Batched matrix operations on distributed GPUs with application in theoretical physics

Nenad Mijić*, Davor Davidović*
* Centre for Informatics and Computing, Ruđer Bošković Institute, Zagreb, Croatia
{nenad.mijic, davor.davidovic}@irb.hr

Abstract—**One of the most important and commonly used operations in many linear algebra functions is matrix-matrix multiplication (GEMM), which is also a key component in obtaining high performance of many scientific codes. It is a computationally intensive function requiring $O(n^3)$ operations, and its high computational intensity makes it well-suited to be significantly accelerated with GPUs. Today, many research problems require solving a very large number of relatively small GEMM operations that cannot utilise the entire GPU. To overcome this bottleneck, special functions have been developed that pack several GEMM operations into one and then compute them simultaneously on a GPU, which is called a batch operation. In this research work, we have proposed a different approach based on linking multiple GEMM operations to Message Passing Interface (MPI) processes and then binding multiple MPI processes to a single GPU. To increase GPU utilisation, more MPI processes (i.e. GEMM operations) are added. We implement and test this approach in the field of theoretical physics to compute entanglement properties through simulated annealing Monte Carlo simulation of quantum spin chains. For the specific use case, we were able to simulate a much larger spin system and achieve a speedup of up to $35\times$ compared to the parallel CPU-only version.**

Keywords*—*matrix multiplication, batched operations, GPU, MPI, HPC

I. Introduction

Matrix multiplications are the fundamental building blocks in many functions of linear algebra as well as in a variety of scientific and industrial codes. The key kernel of linear algebra responsible for the high performance of many computer architectures is matrix-matrix multiplication (GEMM). It is usually one of the first kernels to be optimised and adapted for different processors and computer architectures such as multi-core CPUs, Intel Xeon Phi, ARM or GPUs [1]–[3]. Matrix multiplication is a computationally intensive problem whose performance is limited by processor speed rather than memory bandwidth and latency. The arithmetic intensity (the ratio of floating point operations per data transferred) is defined by the number of rows or columns and, for sufficiently large matrices, can reach a performance close to the theoretical peak performance of the computing system. Due to its high arithmetic intensity, matrix multiplication is well suited to efficiently hide slow data movement from slower memory locations (e.g., disc or main memory) to faster, near-processor memory (e.g., cache memory or global and shared memory on GPUs), but careful implementation of matrix multiplication is required [4]. Highly optimised, fine-tuned implementations of matrix multiplication can be found in numerous numerical computation libraries for multi-threaded and multi CPU systems (BLAS [5], LAPACK [1], OpenBLAS [6], MKL [7]), GPU-accelerated and heterogeneous platforms (cuBLAS [8], MAGMA [2], [9]) and distributed memory systems (ScaLAPACK [10], [11]).

Although very high performance and resource utilisation can be achieved in matrix multiplication, when working with small matrices the impact of overlapping data transfers with useful computations can be significantly reduced. This shortcoming becomes even more apparent when the computation is moved to the GPU devices, as the matrix is usually not large enough to fully utilise the entire GPU. In addition, for each multiplication, the data must be transferred between the main memory and the GPU memory via a relatively slow CPU-GPU interconnection, which is characterised by high latency.

The problem described above becomes even more challenging when solving problems involving a large number of small matrix multiplications. In these problems, the transfer of a large number of small matrices to GPUs can easily become a time-dominant part of the overall execution due to the latency issues. An example where a large number of small matrix computations are required are domain decompositions [12], 3D graphics transformations in the Level 3 Cascading Style Sheets specification in web browsers [13], Astrophysics [14], Finite Element Methods [15] and in Machine Learning and Artificial Intelligence [16]–[18].

To solve these types of problems efficiently on GPUs, the common approach is to group or package many small matrix operations into one larger operation, which is then transferred and processed simultaneously on the GPU. By packing multiple operations into one larger operation, the number of memory transfers can be greatly reduced, which decreases latency and increases GPU utilisation. This type of operation is commonly referred to as batched matrix operations.

In this paper, we present an alternative approach to batched matrix operations (called *batchedGemm* in the rest of this research) based on scheduling multiple MPIs on a single GPU and analyze the achieved performance compared to CPU-only variant. The original contribution of this research are the following:

- Implicitly packing a larger number of small matrices on a single GPU using MPI ranks,

- The proposed model allows easy scaling to a larger number of GPUs and across compute nodes.
- Improved performance compared to the state-of-the-art Numpy batched approach (up to 22%).
- Achieved $35\times$ speedup compared to the CPU-only version for the use case Simulated Annealing Monte Carlo Simulation.

The rest of the paper is organized as follows. In Section II a brief introduction to problem of computing many matrix operations is given, with state-of-the art methods and libraries used to solve them. Our approach is solving batch GEMM operations is presented in Section III together with the targeted use-case. The numerical results and the achieved performance are discussed in Section IV. The final notes and the conclusion of our work is given in Section V

II. Batched GEMM and related work

The aim of this research is to find an efficient solution for many small matrix-matrix multiplications on the GPU. A set of matrix multiplications can be defined as follows:

$$C_i = \alpha A_i B_i + \beta C_i, \quad i = 1, \dots N \tag{1}$$

where $A_i \in \mathbb{C}^{m\times k}$, $B_i \in \mathbb{C}^{k\times n}$, $C_i \in \mathbb{C}^{m\times n}$ are complex matrices and N is the number of matrix-matrix multiplications to be calculated. The common approach to compute a single matrix is to partition C into multiple tiles, each processed independently by a block of threads on the GPU, with each thread computing one element of the matrix C. Parallelism can be exploited between tiles, i.e. several tiles run simultaneously, and within tiles, i.e. each tile is processed by many threads. When C is large, this approach can attain close to the peak performance of the GPU, as there are enough tiles to fully utilise the symmetric multiprocessors of the GPU. However, when C is small, the number of tiles is insufficient to fully utilise the GPU. In addition, processing a large number of small GEMMs requires a large number of data copies between main memory and GPU memory, resulting in the application being memory-bound rather than compute-bound.

In order to exploit the full power of GPU devices, an approach has been developed based on packing many small matrix operations into a larger operation called a batch. The basis for the idea of batch matrix multiplication is the tiling algorithm, which divides the matrix into a 2D grid of tiles, each of which represents a subproblem that is then processed simultaneously on the GPU. The initial matrix is transferred once (a large memory copy), while the division into smaller subproblems is done on the device [19]. Performance can be fine-tuned by adjusting the tile size depending on the available resources on the GPU device. The implementation of this approach can be found in MAGMA [2], [9] dense linear algebra library. One of the main drawbacks of the latter approach is the fixed size of the matrices, e.g. all matrices of A should have the same size and be stored in contiguous memory locations. To overcome this problem, a variable-size GEMM was proposed [20], [21], which divides each GEMM (matrices A, B and C, Eq. 1) into tiles processed by thread blocks. Then each C can be processed by its 2D grid of thread blocks. In the paper [22], the authors propose a special kernel for matrices with size less than 32, which are often encountered in Big Data analytics, machine learning and high-order finite element method (FEM). The approach is based on aggregating multiple thread blocks to compute two or more tiny matrices C (matrices smaller than the warp size, i.e. dimension less than 32) and achieves performance within 90% of a large single GEMM.

In batch computation by tiling and batching, the size of the tiles and the batch size, respectively, are the two most important performance parameters. Although different matrix sizes are possible, the tiling strategy is the same, which is not the best solution for different matrix sizes. To overcome this problem, a novel solution is proposed [23] that allows different tiling strategies (different tile sizes for different C) and batch strategies (the number of tiles allocated to a thread block). The proposed work achieves $1.23\times$ speedup on the GoogleNet case study compared to the state-of-the-art CUBLAS and MAGMA implementations.

In some research areas, such as latency-critical deep neural networks consisting of a large number of small batch sizes, the classical batch approach provides very limited GPU usage. Instead of packing smaller operations into a batch, GPU resource usage can be split across multiple processes or tasks. In [24], the authors experimented with two approaches. The first is to put each process in a separate CUDA context and then use a software scheduler to interleave the executions on the GPU. The second approach is to use the NVIDIA Multi-Process Service (MPS) [25] server to partition multiple processes into separate queries across a pool of CUDA streams. Recent research [26] shows that performing a number of $2k \times 2k$ and $8k \times 8k$ matrix multiplications with MPS can achieve a speedup of up to $4.5\times$ compared to the native CUDA scheduler.

III. Task-based batch GEMM

In this research we tackle a problem of processing a number of embarrassingly parallel processes, each computing a series of small matrix operations, each operation not large enough to fully utilize the GPU. Because of its embarrass parallelism, there is no need for an explicit synchronization between the processes thus any packing of many operations into one batch operations will cause an unnecessary synchronization points in the execution.

An example is the simulated annealing Monte Carlo (MC) [27] simulation of frustrated one-dimensional transverse-field Ising model [28] with S spins, Fig. 1. The simulation starts with the initial calculating of the ground state. The main body of the simulation consists of applying

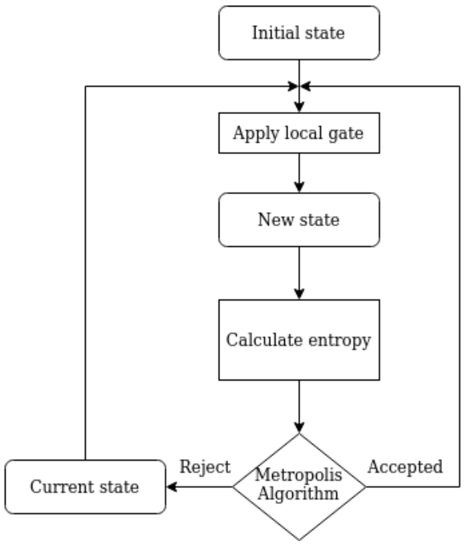

Fig. 1: Simulation model.

local unitary operations (Fig. 1, `Apply Local Gate`) on the neighbouring pair of spins in the lattice, chosen at random, and computing the new entanglement (Fig. 1, `Calculate Entropy`). Finally, the decision is made to accept this new state (Metropolis algorithm, Fig. 1) or keep the current state. Then, repeat the process with the unitary operator on the chosen state. The pseudocode of the Monte Carlo simulation of Ising model is given in Alg. 1.

The main computational bottleneck is the `Calculate Entropy` in which two complex matrices with dimensions depending on the number of spins $\left(m = n = 2^{S-\lfloor S/2 \rfloor}, k = 2^{\lfloor S/2 \rfloor}\right)$ are multiplied. Since the step of the MC simulation depends on the state computed in the previous step, the MC procedure is an inherently sequential code and the loop over MC steps (Alg. 1, line 3) cannot be parallelised. Therefore, only matrix multiplications (line 6) can be parallelised, but only one GEMM at the time can be processed (due to inter-step dependencies). The number of MC steps (N_s) (line 3) can be arbitrary large, but in practice, this number can be counted in millions, thus limiting the performance gain especially when GEMM operations are small.

To increase the statistical significance of the simulation results, a large number of repetitions (N_p) of the Monte Carlo procedure are required (line 2). The MC procedures are mutually independent and no data transfer or communication between them is necessary, making the problem an embarrassingly parallel one and straightforward to parallelise. A typical real-world use-case with $S = 21$ spins, 100 MC procedures and 10^6 steps each, will require to compute a series of GEMM operations of size $m = n = 1024, k = 2048$.

To efficiently solve the above problem, in this research we proposed a solution based on the spatial sharing of the GPU resources among multiple processes and tasks. Within the MC procedures the matrix multiplications are

Algorithm 1 Monte Carlo simulation of Ising model

```
Require: Number of spins $S$
Ensure: Average entropy
1: Compute initial state
2: for $i \in [1 \cdots N_p]$ do                ▷ Loop over MC procedures
3:     for $s \in [1 \cdots N_s]$ do            ▷ Loop over MC steps
4:         Apply unitary operator              ▷ Apply Local Gate
5:         Generate $A, B$
6:         $C \leftarrow GEMM(A, B)$           ▷ Calculate Entropy
7:     end for
8: end for
9: Compute average entropy
```

offloaded to the GPU (Alg. 1, line 6), while MC procedures (line 2) are distributed using Message Passing Interface (MPI), each MPI rank calculating one MC procedure independently. Multiple MPI processes (MC procedures) are assigned to same device in order to saturate it with other ready to execute kernels. This can achieve better utilization at the expense of less speedup per single kernel. This approach is a simple from the programming point of view, because it leaves to the GPU internal scheduler to allocate unused resources for other kernels (processes), a kind of tasked based parallelism of independent simulation procedures. The proposed approach with multiple task utilizing the same GPU has a higher potential of improving throughput and lowering latency, because multiple memory transfer operations and kernels can be arbitrarily and asynchronously overlapped.

The drawback of this approach is that each process creates its unique context on the device that leads to the process synchronization, since exclusively only one context can be active on the device at the time. To overcome this problem NVIDIA has devised Multi Processing Service (MPS), a software layer between CUDA driver and multi process program, which creates single context and routes all CUDA calls/kernels through it. Starting with the Volta architecture, each process sends jobs to the device separately without passing through the MPS server where each process is scheduled in its own queue. It is advisable to use the MPS when GPU utilization per process is low and want to achieve multiple kernels to execute concurrency.

IV. Results

In this section, we present and analyse the performance of our approach in real use cases and compare it with the parallel CPU-only implementation and the GPU-accelerated batch implementation. In the rest of the paper, we refer to our approach as `taskedGEMM` and the classical batch approach as `batchedGEMM`. In both cases, we test how the size of the batch (or in the case of `taskedGEMM` the number of MPI ranks per GPU) affects the performance and how it scales with the addition of more procedures running in parallel for different problem sizes.

The tests were conducted on the GPU partitions of the HPC Vega supercomputer at the Institute of Information Science, Slovenia. Each node is equipped with two 64-core AMD Rome 7H12 CPUs, that can achieve 1.8 TFlops calculating `dgemm`, and four NVIDIA Tesla A100 GPUs each with peak performance of 19.5 TFlops (double precision, using Tensor Cores) and 40 GB of memory.

For benchmarking our MPI-based approach, we extend the original Monte Carlo simulation code written in the Python programming language. The original code takes advantage of highly tuned NumPy and SciPy libraries for multithreaded matrix multiplication on shared memory systems. Although the NumPy implementation of GEMM achieves better results than the sequential code, it turns out that it cannot handle larger simulation sizes ($S > 19$) in a reasonable time. Therefore, the matrix generation and the most time-consuming parts of the code - the matrix multiplications - are offloaded to the GPUs. For the GPU-accelerated computations, we used the CuPy 10.0.0 library with CUDA 11.4.0 and cuBLAS 11.5.2.43. CuPy is an open-source NumPy-like library that uses the cuBLAS, cuSOLVER and cuRAND libraries from the CUDA toolkit. It has implemented a wrapper function for GEMM as well as for batched GEMM, in our case `cublasZgemm` and `cublasZgemmBatched`. To distribute the workload to the compute nodes and GPUs, we use SLURM 20.11 Workload Manager and OpenMPI 4.1.1 (UCX 1.11.2). All the tests were run in complex double precision.

A. *Single GPU benchmark*

In the first benchmark, we want to find the best binding policy of MPI ranks (i.e. number of Monte Carlo procedures N_p) per GPU. For this test, we focused on a single GPU and small problem sizes with the number of MC steps $N_s = 1000$, the number of MC procedures $N_p = \{1, 2, 4, \dots, 16\}$ and the number of spins $S = \{15, 19, 21, 23\}$. Since the execution time of each MC step is approximately the same and they are executed in sequential order, there is a linear dependence between the MC steps and the execution time. We can test with a smaller number of steps without loss of generality and then simply extrapolate execution time linearly to a desired number of MC steps.

Fig. 2 shows the speedup of `bachedGEMM` and `taskedGEMM` compared to the sequential version, i.e. when MC procedures are executed individually on the GPU. The number of MPI ranks in `batchedGEMM` is 1 and the batch size (number of MC procedures packed in a batch) is the number of procedures (represented by the x-axis, Fig. 2). In `taskedGEMM`, the number of MPI ranks is equal to the number of procedures (x-axis), since each MPI rank executes one MC procedure and all MPI ranks use the same GPU. The figure shows that the `taskedGEMM` achieves better speedup than the batch approach in all cases. As expected, as the number of MC procedures executed in parallel increases, we observe a steady speedup up to a certain point. For example, in

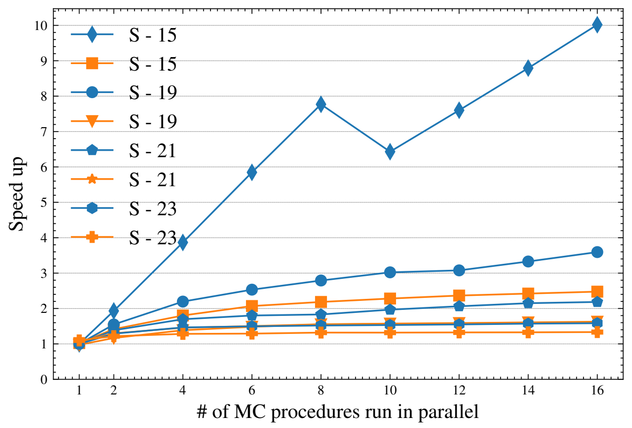

Fig. 2: Speedup of batched (orange) and tasked (blue) variants compared to the sequential execution for a given number of MC procedures on a single GPU.

`taskedGEMM` with 15 spins, the speedup is linear up to 8 MC procedures. This is because the matrices involved in GEMM are too small (256×128) to make full use of the test GPU. When more MC procedures are added, the speedup slows down as the GPU is overused, causing some MCs to end up waiting for GPU resources. In terms of GPU memory, the simulation only requires 24 MB of memory for 3 matrices per GEMM for each of the 16 MC simulations (48 matrices in total). For a larger spin system, e.g. $S = 23$, the overuse of GPU resources occurs even sooner, with only 4 MC procedures, as the matrices involved in the calculations are much larger (4096×2048). The GPU memory requirement is ≈ 6 GB in the largest case (with 16 MC procedures). The higher speedup of `taskedGEMM` suggests that interleaving GEMM operations leads to slightly better resource utilisation and thus higher efficiency.

The average performance (in TFLOPS/s) achieved per GEMM operation is given in Fig. 3. To calculate the performance of a single GEMM in `batchedGEMM`, the total performance is divided by the size of the batch. Since the execution time per GEMM call within each MC procedure in `taskedGEMM` is not constant and depends on the number of GEMM calls executing simultaneously on the GPU, the average execution time per GEMM is given. The constant performance of `batchedGEMM` for sizes $N_s = 21, 23$ indicates that the matrices are large enough to fully utilise the GPU even if only 1 MC procedure is computed, and that adding more matrices to the batch is unlikely to further increase performance. In contrast, the performance of $S = 15, 19$ batched operations increases when the size of the batch is increased. This shows that the size of the computational problem is still too small to fully utilise the GPU, so more operations should be added.

For task-based GEMM operations, there is a constant drop in performance per GEMM call as more MC procedures are added (blue lines in Fig. 3). Performance drops significantly as the number of MC procedures increases. The reason for this behaviour is that interleaving of

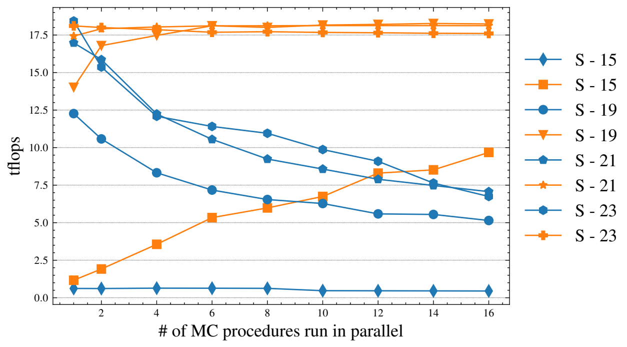

Fig. 3: Performance of the single GEMM operation as a function of the number of MC procedures for batch (orange) and task (blue) variants.

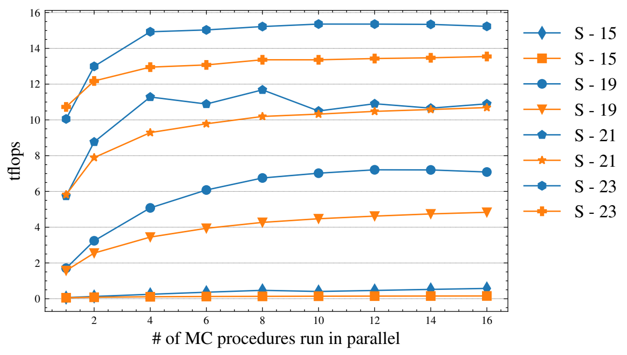

Fig. 4: The performance in TFLOPS/s of the entire simulation as a function of the number of MC procedures for batch (orange) and task (blue).

numerous operations (kernels) leads to an overuse of GPU resources. If there are not enough GPU resources available for a particular kernel (e.g. GEMM function call), such as the number of available registers or computing cores, it remains in the queue of the CUDA scheduler. As soon as the resources are released, the kernel begins execution. Such use of GPU resources consequently leads to fluctuations in the performance of individual executions, which manifest themselves as performance loss (see `taskedGEMM` for $S = 19, 21, 23$ in Fig. 3).

Although the time required to compute single GEMM operation in `taskedGEMM` is slower than in `batchedGEMM`, the overall performance of the whole simulation is shown to be much better for the task-based approach, Fig. 4. This is because interleaving different kernels (operations) on a GPU can better hide latency, reduce GPU idle time and increase utilisation. So in our use case, interleaving many independent operations has a much greater impact on performance than combining many operations into one larger but faster operation. We ran each test configuration 5 times and calculated the average value for the test results to achieve statistical significance.

TABLE I: Total runtime of `taskedGEMM` (in seconds) for 21 spin and 128 MC procedures.

N_p per GPU	Number of GPUs			
	1	2	4	8
4	3680	1840	920	460
8	3392	1696	848	424
16	3296	1648	824	420

B. *Multi-GPU benchmark*

In order to run the tests on multiple GPUs, we have chosen a sufficiently large real-world use case that represents well the performance on many GPUs. For the tests, we chose a case with $S = 21$ spins, $N_s = 10000$ Monte Carlo steps and the fixed number of procedures $N_p = 128$. Although in real-world use cases the number of MC steps is usually more than 10^6, we can test with a smaller number of steps because the steps are interdependent and the time increases linearly with the number of steps. The total execution time for a larger number of steps can be easily estimated by scaling the runtime of the test with a smaller number of steps.

The Table I shows the total execution time for solving the problem with the number of Monte Carlo procedures fixed to 128 and different number of procedures per GPU (4, 8 and 16) using the `taskedGEMM` approach. The test shows the results run on 1, 2, 4 and 8 GPUs. For example, the time to solve the problem with 16 tasks per GPU on only 1 GPU would be 3296 seconds, while the time with 8 GPUs decreases almost linearly and is 420 seconds. The reason is that the MC procedures are independent and 16 procedures can be processed simultaneously on one GPU, so 8 batches have to be executed sequentially. In contrast, in the case of 8 GPUs, all 8 batches, each processing 16 MC procedures, can be executed simultaneously, each on its own GPU.

In the previous tests, we have shown that the best performance is achieved when 16 MC procedures are executed per GPU (for a problem with 21 spin), so we set the number of MC procedures per GPU to 16. The total execution time and speedup of both `batchedGEMM` and `taskedGEMM` are shown in Table II. The batched and tasked GEMM variants achieve speedups of more than 26× and 35×, respectively, compared to the CPU-only version. Note that the CPU version is a parallel code with one MC procedure per MPI rank and the GEMMs are computed using the multi-thread NumPy library. All tests were performed on 2 compute nodes, and the configuration of the CPU-only version is 64 MPI ranks per node and 2 threads per MPI rank to compute the GEMMs.

The fastest execution time we achieved on our test systems for the CPU version was on 32 compute nodes with 4 MPI ranks (i.e. MC procedures) per node and 32 threads per MPI rank. In this case, the total execution time for the parallel CPU variant was 3529 seconds, noting that the result was obtained by extrapolating the

TABLE II: Total execution time and speedup of batch and task GEMM for $S = 21$, $N_s = 10000$, $N_p = 128$ compared to the best possible CPU-only configuration.

	Algoritmic variants		
	CPU	`batchedGEMM`	`taskedGEMM`
time	14735	534	420
speedup	1	27.6	35.1

computation time of 1 compute node. The speedup of `batchedGEMM` and `taskedGEMM` is $6.6\times$ and $8.4\times$ on 2 compute nodes, respectively, compared to the CPU-only variant on 32 compute nodes and 4 MPI ranks per node. This configuration guarantees enough resources to process all 128 MC procedures concurrently. Although the speedup is smaller than indicated in Table II, the CPU-only variant requires much more computing resources to achieve this performance, exactly $16\times$ more compute nodes.

V. Conclusion

In this paper, we have presented a task-based model for the efficient execution of many small to medium-sized GEMM operations on GPUs. Unlike a classical batch model where many small matrix operations are packed into a larger one and executed at once on the GPU, the proposed model assigns the independent kernels/operations to different MPI ranks. Each MPI rank transfers its operations independently to the GPU, resulting in interleaving of multiple operations on the same GPU. The GPU scheduler is then responsible for the optimal distribution of the operations (kernels) on the available GPU resources.

The model was demonstrated by simulating the frustrated one-dimensional transverse-field Ising model, a use case from theoretical physics. The model shows a significant speedup of 22% compared to the batch approach. The benchmark shows that although the batch approach has up to twice the performance per GEMM operation, our task-based approach shows higher overall performance. This shows that a model based on multiple tasks that independently offload computations to the same GPU can better utilise the GPU by running many different kernels simultaneously, reducing GPU idle time and hiding latency.

In addition, two improvements have been made over the CPU implementation. First, the GPU code is more than $35\times$ faster than the CPU-only code and second, fewer computational resources are needed to achieve the same performance or much larger simulations can be performed in terms of number of spins, number of Monte Carlo simulations and number of steps per simulation.

Acknowledgment

This work was supported by the Croatian Science Foundation under grant number HRZZ-UIP-2020-02-4559 and the European Regional Development Fund under the grant KK.01.1.1.01.009 - DATACROSS. The authors gratefully acknowledge the HPC RIVR consortium (www.hpc-rivr.si) and EuroHPC JU (eurohpc-ju.europa.eu) for funding this research by providing computing resources of the HPC system Vega at the Institute of Information Science (www.izum.si).

References

[1] E. C. Anderson, Z. Bai, C. H. Bischof, L. S. Blackford, J. W. Demmel, J. Dongarra, J. J. Du Croz, A. Greenbaum, A. Hammarling, S. McKenney, and D. C. Sorensen, *{LAPACK} Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[2] A. Haidar, S. Tomov, P. Luszczek, and J. Dongarra, "MAGMA Embedded: Towards a Dense Linear Algebra Library for Energy Efficient Extreme Computing," in *High Performance Extreme Computing Conference (HPEC)*, 2015.

[3] M. E. Guney, K. Goto, T. B. Costa, S. Knepper, L. Huot, A. Mitrano, and S. Story, "Optimizing Matrix Multiplication on Intel® Xeon Phi TH x200 Architecture," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*. IEEE, 7 2017, pp. 144–145. [Online]. Available: http://ieeexplore.ieee.org/document/8023080/

[4] K. Goto and R. A. Van De Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, 5 2008. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1356052.1356053

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 14, no. 1, pp. 1–17, 3 1988. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/42288.42291

[6] "OpenBLAS - An optimized BLAS library." [Online]. Available: https://www.openblas.net/

[7] "Intel OneAPI Math Kernel Library." [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

[8] "NVIDIA cuBLAS. Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas." [Online]. Available: https://developer.nvidia.com/cublas

[9] The University of Tenneesse, "{MAGMA} Matrix Algebra on GPU and Multicore Architectures," 2014. [Online]. Available: https://icl.cs.utk.edu/magma/

[10] J. Choi, J. Dongarra, R. Pozo, and D. Walker, "ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers," in *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Comput. Soc. Press, 1 2003, pp. 120–127. [Online]. Available: http://ieeexplore.ieee.org/document/234898/

[11] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, "A proposal for a set of parallel basic linear algebra subprograms," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer Verlag, 1996, vol. 1041, pp. 107–114.

[12] E. Agullo, L. Giraud, and M. Zounon, "On the Resilience of Parallel Sparse Hybrid Solvers," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE, 12 2015, pp. 75–84. [Online]. Available: http://ieeexplore.ieee.org/document/7397621/

[13] N. J. Higham and V. Noferini, "An algorithm to compute the polar decomposition of a 3 × 3 matrix," *Numerical Algorithms*, vol. 73, no. 2, pp. 349–369, 10 2016. [Online]. Available: http://link.springer.com/10.1007/s11075-016-0098-7

[14] O. E. Messer, J. A. Harris, S. Parete-Koon, and M. A. Chertkow, "Multicore and Accelerator Development for a Leadership-Class Stellar Astrophysics Code," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7782 LNCS, pp. 92–106, 2012. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-36803-5_6

[15] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov, "High-performance Tensor Contractions for GPUs," *Procedia Computer Science*, vol. 80, pp. 108–118, 2016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050916306536

[16] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," 10 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[17] A. Abdelfattah, S. Tomov, and J. Dongarra, "Matrix multiplication on batches of small matrices in half and half-complex precisions," *Journal of Parallel and Distributed Computing*, vol. 145, pp. 188–201, 11 2020.

[18] E. Georganas, K. Banerjee, D. Kalamkar, S. Avancha, A. Venkat, M. Anderson, G. Henry, H. Pabst, and A. Heinecke, "High-Performance Deep Learning via a Single Building Block," 6 2019. [Online]. Available: https://arxiv.org/abs/1906.06440v2

[19] E. Agullo, J. Demmel, and J. Dongarra, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, p. 012037, 7 2009. [Online]. Available: https://iopscience.iop.org/article/10.1088/1742-6596/180/1/012037

[20] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, Design, and Autotuning of Batched GEMM for GPUs," 2016, vol. 1, pp. 21–38. [Online]. Available: http://link.springer.com/10.1007/978-3-319-41321-1_2

[21] ——, "Novel HPC techniques to batch execution of many variable size BLAS computations on GPUs," in *Proceedings of the International Conference on Supercomputing*. New York, NY, USA: ACM, 6 2017, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3079079.3079103

[22] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra, "High-performance Matrix-Matrix multiplications of very small matrices," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9833 LNCS. Springer, Cham, 2016, pp. 659–671. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-43659-3_48

[23] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient GEMM on GPUs," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2 2019, pp. 229–241. [Online]. Available: https://dl.acm.org/doi/10.1145/3293883.3295734

[24] P. Jain, X. Mo, A. Jain, H. Subbaraj, R. S. Durrani, A. Tumanov, J. Gonzalez, and I. Stoica, "Dynamic Space-Time Scheduling for GPU Inference," 12 2018. [Online]. Available: https://arxiv.org/abs/1901.00041v1

[25] "Multi-Process Service :: GPU Deployment and Management Documentation." [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[26] I. Oroutzoglou, D. Masouros, K. Koliogeorgi, S. Xydis, and D. Soudris, "Exploration of GPU sharing policies under GEMM workloads," in *Proceedings of the 23rd International Workshop on Software and Compilers for Embedded Systems, SCOPES 2020*. New York, NY, USA: ACM, 2020, pp. 66–69. [Online]. Available: https://doi.org/10.1145/3378678.3391887

[27] D. Vanderbilt and S. G. Louie, "A Monte carlo simulated annealing approach to optimization over continuous variables," *Journal of Computational Physics*, vol. 56, no. 2, pp. 259–271, 11 1984.

[28] B. A. Cipra, "An Introduction to the Ising Model," *https://doi.org/10.1080/00029890.1987.12000742*, vol. 94, no. 10, pp. 937–959, 12 2018. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00029890.1987.12000742