DOI 10.15276/imms.v16.no1.43
UDC 004.4`4: 004.23

Informatics and Mathematical Methods in Simulation
Vol.16 (2026), No. 1, pp. 43-51

PROBLEMS OF AUTOMATIC CODE OPTIMIZATION BY THE COMPILER

I. Zhulkovska[1], O. Zhulkovskyi[1],
T. Rudianova[2], O. Lebid[2], M. Mormul[2]

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

Rational use of modern compiler capabilities, in particular automatic SIMD vectorization, enables significant improvements in computational performance for tasks involving data-array processing and computer modeling of complex processes and systems. The growing demand for software performance in scientific computing, big-data analysis, artificial intelligence, and machine learning emphasizes the importance of exploiting hardware-level data parallelism. This study investigates the efficiency of automatic SIMD vectorization provided by the Microsoft Visual C++ compiler in comparison with manual optimization implemented through AVX2 instructions. To evaluate performance, three implementations were developed: a scalar baseline version, a compiler-optimized automatic SIMD code, and a manually vectorized SIMD version using intrinsic functions. Computational experiments were conducted using the SAXPY operation for arrays sized from 10^5 to 10^9. The results demonstrated that automatic SIMD vectorization provides up to a 7.5x speedup with an efficiency of 0.94 for small- and medium-scale problems, effectively utilizing processor resources through aggressive optimizations such as loop unrolling and efficient use of FMA pipelines Manual SIMD optimization showed stable acceleration of up to 3. for large arrays but with lower efficiency (0.28–0.49 due to memory-bandwidth limitations and less aggressive compiler-level transformations. The comparison revealed that automatic methods are more convenient for developers, significantly reducing the effort required for writing SIMD code, while manual optimizations remain relevant when scaling to large data volumes. The findings indicate that the optimal strategy is a combined use of automatic and manual SIMD transformations, allowing a balance between performance, accuracy, and development effort, thus ensuring both efficiency and scalability of software solutions in high-performance computing and computer modeling. Future research will focus on expanding the experimental base across various processor architectures, analyzing the interaction of SIMD vectorization with other compiler transformations, and applying ML-based methods for adaptive optimization-strategy selection.

Keywords: automatic SIMD vectorization, manual SIMD optimization, AVX2, MSVC compiler, high-performance computing.

Introduction. The rapid development of computing technology [1] and software has significantly increased the requirements for software performance, particularly in scientific computing, computer modeling, big-data analysis, and tasks related to artificial intelligence (AI) and machine learning (ML) [2, 3]. Execution efficiency is critical for high-performance computing (HPC), as well as for engineering and industrial systems, where computation speed and result accuracy directly affect the quality of forecasts and decision-making processes [4].

For a long time, performance growth was achieved by reducing transistor sizes and increasing processor clock frequencies. However, physical limitations rendered this approach ineffective, leading to alternative solutions [1], such as multi-core architectures, parallel computing, and the use of SIMD (Single Instruction, Multiple Data) hardware capabilities.

One of the key directions in improving performance is automatic compiler-based code optimization, which improves execution performance without modifying the source code. Modern compilers implement a wide range of optimizations, including vectorization, loop

unrolling and loop fusion, dead-code elimination, function inlining, constant propagation, and others [5].

Special attention is given to SIMD vectorization, which enables efficient exploitation of data-level parallelism (DLP) [6]. This approach allows a single instruction to be executed simultaneously across an entire vector of elements, significantly accelerating array-based data processing. Such methods are particularly relevant for modeling complex technological processes and systems [4], as well as for numerical algorithms used in large-scale data analysis and AI models.

Until recently, most SIMD optimizations were performed manually by programmers, requiring deep knowledge of hardware architecture and being a labor-intensive process. However, the advancement of modern compilers has gradually enabled automation of this process, making it possible to compare the efficiency of manual and automatic SIMD vectorization.

In this context, the present study focuses on analyzing modern approaches to automatic SIMD vectorization by compilers and comparing them with the outcomes of manual optimization in order to assess the advantages and limitations of both approaches. This allows identification of development trends in code-optimization technologies and directions for their further improvement.

Related works. Modern compilers perform multistage program transformations at different representation levels (source – intermediate representation (IR) – machine code), applying a set of optimization passes. The objective of these passes is to improve execution performance and hardware-resource utilization without altering the program's semantics. In recent years, compiler developers have emphasized combining program data-flow analysis with processor architectural features to automatically select and tune optimizations [7].

A particularly important direction involves optimizations aimed at exploiting DLP parallelism [8, 9]. These include [5]: vectorization, SLP transformations (Superword-Level Parallelism), loop unrolling/fusion, and related transformations that reduce the number of instructions and improve the utilization of vector-register resources in modern CPUs (SSE, AVX/AVX2/AVX-512 for Intel, NEON for ARM architectures, etc.). The practical effectiveness of such transformations depends on the accuracy of dependence analysis within loops, the availability of memory-aliasing information, and support for a specific Instruction Set Architecture (ISA) [10].

The assessment of compilers' automatic vectorization capabilities is an active area of research. In [5], a systematic methodology for evaluating auto-vectorizers was proposed, demonstrating that the presence or absence of useful information in the code strongly affects the results of auto-vectorization. Moreover, synthetic benchmarks (e.g., the Test Suite for Vectorizing Compilers, TSVC) do not always capture the practical constraints of real-world applications. This underlines the necessity of thorough testing and specialized approaches for measuring compiler capabilities.

An important direction of development involves combining SIMD vectorization with other compiler transformations, such as loop tiling (which improves cache locality and the utilization efficiency of the memory hierarchy), software pipelining (which overlaps data dependencies and balances instruction pipeline utilization), and memory-access optimizations aimed at reducing latency and avoiding memory-bank conflicts. As shown in [11], the integrated application of these approaches enables performance levels approaching those of manual optimization, confirming the potential of multilevel strategies for program-code optimization.

Recent studies increasingly focus on the application of ML methods for selecting optimization passes. In [12], an ML model was proposed that predicts the suitability of vectorization and other optimizations based on code characteristics, allowing compilers to dynamically adapt their strategies. This opens new opportunities for creating «intelligent

compilers», capable of learning from examples and accounting for both code properties and hardware features.

Although modern compilers implement multistage optimization passes and achieve significant acceleration in many cases, the literature review identifies a number of systemic limitations that substantially affect the effectiveness of automatic transformations.

For instance, automatic vectorization is effective for regular access patterns, such as linear matrix indices. However, complex address expressions, indirect indexing through arrays, or branches within loops limit the compiler's ability to generate efficient vector code [13]. Preliminary data transformations (e.g., tiling, data-layout adjustments) are often required, complicating automation.

The generation of efficient code also depends on the specific hardware architecture, including vector width, instruction set, and support for predication. Optimizations designed for the AVX2 architecture may not deliver performance gains on platforms with ARM SVE, and vice versa. This complicates the creation of universal auto-vectorizers, since each architecture has unique characteristics that influence vectorization efficiency [14].

Finding the optimal combination of passes and compiler parameter settings considerably increases compilation time. In ML-based approaches, additional offline training of models is required to predict the usefulness of optimization passes. This increases build time and necessitates a trade-off between code quality and compilation cost, especially in industrial workflows [7].

While ML-based methods for compiler optimization-pass selection show promising results, the main challenge lies in the need for large training datasets, which may be difficult to obtain in specific domains or for new architectures. Furthermore, ML models may have limited generalization ability to unseen programs, reducing their effectiveness in real-world scenarios. The lack of transparency in decision-making (explainability) within complex ML models further complicates their integration into industrial compilers, as predicting and controlling their behavior across different scenarios becomes difficult. These factors increase the risk of overfitting, where a model performs well on training data but fails to efficiently handle new or unexpected inputs. Thus, although ML approaches promise improved optimization efficiency, their application requires careful dataset collection, model tuning, and ensuring transparency of decisions [15].

All optimization transformations must preserve program semantics. Aggressive transformations, such as memory-access reordering or speculative vectorization, may require additional control mechanisms – including memory fences or runtime assertions – to prevent correctness violations. However, introducing such safeguards can reduce runtime performance and partially offset the benefits of automatic optimizations [16].

Research Objective. Computational efficiency in computer-modeling tasks largely depends on the use of SIMD instructions, which enable data-level parallelism in array processing. The traditional approach of manual vectorization (explicit SIMD) ensures a high degree of control but requires considerable time investment and architectural expertise. In contrast, modern compilers – particularly Microsoft Visual Studio C++ (MSVC) – implement automatic vectorization (implicit SIMD), which can significantly reduce development effort.

The purpose of this study is to investigate the efficiency of automatic SIMD vectorization in modern compilers using MSVC as a case study and to compare it with manual code optimization. The work aims to identify the advantages and limitations of automatic and manual optimization strategies, as well as to develop practical recommendations for improving program performance in high-performance computing, particularly in the computer modeling of complex processes and systems.

Main Part. Vectorization is the process of transforming sequential scalar instructions into vector instructions, which – unlike multithreading models – implements data-level parallelism (DLP) within a single processor core. This allows a single instruction to be executed over multiple elements simultaneously, using SIMD instructions as the hardware foundation.

Modern processors support SIMD through the following extensions [17]: SSE (Streaming SIMD Extensions) – 128 bit, AVX/AVX2 (Advanced Vector Extensions) – 256 bit, AVX-512 – 512 bit, ARM NEON, and SVE (Scalable Vector Extension).

Theoretical acceleration (S_{max}) depends on the ratio of the vector-register width to the size of the data element (W_{reg} / W_{elem}). For example, for AVX2 with 32-bit elements, up to eight operations can be performed per clock cycle.

Two approaches to SIMD are distinguished: explicit vectorization (explicit SIMD) – using intrinsic functions and inline assembly – and automatic vectorization (implicit SIMD, auto-vectorization), in which the compiler automatically analyzes and transforms the appropriate code into vectorized instructions without changes to the program source text. Explicit vectorization ensures full control but requires deep knowledge of the Instruction Set Architecture (ISA) and reduces code portability. The advantage of automatic vectorization is reduced development effort; however, its efficiency depends on dependence analysis algorithms and support for the specific hardware architecture.

The conditions for effective vectorization include regular memory access, the absence of loop-carried dependencies, proper data alignment, and correct pointer handling (alias analysis).

The primary performance metrics are execution time (τ), speedup (S), and acceleration efficiency (E, $0 \le E \le 1$):

$$S = \tau_{scal} / \tau_{vec},$$

$$E = S / S_{max},$$

where τ_{scal}, τ_{vec} – are the execution times of the scalar (non-vectorized) and vectorized versions, respectively.

Most modern compilers implement multi-level optimization – from high-level transformation (source – IR) to machine-code generation. Vectorization is a component of loop optimizations and machine-dependent optimizations.

GCC supports auto-vectorization at optimization level -O3 and with options such as -ftree-vectorize and -funroll-loops. It also has good integration with OpenMP SIMD.

Clang/LLVM provides a Loop Vectorizer and an SLP Vectorizer and is oriented toward flexible tuning. It is widely used in scientific computing projects and AI frameworks.

Intel ICC/ICX is regarded as a reference standard for high-performance computing, offering advanced heuristics for dependence analysis and optimized generation of AVX/AVX-512 instructions.

MSVC supports auto-vectorization for loops when using the /O2 or /Ox. optimization flags. Built-in intrinsic functions in <immintrin.h> allow for explicit SIMD implementation.

Starting with C++17, MSVC also integrates parallel STL algorithms (Parallel Patterns Library), which combine multithreading with vectorization. This represents a higher level of automation, relying not only on the compiler's internal optimization mechanisms but also on library-level abstractions.

In MSVC, vectorization is implemented based on loop analysis in an SSA-style intermediate representation (IR). The algorithm checks for loop-iteration independence, the feasibility of applying predication to conditional branches, the regularity of array indexing, and proper memory-access alignment. If alignment cannot be guaranteed, the compiler inserts so-called «safe loads» to ensure correctness.

MSVC applies a combination of strip-mining and vectorization – splitting the loop into a «vector» part and a «remainder» (tail) executed in scalar mode. This ensures correctness even for arrays whose length is not a multiple of the vector width. A distinctive feature of MSVC is its integration of the auto-vectorizer with the Profile-Guided Optimization (PGO) system: during preliminary program runs, execution statistics are collected, allowing the compiler to more accurately select loops for vectorization.

To evaluate the efficiency of compiler auto-vectorization and manual SIMD optimization, three groups of tests were implemented:

1) a scalar baseline implementation of array-processing algorithms;

2) an automatically optimized implementation compiled with the /O2 and /Ox, optimization flags that activate MSVC's auto-vectorizer;

3) a manual SIMD implementation, where the same algorithms were vectorized using AVX2 instructions from the <immintrin.h> library.

As an example, the SAXPY operation (Single-Precision A×X Plus Y) was computed:

$$y_i = \alpha x_i + y_i, i = 1,..., N,$$

where N is the array size, α – a constant, x, y – are vectors.

In the scalar baseline version (C++):

```
for (int i = 0; i < N; ++i) y[i] = a * x[i] + y[i];
```

The loop contains a simple linear indexing pattern, no conditional branches, and regular memory access, making it suitable for automatic SIMD vectorization.

When applying automatic optimization, the Microsoft Visual C++ compiler transforms the scalar loop into a vectorized loop using AVX2 SIMD instructions. The basic approach involves loading eight float elements into a 256-bit YMM register and performing multiplication and addition in vectorized form.

However, in practical implementations within the MSVC IDE, the compiler applies a more aggressive strategy – loop unrolling and scheduling multiple vector blocks per iteration. This means that, instead of processing only eight elements at a time, MSVC simultaneously processes several groups of eight elements, distributing them across different YMM registers (e.g., ymm1, ymm2, ymm3, etc.). Such an approach reduces loop-control overhead, maximally loads processor pipelines capable of executing several SIMD operations in parallel, and more effectively exploits FMA (fused multiply-add) instructions, which combine multiplication and addition in a single cycle.

Thus, automatically generated MSVC code not only vectorizes computations but also applies advanced optimizations at both the memory and instruction-flow levels. This approach can be characterized as an aggressive SIMD-vectorization strategy.

SIMD version (AVX2):

```
1   __m256 avx_a = _mm256_set1_ps(a);
2   int i;
3   for (i = 0; i + 7 < N; i += 8)  // vector loop
4   {
5       __m256 avx_x = _mm256_loadu_ps(&x[i]);
6       __m256 avx_y = _mm256_loadu_ps(&y[i]);
7       avx_y = _mm256_fmadd_ps(avx_a, avx_x, avx_y);
8       _mm256_storeu_ps(&y[i], avx_y);
9   }
10  for (; i < N; ++i) y[i] = a * x[i] + y[i]; // tail loop
```

In this implementation, vectorization of computations is carried out using AVX2 SIMD intrinsics from Intel. A 256-bit vector type __m256 s employed, corresponding to the hardware YMM registers of the Intel architecture [18, 19]. Each YMM register can hold eight single-precision floating-point numbers (float), enabling the simultaneous processing of eight elements of a float array within a single instruction.

At the beginning, a broadcast operation is performed using _mm256_set1_ps(a) (line 1), which loads the scalar coefficient a into all eight positions of the vector register avx_a. Next, the vectorized loop (lines 3–9) is executed: _mm256_loadu_ps(&x[i]) and _mm256_loadu_ps(&y[i]) (lines 5 and 6, respectively) load contiguous subarrays of x and y into YMM registers. The instruction _mm256_fmadd_ps(avx_a, avx_x, avx_y) implements the fused multiply-add (FMA) operation (line 7), which is executed in hardware without intermediate storage of the product, thereby reducing execution overhead and improving computational accuracy. The result is stored back into memory using _mm256_storeu_ps (line 8).

Since the array length N may not be a multiple of the vector register size (eight elements), a tail loop is used to process the remainder with scalar instructions. This guarantees correctness for any input size. The same principle is applied in automatic compiler optimization.

The presented code illustrates a typical SIMD-programming structure, where vector processing of the main data portion using YMM registers is combined with a tail section for residual elements. Such an approach efficiently exploits hardware-level data-parallelism (DLP) while maintaining universal applicability.

Thus, manual SIMD code is optimal in terms of the «purity» and simplicity of SIMD, but it does not exploit the potential of loop unrolling. By contrast, MSVC auto-vectorization generates more complex yet more aggressive code, capable of delivering higher performance on modern CPUs thanks to loop unrolling and maximized utilization of FMA pipelines.

Results and Discussion. The experiments were conducted on a laptop equipped with a 12th Gen Intel Core i5-12500H 2.50 GHz processor (12 cores / 16 threads), 16 GB of DDR4-3200 MHz RAM, running Microsoft Windows 10. Program development and compilation were performed in MSVC 2022, targeting the 64-bit (x64) architecture with SIMD AVX2 extensions enabled. Execution time was measured using the standard clock() function from the <time.h> library.

The program was implemented in MSVC with the following settings:

– Enable Enhanced Instruction Set = Advanced Vector Extensions 2 (x86/x64) (/arch:AVX2);

– Floating Point Model = Fast (/fp:fast)

Analysis of the obtained results (Table 1) reveals significant differences between scalar execution, manual SIMD vectorization, and compiler auto-optimization. The execution time in the scalar version grows nearly linearly with increasing problem size, reaching 4.493 seconds for $N = 10^9$ (Fig. 1). This is expected, since the absence of data-level parallelism limits performance to sequential execution of instructions.

Table 1.

Performance metrics of computational implementations

N	1.0×10^5	1.0×10^6	1.0×10^7	1.0×10^8	1.0×10^9
τ_{scal}	0.000121	0.00135	0.01445	0.10685	4.493
τ_{vec}	0.000046	0.00049	0.00656	0.04635	1.157
τ_{auto}	0.000016	0.00035	0.00433	0.04599	0.855
S_{vec}	2.61	2.76	2.20	2.31	3.88
S_{auto}	7.50	3.92	3.34	2.32	5.25
E_{vec}	0.33	0.34	0.28	0.29	0.49
E_{auto}	0.94	0.49	0.41	0.29	0.66

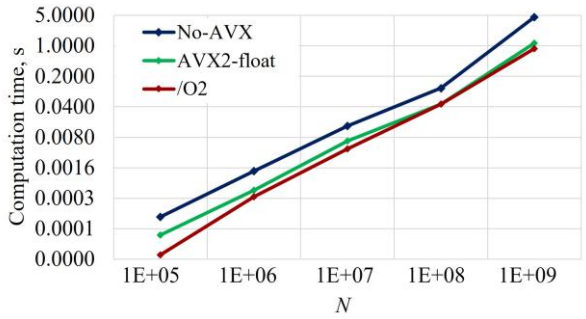

Fig. 1. Execution-time dependence on problem size

Manual SIMD vectorization provides substantial acceleration. For $N = 10^9$, execution time is reduced almost fourfold compared to the baseline scalar version. The corresponding speedup ranges from 2.2 to 3.88 depending on the problem size (Fig. 2). At the same time,

hardware-utilization efficiency remains in the range of 0.28–0.49, indicating incomplete loading of vector registers and potential performance losses due to irregular memory access or synchronization overheads.

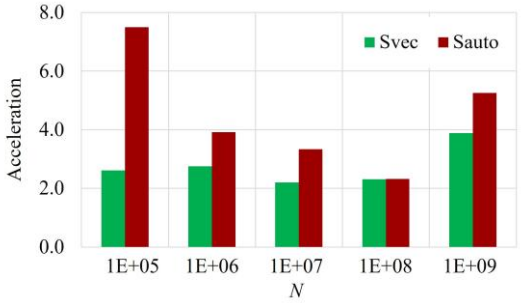

Fig. 2. Comparison of manual and automatic vectorization performance

Compiler auto-optimization demonstrates even higher performance for small and medium problem sizes. For $N = 10^5$, a maximum speedup of 7.5x is achieved with efficiency of 0.94, which is close to the theoretical limit (Fig. 3). However, as data size increases, efficiency decreases. For instance, at $N = 10^8$ it drops to only 0.29, which can be attributed to memory-bandwidth limitations and the influence of data-layout organization on the performance of vector computations.

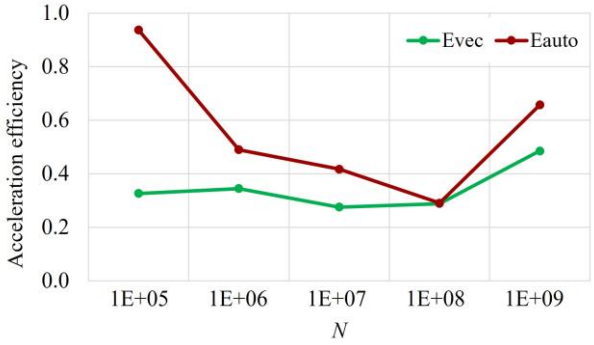

Fig. 3. Comparison of SIMD-resource-utilization efficiency

The results indicate that automatic SIMD vectorization in modern compilers can deliver performance comparable to or even exceeding manual optimization, particularly for small- and medium-sized tasks. At the same time, when scaling to larger datasets, the advantages of manual optimization become more evident due to its ability to better account for memory-organization specifics and to fine-tune loop parameters. These findings confirm the necessity of a combined approach, integrating automatic optimization with selective manual SIMD techniques in performance-critical code sections.

Conclusions. This study analyzed the efficiency of automatic and manual SIMD vectorization for array-processing tasks in the MSVC environment using AVX2 extension instructions.

Automatic vectorization in modern compilers demonstrates a high level of performance, particularly for small- and medium-scale tasks. The achieved speedup of up to 7.5x with an efficiency of 0.94 highlights MSVC's ability to fully utilize processor SIMD resources through aggressive optimizations, including loop unrolling and the effective use of FMA pipelines.

Manual SIMD optimization provides stable performance gains when scaling to larger problem sizes, achieving up to 3.88x acceleration for large arrays. However, its efficiency remains lower (0.28–0.49) due to memory-bandwidth limitations and the less aggressive nature of the transformations compared to automatic compiler optimizations.

The optimal strategy for high-performance applications is a combined use of automatic and manual SIMD optimization methods, which enables a balance between

performance and development effort, while ensuring the scalability of software solutions in computational modeling of complex processes and systems.

Future research directions include expanding the experimental base to cover various processor architectures (Intel, AMD, ARM), analyzing the interaction of SIMD vectorization with other compiler transformations, and applying ML-based methods for the adaptive selection of optimization strategies in HPC and computational modeling tasks.

References

1. Hennessy J.L., Patterson D.A. A New Golden Age for Computer Architecture. *Communications of the ACM.* 2019. Vol. 62, №2. P. 48–60. DOI: https://doi.org/10.1145/3282307
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. 102. P. 67–81. DOI: https://doi.org/10.1007/978-3-031-33309-5_6
3. Imbert C. Computer Simulations and Computational Models in Science. In: *Springer Handbook of Model-Based Science. Springer Handbooks.* Cham: Springer, 2017. P. 735–781. DOI: https://doi.org/10.1007/978-3-319-30526-4_34
4. 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. DOI: https://doi.org/10.3103/s0967091222050138
5. Siso S., Armour W., Thiyagalingam J. Evaluating Auto-Vectorizing Compilers Through Objective Withdrawal of Useful Information. *ACM Transactions on Architecture and Code Optimization.* 2019. Vol. 16, No.4. Article 40. P. 1–23. https://doi.org/10.1145/3356842
6. Zheng R., Pai S. Efficient Execution of Graph Algorithms on CPU with SIMD Extensions. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 2021. P. 262–276. DOI: https://doi.org/10.1109/CGO51591.2021.9370326
7. Haj Ali A. Machine Learning in Compiler Optimization. Berkeley: EECS Department, University of California, 2021. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-2.html
8. Zhulkovskyi O., Zhulkovska I., Vokhmianin H., et al. Application of SIMD-Instructions to Increase the Efficiency of Numerical Methods for Solving SLAE. *Computer Systems and Information Technologies.* 2024. No.4. P. 126–133. DOI: https://doi.org/10.31891/csit-2024-4-15
9. Zhulkovskyi O.O., Vokhmianin H.Ya., Zhulkovska I.I., et al. Acceleration of Image Processing Algorithms Using SIMD Technology. *Informatics and Mathematical Methods in Simulation.* 2025. Vol. 15, №1. P. 15–23. DOI: https://doi.org/10.15276/imms.V15.Vo1.15
10. Feng J., He Y., Tao Q. Evaluation of Compilers' Capability of Automatic Vectorization Based on Source Code Analysis. *Scientific Programming.* 2021. P. 1–15. DOI: https://doi.org/10.1155/2021/3264624
11. Aleen F., Zakharin V.P., Krishnaiyer R., et al. Automated Compiler Optimization of Multiple Vector Loads/Stores. *International Journal of Parallel Programming.* 2018. Vol. 46. P. 471–503. DOI: https://doi.org/10.1007/s10766-016-0485-7
12. Ashouri A.H., Killian W., Cavazos J., et al. A Survey on Compiler Autotuning Using Machine Learning. *ACM Computing Surveys.* 2018. Vol. 51, №5. Article 96. P. 1–42. DOI: https://doi.org/10.1145/3197978
13. Cho D., Pasricha S., Issenin I., et al. Compiler Driven Data Layout Optimization for Regular/Irregular Array Access Patterns. *ACM SIGPLAN Notices.* 2008. Vol. 43, №7. P. 41–50. https://doi.org/10.1145/1379023.1375664

14. Sakib N., Prabhu T., Santhi N., et al. Comparison of Vectorization Capabilities of Different Compilers for x86 and ARM CPUs. *arXiv*. 2025. DOI: https://doi.org/10.48550/arXiv.2502.11906
15. Wang Z., O'Boyle M. Machine Learning in Compiler Optimization. In: *Proceedings of the IEEE*. 2018. Vol. 106, №11. P. 1879–1901. DOI: https://doi.org/10.1109/JPROC.2018.2817118
16. Vu S.T., Heydemann K., de Grandmaison A., Cohen A. Secure Delivery of Program Properties Through Optimizing Compilation. In: *Proceedings of the 29th International Conference on Compiler Construction (CC 2020).* 2020. P. 14–26. DOI: https://doi.org/10.1145/3377555.3377897
17. Wang J., Yu L., Zhuang W., Yang X., Zhang S., Qin Z. Research on Vector Extension of Instruction Set Architecture. In: *2024 3rd International Conference on Cloud Computing, Big Data Application and Software Engineering (CBASE).* Hangzhou, China, 2024. P. 378–385. DOI: https://doi.org/10.1109/CBASE64041.2024.10824427
18. Van Hoey J. AVX. In: *Beginning x64 Assembly Programming.* Berkeley, CA: Apress, 2019. P. 307–315. https://doi.org/10.1007/978-1-4842-5076-1_35
19. Intel Intrinsics Guide. Intel. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

ПРОБЛЕМИ АВТОМАТИЧНОЇ ОПТИМІЗАЦІЇ ПРОГРАМНОГО КОДУ КОМПІЛЯТОРОМ

І.І. Жульковська[1], О.О. Жульковський[1],
Т.М. Рудянова[2], О.Ю. Лебідь[2], М.Ф. Мормуль[2]

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

Раціональне використання можливостей сучасних компіляторів, зокрема автоматичної SIMD-векторизації, дозволяє значно підвищити продуктивність обчислень у задачах обробки масивів даних, комп'ютерного моделювання складних процесів та систем. Зростання вимог до продуктивності програмного забезпечення у наукових обчисленнях, аналізі великих даних, задачах штучного інтелекту та машинного навчання робить актуальним використання апаратного паралелізму на рівні даних.

У роботі досліджується ефективність автоматичної SIMD-векторизації компілятором Microsoft Visual C++ у порівнянні з ручною оптимізацією, що реалізується через AVX2-інструкції. Для оцінки продуктивності були розроблені три реалізації обчислень: скалярна базова версія, автоматично оптимізований код компілятора, а також ручна SIMD-версія із застосуванням intrinsic-функцій. Обчислювальні експерименти проведено на прикладі операції SAXPY для масивів розміром 10^5–10^9.

Результати показали, що автоматична SIMD-векторизація забезпечує прискорення до 7.5x із ефективністю 0.94 для задач малої та середньої розмірності, максимально використовуючи ресурси процесора завдяки агресивним оптимізаціям, таким як розгортання циклів та ефективне використання FMA-конвеєрів. Ручна SIMD-оптимізація демонструє стабільне прискорення до 3.88x для великих масивів, проте з нижчою ефективністю (0.28–0.49) через обмеження пропускної здатності пам'яті та меншу агресивність трансформацій. Порівняння показало, що автоматичні методи є більш зручними для розробника, дозволяючи значно зменшити трудомісткість написання SIMD-коду. Водночас ручні оптимізації залишаються актуальними при масштабуванні задач на великі обсяги даних. Результати роботи свідчать, що оптимальною стратегією є комбіноване застосування автоматичних і ручних SIMD-трансформацій, що дозволяє досягти балансу між продуктивністю, точністю та зручністю розробки, забезпечуючи ефективність і масштабованість програмних рішень у високопродуктивних обчисленнях і комп'ютерному моделюванні. Перспективи подальших досліджень пов'язані з розширенням експериментальної бази на різні архітектури процесорів, аналізом взаємодії SIMD-векторизації з іншими компіляторними трансформаціями та застосуванням ML-методів для адаптивного вибору оптимізаційних стратегій.