10.24425/acs.2026.158422

Archives of Control Sciences
Volume 36(LXXII), 2026
No. 1, *pages 85–98*

Practical aspects of fast matrix multiplication

Marek KUBALE and Damian NIEMCZYK

The aim of this paper is to analyze the development of algorithms for *Fast Matrix Multiplication* (FMM) in both historical and technical contexts, as well as to compare available solutions on consumer-grade computer hardware. We review advancements in estimating the theoretical computational complexity of FMM and optimization techniques that are used in widely adopted algorithms, with a particular focus on optimal cache memory usage and leveraging *Graphics Processing Units* (GPU). The methodology of tests and their analysis highlight the performance differences of the considered algorithms depending on the matrix size and the nature of the data stored in them. Results indicate the significant role of tailoring the chosen algorithm to the available hardware and the specific application in which the algorithm is being performed. Also, we emphasize that the FMM algorithms can be applied not only to linear algebra problems but also to current problems in science and engineering, such as artificial intelligence, databases, parallel computations, computational biology, pattern recognition, and compiler construction, to mention just a few examples.

Key words: AlphaTensor, computational complexity, fast matrix multiplication, linear algebra

1. Introduction

Matrix multiplication is a fundamental operation of linear algebra [10]. This is one of the most frequently performed operations both on personal computers and on clusters or supercomputers. The popularity of matrix multiplication results from the versatility of its applications. Over the years, many multiplication algorithms have been created, various implementations of which are currently used in practice. It is not easy to determine which solutions behave best in a specific application. This depends on many factors related to the input data, the available computer hardware, and even the characteristics of the execution of

M. Kubale (corresponding author, e-mail: m.kubale@wi.umg.edu.pl) is with Gdynia Maritime University, Faculty of Computer Science, Gdynia, Poland.

D. Niemczyk (e-mail: s188867@student.pg.edu.pl) is with Gdańsk University of Technology, Faculty of Electronics, Telecommunications and Informatics, Gdańsk, Poland.

Received 22.06.2025. Revised 30.10.2025.

multiplications within most of the program. The aim of this paper is to analyze the impact of some of these factors on the efficiency of computations and to present the development of algorithms in a historical and technical context.

We recall that the speed of execution is not the only important issue of numerical algorithms. Calculations are performed on computer hardware, and therefore the results are burdened with certain errors. It is therefore very important to create numerically stable algorithms, i.e. such that do not introduce significant errors during calculations. This is particularly important where the goal is to obtain the most accurate result possible. Generally, computer science is constantly developing and affects more and more areas of human life. Numerical errors in computing have often caused unforeseen and unacceptable damage, as exemplified by the error in the American Patriot air defense system [11]. That is why the risk associated with the lack of numerical stability should not be underestimated.

Fast matrix multiplication algorithms cannot achieve component-wise stability, but some can be shown to exhibit norm-wise stability. It is very useful for large matrices over exact domains such as finite fields, where numerical stability is not an issue, e.g. multimedia where imprecise results are acceptable. One can therefore consider creating separate algorithms for them, where reduced accuracy allows for increased performance.

In the next section we give a brief history of the development of the FMM algorithms. The motivations behind this development and the progress in more specialized algorithms are discussed. The progress in theoretical time complexity is depicted in a tabular form. The role of computer hardware in constructing practical algorithms is highlighted and some hardware optimization techniques used today are cited.

The main Sections 3 and 4 contain information on the computer experiments with FMM. The test environments and the method of performing measurements are presented along with comparisons of available solutions. The performance of implementations was compared with respect to the size of the matrices, the type of data stored in the multiplied matrices, the density of the matrices and the computer components used to calculate them. The influence of the communication time between the processor and the graphics card on the speed of multiplication was also discussed. Section 4 is devoted to test results and plots, while the last section summarizes the obtained results.

2. History of FMM

Matrix multiplication is a fundamental operation in numerous modern applications. However, it is very time-consuming compared to other operations that are performed in computer programs. It is often the main performance limitation

of more complex operations. In such cases, the only way to significantly speed up the whole computation is to speed up this multiplication.

Directly applying the mathematical definition of matrix multiplication gives an algorithm that requires n^3 field operations to multiply two $n \times n$ matrices over that field, in symbols $O(n^3)$. There are algorithms that require a lower number of multiplications than this naive algorithm. The first to be discovered was Strassen's algorithm, devised by Volker Strassen in 1969 and often referred to as "fast matrix multiplication" [12]. The optimal number of field operations needed to multiply two square $n \times n$ matrices up to constant factors is still unknown. This is a major open question in theoretical computer science.

As of January 2024, the best bound on the asymptotic complexity of a matrix multiplication algorithm is $O(n^{2.371339})$ [2]. However, this and similar improvements to Strassen are not used in practice, because they are galactic algorithms: the constant coefficient hidden by the big O notation is so large that they are only worthwhile for matrices that are too large to handle on present-day computers. Moreover, these algorithms are difficult to optimize for modern hardware architectures, making them impractical despite their theoretical efficiency.

In Table 1 we cite after [13] the state-of-the-art concerning the *matrix multiplication exponent* ω, where $n^{\omega+o(1)}$.

Unlike most algorithms with faster asymptotic complexity, Strassen's algorithm is used in practice. The numerical stability is reduced compared to the naive algorithm, but it is generally faster in cases where n is large, and appears in some modern libraries, implementing the *Basic Linear Algebra Subprograms* (BLAS) specification [7, 9]. These implementations, however, benefit from hardware-specific optimization techniques, which make them significantly faster than a direct implementation of the algorithm from its theoretical definition. The naive algorithm is exceptionally easy to optimize this way and, as a result, is still the fastest choice for multiplying matrices that are not very large.

Optimization techniques primarily focus on parallel computation and reducing communication overhead between processes and hardware components. In fact, communication complexity has become the dominant factor in determining the practical efficiency of matrix multiplication algorithms. Theoretical studies in [3] and [4] explore this aspect. The results align closely with the execution times observed in practice. Some of the most important optimization concepts include: *cache-awareness*, *auto-tuning*, *block algorithms*, and *algorithm switching*. Moreover, efficient implementations might either be *cache-oblivious* or parametrized to allow for optimal memory utilization on a given machine. For more details one might refer to [1].

In October 2022, an article [5] was published, presenting a groundbreaking approach to finding new matrix multiplication algorithms obtained using

Table 1: Timeline of matrix multiplication exponent ω; for relevant references see [13]

Year	ω	Authors
?	3.000	unknown
1969	2.8074	Strassen
1978	2.796	Pan
1979	2.780	Bini, Capovani, Romani
1981	2.522	Schönhage
1981	2.517	Romani
1981	2.496	Coppersmith, Winograd
1986	2.479	Strassen
1990	2.3755	Coppersmith, Winograd
2010	2.3737	Stothers
2012	2.3729	Williams
2014	2.3728639	Le Gall
2020	2.3728596	Alman, Williams
2022	2.371866	Duan, Wu, Zhou
2024	2.371552	Williams, et al.
2024	2.371339	Alman, et al.

Artificial Intelligence (AI) (for the first Polish mentioning of this achievement, see [6]). Authors of that paper applied the *AlphaTensor* agent trained using deep reinforcement learning to find new algorithms that can be represented as a matrix multiplication tensor. The algorithms with a tensor decomposition include, among others, the naive algorithm and Strassen's algorithm. The agent was trained to play a single-player game in which it was rewarded for finding algorithms that performed a small number of operations. Many thousands of correct algorithms were found and combined to produce algorithms for larger tensors. For over 70 sizes (not necessarily square) of matrices, algorithms with better complexity than those previously known were found. The most significant were algorithms that allowed multiplication of matrices of size 4×4 using only 47 multiplications. Previously known algorithms required at least 49 multiplications for this purpose like the one based on Strassen's approach. The practical implementation of the *AlphaTensor* algorithm is based on matrices of size 4×4, thus obtaining the overall complexity of $O(n^{\log_4 47}) \approx O(n^{2.777})$. This approach also enabled tailoring algorithms to specific hardware by adjusting the reward function to additionally account for execution time on the target device.

3. Computer experiments

Libraries

The naive algorithm has been implemented in C++20 without any optimizations used in modern libraries. The tests were intended to show the difference in performance compared to contemporary optimizations. Consequently, the following libraries implementing BLAS specification for CPU were selected:

1. ATLAS
2. BLIS OPENMP
3. BLIS SERIAL
4. BLIS THREADS
5. NETLIB
6. OPENBLAS OPENMP
7. OPENBLAS SERIAL
8. OPENBLAS THREADS
9. Intel oneMKL

The Intel oneMKL library was tested in the MATLAB environment, and the remaining libraries in the R environment. Additionally, tests were performed using Intel oneMKL for sparse matrices with 1% and 10% density. In the following, the name Intel oneMKL is shortened to MKL.

Also, the MAGMA library, implementing BLAS for GPU, was tested. The tests were performed in the MATLAB environment for sparse matrices with density of 1% and 10%. The tests were also performed for the AlphaTensor algorithm tailored for GPU and *Tensor Processing Unit* (TPU) and for the implementation of the classical multiplication of the JAX library for Python. In the Python environment, all tests were optimized by JIT (just-in-time) compilation of the JAX library. The standard algorithm from the JAX library with JIT compilation will be referred to as JAX JIT, for short.

Hardware

A computer equipped with an Intel Core i5-13400F processor, Nvidia RTX 3060 12 GB graphics card, and 32 GB of Kingston Renegade 3200 MHz RAM was used. Operating system Ubuntu 22.04.2 LTS x86_64 was used. Tests using Intel MKL and MAGMA libraries were performed in MATLAB R2023b. Tests of other libraries were performed using FlexiBLAS 3.4.0 library in R 4.1.2 environment. AlphaTensor and JAX JIT tests were performed in Python 3.11 environment, modified in a way that allows performing necessary measurements without modifying the operation of algorithms.

Measurement

The following quantities were measured: (1) matrix multiplication time – the time from the moment immediately before the multiplication operation is performed to the moment the result is saved in the CPU or GPU memory. Before the multiplication was performed, the matrices were already initialized. (2) communication time between the CPU and GPU – the time of sending a single dense matrix from one component to another.

Measurements were performed for single- and double-precision floating-point numbers with random values based on the normal distribution. In addition to ordinary dense matrices, the multiplication of sparse matrices with densities of 1% and 10% was tested. Tests were performed for square matrices of sizes taken from the set: {2, 3, 4, 5, 6, 7, 8, 10, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 512, 1024, 1536, 2048, 3072, 4096, 5120, 6144, 7168, 8192, 9216, 10240, 11264, 1288, 13312, 14336, 15360, 16384}. Some of these values were omitted for some algorithms due to performance limitations or limitations of the algorithms themselves. Tests were performed with minimal load on the operating system by other processes. Each measurement was taken 1000 times for matrices of order at most 1024 and 100 times for larger matrices with the average value computed in each case.

4. Graphical plots

In this section we give several graphs obtained while testing selected algorithms on certain platforms. In all the plots time is given in seconds, i.e. y-axis, and x-axis corresponds to the size of matrices. For full data and details, we refer the reader to [8]. Some results for small matrix sizes are characterized by a relatively high standard deviation, often comparable in size to the result itself. This concerns mainly algorithms tested in R and MATLAB environments.

Figure 1 shows a comparison of GPU versus *Central Processing Unit* (CPU) for single- and double-precision numbers. Multiplication of matrices on CPU used the implementation of MKL from Intel library, while multiplication on GPU used MAGMA. All measurements were made in MATLAB. The multiplication of numbers with double-precision representation is slower for both CPU and GPU, with the difference being greater on the GPU for large matrices. For small matrices, the CPU performs multiplication faster for both types of numbers, and for large matrices only for double-precision numbers.

Figures 2 to 4 depict a comparison of the efficiency of available BLAS implementations on CPU. Intel MKL implementation turned out to be the best overall. In some cases, the difference between implementations were of 2 orders

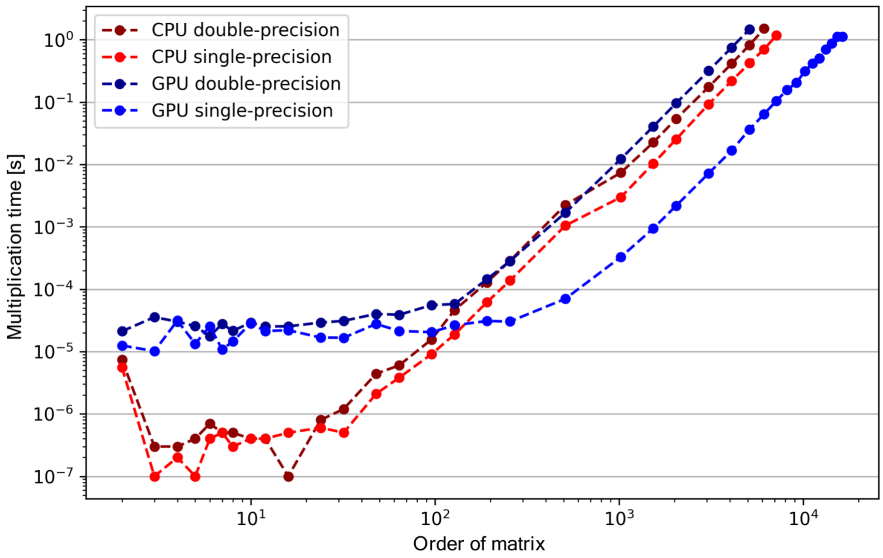

Figure 1: Relationship between the size of matrix and the average time of multiplication for single- and double-precision for CPU and GPU

of magnitude. For very small matrices, the standard deviation of the results was large, comparable to the magnitude of the result itself.

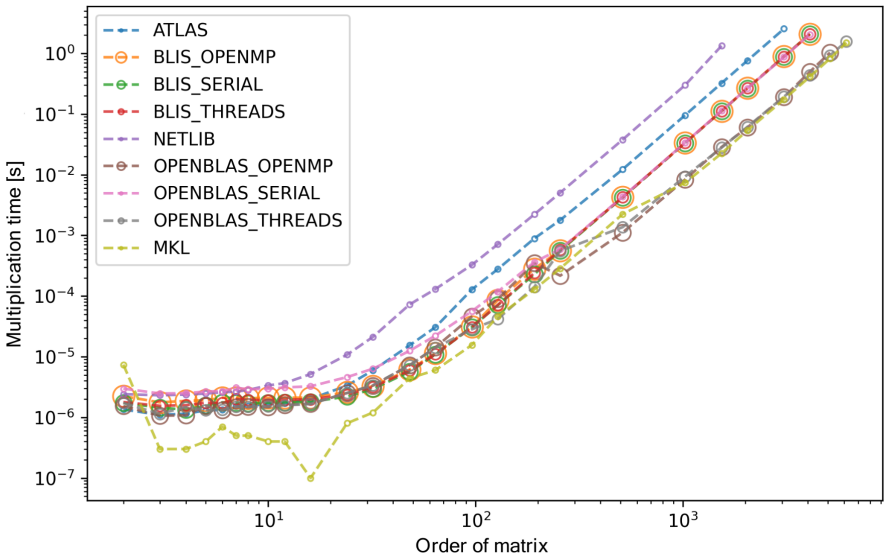

Figure 2: Influence of the size of matrices on average time of multiplication for available implementations on CPU in logarithmic scale

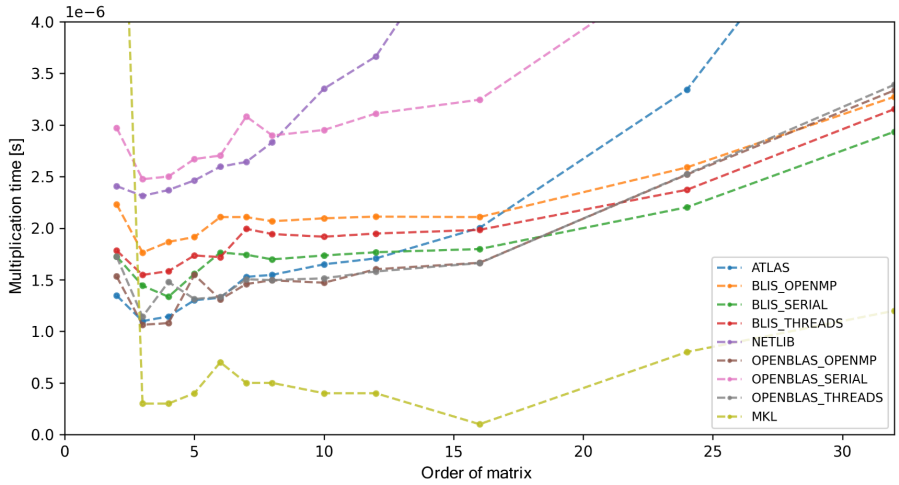

Figure 3: Relationship between matrix size and average multiplication time for available CPU implementations for small matrices in linear scale

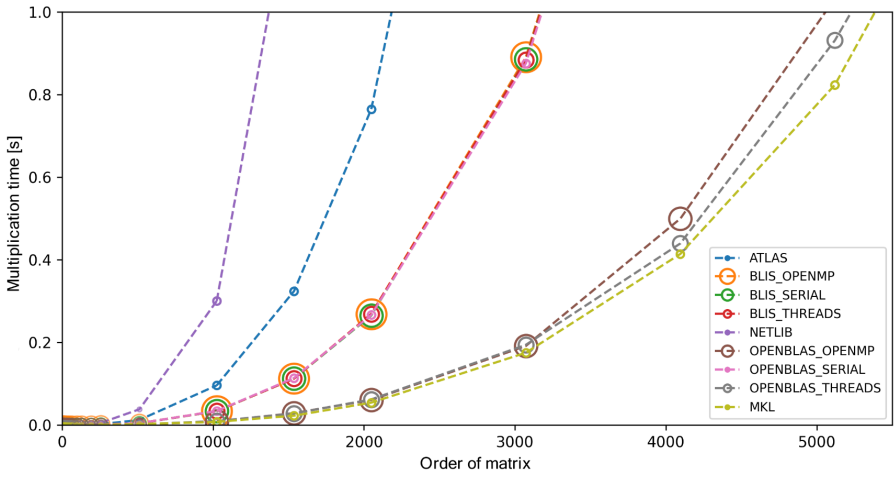

Figure 4: Relationship between matrix size and average multiplication time for available CPU implementations for medium sizes of matrices in linear scale

Figure 5 presents a performance comparison between classical implementation without optimizations and an implementation from the Intel MKL library, which proved to be the fastest among those tested on the CPU. The Intel MKL library performed better for almost all matrix sizes, with its advantage increasing as the matrix size grew. The naive algorithm was significantly better only for matrices of size 2.

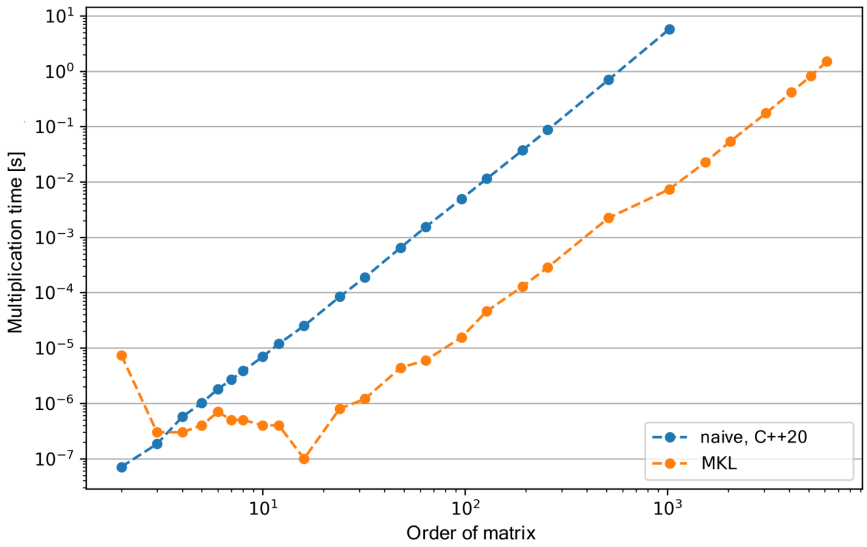

Figure 5: Relationship between matrix size and average multiplication time for naive algorithm without optimizations and Intel MKL implementation

Figure 6 presents a performance comparison of algorithms for dense and sparse matrices. Matrix multiplication on CPU used the Intel oneMKL algorithm, while on GPU the MAGMA algorithm was used. All measurements were

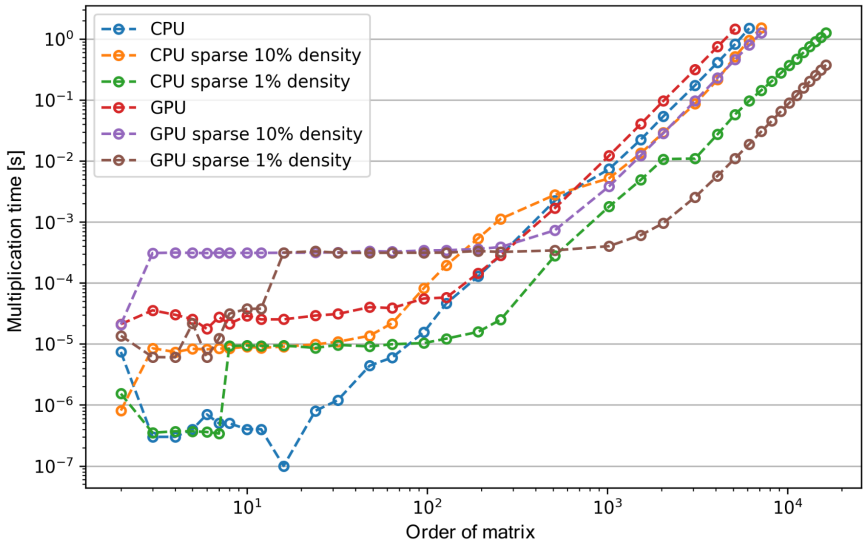

Figure 6: Relationship between matrix size and average multiplication time for sparse matrices on CPU and GPU

conducted in the MATLAB environment. For small matrices, multiplication of dense matrices was faster on both the CPU and GPU units. For larger matrix sizes, sparse matrix multiplication was more efficient—the sparser the matrix, the better the performance.

Figure 7 shows a performance comparison between the MAGMA algorithm and the JAX JIT algorithm. For small matrices, the MAGMA library proved to be more efficient, while for large matrices, the JAX JIT algorithm performed better.

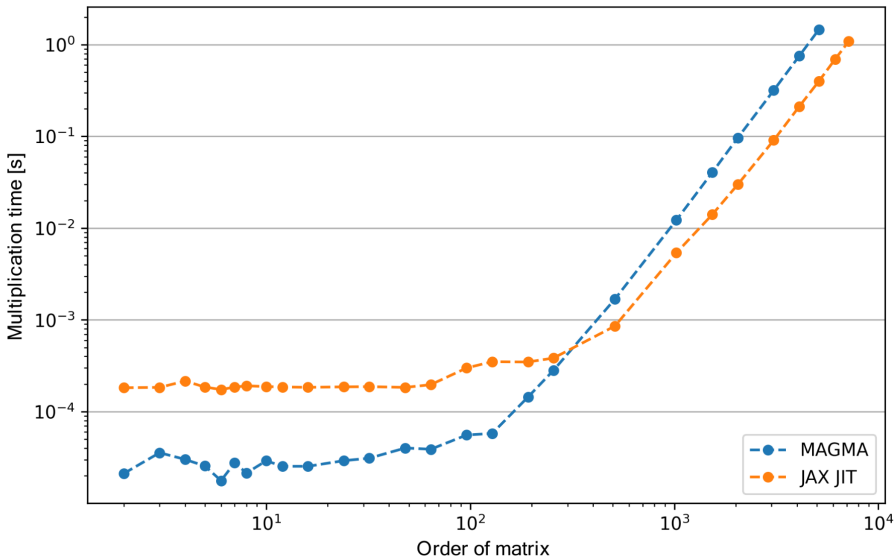

Figure 7: The relationship between matrix size and the average execution time of multiplication for MAGMA and JAX JIT

Figures 8 and 9 present a comparison between JAX JIT and the implementation of AlphaTensor algorithms from [5]. For small and medium matrix sizes, JAX JIT was the fastest, but for large sizes, the AlphaTensor algorithms were more efficient. The AlphaTensor algorithm optimized for GPU was executing faster than the one optimized for TPU.

Figure 10 shows a comparison of matrix multiplication performance on GPU under different communication scenarios with CPU. The baseline MAGMA case represents the time it takes to perform the multiplication on GPU. The "communication" value refers to the time required to transfer a single matrix from one component to the other. "MAGMA + gather" refers to the case where the matrices to be multiplied are generated directly on GPU, and the result is transferred to the CPU after multiplication. "MAGMA + gather + send" refers to the case where the matrices are generated on the CPU, transferred to the GPU, and then the result

is also transferred back to the CPU. Data transfer between the GPU and CPU has the greatest impact on small matrix sizes, and its influence decreases as the size increases.

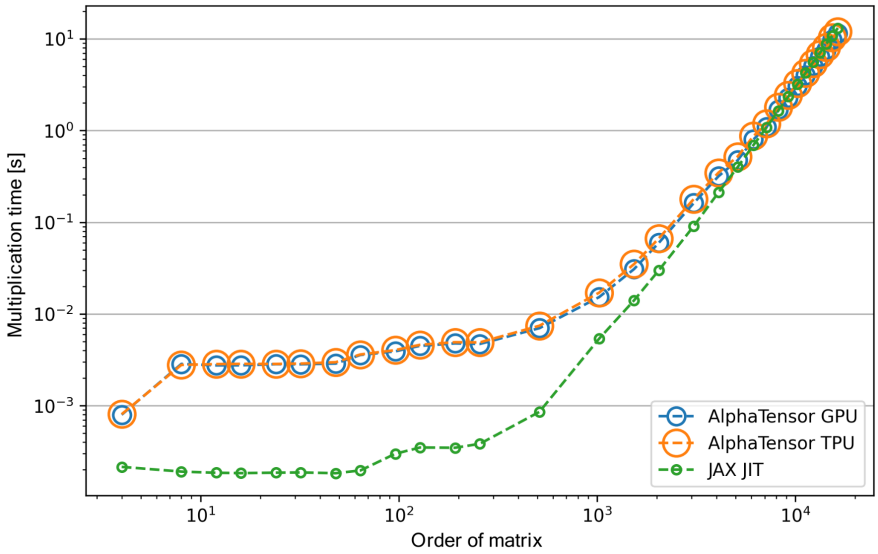

Figure 8: Relationship between matrix size and average execution time of multiplication for AlphaTensor optimized for GPU and TPU and JAX JIT in logarithmic scale

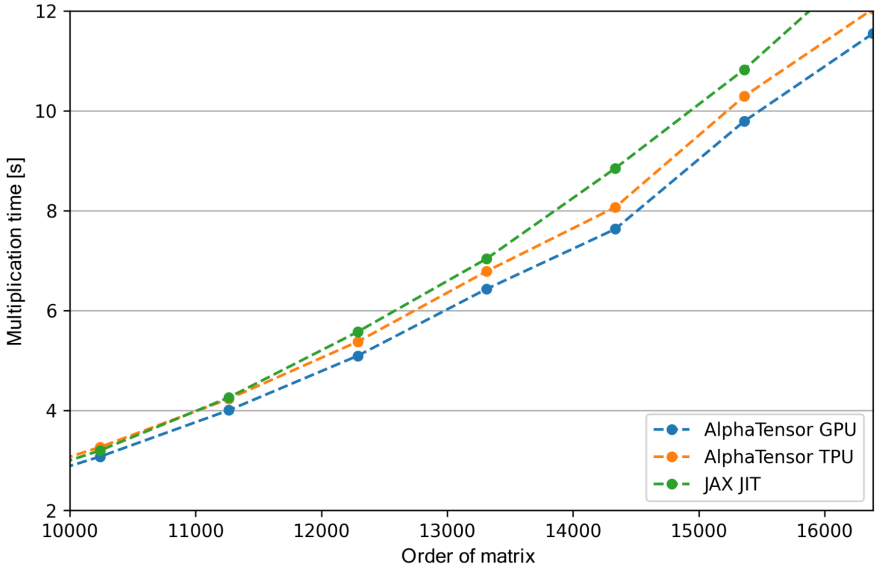

Figure 9: Relationship between matrix size and average execution time of multiplication for AlphaTensor optimized for GPU and TPU and JAX JIT for large matrices in linear scale

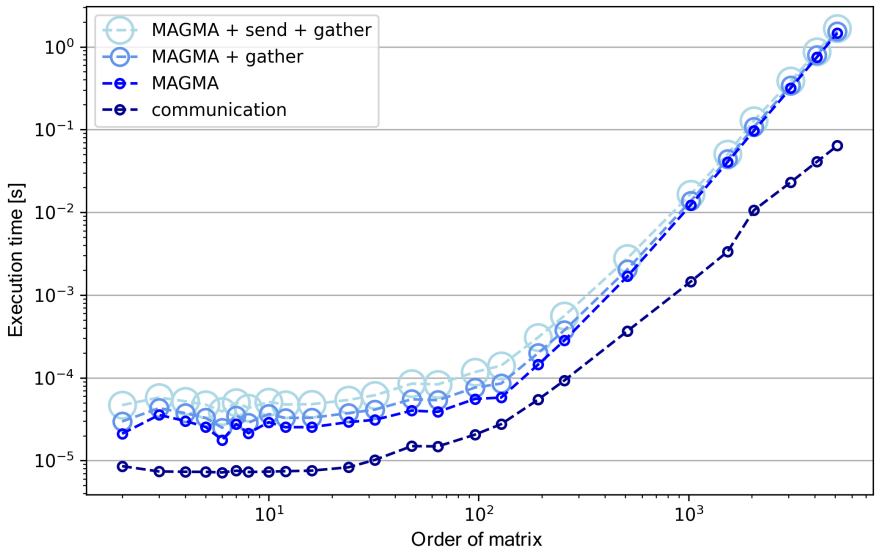

Figure 10: Relationship between matrix size and average time of execution for various participations of communication

5. Final remarks

Previous studies and tests described in this paper confirm that the efficiency of matrix multiplication algorithms depends mainly on the hardware optimizations used in the implementation. These optimizations focus on reducing the access times to memory and communication between the components that participate in the calculations, as well as on the appropriate parallelization of operations. This is in line with the current trend of computer hardware development, which provides more cores and larger sizes of fast memories. Theoretical time complexity of algorithms is important in the case of multiplying large matrices. In the results of the research presented in [5] there was a noticeable increase in efficiency relative to the classical, more communication-optimized algorithms for all matrices of order at least 8192 (smaller matrices were not considered), and in the case of the results obtained in this work – for orders equal to at least 9216. This means that some matrices of practical sizes can be multiplied using the AlphaTensor algorithm more efficiently than optimized implementations of the classical algorithm. Such large matrices are used mainly in physical simulations, machine learning and image processing algorithms. When multiplication is performed particularly often, it can be beneficial to match the algorithm used to the hardware components used and to the matrices being multiplied. This can reduce execution times by several orders of magnitude. For most matrices, the performance gain should be provided

by techniques related to auto-tuning of existing solutions and comparing them on the target hardware. When multiplying special types of matrices, it is well worth considering using implementations corresponding to them. For large matrices, it is worth using run-time compiled algorithms and the AlphaTensor algorithm. It is also possible to tune algorithms to the target hardware using artificial intelligence, as in the case of [5]. For small matrices, standard implementations of BLAS specification from currently developed libraries turned out to be more efficient. It is particularly important to avoid unnecessary data transfer between components. The ultimate performance differences between the solutions depend on the exact hardware specifications and the optimizations that the compiler can perform in the context of larger program subroutines.

References

[1] A. Abdelfattah, A. Haidar, S. Tomov and J. Dongarra: Design, and autotuning of batched GEMM for GPUs. In: J. Kunkel, P. Balaji and J. Dongarra (eds): *High Performance Computing. ISC High Performance 2016*. Lecture Notes in Computer Science, **9697**, Springer, Cham, 2016. DOI: 10.1007/978-3-319-41321-1_2

[2] J. Alman, R. Duan, V. Vassilevska Williams, Y. Xu, Z. Xu and R. Zhou: More asymmetry yields faster matrix multiplication. *arXiv*, (2024). DOI: 10.48550/arXiv.2404.16349

[3] Z.A. Alqadi, M. Aqel and I.M.M. El Emary: Performance analysis and evaluation of parallel matrix multiplication algorithms. *World Applied Science Journal*, **5**(2), (2008), 211–214.

[4] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight and O. Schwartz: Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, **23** (2014), 1–155. DOI: 10.1017/S0962492914000038

[5] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F.J.R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis and P. Kohli: Discovering faster matrix multiplication algorithms with reinforcement learning, *Nature*, **610**, (2022), 47–53. DOI: 10.1038/s41586-022-05172-4

[6] M. Kubale: Grafo-mania, czyli rzecz o grafach i algorytmach. Szybkie mnożenie macierzy. *Pismo PG*, **5** (2022), 36. (in Polish).

[7] C.L. Lawson, R.J. Hanson, F.T. Krogh and D.R. Kincaid: Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software (TOMS)*, **5**(3), 324–325. DOI: 10.1145/355841.3558

[8] D. Niemczyk: *Implementacja i testowanie starych i nowych algorytmów mnożenia macierzy*. Engineering Thesis, Gdańsk University of Technology, WETI Faculty, 2022, (in Polish).

[9] W.H. Press and S.A. Teukolsky: *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007.

[10] J.S. Respondek: *Fast Matrix Multiplication with Applications*, Springer, **166**, 2025. DOI: 10.1007/978-3-031-76930-6

[11] R. Skeel: Round-off error and the Patriot missile. *SIAM News*, **25**(4), (1992).

[12] V. Strassen: Gaussian elimination is not optimal. *Numerische Mathematik*, **13**(4), (1969), 354–356. DOI: 10.1007/BF02165411

[13] Wikipedia, Computational complexity of matrix multiplication, https://en.wikipedia.org/wiki/