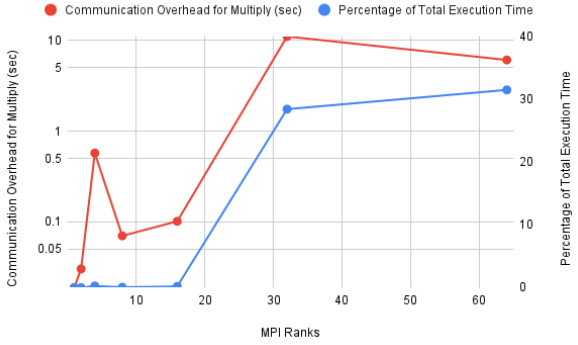

Figure 3: MPI Matrix Multiply Communication Overhead (Strong Scaling, N = 4096)

(Figure 5), which sharply increased the overhead to about 1 second and 3% of the total execution time. We believe this is because the node which has 4 GPUs is only partially allocated, so other running programs can interfere. For the other cases afterwards, each node used was completely allocated by our code since they are all multiples of 4. In all of these tests and all future CUDA tests, we chose to use **1024 threads per block** as our CUDA thread block size. Table 3 shows our results. In most cases, 1024 threads/block had the fastest execution time, and when this wasn't the case the difference was extremely small. Figures 4 and 5 and table 4 show the results for MPI-CUDA strong scaling.

Table 2: Summary of Best Matrix Multiplication Performance (Strong Scaling, N = 4096)

	Execution time (sec)
Serial	7.77E+02
MPI (64 ranks)	1.32E+01
MPI CUDA (32 GPUs)	4.44E+00

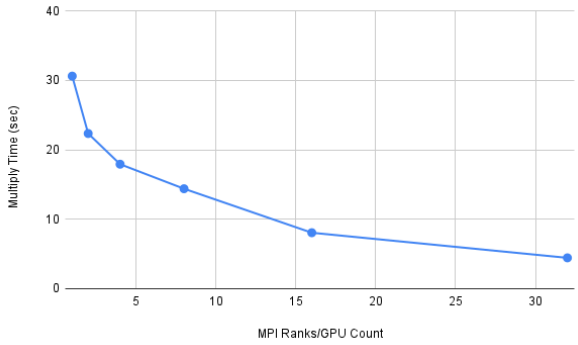

Figure 4: MPI-CUDA Matrix Multiply Time (Strong Scaling, N = 4096)

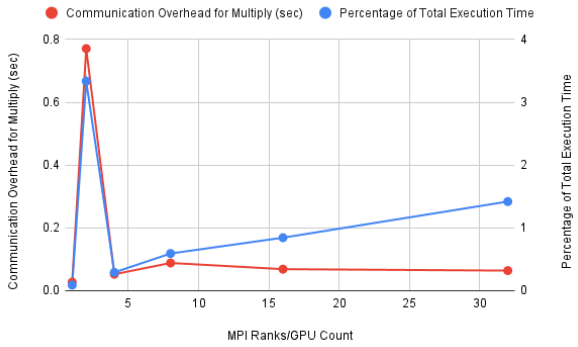

Figure 5: MPI-CUDA Matrix Multiply Communication Overhead (Strong Scaling, N = 4096)

Table 3: Matrix Multiplication Runtimes for Blocksize Configurations (sec)

MPI Ranks/GPU	Blocksize (threads/block)			
	16	256	512	1024
1	\	\	30.6	30.6
2	\	\	22.0	22.3
4	21.5	19.8	17.8	14.8
8	\	\	14.8	14.4

5.2 Weak Scaling

MPI Weak Scaling: When looking at weak scaling, we gave each MPI rank 512×512 cells and increased as we added more ranks. We saw a quick increase in the multiplication time, getting more exponentially larger as we added more ranks. Due to the student allocation limits, we were unable to run the 64 rank case. Using the execution time differences, we estimate that the code would take about **36000 seconds** not taking into account added overhead with using more than 1 node.

As the matrix size doubles, the number of elements increases by a factor of 4. So the increase in the number of elements being written overtakes the speedup from increasing the number of ranks. We can see this in the similarly exponential increase in parallel IO time. While not as sharp, the write time is definitely dominated by the matrix size. If we look at the communication overhead plot (left y axis is log-scaled) in figure 7, we can see that there is a similar shooting up of the overhead time and percentage of total runtime at 32 ranks. Figures 6, 7, and 8 and table 5 show the results for MPI Weak Scaling.

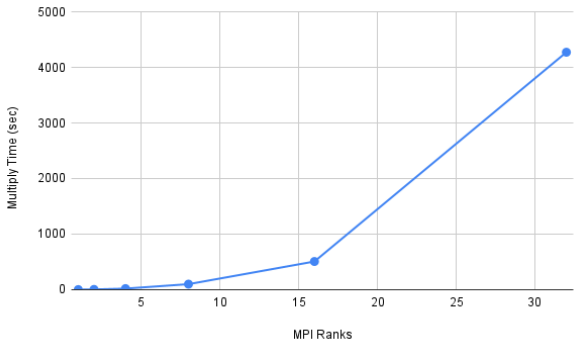

Figure 6: MPI Matrix Multiply Time (Weak Scaling, N = MPI Ranks * 512)

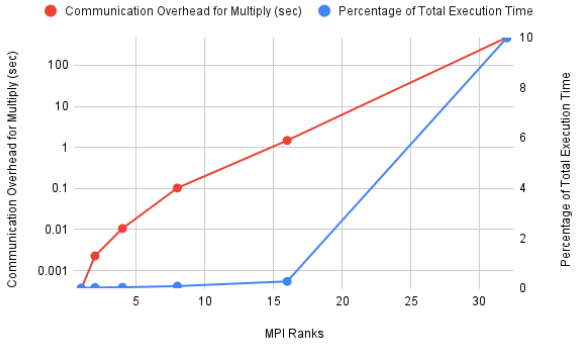

Figure 7: MPI Matrix Multiply Communication Overhead (Weak Scaling, N = MPI Ranks * 512)

MPI-CUDA Weak Scaling: Looking at our CUDA implementation, there is still an increase in the time taken to multiply but the increase isn't nearly as rapid as the pure MPI implementation. There is less serial CPU work to be done so the parallelization is able to keep up more. The overhead plot is similar to the strong scaling case. When we move to 32 ranks and 2 CPU's, it takes longer to communicate between CPUs and that dominates the overhead. But at over 500 seconds, we believe that there are external factors that caused this. The data for MPI-CUDA Weak Scaling is in figures 9 and 10 and in table 6. The left y-axis, the communication overhead for multiply, is log-scaled in figure 10.

Table 4: Performance of MPI-CUDA strong scaling with N = 4096

MPI Ranks/GPU	Multiply Time (sec)	Communication Overhead for Multiply (sec)	Percentage of Overhead
1	3.06E+01	2.72E-02	8.92E-02
2	2.23E+01	7.71E-01	3.34E+00
4	1.79E+01	5.23E-02	2.91E-01
8	1.44E+01	8.83E-02	5.92E-01
16	8.06E+00	6.86E-02	8.44E-01
32	4.44E+00	6.39E-02	1.42E+00

Table 5: Performance of MPI weak scaling with N (matrix size) = MPI Ranks * 512

Matrix Size	MPI Ranks	Multiply Time (sec)	NVME IO Write Time (ms)	Communication Overhead for Multiply (sec)	Percentage of Overhead
512	1	1.48E+00	3.05E-01	3.62E-04	2.44E-02
1024	2	4.75E+00	6.13E-01	2.27E-03	4.79E-02
2048	4	1.96E+01	9.74E-01	1.06E-02	5.40E-02
4096	8	9.96E+01	2.23E+00	1.02E-01	1.03E-01
8192	16	5.07E+02	6.30E+00	1.47E+00	2.90E-01
16384	32	4.27E+03	2.28E+01	4.74E+02	9.99E+00

Table 6: Performance of MPI-CUDA weak scaling with N (matrix size) = MPI Ranks (GPU Count) * 512

Matrix Size	MPI Ranks/GPU Count	Multiply Time (sec)	Communication Overhead for Multiply (sec)	Percentage of Overhead
512	1	8.17E-01	4.07E-04	4.98E-02
1024	2	1.07E+00	3.46E-01	2.45E+01
2048	4	2.92E+00	8.18E-01	2.19E+01
4096	8	1.44E+01	8.83E-02	5.92E-01
8192	16	5.73E+01	2.40E-01	4.16E-01
16384	32	1.02E+02	5.20E+02	8.36E+01

6 SUMMARY

To reiterate, we took a closer look at the analysis of performances with serial, MPI and MPI/Cuda within matrix multiply. We also tested weak scaling benchmarks in which the number of MPI ranks and GPUs used changed with constant matrix size and and strong scaling benchmarks in which the matrix size scaled with the number of MPI ranks or GPUs. To further explain, we looked at the time for the matrix multiplication computation and the write operation with MPI I/O. We also studied message passing overhead with MPI with both the time spent in broadcast and synchronization while also focusing on the percentages of the time with respect to the total execution time of matrix multiplication. Another portion of our work looks at determining the performance impact of increasing the parallelism with the same matrix size as well as the parallelism with the proportionally scaled matrix size.

To begin, we conducted a separate performance investigation of MPI I/O with the MPI-only version in both, the strong and weak scaling cases. From that implementation we note the time for parallelized MPI I/O write operation of each rank's matrix chunk result to a file, which forms a full result matrix in the file. Adding onto that, the overhead time of the MPI synchronization was not measured with MPI I/O. This is because the measured time is only the write operation along with the necessary settings for the operation's correctness.

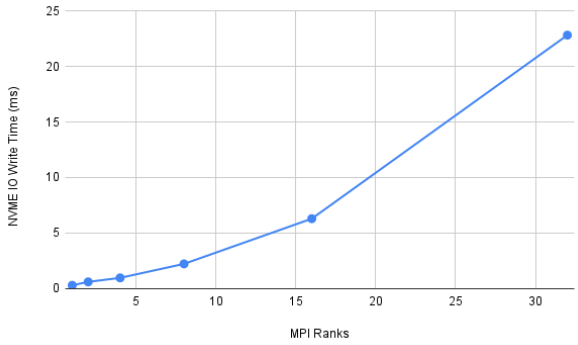

Figure 8: Matrix Multiply MPI I/O Write Time (Weak Scaling, N = MPI Ranks * 512)

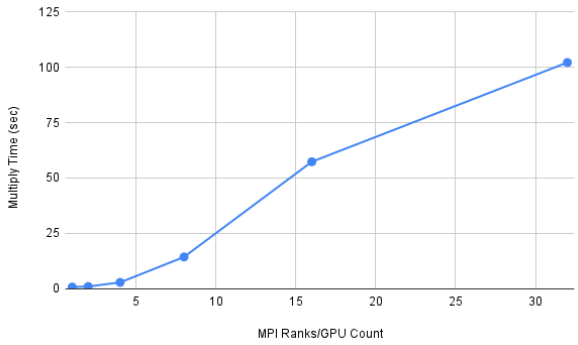

Figure 9: MPI-CUDA Matrix Multiply Time (Weak Scaling, N = MPI Ranks (GPU Count) * 512)

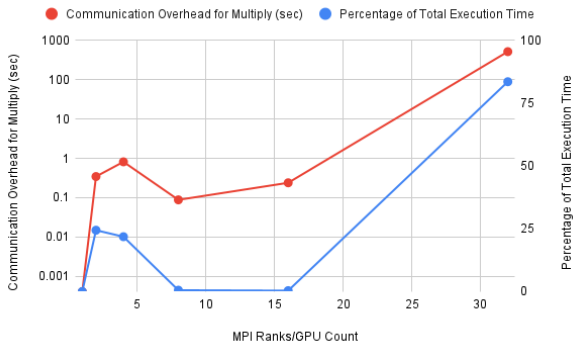

Figure 10: MPI-CUDA Matrix Multiply Communication Overhead (Weak Scaling, N = MPI Ranks (GPU Count) * 512)

Covering our implementation process, our experiments were all conducted on the AiMOS Supercomputer. We began by attempting to do performance analysis with MPI and MPI/CUDA with GkeyllZero, but the transfer to AiMOS was infeasible. We then switched over toward investigating matrix multiply as it can become an intractable problem as the matrix size increases. We were also attempting to incorporate cuBLAS to parallaize the matrix multiplication along with MPI; however, this has also proved to be troublesome due to the dichotomy of how we had to allocate memory for MPI, which is a 1-D case, compared to what is required of cuBLAS, a 2-D case.

In our related works portion, we look at other mechanisms of matrix multiply from existing literature. Some of these topics include KBLAS, creating efficient SmVM, OpenMP, atomic operations, and so forth. Some of the tangential studies done on matrix multiply would include numerical computations with GPUs as well as matrix multiplication in the application of machine learning.

7 CONCLUSION

To conclude, our work conducted strong scaling benchmarks on MPI and MPI-CUDA. For MPI strong scaling we have noted that the time for the matrix multiplication operation decreased by around a factor of 2 for each additional MPI rank. Additionally, the MPI I/O write time showed no consistent trends as the number of MPI ranks increased on NVMe. Initially, as the rank count was low, the write time decreased as rank count increased. However, for higher rank counts the write time increased. To make this concrete, as the number of MPI ranks increases, the time taken for MPI communication for matrix multiplication increases. Thus, the proportion of the overhead of the total execution time of the matrix multiplication increases. As we looked at MPI-CUDA strong scaling, as the number of MPI ranks increased, the time for the matrix multiplication operation decreased. It should be noted that CUDA significantly decreased the operation's time compared to that of the MPI non-CUDA version. We noticed that the MPI I/O began to slow down a bit after 16 ranks due to the length of time it could take to write the data. In terms of our weak scaling implementations for both MPI and MPI I/O, as the number of elements increase, the multiplication time increased. While parallelism helps with stable input size, it appears that only with CUDA could the exponential rise in execution time be mitigated (down to a subexponential increase) with increasing matrix size. Tables 1, 4-6 detail our exact timing results and were used to generate all the plots.

8 FUTURE WORK

There is still more work that can be done to further explore the various implementations and the impact on execution time. A similar analysis can be done on rectangular matricies (M by K matrix times K by N matrix) to see if there is a difference in execution time. Looking at common CUDA implementations of block multiplication, there are opportunities to make the algorithm more efficient by optimizing the block size. Furthermore, the use of cuBLAS in the CUDA implementation varies from the one we have made, and comparing the two could yield in important results. Regarding parallel I/O, all of our testing have used AIMOS' NVMe storage, but comparing to the disk storage on the clusters could also be done. Looking at our non-CUDA MPI weak scaling results, it is clear that the code is bound by the serial portion. Further improving the code to reduce the serial bound would be a nontrivial and fruitful endeavour.

REFERENCES

[1] Ahmad Abdelfattah, David Keyes, and Hatem Ltaief. 2016. Kblas: An optimized library for dense matrix-vector multiplication on gpu accelerators. *ACM Transactions on Mathematical Software (TOMS)* 42, 3 (2016), 1–31.

[2] Muthu Manikandan Baskaran and Rajesh Bordawekar. 2009. Optimizing sparse matrix-vector multiplication on GPUs. *IBM research report RC24704* W0812–047 (2009).

[3] Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA.* Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

[4] Ian Buck. 2007. Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses*. 6–es.

[5] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. 2008. Parallel I/O prefetching using MPI file caching and I/O signatures. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 1–12.

[6] Zhezhe Chen, Xinyu Li, Jau-Yuan Chen, Hua Zhong, and Feng Qin. 2012. SyncChecker: detecting synchronization errors between MPI applications and libraries. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 342–353.

[7] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2007. General-purpose sparse matrix building blocks using the NVIDIA CUDA technology platform. In *First workshop on general purpose processing on graphics processing units*. Citeseer, 32.

[8] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snirt, Bernard Traversat, and Parkson Wong. 1996. Overview of the MPI-IO parallel I/O interface. *Input/Output in Parallel and Distributed Computer Systems* (1996), 127–146.

[9] Peter Corbett, Dror Feitelson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. 1995. *Mpi-io: A parallel file i/o interface for mpi version 0.3.* Technical Report.

[10] Hoang-Vu Dang and Bertil Schmidt. 2013. CUDA-enabled Sparse Matrix–Vector Multiplication on GPUs using atomic operations. *Parallel Comput.* 39, 11 (2013), 737–750.

[11] Jack Dongarra, Sven Hammarling, Nicholas J Higham, Samuel D Relton, Pedro Valero-Lara, and Mawussi Zounon. 2017. The design and performance of batched BLAS on modern high-performance computing systems. *Procedia Computer Science* 108 (2017), 495–504.

[12] Jack J Dongarra, Steve W Otto, Marc Snir, David Walker, et al. 1995. An introduction to the MPI standard. *Commun. ACM* 18 (1995).

[13] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 133–137.

[14] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 97–104.

[15] Tigran M Galstyan. 2018. Performance Analysis of Matrix Multiplication Algorithms Using MPI and OpenMP. *Mathematical Problems of Computer Science* 50 (2018), 111–118.

[16] Zhibin Huang, Ning Ma, Shaojun Wang, and Yu Peng. 2019. GPU computing performance analysis on matrix multiplication. *The Journal of Engineering* 2019, 23 (2019), 9043–9048.

[17] Dana Jacobsen, Julien Thibault, and Inanc Senocak. 2010. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. 522.

[18] NP Karunadasa and DN Ranasinghe. 2009. Accelerating high performance applications with CUDA and MPI. In *2009 international conference on industrial and information systems (ICIIS)*. IEEE, 331–336.

[19] David Kirk et al. 2007. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, Vol. 7. 103–104.

[20] Alfio Lazzaro, Joost VandeVondele, Jürg Hutter, and Ole Schütt. 2017. Increasing the efficiency of sparse matrix-matrix multiplication with a 2.5 D algorithm and one-sided MPI. In *Proceedings of the Platform for Advanced Scientific Computing Conference*. 1–9.

[21] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings 5*. Springer, 111–125.

[22] Guillermo Oyarzun, Ricard Borrell, Andrey Gorobets, and Assensi Oliva. 2014. MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. *Computers & Fluids* 92 (2014), 244–252.

[23] Sreeram Potluri, Hao Wang, Devendar Bureddy, Ashish Kumar Singh, Carlos Rosales, and Dhabaleswar K Panda. 2012. Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 1848–1857.

[24] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using Cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, 73–82.

[25] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[26] Gopal Santhanaraman, Sundeep Narravula, and Dhabaleswar K Panda. 2008. Designing passive synchronization for MPI-2 one-sided communication to maximize overlap. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.

[27] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*. 23–32.

[28] AV Utkin. 2017. Analysis of parallel molecular dynamics for MPI, CUDA and CUDA-MPI implementation. *Mathematica Montisnigri* 39 (2017), 101–109.

[29] David W Walker and Jack J Dongarra. 1996. MPI: a standard message passing interface. *Supercomputer* 12 (1996), 56–68.

[30] Nicholas Wilt. 2013. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education.

[31] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. 2011. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications* 182, 1 (2011), 266–269.