Accelerate Dense Matrix Multiplication on Heterogeneous-GPUs

1st Jianan Sun
Institute of Automation,
Chinese Academy of Sciences.
School of Artificial Intelligence,
University of Chinese Academy of Sciences.
Beijing, China
sunjianan2021@ia.ac.cn

2nd Mingxue Liao
Institute of Automation,
Chinese Academy of Sciences.
Beijing, China
mingxue.liao@ia.ac.cn

3rd Yongyue Chao
Institute of Automation,
Chinese Academy of Sciences.
Beijing, China
chaoyongyue2020@ia.ac.cn

4th Pin Lv
Institute of Automation,
Chinese Academy of Sciences.
Beijing, China
pin.lv@ia.ac.cn

***Abstract*—Matrix multiplication is crucial in scientific computing, but it demands substantial resources. We propose a framework for effectively utilizing heterogeneous GPUs to large matrix multiplication. By splitting matrices into small blocks and using Douglas's variant of Strassen's algorithm, we enable concurrent tasks on heterogeneous systems. Our framework improves speed by 89.5% on homogeneous GPU servers and by 108% in multi-server heterogeneous GPU setups.**

***Index Terms*—GPU, parallel computing, heterogeneous system, matrix multiplication**

I. Introduction

Matrix multiplication is a crucial operation in various scientific fields, but as the size of matrices increases, computation time becomes impractical and a single GPU's memory may not be sufficient. This necessitating multi-GPU parallel computing. Most of algorithms nowdays focus on homogeneous systems ([1]–[3], [7], [8]), ignoring the prevalence of heterogeneous systems comprising different types of GPUs with varying computing capabilities, and these algorithms treat GPUs as a square or a cube, that is the number of GPUs should be denoted as [m, n] or [m, n, p]. While these algorithms improve the efficiency of matrix multiplication, they are ideal for specific application scenarios that use a parallel system with the same GPUs and the number of GPUs that is just right for a square or a cube.

We propose a framework that can effectively utilize systems with different types of GPUs. It use Douglas's algorithm, one variant of Strassen's algorithm ([4]–[6]), to improve concurrency of one GPU. To handle large input matrices effectively, a divide-and-conquer strategy is often necessary. Our framework achieves this by dividing the matrices into smaller blocks and utilizing Douglas's variant to multiply corresponding blocks.

Here is contributions of this paper.

- We implemented a framework to execute large matrix multiplication on heterogeneous GPU systems. It is responsible for matrix decomposition and task allocation.
- To enhance the performance of the framework, we have made improvements from multiple dimensions. For single GPU matrix multiplication, we implemented the Douglas algorithm based on cuBLAS. For single-server multi-GPU matrix multiplication, we enhanced the task allocation algorithm of cuBLASXt. In the case of multi-server heterogeneous GPU matrix multiplication, we allocated tasks according to the performance of each GPU.
- We conducted experiments for three scenarios which illustrated that our framework is effective to execute matrix multiplication on heterogeneous systems.

II. Design and Implementation

A. Matrix Decomposition

To accelerate multiplication on multi-GPU systems and multi-server systems, we employed a two-layer decomposition method. In the first layer of decomposition, we divided the input matrices $\mathbf{A}$ and $\mathbf{B}$ into block matrices that were larger than the cut-off point of Strassen's algorithm.

In the second layer of decomposition, we further divided the block matrices $\mathbf{A_{ik}}$ and $\mathbf{B_{kj}}$ into 2×2 sub-block matrices, which could then be multiplied using Strassen's algorithm. which is implemented on the basis of cuBLAS as detailed in algorithm1. To calculate multiplication of $\mathbf{AB} = \mathbf{C}$, these matrices are Firstly divided into 2×2 block matrices. Then complete the conputing produces detailed in reference [5].

Our two-layer decomposition method proved effective in maximizing the performance of matrix multiplication on multi-GPU servers and multiple servers.

Algorithm 1: Implementation of Strassen's algorithm

Input: Matrix A and B
Output: Matrix C

```
A11, A12, A21, A22 = did2(A)
B11, B12, B21, B22 = did2(B)
one = 1
minus_one = -1
zero = 0
cublasSgeam(one, A11, minus_one, A21, T1)
cublasSgeam(one, B22, minus_one, B12, T2)
cublasSgemm(one, T1, T2, zero, C21)
...(complete produces of Douglas's algorithm)
```

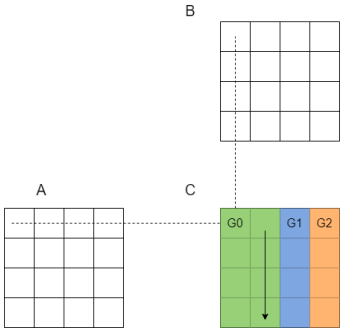

Fig. 1. Overview of tasks allocation. This figure shows how tasks are allocated across three GPUs, with matrices stored in column-first format.

B. Tasks Allocation

The cuBLASXt Library provides a multi-GPU capable Host interface, allowing applications to allocate memory space on the Host side while the GPU-related work is done by cuBLASXt. To compute matrix multiplications on multi-GPU systems, cuBLASXt divides matrices into square tiles and uses a round-robin fashion to calculate the resulting tile. However, this strategy can result in performance decrement due to irrelevant tiles being processed by a single GPU, leading to more redundant memory transfers. Instead, we conducted a sequential task allocation approach, allowing block matrices to be reused in multiple tasks (See Figure 1 for an illustration). GPU0 computes tasks of block matrix $\mathbf{C_{00}}$ to $\mathbf{C_{31}}$, so a column of matrix $\mathbf{B}$ is only needed to transfer to GPU0 once, reducing the transferring time.

C. Thread Pool

We created a thread pool to manage the status of the entire system that includes multiple servers, each equipped with a number of specific GPUs. The system is composed of three main components, which are illustrated in Figure 2.

Figure 2 illustrates the thread structure of the main server, which controls the entire system through the system thread. The system thread is responsible for managing the multi-server system and takes three matrices as input, which are divided into blocks. In addition, it takes as input the performance ratios of different servers and the number of GPU that each server

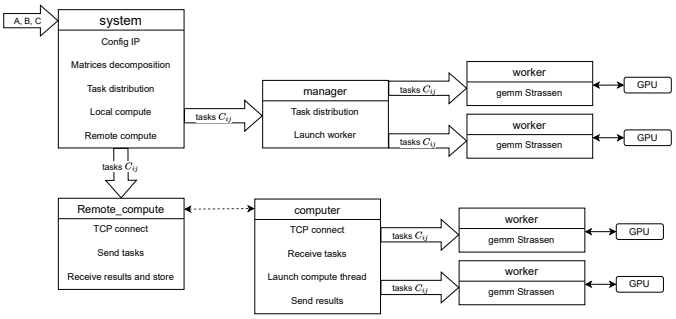

Fig. 2. Overview of the thread pool. The framework consists of two main components: the local part and the remote part. The local part is executed by the main server, while the remote part is executed by subordinate servers.

TABLE I
CONFIGURATIONS OF SERVERS

CPU	Memory	GPU
Intel(R) Xeon(R) Gold 5218R	1 TB	8 × GeForce RTX 3090
Intel(R) Xeon(R) Gold 5117	30 GB	3 × GeForce RTX 2080 Ti
Intel(R) Xeon(R) Gold 5117	30 GB	2 × GeForce RTX 2080 Ti

is equipped to calculate the number of tasks for each server. The system thread divides the result matrix into blocks and creates a list that contains all the tasks. This task list is then divided into several parts, with each part corresponding to a specific GPU.

System launches a local compute routine to execute tasks on local GPUs, which executes a manager thread that controls all the GPUs on the local server and distributes tasks to worker threads. A worker thread is responsible for computing a series of specific tasks on a given GPU. It handles GPU memory management, computes block matrix multiplications using Strassen's algorithm, and transfers the results from GPU memory to the Host memory. System also launches remote_compute routines to connect with other servers via TCP connections.

On other servers, a computer thread controls the entire computing of this server. It connects to the main server to receive tasks and distributes these tasks to GPUs. Worker threads are launched to compute tasks on GPUs by computer thread. After a worker finishes a task, the compute thread sends the result back to main server.

III. EXPERIMENT

A. Experimental Platform

All experiments were conducted on the cluster of three servers, connected through Gigabit networks. The configurations of the servers are detailed in Table I.

B. Single GPU Multiplication

We implemented Strassen's algorithm based on cuBLAS and, we compare its performance to the cuBLAS general matrix-to-matrix multiply (gemm) routine. Specifically, we compute $\mathbf{AB} = \mathbf{C}$ where $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{n \times n}$ and record the computing time of multiplication operations for varying

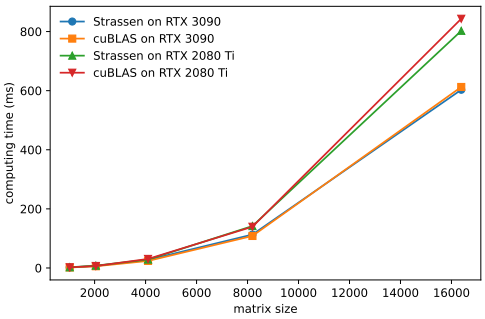

Fig. 3. Performance of our implementation of Strassen's algorithm and cuBLAS gemm routine.

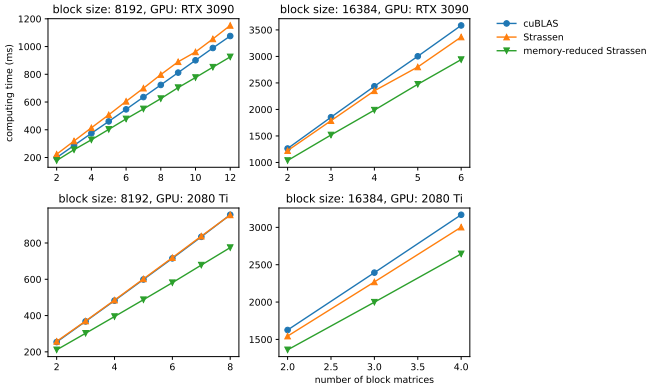

Fig. 4. Performance of a series of block matrix multiplications $\sum_k \mathbf{A_{ik}B_{kj}}$.

matrix sizes. Figure 3 shows the performance of our Strassen's algorithm implementation and the cuBLAS routine. As the matrix size increases, the performance of Strassen's algorithm surpasses that of cuBLAS. When the matrix size is less than 8192, the computing time of Strassen's algorithm and the cuBLAS gemm routine are almost equal on both GPUs. When the matrix size is 16384, the computing time of Strassen's algorithm is less than that of cuBLAS on both GPUs. For the RTX 2080 Ti, Strassen's algorithm runs 5% faster than the cuBLAS routine, and for the RTX 3090, Strassen's algorithm runs 1.5% faster.

Section II-B describes task allocation method that outperforms cuBLASXt, by avoiding redundant memory transfers. We illustrate the performance of this method by multiplying a series of matrices $\sum_k \mathbf{A_{ik}B_{kj}}$. We reduce memory transfers by using the same $\mathbf{B_{kj}}$ for all k, and Strassen's algorithm is used in these multiplications, compared with regular operations (with excessive memory copy) of the series of matrix multiplications that use both cuBLAS routines and Strassen's algorithm, that is when multiplying $\mathbf{A_{ik}B_{kj}}$ our implementation of Strassen's algorithm or cuBLAS gemm routine is utilized. Figure 4 shows the performance of a series of block matrix multiplications $\sum_k \mathbf{A_{ik}B_{kj}}$ with different block sizes executed on both RTX 3090 and RTX 2080 Ti. When the block size is 8192, the computing time of cuBLAS routines and Strassen's algorithm is roughly equal, and that time increases linearly with the number of block matrices. When the block size is 16384, our implementation of Strassen's algorithm outperforms cuBLAS routines in every number of block matrices, and the gap between these two methods becomes larger as the number increases.

C. multi-GPU Server Multiplication

We evaluate the performance of our framework for multiplying large matrices on a multi-GPU server. The results of three types of experiments are presented in Figure 5, demonstrating the speedup achieved by our framework. In the first type of experiment, we use cuBLASXt to multiply two large matrices. In the second type of experiment, our framework is responsible for dividing matrices and memory transfer, while cuBLAS gemm routine multiplies two block matrices. This allows us to illustrate the impact of Strassen's algorithm and our task allocation method. The last type of experiment also utilizes our framework, and Strassen's algorithm is used when multiplying two block matrices.

Our framework outperforms cuBLASXt when executing matrix multiplication on Server 1, which contains 8 RTX 3090 GPUs. This improvement in performance can be attributed to our task allocation method, which reduces the need for memory transfers from CPU to GPU. When the block size is 8192, our framework using Strassen's algorithm performs as fast as that using cuBLAS routine. Moreover, when the block size is increased to 16384, our framework using Strassen's algorithm outperforms that using cuBLAS routine.

Similar results were observed on the other two servers, each containing two or three RTX 2080 Ti GPUs. Our framework outperforms cuBLASXt, especially when the block size is 16384. However, the performance improvement is not as significant as on server 1. This can be attributed to two reasons. Firstly, the RTX 2080 Ti exhibits lower performance, resulting in more time being spent on computing matrix multiplication. This reduces the impact of memory transfer on the overall computation time, as the total time is primarily determined by the calculations. Secondly, our task allocation method has limited effectiveness on servers with a small number of GPUs.

D. Multi-Server Multiplication

In this section, we evaluate the performance of our framework for large matrix multiplication on a heterogeneous system consisting of multiple servers with different types of GPUs. Task partitioning has a significant impact on the system, so we conduct two experiments. The first experiment divides the tasks evenly, calculating similar tasks across three servers. The second experiment involves manually dividing the tasks to optimize performance by adjusting the amount of tasks per server. We also compare the performance of multiple servers with that of a single server (server 1) to demonstrate the benefits of using a multiple server system for matrix multiplication, as there is limited research on accelerating matrix multiplication on heterogeneous systems. We choose

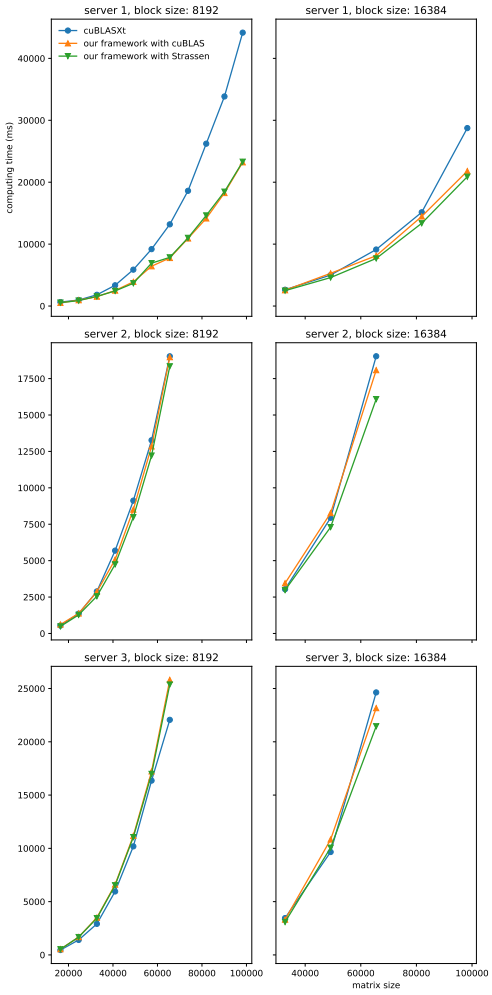

Fig. 5. Performance of large matrix multiplication on a server with multiple GPUs.

the block size of 8192 to maximize the number of tasks with a smaller size.

Figure 6 presents the experimental results. The computation time significantly decreases when using multiple servers for matrix multiplication. As the matrix size increases, it becomes more flexible to schedule tasks for each server, and choosing a better scheduling method can further accelerate performance. The speedup of multiple servers reaches 2.08 when the input matrix size is 98304.

IV. Conclusion

In this paper, we propose a framework for matrix multiplication on heterogeneous systems that contain different types of GPUs. Our framework leverages the resources of each GPU by assigning tasks that optimize the occupancy of the entire system. The framework sequentially allocates tasks to each GPU to avoid unnecessary memory transfers, which can improve system performance. Our experiments show that our implementation of Strassen's algorithm outperforms cuBLAS gemm routine when the input matrix is larger

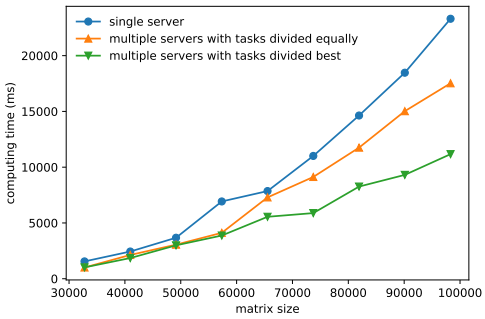

Fig. 6. Performance of large matrix multiplication on multiple servers.

than 8192. Our framework also outperforms cuBLASXt when multiplying large matrices on a server with homogeneous GPUs, achieving a speedup of 89.5%. Our framework is capable of efficiently utilizing multiple servers to compute large matrix multiplication, with a 108% speedup compared to using a single server. In the future work, we anticipate using the framework to accelerate matrix multiplication and improve tensor parallelism in training deep learning models. The source code of our framework is available at https://github.com/Sunjnn/mutGPUStrassen.

References

[1] B. Wang, Q. Xu, Z. Bian, and Y. You, "2.5-dimensional Distributed Model Training," ArXiv, vol. abs/2105.14500, 2021.

[2] L. E. Cannon, "A Cellular Computer to Implement the Kalman Filter Algorithm," PhD Thesis, Montana State University, USA, 1969.

[3] E. Solomonik and J. Demmel, "Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms," in Euro-Par 2011 Parallel Processing, E. Jeannot, R. Namyst, and J. Roman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 90–109.

[4] V. Strassen, "Gaussian elimination is not optimal," Numerische Mathematik, vol. 13, pp. 354–356, 1969, doi: 10.1007/BF02165411.

[5] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith, "GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm," Journal of Computational Physics, vol. 110, no. 1, pp. 1–10, 1994, doi: https://doi.org/10.1006/jcph.1994.1001.

[6] D. Coppersmith and S. Winograd, "Matrix Multiplication via Arithmetic Progressions," in Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, in STOC '87. New York, NY, USA: Association for Computing Machinery, 1987, pp. 1–6. doi: 10.1145/28395.28396.

[7] Z. Bian, Q. Xu, B. Wang, and Y. You, "Maximizing Parallelism in Distributed Training for Huge Neural Networks," ArXiv, vol. abs/2105.14450, 2021.

[8] R. A. Van De Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," Concurrency: Practice and Experience, vol. 9, no. 4, pp. 255–274, 1997, doi: https://doi.org/10.1002/(SICI)1096-9128(199704)9:4¡255::AID-CPE250¿3.0.CO;2-2.