DOI 10.15276/imms.v15.no3.327
UDC 519.6:004.42: 004.31

Informatics and Mathematical Methods in Simulation Vol.15 (2025), No. 3, pp.327-336

PERFORMANCE IMPROVEMENT OF PARALLEL ALGORITHMS FOR SOLVING SLAEs VIA GPU COMPUTING

O. Zhulkovskyi[1], I. Zhulkovska[2], Y. Ulianovska[2], O. Kosukhina[1], V. Riabovolenko[2]

[1]Dniprovsky State Technical University
2, Dniprobudivska str., Kamianske city, 51918, Ukraine
[2]University of Customs and Finance
2/4, Volodymyr Vernadskyi str., Dnipro, 49000, Ukraine
Email: olalzh@ukr.net

Efficient organization of computational architecture and the optimization of algorithms for solving systems of linear algebraic equations on GPUs make it possible to significantly improve program performance and achieve efficient scalability when working with large sparse matrices. This study investigates the efficiency of implementing Cholesky, QR, and LU matrix factorization algorithms on the CUDA architecture using the JCuda library for parallel computations. Special attention is given to the impact of matrix row and column reordering methods (AMD and RCM) on the performance of algorithms on both GPU and CPU. Numerical experiments were carried out on matrices of different sizes and sparsity levels, which enabled the evaluation of algorithm performance and the influence of data preprocessing. The results show that the GPU implementation of the Cholesky algorithm provides acceleration of up to 5.2x for large sparse matrices, while QR demonstrates a GPU-over-CPU advantage on most test cases, with a maximum speedup of 7.6x. The AMD reordering strategy is more effective for the Cholesky algorithm, whereas RCM is preferable for QR. The scientific novelty of the paper lies in the comprehensive comparison of factorization-based methods for solving SLAEs in the context of hybrid CPU/GPU computing, considering matrix reordering strategies. The practical significance of the obtained results is in identifying the feasibility of GPU utilization for specific classes of problems, enabling optimization of computation time and memory usage. Prospects for further research include the optimization of GPU-based LU factorization, the application of hybrid CPU/GPU strategies to improve scalability and energy efficiency, as well as the use of mixed-precision techniques to balance computational accuracy and performance.
Keywords: GPU acceleration, systems of linear algebraic equations, matrix factorizations, CUDA, parallel computing, matrix reordering (AMD, RCM).

Introduction. The rapid development of computer technology and the advancement of personal computer architecture, particularly through multicore processors, increased cache sizes, and system memory, constantly stimulate the development of software capable of efficiently exploiting these resources [1]. Modern computational modeling tasks that require processing large volumes of data often exceed the capabilities of single-core processors, which has encouraged the active adoption of parallel computing technologies [2, 3].

In recent years, graphics processing units (GPUs) have attracted considerable attention, as their computational power has been growing much faster than that of central processing units (CPUs). This enables GPUs to be employed as a cost-effective and high-performance platform for general-purpose scientific and engineering computations (General-Purpose Computing on Graphics Processing Units, GPGPU), significantly accelerating algorithms that support parallel execution. The use of GPUs in hybrid computing systems allows partially offloading computationally intensive tasks from CPUs, thereby enhancing overall program performance [4].

Modern standards and technologies for GPU-accelerated computing include [5, 6]: OpenCL, DirectCompute, C++ AMP, AMD FireStream, and CUDA (Compute Unified Device Architecture). In particular, NVIDIA’s CUDA architecture provides efficient

implementation of parallel computation methods owing to a large number of computational threads and processing pipelines, enabling simultaneous processing of vast amounts of data.

The relevance of accelerating the solution of SLAEs on GPUs [7] is driven by the need for high-performance computational methods in scientific modeling, engineering calculations, and processing of large datasets. The deployment of parallel methods on the CUDA architecture [6] allows for significant improvements in computational efficiency and opens new prospects for software–hardware optimization of SLAEs.

Related works. The solution of SLAEs occupies a central place in many problems of applied mathematics, computational physics, chemistry, mechanics, and engineering. The efficiency of SLAE algorithms largely determines the performance of numerical simulations of processes underlying modern scientific and technical research. Traditional methods for solving SLAEs, such as the Gaussian elimination method, LU factorization, or Cholesky factorization, provide reliable accuracy; however, their computational complexity becomes critical for large-scale problems. Therefore, iterative methods come to the forefront, including the Conjugate Gradient (CG) method, BiCGStab (Biconjugate Gradient Stabilized Method), GMRES (Generalized Minimal Residual Method), and others, which are more suitable for parallel implementation [1].

The advancement of hardware, particularly the emergence of GPUs with CUDA architecture, has opened new opportunities for high-performance computing [8]. CUDA provides a programming model that enables massively parallel execution of operations, which is especially effective for operations on large matrices. As noted in [9], the performance of GPU applications strongly depends on the CUDA version, since updates to libraries and drivers affect the performance of core operations, particularly matrix–vector multiplications, which constitute the core of SLAE solution methods.

Beyond performance, energy efficiency has also attracted considerable research attention. Mixed-precision computation has emerged as a promising direction, as it reduces energy consumption without significant loss of accuracy. As demonstrated in [7], applying combined high- and low-precision methods in GPU computations results in noticeable reductions in both solution time and energy consumption, which is critically important for large-scale scientific computing and machine learning applications.

The automation of parallel algorithm development represents another important research direction. As highlighted in [9], the use of the FEniCS system in conjunction with GPU architectures enables automated generation of efficient code for finite element computations. This allows applied researchers to benefit from high-performance computing without requiring deep expertise in parallel programming, thus bridging the gap between hardware complexity and user needs.

Another promising direction is the development of hybrid algorithms that combine different methods or architectures. For example, [4] presents the HyLAC (Hybrid Linear Assignment Solver in CUDA) algorithm, which demonstrates the effectiveness of combining various optimization strategies. Although the study focuses on the linear assignment problem, it confirms the potential of hybrid approaches and can be extended to linear algebra problems, including SLAE solving.

Considerable attention is also given to the scalability of parallel algorithms [2]. Key challenges include load balancing among threads, memory usage optimization, and communication efficiency for large-scale problems. Further progress in these directions is possible through tight integration of hardware and software solutions, highlighting the relevance of research aimed at adapting methods to GPU-specific architectures.

The literature also reveals a clear trend toward the development of specialized libraries and frameworks that provide broader access to parallel computing. A notable example is the CUDA Toolkit [5], which includes a set of libraries, tools, and code samples for developing high-performance applications. This environment has become a de facto standard in the

GPGPU domain, offering extensive opportunities for optimization and testing of computational kernels.

Recent research has also made significant progress in the field of direct solvers for sparse systems. For instance, the NVIDIA cuDSS library, a direct sparse solver, demonstrates that GPU acceleration allows efficient handling of symmetric, positive-definite, and general sparse matrices, including multi-user and multi-node configurations [10]. This indicates that not only dense systems but also sparse systems are gaining increasing priority.

Another active research area is the optimization of iterative solvers with an aggregation-based AMG preconditioner, which ensures good scalability on multi-GPU systems [11]. This is particularly important since many applied SLAEs exhibit properties well-suited for AMG, such as large size, irregularity, and weak connectivity.

Another example is tfQMRgpu, a solver for block-sparse systems optimized for throughput, which allows simultaneous solution of multiple systems with different right-hand sides (multiple RHS), thus increasing efficiency by enhancing arithmetic intensity (the ratio of computations to data transfers) [12].

Studies show that memory optimization, reducing the overhead of kernel launches, adopting a hybrid Host/Device/Host multithreading model, and employing effective sparse-matrix reordering strategies are crucial for improving performance [10–12].

Overall, the current state of research indicates a high level of interest in enhancing the efficiency of parallel methods for solving SLAEs [13]. However, open questions remain regarding the adaptation of algorithms to new GPU generations, balancing accuracy and performance, and ensuring user-friendly computational technologies in applied fields, primarily in computer modeling and predictive analysis of engineering systems [14].

Research Objective. The objective of this study is to enhance the efficiency of solving SLAEs by implementing and optimizing matrix factorization algorithms (Cholesky, QR, and LU) on GPUs with CUDA architecture, accompanied by a comparative evaluation of their performance relative to CPU-based computations.

Main Part. The GPU architecture features shader cores organized into Texture Processor Clusters, each comprising a texture unit and two streaming multiprocessors. Each multiprocessor includes an interface for decoding and launching instructions and a backend with eight arithmetic units and two SFUs (Special Function Units) that execute instructions in SIMT (Single Instruction, Multiple Threads) mode. To maximize hardware utilization, instructions are interleaved between arithmetic and SFU operations.

Each multiprocessor contains 16 KB of shared memory for data exchange between threads within a block, and about 8 KB of cache memory per multiprocessor for accessing constants and textures. Global memory is accessible to all multiprocessors but suffers from higher latency and lower bandwidth.

The CUDA programming model is based on lightweight extensions to C/C++ to support parallelism. A program consists of a sequential part executed on the CPU and a parallel part executed on the GPU. A kernel is a function executed simultaneously by many threads, which are organized into blocks and grids. Synchronization within a block occurs via shared memory, while communication across blocks relies on global memory. CUDA ensures that parallel code scales across varying numbers of cores and multiprocessors.

CUDA extends C/C++ with function and variable qualifiers such as:

__global__ – kernel function invoked from the CPU and executed on the GPU;

__device__ – function executed on the GPU, callable only from the GPU;

__host__ – conventional CPU function.

Variables can also be declared with the __shared__ qualifier, denoting multiprocessor shared memory. Kernel invocation requires specifying grid and block dimensions. The CUDA API provides GPU memory management functions (e.g., cudaMalloc, cudaFree, cudaMemcpy) and supports data exchange between host RAM and GPU VRAM.

JCuda is a collection of Java bindings for the CUDA platform, enabling high-performance GPU computations without direct use of C/C++. It provides Java interfaces to native CUDA libraries (e.g., cuBLAS, cuFFT, cuSPARSE, cuSOLVER), allows launching custom CUDA kernels, managing GPU memory, transferring data between host and device, and executing parallel operations on vectors and matrices. By combining the portability of Java with the performance of CUDA, JCuda serves as an effective framework for numerical modeling, linear algebra, and large-scale data processing. Development was carried out in IntelliJ IDEA by JetBrains.

Three algorithms for solving SLAEs were selected for this study: Cholesky, QR, and LU, which differ substantially in their mathematical structure, enabling a comprehensive evaluation of their efficiency in parallel data processing. Two reordering algorithms were employed for preprocessing matrices: Approximate Minimum Degree (AMD) and Reverse Cuthill–McKee (RCM).

Cholesky factorization is applied to symmetric positive-definite matrices, decomposing them into the product of a lower triangular matrix and its transpose. This ensures efficient and stable numerical solutions of SLAEs, reducing computational costs compared to direct methods for general matrices. QR factorization expresses a matrix A as the product of an orthogonal (or unitary) matrix Q and an upper triangular matrix R. It is used for stable numerical solutions of SLAEs and eigenvalue computations, providing high accuracy in cases of sparse or ill-conditioned systems. LU factorization decomposes a matrix A into the product of a lower triangular matrix L and an upper triangular matrix U. This method is widely applied for solving SLAEs, computing determinants, and inverting matrices, ensuring efficiency through straightforward sequential execution.

The AMD algorithm is designed to reorder rows and columns of sparse matrices to minimize fill-in during factorization. It heuristically selects nodes of minimal degree in the matrix dependency graph, thereby optimizing sparse matrix structure for efficient execution of factorizations such as LU or Cholesky, reducing both computational cost and memory usage.

The RCM algorithm is used to reduce the bandwidth of sparse matrices. It orders rows and columns so that nonzero elements are positioned closer to the main diagonal, reducing fill-in during factorization and improving both computational efficiency and cache utilization. The aim of the study is to compare the efficiency of different SLAE-solving algorithms on a unified dataset, considering both CPU and GPU performance. Numerical experiments were implemented in JCuda for the three matrix factorizations (Cholesky, QR, LU), along with two matrix preprocessing strategies: AMD and RCM reordering. The computational model involved the following steps: loading the matrix data; measuring execution time of Cholesky, QR, and LU algorithms on CPU; measuring execution time of Cholesky and QR algorithms on GPU; applying AMD reordering and repeating execution time measurements for all algorithms; applying RCM reordering and repeating execution time measurements for all algorithms. This approach makes it possible to obtain comparative performance metrics, evaluate the effect of column ordering on computational efficiency, and establish execution-time dependencies for different algorithms and architectures.

Results and Discussion. Computational experiments were conducted using the following test environment: CPU Intel Core i5-9300H, 2.40 GHz, 4 cores, 8 threads; GPU Nvidia GTX 1660TI with 1536 CUDA cores and 6 GB of video memory.

During data processing, the analysis focused on the impact of matrix size, preprocessing, and the number of nonzero elements on algorithm execution time, as well as a comparison of CPU vs GPU performance (Table 2).

The experiments were performed on input data described in Table 1.

Table 1.

Characteristics of input matrices for algorithm testing

Experiment No.	Matrix size (rows × columns)	Number of non-zero elements
1	1000 × 1000	3750
2	1074 × 1074	12918
3	1083 × 1083	18437
4	1050 × 1050	19918
5	1282 × 1282	22878
6	10000 × 10000	49600
7	8000 × 8000	53600

Table 2.

Performance of factorization algorithms

Additional preprocessing	Non-zero elements	Algorithm	CPU, ms	GPU, ms	Additional preprocessing	Non-zero elements	Algorithm	CPU, ms	GPU, ms
-	3750	Cholesky	4.816	9.266	–	12918	Cholesky	55.267	23.164
		QR	7.877	16.751			QR	247.893	70.145
		LU	2.897				LU	166.515	n/a
AMD sorting		Cholesky	1.08	6.418	AMD sorting		Cholesky	4.207	7.175
		QR	18.45	16.604			QR	139.571	60.511
		LU	1.143				LU	25.692	n/a
RCM sorting		Cholesky	1.976	7.397	RCM sorting		Cholesky	60.404	22.801
		QR	6.597	14.585			QR	266.238	61.674
		LU	2.228				LU	134.589	n/a
-	18437	Cholesky	6.671	12.934	–	19918	Cholesky	3.297	6.487
		QR	18.77	26.933			QR	218.281	49.314
		LU	6.678	n/a			LU	2.49	n/a
AMD sorting		Cholesky	7.721	9.378	AMD sorting		Cholesky	2.408	6.024
		QR	106.596	51.191			QR	40.968	41.301
		LU	8.503	n/a			LU	4.028	n/a
RCM sorting		Cholesky	8.136	13.661	RCM sorting		Cholesky	34.585	16.929
		QR	27.38	29.332			QR	116.23	36.035
		LU	9.908	n/a			LU	55.479	n/a
-	22878	Cholesky	3.472	9.048	–	49600	Cholesky	124.957	127.781
		QR	9.79	17.079			QR	442.195	267.734
		LU	4.736	n/a			LU	228.887	n/a
AMD sorting		Cholesky	4.605	6.793	AMD sorting		Cholesky	27.853	22.1
		QR	51.666	31.438			QR	2406.685	750.128
		LU	7.083	n/a			LU	25.287	n/a
RCM sorting		Cholesky	3.166	9.673	RCM sorting		Cholesky	67.622	79.642
		QR	9.944	18.501			QR	241.055	205.664
		LU	5.006	n/a			LU	105.692	n/a
-	53600	Cholesky	1278.723	245.499					
		QR	4674.122	614.58					
		LU	2330.692	n/a					
AMD sorting		Cholesky	370.923	83.306					
		QR	14756.99	2319.035					
		LU	559.962	n/a					
RCM sorting		Cholesky	506.049	157.155					
		QR	1802.826	359.787					
		LU	882.141	n/a					

When comparing CPU and GPU implementations without preprocessing, the Cholesky factorization on GPU outperformed the CPU only for matrices with 12918 (2.4x speedup) and

53600 (5.2x speedup) nonzero elements. For all other datasets, GPU execution did not provide a performance advantage. In contrast, the QR factorization showed GPU superiority in 4 out of 7 benchmark cases, with the maximum observed speedup of 7.6x for the matrix containing 53600 nonzero elements. The LU factorization was not executed on GPU due to library limitations.

The application of AMD preprocessing for Cholesky (Fig. 1) showed that GPU execution is advantageous only for datasets with 49600 (1.3x speedup) and 53600 (4.5x speedup) nonzero elements.

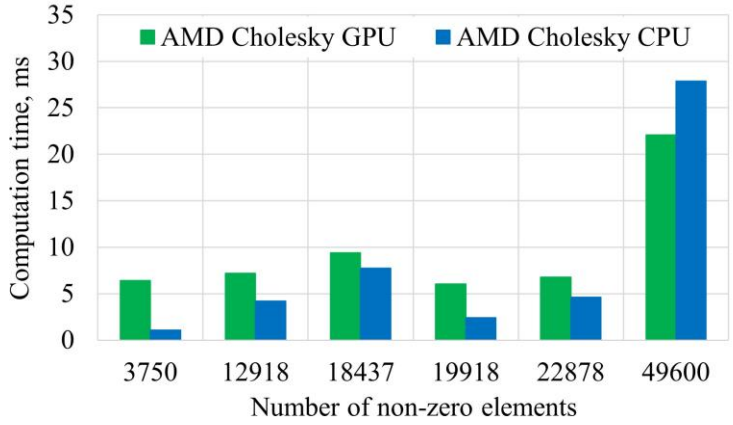

Fig. 1. Effect of AMD sorting on GPU performance in Cholesky factorization

The application of AMD preprocessing for QR (Fig. 2) demonstrated GPU superiority over CPU on 4 out of 7 benchmark cases, as in the scenario without preprocessing. Significant GPU speedups were achieved: 2.3x for the 12918-element matrix and 6.4x for the 53600-element matrix.

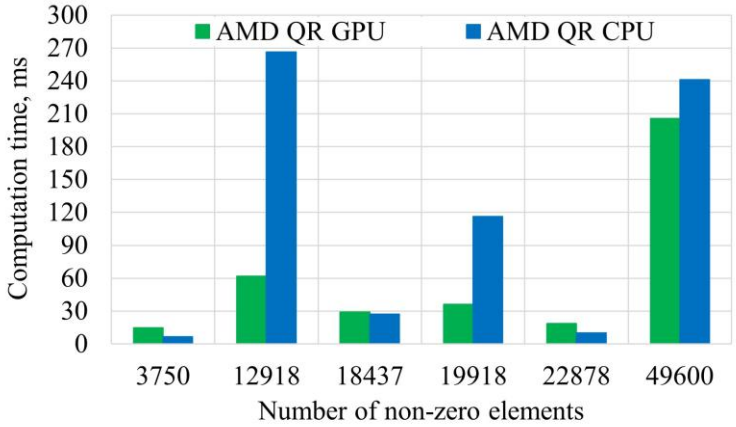

Fig. 2. Effect of AMD sorting on GPU acceleration for QR

The use of RCM preprocessing for Cholesky (Fig. 3) demonstrated GPU superiority over CPU only for datasets with 12918 (2.6x speedup), 19918 (2x speedup), and 53600 (3.2x speedup) nonzero elements.

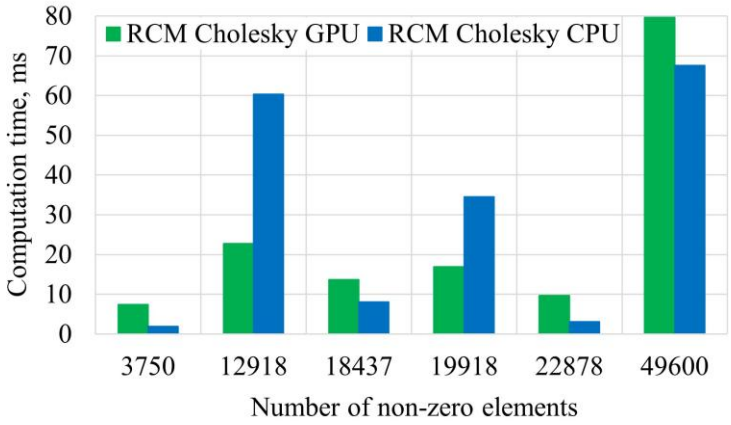

Fig. 3. Effect of RCM sorting on GPU acceleration for Cholesky

The use of RCM preprocessing for QR (Fig. 4) demonstrated GPU superiority over CPU on 4 out of 7 benchmark cases, similar to AMD preprocessing. Notable GPU speedups were 4.3x for the 12918-element matrix, 3.2x for the 19918-element matrix, and 5x for the 53600-element matrix.

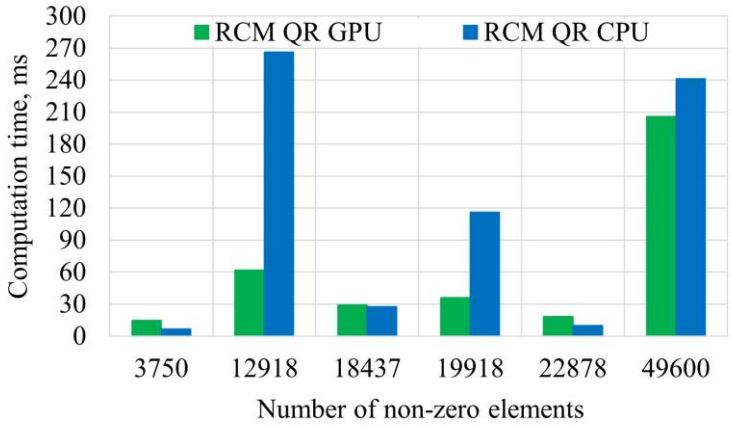

Fig. 4. Effect of RCM sorting on GPU acceleration for QR

As the results show, AMD preprocessing has a stronger impact on GPU performance in the Cholesky implementation (Fig. 4), while RCM preprocessing is more effective in the QR implementation (Fig. 5).

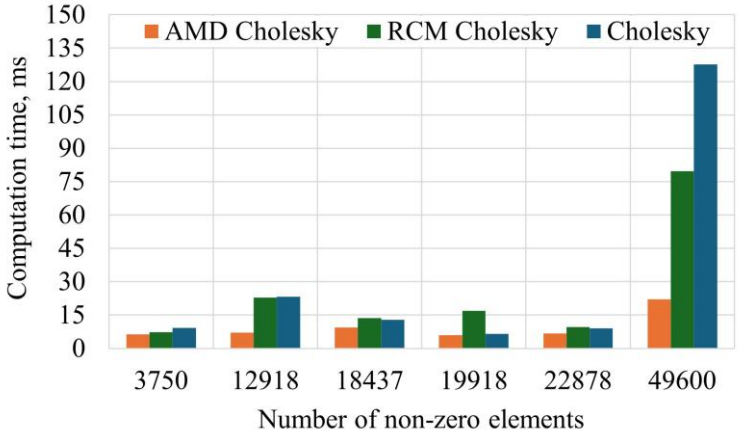

Fig. 5. Comparison of Cholesky performance

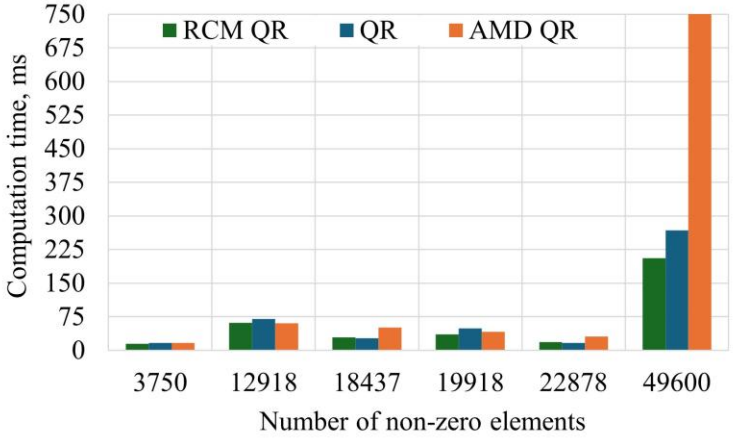

Fig. 6. Comparison of QR performance

Conclusions. The study investigated the efficiency of implementing matrix factorization algorithms (Cholesky, QR, and LU) for solving SLAEs on GPU architecture using CUDA and JCuda libraries. A comparative analysis of CPU and GPU performance was carried out on multiple datasets, considering preprocessing with AMD and RCM methods.

The GPU-based Cholesky implementation demonstrated speedups for large sparse matrices (up to 5.2x), while for smaller datasets GPU execution did not provide a performance advantage. The QR algorithm proved more suitable for parallel execution, providing GPU-over-CPU advantages on most benchmark cases (with a maximum speedup of 7.6x).

The use of AMD and RCM reorderings affected performance differently—AMD was more effective for Cholesky, whereas RCM proved more beneficial for QR.

The scientific novelty of the work lies in a comparative analysis of factorization methods for sparse SLAEs in the context of hybrid CPU/GPU computing, taking into account different matrix reordering strategies.

The practical significance of the results is in identifying the feasibility of GPU usage for specific classes of problems: for large, highly sparse matrices, and when applying suitable reorderings, GPU implementations demonstrate substantial advantages over CPU.

Further research should focus on optimizing GPU implementations of LU factorization using modern libraries (cuDSS, cuSPARSE), developing hybrid CPU/GPU strategies to improve scalability, analyzing computational energy efficiency, and applying mixed-precision methods to balance accuracy and performance.

Thus, GPU acceleration is a promising tool for solving large-scale SLAEs, opening new opportunities in high-performance computing [1, 2].

References

1. Masciari E., Napolitano E.V. Sustainability and High Performance Computing. *Lecture Notes in Computer Science*. 2024. P. 237–242. URL: https://doi.org/10.1007/978-3-031-78093-6_21
2. Bouras M., Idrissi A. A Survey of Parallel Computing: Challenges, Methods and Directions. *Modern Artificial Intelligence and Data Science. Studies in Computational Intelligence*. 2023. Vol. 1102. P. 67–81. URL: https://doi.org/10.1007/978-3-031-33309-5_6
3. Zhulkovskyi O., Zhulkovska I., Kurliak P., Sadovoi O., Ulianovska Y., Vokhmianin H. Using Asynchronous Programming to Improve Computer Simulation Performance in Energy Systems. *Energetika.* 2025. Vol. 71, №1. P. 23–33. URL: https://doi.org/10.6001/energetika.2025.71.1.2
4. Kawtikwar A., Nagi R. HyLAC: Hybrid Linear Assignment Solver in CUDA. *Journal of Parallel and Distributed Computing*. 2024. Vol. 187. 104838. URL: https://doi.org/10.1016/j.jpdc.2024.104838

5. CUDA Toolkit Archive. URL: https://developer.nvidia.com/cuda-toolkit-archive
6. Yoshida K., Miwa S., Yamaki H., Honda H. Analyzing the Impact of CUDA Versions on GPU Applications. *Parallel Computing*. 2024. Vol. 120. 103081. URL: https://doi.org/10.1016/j.parco.2024.103081
7. Mukunoki D., Ogita T. Performance and Energy Consumption of Accurate and Mixed-Precision Linear Algebra Kernels on GPUs. *Journal of Computational and Applied Mathematics*. 2020. Vol. 372. 112701. https://doi.org/10.1016/j.cam.2019.112701
8. Mohamed K. S. Parallel Computing: OpenMP, MPI, and CUDA. In: *Neuromorphic Computing and Beyond*. Cham, 2020. P. 63–93. URL: https://doi.org/10.1007/978-3-030-37224-8_3
9. Trotter J.D., Langguth J., Cai X. Targeting Performance and User-Friendliness: GPU-Accelerated Finite Element Computation With Automated Code Generation in FEniCS. *Parallel Computing*. 2023. Vol. 118. 103051. URL: https://doi.org/10.1016/j.parco.2023.103051
10. NVIDIA. NVIDIA cuDSS (Preview): A High-performance CUDA Library for Direct Sparse Solvers. Documentation. URL: https://docs.nvidia.com/cuda/archive/12.8.1/cudss/index.html
11. Bernaschi M., Celestini A., Vella F., D'Ambra P. A Multi-GPU Aggregation-Based AMG Preconditioner for Iterative Linear Solvers. *IEEE Transactions on Parallel and Distributed Systems.* 2023. Vol. 34, No. 8. P. 2365–2376. URL: https://doi.org/10.1109/TPDS.2023.3287238
12. Baumeister P.F., Nassyr S. tfQMRgpu: A GPU-Accelerated Linear Solver With Block-Sparse Complex Result Matrix. *Journal of Supercomputing*. 2025. Vol. 81. 663. URL: https://doi.org/10.1007/s11227-025-07145-6
13. Alsuwaiyel, M.H.: Parallel Algorithms. *Lecture Notes Series on Computing*. Vol. 16. New Jersey: World Scientific, 2022. URL: https://doi.org/10.1142/12744
14. Zhulkovskii O., Panteikov S., Zhulkovskaya I. Information-Modeling Forecasting System for Thermal Mode of Top Converter Lance. *Steel in Translation*. 2022. Vol. 52, №5. P. 495–502. URL: https://doi.org/10.3103/s0967091222050138
15. Zhulkovskyi O., Bilio V., Zhulkovska I. Improve Efficiency of Computer Simulation Using GPU Programming. *Праці VII Всеукраїнської науково-практичної конференції молодих науковців «Інформаційні технології»*. 2020. URL: https://informationtechn2020.blogspot.com/2020/05/zhulkovskyi-o-bilio-v-zhulkovska-i.html

O. Zhulkovskyi, I. Zhulkovska, Y. Ulianovska, O. Kosukhina, V. Riabovolenko

ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ РОЗВ'ЯЗАННЯ СЛАР ЗА РАХУНОК GPU-ОБЧИСЛЕНЬ

О.О. Жульковський[1], І.І. Жульковська[2], Ю.В. Ульяновська[2],
О.С. Косухіна[1], В.А. Рябоволенко[2]

Дніпровський державний технічний університет
2, Дніпробудівська вул., Кам'янське, 51918, Україна
Університет митної справи та фінансів
2/4, Володимира Вернадського вул., Дніпро, 49000, Україна
Email: olalzh@ukr.net

Раціональна організація обчислювальної архітектури та оптимізація алгоритмів розв'язання систем лінійних алгебраїчних рівнянь на GPU дозволяє суттєво підвищити продуктивність програм і забезпечити ефективну масштабованість при роботі з великими розрідженими матрицями. У роботі досліджується ефективність реалізації алгоритмів матричних розкладів Холецького, QR та LU на архітектурі CUDA із застосуванням бібліотеки JCuda для паралельних обчислень. Особлива увага приділяється впливу методів перестановки рядків і стовпців матриць (AMD та RCM) на швидкодію алгоритмів на GPU та CPU. Чисельні експерименти виконувалися на матрицях різного розміру та розрідженості, що дозволило оцінити продуктивність алгоритмів та вплив попередньої підготовки даних. Результати показали, що реалізація Холецького на GPU забезпечує прискорення до 5.2x для великих розріджених матриць, тоді як QR демонструє перевагу GPU над CPU на більшості тестових вибірок із максимальним прискоренням 7.6x. Застосування AMD-перестановки ефективніше для алгоритму Холецького, а RCM – для QR. Наукова новизна роботи полягає у комплексному порівнянні продуктивності факторизаційних методів для систем лінійних алгебраїчних рівнянь у контексті гібридних обчислень CPU/GPU із врахуванням стратегій перестановки матриць. Практичне значення отриманих результатів полягає у визначенні доцільності застосування GPU для конкретних класів задач, що дозволяє оптимізувати час обчислень та використання ресурсів пам'яті. Перспективи подальших досліджень передбачають оптимізацію GPU-реалізацій LU-розкладу, використання гібридних стратегій CPU/GPU для підвищення масштабованості, енергоефективності обчислень та застосування методів змішаної точності для забезпечення балансу між точністю результатів і продуктивністю обчислень.
Ключові слова: GPU-прискорення, системи лінійних алгебраїчних рівнянь, матричні розклади, CUDA, паралельні обчислення, перестановка матриць (AMD, RCM).