2020 Global Smart Industry Conference (GloSIC)

Matrix-Matrix Multiplication Using Multiple GPUs Connected by Nvlink

Yea Rem Choi
National Research University Higher School of Economics
Moscow, Russia
yerem5884@gmail.com

Vsevolod Nikolskiy
National Research University Higher School of Economics
Moscow, Russia
thevsevak@gmail.com

Vladimir Stegailov
Joint Institute for High Temperatures of Russian Academy of Sciences
Dolgoprudny, Russia
stegailov@gmail.com

***Abstract*—In this work we present an original GPU-only parallel matrix-matrix multiplication algorithm ($C = \alpha A * B + \beta C$) for servers with multiple GPUs connected by NVLink. The algorithm is implemented using CUDA. The data transfer patterns, the communication and computation overlap, and the overall performance of the algorithm are considered. By regulating the commands call order and the sizes of tiles, we tune the uninterrupted asynchronous data transmission and kernel execution. Two cases are considered: when all the data are stored in one GPU and when the matrices are distributed among several GPUs. The execution efficiency of this new algorithm is compared with cuBLAS-XT from the Nvidia CUDA Toolkit library.**

***Keywords*—parallel computing, CUDA, GEMM, high-speed GPU interconnect**

I. Introduction

Recent progress in high-performance computing systems shows the leading role of GPU computing as one of the major future trends of technological development. The driven force is the high efficiencies demanded by large supercomputer machines and the slowdown of Moore's law of scaling. The major example of this kind is the CORAL systems Summit and Sierra that address the design challenges with a heterogeneous approach [1]. A key feature of CORAL nodes is the high-performance NVLink 2.0 interconnect enabling seamless GPU/CPU integration.

After the emergence of GPGPU computing, many applied algorithms and program codes have been adapted for GPU acceleration: molecular dynamics codes such as GROMACS [2], electronic structure codes such as Quantum Espresso [3], [4], particle-in-cell plasma simulation codes such as PICADOR [5], astrophysical hydrodynamics codes such as GPUPEGAS [6], [7]. This acceleration is based on using GPUs for offloading certain computationally intensive parts of the code from CPUs. The most widely used parallel programming strategy is MPI parallelization across the nodes and CUDA/OpenCL parallelization inside each node of a hybrid supercomputer (e.g. see [8], [9]). The efficiency of such an offloading is limited by the CPU-GPU data transfer bandwidth that usually limits the level of GPU utilization essentially [10], [11]. The CPU-GPU data transfers via NVLink helps with automatic GPU offloading (e.g. see [12]). In some cases, using NVlink instead of PCIe provides no effect on performance as it has been shown for VASP [13]. However, the NVLink connection between CPU and GPU is a unique feature available in IBM Power CPUs only. A much more widespread type of multi-GPU servers is represented by Nvidia DGX servers that are based on 8-16 GPUs interconnected via NVLink with PCIe connections between GPUs and CPUs.

Highest levels of computational performance of GPUs and ultrahigh bandwidth and low latency of NVLink make such multi-GPU systems a very attractive option for the development of novel high performance computing algorithms. The algorithms for mathematical modelling that perform all calculations inside GPUs only (e.g. [14]–[16]) demonstrate high efficiency. Despite promising capabilities, there are still not many software packages for mathematical modelling that make use of the high-speed GPU interconnects (e.g. see [17]).

A large number of mathematical modeling problems are fully or partially based on typical calculations using linear algebra methods, that is, on operations with matrices and vectors. Among these methods, the matrix multiplication is the most computationally expensive. In this work we describe a new GPU-only parallel matrix multiplication algorithm for multi-GPU systems with NVlink.

II. Related work

The block partitioning improves the performance of matrix multiplication algorithms for CPUs by using the processor's cache (highly optimized CPU algorithms are available, for example, in Intel MKL, IBM ESSL and openBLAS libraries). The reviews of the underlying algorithms can be found in [18]. Effective matrix multiplication algorithms for GPU accelerators require the use of multithreaded parallelism. Highly optimized versions of such algorithms are available (e.g. cuBLAS from the Nvidia CUDA Toolkit library and rocBLAS from the AMD ROCm framework).

Parallel matrix-matrix multiplication algorithms requires optimization of data transfers between the nodes of a distributed memory system [19]. The recent state-of-the-art algorithms are reviewed in [20]. Several implementations are available (e.g. ScaLAPACK, PLAPACK, Elemental, DPLASMA, SLATE, PARSEC).

GPGPU algorithms for matrix-matrix multiplication are under development since the beginning of the GPGPU era [21]– [24]. There has been the only publicly available library for dense linear algebra kernels on multi-GPU accelerated distributed memory platforms called SLATE [25]. Recently, a further development has appeared called PARSEC that is aimed for matrices unrestricted by the size of the GPU memory [26].

SLATE and PARSEC are complex and multipurpose software projects. The aim of this work is to develop a much simpler matrix-matrix algorithm that suits better for the purpose of benchmarking different types of GPUs and GPU-GPU interconnects.

III. Testing Platform

The results reported in this study have been obtained on the nodes of the Harisma supercomputer in NRU HSE [27]. The nodes are based on the DELL PowerEdge C4140 servers with two Intel Xeon Gold 6152 CPUs and four Nvidia Tesla V100 GPUs (Fig. 1). Each GPU has 32 Gb of HBM2 memory and four GPUs are connected by NVLINK 2.0 forming a fully connected ('all-to-all') topology.

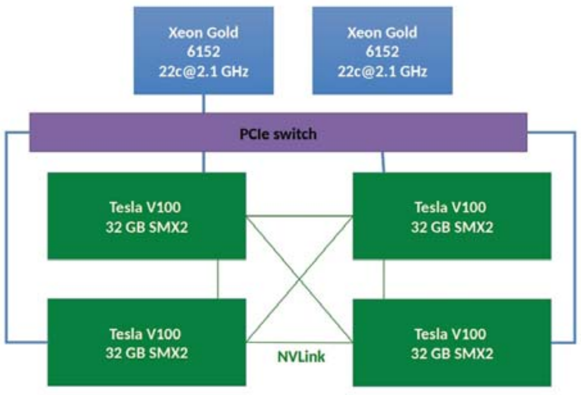

Fig. 1. The topology of the DELL PowerEdge C4140M server with two CPUs and four Nvidia Tesla V100 GPUs connected by NVLINK 2.0.

The benchmarking studies presented in this work have been carried out using the standard HPC software stack based on CentOS Linux release 7.6.1810, the GNU compilers 7.3, and the CUDA Version 10.2.89 with the driver ver. 440.33.01.

IV. Parallel Matrix-Matrix Multiplication Algorithm for Multiple GPUs

In this work we describe an original general matrix-to-matrix multiplication algorithm for the following matrix operation

$$C = \alpha A * B + \beta C.$$

Here we use 2 column-oriented square matrices A and B with $N * N$ elements, where each matrix is divided to some number of equal sized bands to be able to share computational load between different GPUs. Then from the pairs of bands we find the appropriate tiles of resulting matrix C (see Fig. 2).

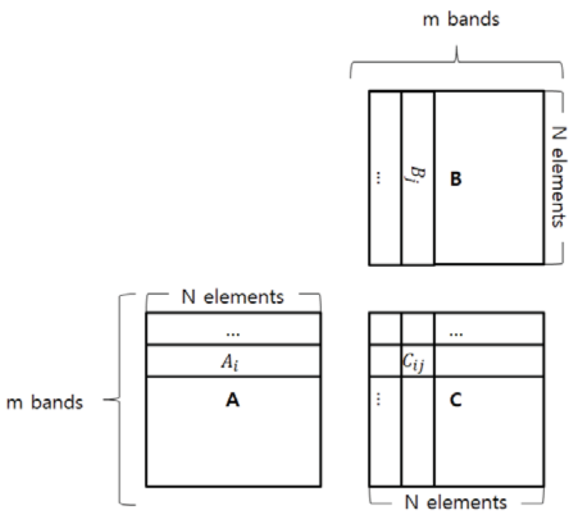

Fig. 2. The scheme shows the matrices A, B divided into bands (A_i and B_j) to compute the tiles C_{ij} of the resulting matrix C.

During the whole process, we use only GPUs for calculation and data storage. This restriction strongly reduces the amount of available memory, but we can involve high performance communication via NVLink that connects GPUs within a node. In this work we do not discuss the application of the unified memory technology that will be considered in further studies. The division of matrices and storage of their parts in memories of different GPUs are not suitable for our research purpose. We store the data of each matrix A, B and C fully in the memory of the selected GPU. The master-worker scheme is applied where the master does computations as well.

The general scheme of the process is as follows:

- the devices that store matrices A and B send bands A_i and B_j to other GPUs,
- the GPUs perform matrix multiplication with α using the received bands and get the intermediate tiles C_{ij},
- the device with the matrix C gathers tiles C_{ij} and sums the resulting matrix with βC.

To optimize the data transfer process, the data access and the data reuse, we allocate two bands for matrices A, B, and C in each GPU except the ones where matrices are stored (bands of C are allocated in all devices including the one with matrix C). It allows us to perform computations with one set of bands and to perform communications with others simultaneously.

We do not transfer the original data of matrix C to do addition with the purpose of complexity reduction. Nevertheless, the bands of matrix C are needed to collect tiles and are sent in a long data queue at one step. This has three advantages: firstly, the data transmission for a long line is more effective than for several short ones. Secondly, send or receive operations cause barriers, so a broadcast makes the algorithm more complicated if more GPUs are deployed. Thirdly, that is simpler to profile and pick up a suitable case with a more optimized transferring order.

We can explain the algorithm as a nested loop (see Alg. 1, 2). In Alg. 1 if the considered GPU does not store matrices, during the outer loop we receive a band of matrix *A*, then during the inner loop we alternately receive bands of *B*, compute an appropriate tile of *C* and locate it in a band *C* . The obtained band *C* is sent then at the stage of the outer loop to the device where matrix *C* is stored. Speaking of devices with matrices *A* or *B*, the data transmission step is skipped and the direct access to the required data is set instead. If the memory of the device contains matrix *C*, then on the stage of the outer loop it receives bands C_i from other devices, performs the addition of βC_i, and stores the result in *C*. In Alg. 2 the full algorithm is illustrated.

Algorithm 1 The schematic process in a default worker GPU, *m* is the number of bands in a row (column).

```
for outerloop = 0 to m/NumOfGpus do
    receive (bandA);
    for innerloop = 0 to m do
        recieve (bandB);
        GEMM (bandA, bandB, alpha, tileC);
        write TileToBand (tileC, bandC);
    end for
    send (bandC);
end for
```

Algorithm 2 The full algorithm with the changed data transfer order and different commands for different devices.

```
if (device = device withA) then
    receive (bandA);
end if
for outerloop = 0 to m/NumOfGpus do
    for innerloop = 0 to m do
        if (innerloop < m 1) and (device = device withB)
        then
            recieve (bandB);
        end if
        GEMM (bandA, bandB, alpha, tileC);
        if (innerloop = 0) then
            if (device = device withB) then
                recieve (bandB);
            end if
            if (outerloop < m/NumOfGpus 1) and (device =
            device withA) then
                Receive (bandA);
            end if
        end if
    end for
    send (bandC);
    if (device == device withC) then
        AddbetaC (C, beta, bandC);
    end if
end for
```

We use asynchronous data transfers and use different queues (Fig. 3). We do verification whether data is obtained only before computation and the data are used completely before rewriting during the next step.

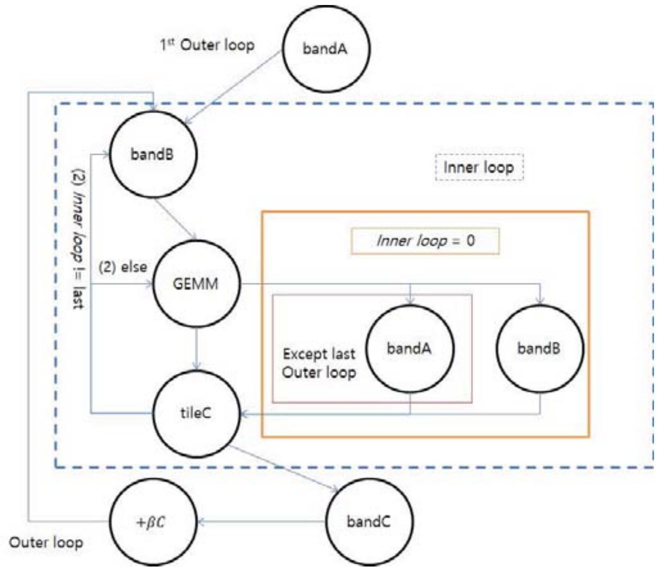

Fig. 3. The graphical scheme of the algorithm version. The whole algorithm works in the frame of the outer loop. Data transfer commands may be called in a different order if the data usage before rewriting is verified.

In Fig. 3 we can see the necessity of 2 bands allocation for a same matrix. In the beginning of a loop the device gets data into one of the bands. Then it executes the GEMM function from cuBLAS library with the received data. During the kernel computation, the next part of the matrix is stored into another band. Correspondingly, the next GEMM is reserved in the queue, which the device launches it when the able computational resource shows out. When all works with the previous band usage have been done, we can free it and repeat the process for next data set. To remark how the data is reused, bands for different matrices have different lengths of the changing cycle. Ones are changed more frequently whenever others have been reused to be matched with all from the first matrix band line.

Here we also should make a notice, that for simple transmission of data, matrix *A* is stored in the transposed form. It allows us to transfer data of bands *A* as long lines.

V. Parallel Matrix-Matrix Multiplication Algorithm Performance Benchmarks

Nvidia provides a set of libraries for the convenient use of accelerators. One of such is cuBLAS, which contains a list of simple matrix operations. The general matrix-to-matrix multiply (GEMM) operations are of particular interest. Here we can note that it is enough to use the cuBLAS library in case of single GPU system, but it does not support the implementation for multiple GPUs. Multiple GPU functionality is supported by another library called cuBLASXT. It divides matrices into tiles which are sent to GPUs for computation and gathers the results. Here we note, that in this research we observed the cases when all available devices do computation whatever they store matrix or not.

A. with cuBLAS-XT

Exploring the performance, we found, that the efficiency falls while we work with rather small matrices as expected. However, there are issues that do not depend on the size. The crucial one turned up when the experiments with more than 2 GPUs were being held. On average, the computation performance fell down to 60% of peak performance with 3 GPUs, 40% with 4 GPUs, in contrast to the case of 1 or 2 GPUs, when performance was close to the maximum ability of the devices.

The size of tiles was the parameter that we were able to regulate. Fig. 4 and Fig. 5 shows the most efficient cuBLASXT GEMM executions. In the experiments it was established that in the case when matrices are located in one device, the best performance can be achieved when size of tiles is a quarter to an original square matrix. Otherwise, there was some interval of the parameter ($N_i \in (8000; 10000)$, N_i=N/m) when it shows the peak.

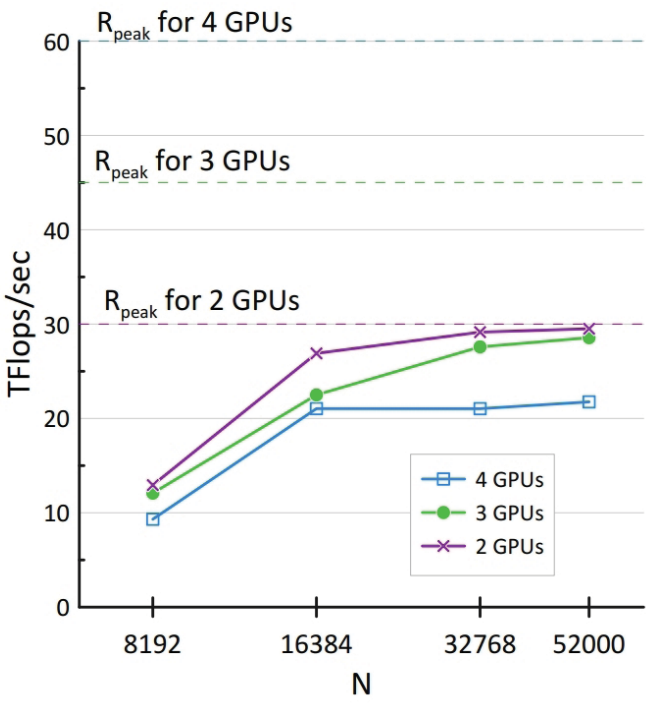

Fig. 4. The graph of multi-GPU GEMM operation in cuBLAS-XT performance speed on 2, 3, 4 GPUs by numbers of elements of matrices (N) in a row (column). Matrices A, B, and C are stored together in the device with id 0. The dashed lines show the total single precision peak performance of 2, 3 and 4 GPUs respectively.

To analyze the issue we did profiling by Nvidia Visual Profiler (see Fig. 6). We got some confusing results as, despite the purpose of cuBLAS-XT is effective implementation of multiple GPUs, it does synchronous launching if more than 2 GPUs are in work. We tried to regulate parameters and other launch characteristics to fix this issue, but we could not find the excuse outside of the executable GEMM function.

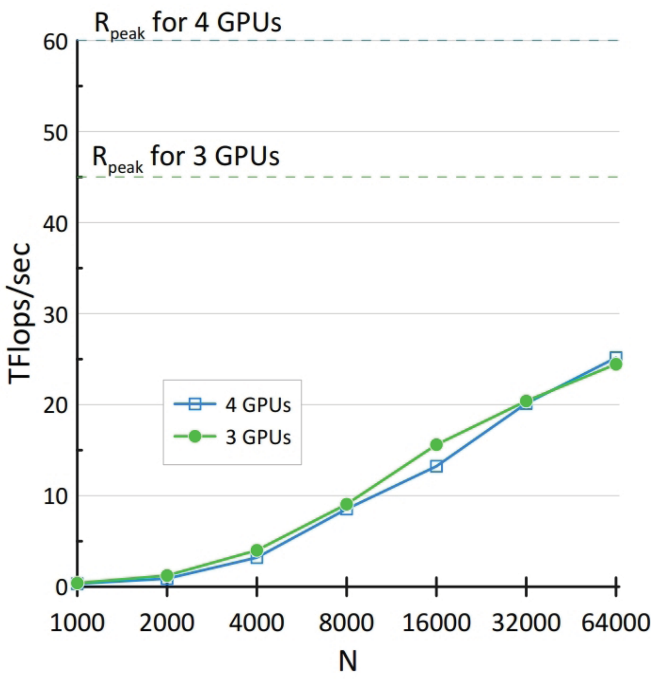

Fig. 5. The graph of multi-GPU GEMM operation in cuBLAS-XT performance speed on 3 and 4 GPUs by numbers of elements of matrices (N) in a row (column). Matrices A, B, and C are stored respectively in devices with id 0, 1, and 2. The dashed lines show the total single precision peak performance of 3 and 4 GPUs respectively.

During the work on the original algorithm, special attention was paid to the asynchronous memory operating functions. There are several reasons, firstly, if they were called in one device, each call interfere the others until it has not completed its job. In Fig. 6 by the behavior of cuBLAS-XT we can suspect that the problem is related. Secondly, we had to place barriers to make kernel work with the necessary information. Therefore, to supply data before GEMM was called the additional bands were allocated in each device which receives data while the other was involved in the computation.

B. with the proposed algorithm

To deal with the simplest cases, with such restrictions as the kernel dimensions limits and task division in equal parts between devices, we used the matrices with $N = 2^x$, where x=10-16. By this moment we have developed the program that is able to utilize more than 80% of peak performance of the device on multiple GPUs for rather large matrices ($N \geq 2^{15}$). The main strategy was reducing waiting time of each device, by continuous kernel execution and data supply via NVlink (see Fig. 7).

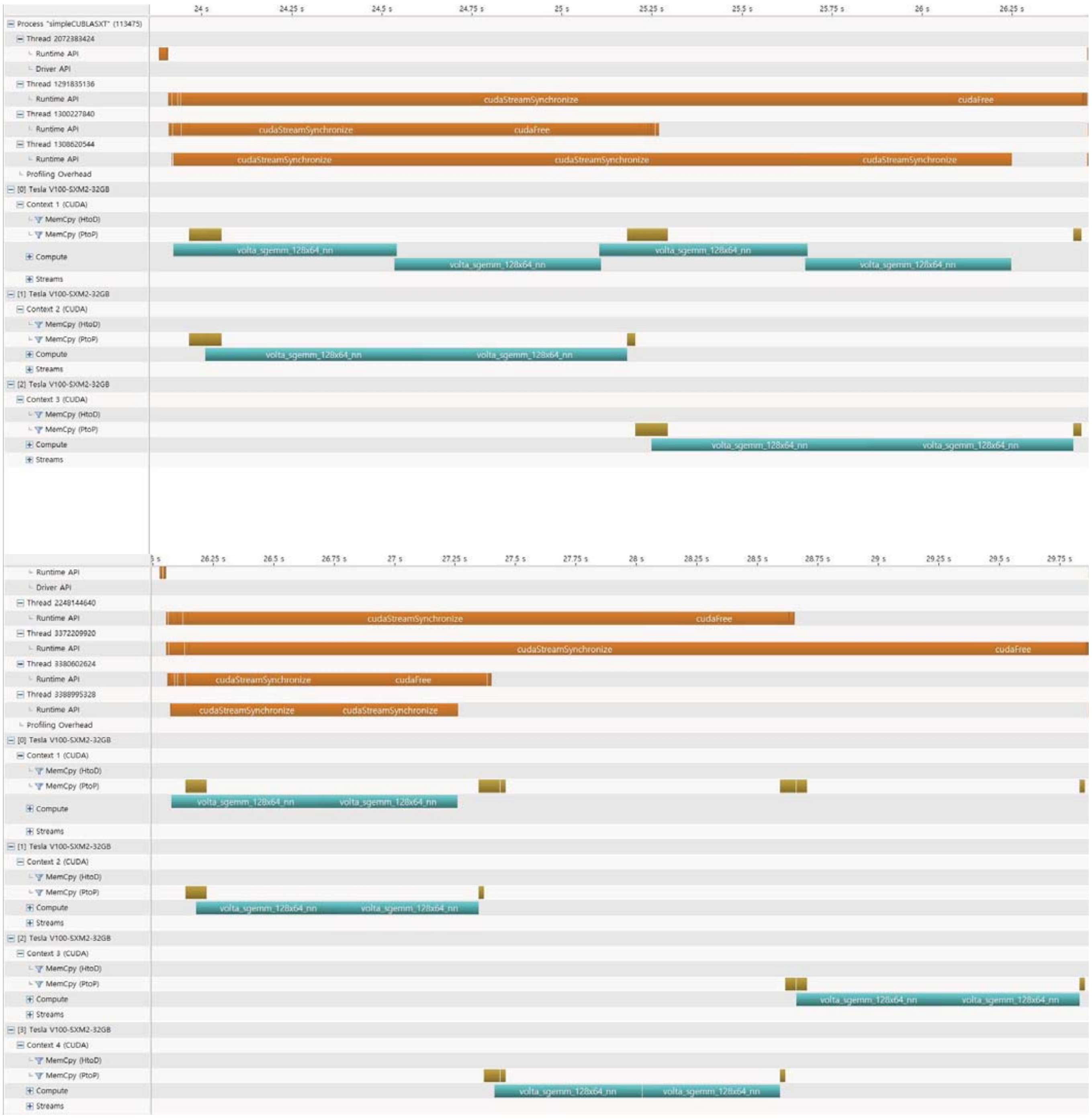

Fig. 6. The profile of multi-GPU GEMM operation in cuBLAS-XT performance on 3 GPUs (upper) and 4 GPUs (lower). Number of elements ($N = 2^{15}$) in a row (column) of matrices. Only up to 2 devices are working simultaneously and then it does synchronous kernel execution.

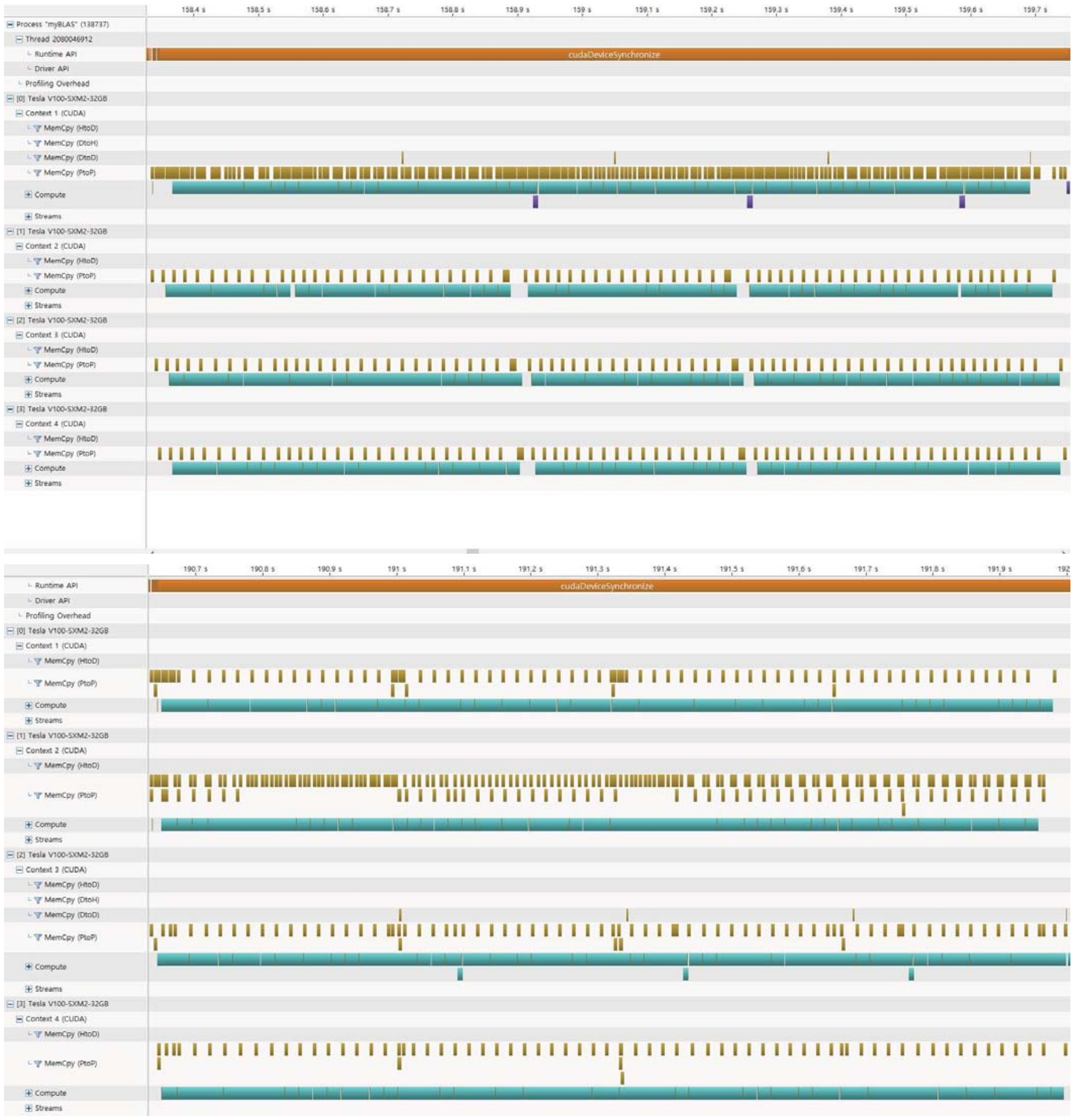

Fig. 7. The profile of multi-GPU GEMM operation with the proposed algorithm performing on 4 GPUs. Number of elements ($N = 2^{15}$) in a row (column) of matrices and number of bands ($m = 16$, $N_t^2 = (2^{11})^2$ elements in a tile). Matrices A, B, and C are stored in the device with id 0 (upper) or stored in devices with id 0, 1, and, 2 (lower).

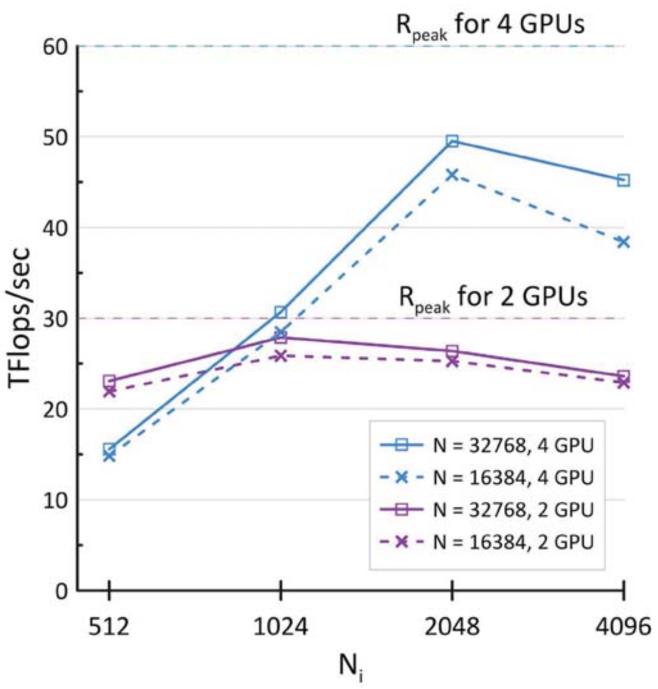

Fig. 8. The graph of multi-GPU GEMM operation with the proposed algorithm performance speed on 2 and 4 GPUs by size of tiles (N_i) for different numbers of elements (N) in a row (column) of matrices. Matrices *A*, *B*, and *C* are stored in the device 0. The dashed lines show the total single precision peak performance of 2 and 4 GPUs respectively.

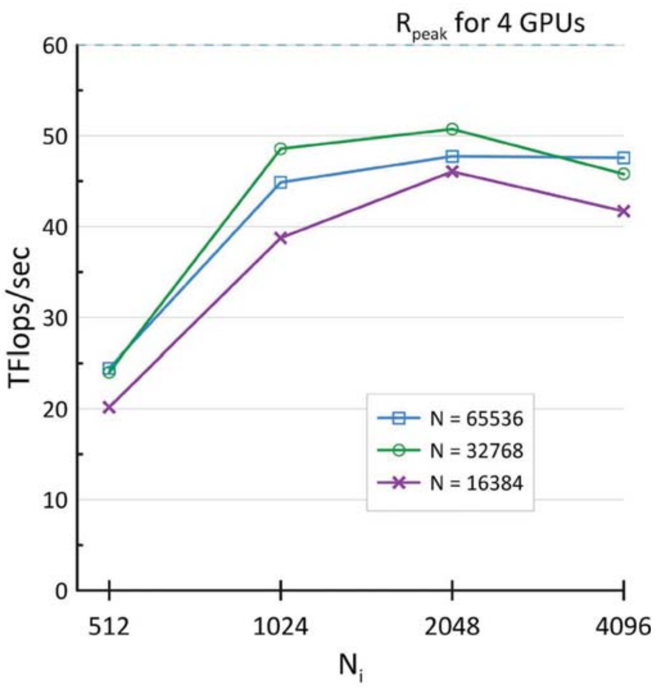

Fig. 9. The graph of multi-GPU GEMM operation with the proposed algorithm performance speed on 4 GPUs by size of tiles (N_i) for different numbers of elements (N) in a row (column) of matrices. Matrices *A, B*, and, *C* stored respectively in devices 0, 1, and 2. The dashed line shows the total single precision peak performance of 4 GPUs respectively.

First of all, it works with bands of matrices for easy data access. This spreads GEMM execution in multiple devices based on the expectation that we have fast enough data transmission instrument. To research the performance we analyzed the dependency of computation time by task amount and size of bands in cases when all matrices are stored in one device or when stored individually in several devices.

Because we met a high data limit (32 Gb in a device) we could launch tests with the largest size of a matrix $N = 32768$ when they were stored together and $N = 65536$ when they were stored separately. In Fig. 8 and Fig. 9 for the case of NVidia Tesla V100 GPUs connected by NVlinks the found optimal size of tiles is $N_i = 2^{11}$ when we use 4 GPUs and $N_i = 2^{10}$ when use 2 GPUs. Nonetheless, the behavior of the performance graph makes us suppose that it could be bigger for larger matrices if we had devices with more memory. Primarily two facts may affect. One is the execution efficiency of GEMM operation and total data transfer time increase when devices work with smaller matrices. Another is that the larger matrix require more time to data transfer. It makes the devices where no original matrix is stored wait longer before beginning computation. We found, that the best performance we get when N = 32768 (see Fig. 10).

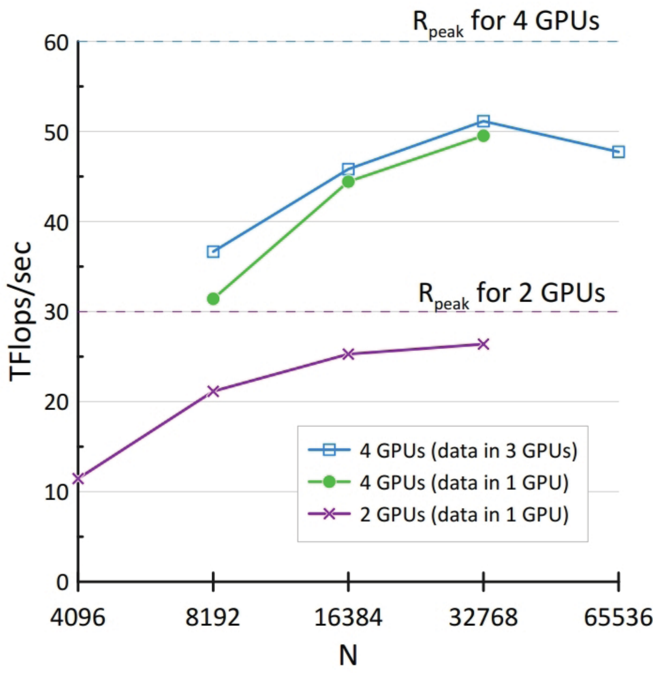

Fig. 10. The graph of multi-GPU GEMM operation with the proposed algorithm performance speed on 2 and 4 GPUs by numbers of elements of matrices (N) in a row (column) for optimal size of tiles (N_i = 1024 for 2 GPU and N_i = 2048 for 4 GPU). Matrices *A, B*, and, *C* stored together in the device 0 or stored respectively in devices 0, 1, and, 2. The dashed lines show the total single precision peak performance of 2 and 4 GPUs respectively.

VI. Conclusion

The algorithm for multiple GPUs GEMM computation working without exploiting the host processor is developed. The advantage is the continuous data transmission and kernel execution provision. Despite the fact that GPU-only algorithm allows one to use less memory, it could reach a large percentage of peak performance about 80% for 4 GPUs and 86% for 2 GPUs for sufficiently large matrices. Also, we have found the optimal sizes of tile matrices when the computation speed is definitely superior.

The performance of the proposed algorithm is higher than cuBLAS-XT when 4 GPUs are used. After profiling we have explained this difference by the inefficient communication patterns in cuBLAS-XT. On the contrary, our new algorithm has demonstrated very efficient overlapping of communication and computation. Among observed there is the case with the scattered data storage in several GPUs that is faster than the case when all the data are stored in one GPU. The reason is that the communication workload is shared among several NVlink links that results in a more balanced communication pattern.

ACKNOWLEDGMENT

The article was prepared within the framework of the HSE University Basic Research Program. This research was supported in part through resources of supercomputer facilities provided by NRU HSE.

REFERENCES

[1] W. A. Hanson, “The CORAL supercomputer systems,” IBM Journal of Research and Development, vol. 64, no. 3/4, pp. 1:1–1:10, 2020.

[2] M. J. Abraham, T. Murtola, R. Schulz, S. P´all, and J. C. Smith, “GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers,” SoftwareX, vol. 1–2, pp. 19–25, 2015.

[3] F. Spiga and I. Girotto, “phiGEMM: A CPU-GPU library for porting Quantum ESPRESSO on hybrid systems,” 2012 20th Euromicro Int. Conf. on Parallel, Distributed and Network-based Processing, pp. 368–375, 2012.

[4] J. Romero, E. Phillips, G. Ruetsch, M. Fatica, F. Spiga, and P. Giannozzi, “A performance study of Quantum ESPRESSO’s PWscf code on multicore and GPU systems,” High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation, pp. 67–87, 2018.

[5] S. Bastrakov, R. Donchenko, A. Gonoskov, E. Efimenko, and A. Malyshev, “Particle-in-cell plasma simulation on heterogeneous cluster systems,” Journal of Computational Science, vol. 3, no. 6, pp. 474–479, 2012.

[6] I. Kulikov, “GPUPEGAS: A new GPU-accelerated hydrodynamic code for numerical simulations of interacting galaxies,” The Astrophysical Journal Supplement Series, vol. 214, no. 1, 2014.

[7] E. Akimova, V. Misilov, I. Kulikov, and I. Chernykh, “Hydrodynamical simulation of astrophysical flows: High-performance GPU implementation,” Journal of Physics: Conference Series, vol. 1336, 2019.

[8] V. Stegailov, E. Dlinnova, T. Ismagilov, M. Khalilov, and N. Kondratyuk, “Angara interconnect makes GPU-based desmos supercomputer an efficient tool for molecular dynamics calculations,” The Int. Journal of High Performance Computing Applications, vol. 33, no. 3, pp. 507–521, 2019.

[9] V. P. Nikolskiy and V. V. Stegailov, “GPU acceleration of four-site water models in LAMMPS,” Advances in Parallel Computing, Parallel Computing: Technology Trends. Proceedings of PARCO-2019, vol. 36, pp. 565–573, 2019.

[10] I. Morozov, A. Kazennov, R. Bystryi, G. Norman, V. Pisarev, and V. Stegailov, “Molecular dynamics simulations of the relaxation processes in the condensed matter on GPUs,” Computer Physics Communications, vol. 182, no. 9, pp. 1974–1978, 2011.

[11] G. Smirnov and V. Stegailov, “Efficiency of classical molecular dynamics algorithms on supercomputers,” Mathematical models and computer simulations, vol. 8, no. 6, pp. 734–743, 2016.

[12] S. Mal’kovskii, A. A. Sorokin, S. P. Korolev, A. Zatsarinnyi, and G. Tsoi, “Performance evaluation of a hybrid computer cluster built on IBM POWER8 microprocessors,” Programming and Computer Software, vol. 45, no. 6, pp. 324–332, 2019.

[13] V. Stegailov, G. Smirnov, and V. Vecher, “VASP hits the memory wall: Processors efficiency comparison,” Concurrency and Computation: Practice and Experience, vol. 31, no. 19, 2019.

[14] J. A. Anderson, C. D. Lorenz, and A. Travesset, “General purpose molecular dynamics simulations fully implemented on graphics processing units,” Journal of Computational Physics, vol. 227, no. 10, pp. 5342–5359, 2008.

[15] N. Luehr, I. S. Ufimtsev, and T. J. Mart´ınez, “Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs),” Journal of Chemical Theory and Computation, vol. 7, no. 4, pp. 949–954, 2011.

[16] K. Rojek, R. Wyrzykowski, and L. Kuczynski, “Systematic adaptation of stencil-based 3D MPDATA to GPU architectures,” Concurrency and Computation: Practice and Experience, vol. 29, no. 9, 2016.

[17] J. Glaser, P. S. Schwendeman, J. A. Anderson, and S. C. Glotzer, “Unified memory in HOOMD-blue improves node-level strong scaling,” Computational Materials Science, vol. 173, 2020.

[18] K. Goto and R. A. V. d. Geijn, “Anatomy of high-performance matrix multiplication,” ACM Trans. Math. Softw., vol. 34, no. 3, pp. 12–1–12–25, 2008.

[19] M. D. Schatz, R. A. Van de Geijn, and J. Poulson, “Parallel matrix multiplication: A systematic journey,” SIAM Journal on Scientific Computing, vol. 38, no. 6, pp. C748–C781, 2016.

[20] G. Kwasniewski, M. Kabi´c, M. Besta, J. VandeVondele, R. Solc`a, and T. Hoefler, “Red-blue pebbling revisited: Near optimal parallel matrixmatrix multiplication,” Proc. of the Int. Conf. for High Performance Computing, Networking, Storage and Analysis, pp. 24–1–24–22, 2019.

[21] J. Dongarra, J.-F. Pineau, Y. Robert, and F. Vivien, “Matrix product on heterogeneous master-worker platforms,” Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 53–62, 2008.

[22] A. DeFlumere and A. Lastovetsky, “Searching for the optimal data partitioning shape for parallel matrix matrix multiplication on 3 heterogeneous processors,” 2014 IEEE Int. Parallel & Distributed Processing Symposium Workshops, pp. 17–28, 2014.

[23] D. Rohr and V. Lindenstruth, “A flexible and portable large-scale DGEMM library for linpack on next-generation multi-gpu systems,” 2015 23rd Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing, pp. 664–668, 2015.

[24] S. Ryu and D. Kim, “Parallel huge matrix multiplication on a cluster with GPGPU accelerators,” 2018 IEEE Int Parallel and Distributed Processing Symposium Workshops, pp. 877–882, 2018.

[25] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, “Slate: Design of a modern distributed and accelerated linear algebra library,” Association for Computing Machinery, pp. 26–1–26–18, 2019.

[26] T. Herault, Y. Robert, G. Bosilca, and J. Dongarra, “Generic matrix multiplication for multi-gpu accelerated distributed-memory platforms over parsec,” 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, pp. 33–41, 2019.

[27] N. Kondratyuk, G. Smirnov, A. Agarkov, and A. Osokin, “Performance and scalability of materials science and machine learning codes on the state-of-art hybrid supercomputer architecture,” Communications in Computer and Information Science. Supercomputing, pp. 597–609, 2019.