https://orcid.org/0009-0000-4743-4023
https://orcid.org/0000-0002-9639-3544

УДК 004.4`2

М.О. БОГДЮК[*], О.Є. КОВАЛЕНКО[*]

АВТОМАТИЗАЦІЯ ВЕКТОРИЗАЦІЇ КОДУ ПРОГРАМ ДЛЯ ВИКОРИСТАННЯ SIMD-ІНСТРУКЦІЙ

[*]Національний університет біоресурсів і природокористування України, м. Київ, Україна

Анотація. *Суттєве уповільнення темпів зростання продуктивності процесорів пов'язане з наближенням до фізичних обмежень мінімізації розмірів транзисторів. Це призвело до того, що традиційні шляхи підвищення швидкодії (збільшення тактової частоти чи кількості ядер) вже не в повній мірі забезпечують приріст продуктивності. У сучасних процесорах важливу роль у підвищенні продуктивності обчислень відіграють інструкції типу SIMD (Single Instruction, Multiple Data), що дозволяють процесору виконувати одну операцію над кількома потоками даних одночасно на одному ядрі. У такій ситуації ключового значення набуває паралельна обробка даних, що реалізується шляхом використання SIMD-інструкцій. Спрямування процесу компіляції за допомогою спеціальних директив, опцій та правильної організації даних дозволяє отримати оптимізований векторизований код без необхідності явного використання SIMD-інструкцій у вихідному тексті програми. Але ефективність таких механізмів значною мірою залежить від низки чинників: архітектурних особливостей процесора, версії компілятора, якості початкового коду та стану оптимізатора. У статті проведено аналіз існуючих методів та засобів векторизації коду для використання SIMD-інструкцій та запропоновано підхід до автоматизації векторизації коду програм шляхом вставки міток у вихідний код, виявлення меж ділянок у скомпільованому коді, аналізу інструкцій у межах ділянок і формування звіту з прив'язкою до вихідного коду. Проведено порівняльний аналіз існуючих засобів векторизаці з розробленим прототипом. Запропонований підхід дозволяє автоматизувати перевірку факту векторизації коду з точністю до рядка і працює навіть при використанні агресивної оптимізації, включаючи вбудовування функцій.*

Ключові слова: *паралельні обчислення, SIMD, оптимізація коду, векторизація коду, автоматизація векторизації.*

Abstract. *A significant slowdown in processor performance growth is associated with approaching the physical limits of reducing transistor size. This has led to the situation where traditional ways of increasing performance (increasing the clock frequency or the number of cores) no longer fully deliver performance gains. In modern processors, SIMD (Single Instruction, Multiple Data) instruction-set architectures play an important role in increasing computing performance. They allow a processor to perform one operation on multiple data streams simultaneously on a single core. In such a situation, parallel processing at the data level, implemented using SIMD instructions, becomes key. Directing the compilation process with special directives, options, and proper data organization makes it possible to obtain an optimized, vectorized code without explicitly using SIMD instructions in the program source code. However, the effectiveness of such mechanisms largely depends on a number of factors: the processor architectural features, the compiler version, the source code quality, and the optimizer state. The article analyzes some existing methods and tools for vectorizing code using SIMD instructions and proposes an approach to automating vectorization by inserting labels into source code, detecting section boundaries in compiled code, analyzing instructions within sections, and generating a report with a link to the source code. A comparative analysis of existing vectorization tools and the developed prototype is conducted. The proposed approach allows for automating the verification of code vectorization with line-level accuracy and works even with aggressive optimization, including function embedding.*

ISSN 1028-9763. Математичні машини і системи. 2026. № 1

Keywords: *parallel computing, SIMD, code optimization, code vectorization, vectorization automation.*

DOI: 10.34121/1028-9763-2026-1-21-28

1. Вступ

Паралельна обробка даних є основним методом підвищення ефективності обробки інформації. У сучасних процесорах важливу роль у підвищенні продуктивності обчислень відіграють інструкції типу SIMD (Single Instruction, Multiple Data), що дозволяють процесору виконувати одну операцію над кількома потоками даних одночасно на одному ядрі. Вони присутні в процесорах, орієнтованих на обробку мультимедійних, графічних та наукових даних. Зокрема, обробка SIMD реалізована у найпопулярніших архітектурах x86-64 (x86_64, x64, AMD64, Intel 64), x86 з розширеннями MultiMedia eXtensions [1] (з 1997 року) та ARM з розширеннями Neon [2].

Наразі немає інструменту, який би автоматично перевіряв факт векторизації ділянки коду з точністю до рядка. Тому актуальною є задача створення засобів, що дозволяють автоматично перевіряти наявність SIMD-інструкцій у заданих ділянках коду. Якщо оптимізація була скасована, такі інструкції будуть відсутні, і засіб поверне помилку. Такий засіб призначений для інтеграції в системи CI (Continuous Integration) разом із модульними тестами, щоб при відсутності оптимізації збірка продукту припинялась.

Метою статті є розробка алгоритму і програмного засобу, який забезпечує контроль наявності чи відсутності певних інструкцій у заданих ділянках скомпільованого коду.

2. Аналіз попередніх робіт

Протягом останніх років спостерігається суттєве уповільнення темпів зростання продуктивності процесорів [3], пов’язане із сповільненням дії закону Мура через наближення до фізичних обмежень мінімізації розмірів транзисторів [4]. Це призвело до того, що традиційні шляхи підвищення швидкодії (збільшення тактової частоти чи кількості ядер) вже не в повній мірі забезпечують приріст продуктивності. У такій ситуації ключового значення набуває паралельна обробка даних, що реалізується шляхом використання SIMD-інструкцій. Саме автоматична векторизація компіляторів стає основним механізмом більш повного використання потенціалу апаратної платформи, особливо у наукових, інженерних та мультимедіа застосунках.

Для оцінки якості векторизації у статті [5] було запропоновано методику оцінювання компіляторів шляхом систематичного виділення корисної інформації, яка зазвичай полегшує прийняття рішень оптимізатором (межі циклів, параметри індексів масивів, атрибути змінних). Такий підхід дозволив виявити, що саме дефіцит інформації є ключовою причиною провалу векторизації в реальних умовах. Наприклад, відсутність індексних параметрів або атрибутів змінних суттєво знижує ефективність векторизації, а в окремих випадках навіть вилучення умов могло несподівано сприяти використанню SIMD-інструкцій. Попри це, дослідження має певне обмеження: воно оцінює векторизацію на рівні бенчмарків і функцій, але не дозволяє прив’язати результат до конкретних рядків вихідного коду, що є важливим для практичних задач векторизації у великих проєктах. Важливим висновком цієї роботи є те, що швидкодія після застосування автовекторизації все ще далека від пікової швидкодії архітектури.

Ще одним популярним напрямом для покращення векторизації є виконання додаткових перевірок разом із компілятором. Наприклад, на конференції PACT '22 [6] було представлено метод VecRC (Vectorization with Run-time Checks), що поєднує compile-time аналіз із динамічними перевірками під час виконання. Завдяки цьому векторизація стає можливою навіть у випадках, коли код містить складний потік керування і статичний аналіз вказує на його потенційну небезпеку. Використання ймовірнісної моделі вартості дозволяє прийняти рішення, коли додавання run-time перевірок є виправданим. Результати експериментів на

бенчмарках SPEC2017, NPB, TSVC і Rodinia показали відчутний приріст продуктивності в порівнянні з GCC, Clang та ICC. Водночас обмеженням підходу є накладні витрати, пов'язані з перевірками, та необхідність запуску коду, що ускладнює його використання для статичного аналізу або інтеграції у CI/CD.

Окремим напрямом стали спроби застосування методів штучного інтелекту, зокрема великих мовних моделей. Так, LLM-Vectorizer [7] вперше продемонстрував, що LLM можна використати для автоматичного перетворення скалярних циклів у векторизовану форму. У цьому підході трансформації, згенеровані мовною моделлю, перевіряються на рівні проміжного представлення (IR) за допомогою Alive2, що гарантує семантичну еквівалентність між вихідним і оптимізованим кодами. Дослідження підтвердило, що такий підхід дозволяє векторизувати низку функцій, які традиційні компілятори залишали у скалярному вигляді, тим самим відкриваючи нові можливості для інтеграції AI-засобів у компіляторні пайплайни. Обмеженнями лишаються залежність від тестового покриття та ризик генерації некоректного коду, однак запропонований метод продемонстрував високу перспективність у підвищенні ступеня векторизації.

Логічним продовженням цього напряму став фреймворк VecTrans, описаний у роботі [8]. Він поєднує статичний аналіз компілятора та трансформації коду за допомогою LLM. Алгоритм VecTrans ідентифікує регіони, де компілятор не здійснює векторизацію, намагається їх рефакторити за допомогою LLM і перевіряє коректність результатів на рівні IR та юніт-тестів. На практиці це дозволило векторизувати близько 46 % функцій, які раніше не піддавалися оптимізації, із середнім прискоренням виконання близько двох разів. Важливою перевагою є низька вартість трансформацій (≈ $0,012 за функцію). Водночас метод залишається залежним від якості LLM і може генерувати некоректний код, що вимагає ретельної верифікації.

3. Опис методики удосконалення

Спрямування процесу компіляції за допомогою спеціальних директив, опцій та правильної організації даних дозволяє отримати оптимізований векторизований код без необхідності явного використання SIMD-інструкцій у вихідному тексті програми. Але ефективність таких механізмів значною мірою залежить від низки чинників: архітектурних особливостей процесора, версії компілятора, якості початкового коду та стану оптимізатора. Це зумовлює їхню ненадійність і вимагає систематичної перевірки факту застосування векторизації у практичних сценаріях розробки програмного забезпечення.

Загальний підхід до удосконалення векторизації коду було представлено на конференції TAACSD-2025 [9] і враховує такі особливості інструментів векторизації:

1. Автоматична векторизація компілятора, «#pragma loop(…)» в C++, вміє видавати попередження, якщо векторизація не була успішною, однак є випадки, коли не буде ні векторизації, ні попередження:

– код був видалений оптимізатором (неініціалізована змінна, невикористаний результат);

– застосовано SIMD іншого типу (SSE, а не очікуване AVX).

2. Інструменти статичного аналізу:

– напівавтоматичний аналіз «*.s» файлів: компілятор разом із бінарним кодом може видати файл з ассемблерними інструкціями. Такий файл можна відфільтрувати grep'ом, однак із цим підходом важко прив'язати позицію інструкцій до функцій, особливо, якщо було виконано «inlining» (вбудовування) функції;

– llvm-mca є потужним інструментом бінарного коду. Для кожної інструкції він виводить затримки, факт запису/читання з пам'яті, кількість мікрооперацій;

– Godbolt є онлайн інструментом, теж корисний на етапі підгонки коду, однак не може бути автоматизованим.

3. Інструменти динамічного аналізу (профайлери).

Всі ці інструменти мають такі недоліки:

• слабко автоматизовані й важко інтегруються з CI;

• низька деталізація (з точністю до функції, а не до рядка коду);

• складнощі з вбудовуванням inline.

Для усунення зазначених недоліків запропоновано такі ключові елементи удосконалення:

• можливість маркувати ділянки коду;

• пошук ділянок, позначених SIMD_BEGIN.. SIMD_END;

• перевірка факту наявності/відсутності SIMD-інструкцій у знайдених ділянках;

• підтримка різних компіляторів, архітектур, наборів інструкцій;

• командний інтерфейс, можливість інтеграції у CI/CD.

Запропонована методика передбачає такі етапи:

• вставка міток у вихідний код;

• виявлення меж ділянок у скомпільованому коді;

• аналіз інструкцій у межах ділянок;

• формування звіту з прив'язкою до вихідного коду.

Критеріями оцінки ефективності запропонованих інструментів удосконалення у порівнянні з існуючими є:

• рівень деталізації (рядок, функція, модуль);

• як працює, коли функція вбудована (inline);

• простота інтеграції (розмір, залежності);

• покриття і деталізація інструкцій (конкретні інструкції, набір);

• ступінь автоматизації.

4. Реалізація прототипу

Для реалізації прототипу було обрано мову програмування Rust [10], оскільки це мова системного програмування з акцентом на мінімізацію вразливостей роботи з пам'яттю і ресурсами. Використані дві зовнішні бібліотеки з мінімальними залежностями:

– goblin — для читання контейнерів скомпільованого коду різних архітектур: PE (Windows, EXE/DLL), ELF (Linux), Mach-O (macOS) [11];

– iced-x86 — дисасемблер інструкцій x86 та amd64 [12].

Діаграма послідовності виконання прототипу представлена на рис. 1.

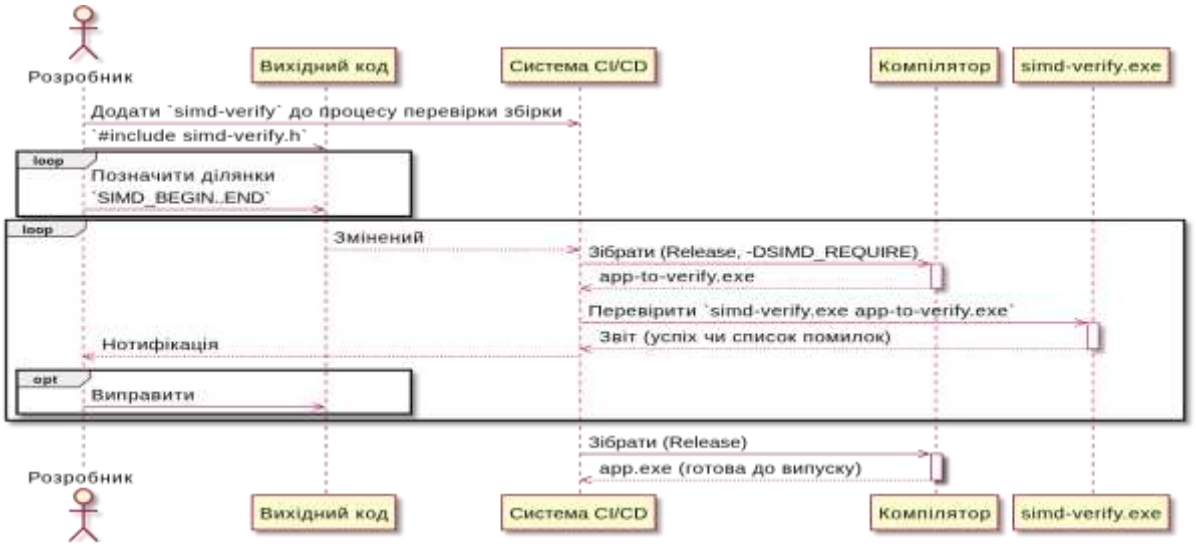

Рисунок 1 — Діаграма послідовності виконання прототипу для перевірки коду C/C++ і CI/CD для Windows

Узагальнений алгоритм роботи самого аналізатора SIMD-verify представлений діаграмою діяльності на рис. 2.

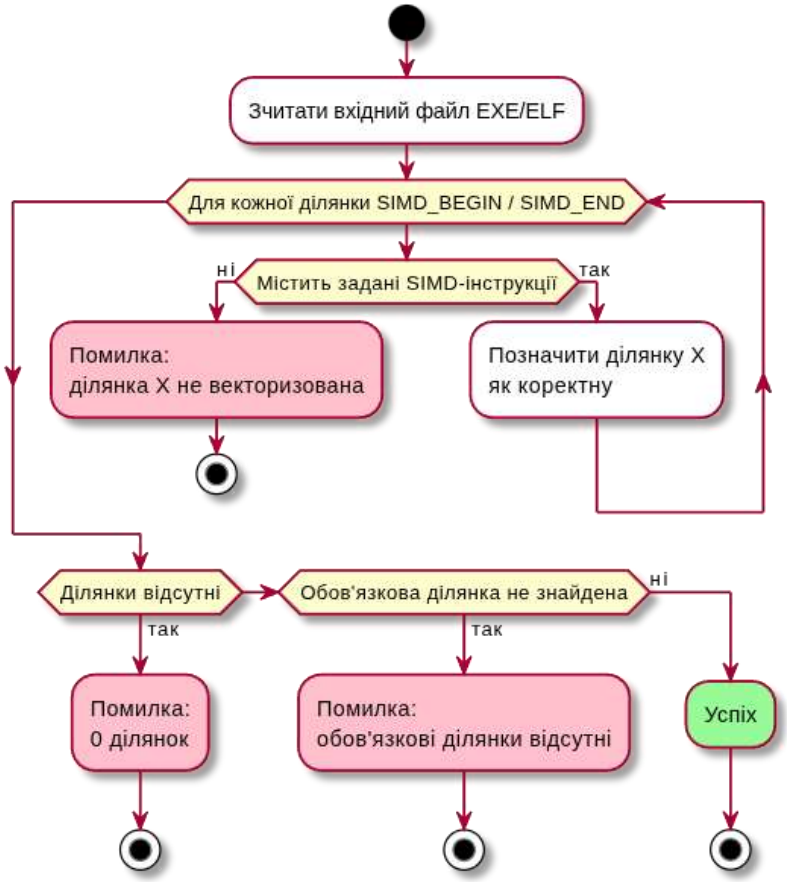

Рисунок 2 — Узагальнений алгоритм роботи аналізатора коду

Найскладніша задача — позначити ділянки так, щоб це мало мінімум побічних ефектів часу виконання. Пропонована реалізація передбачає глобальну змінну, запис до якої використовується як маркер (рис. 3):

```
__attribute__((used)) static volatile const char* simd_verify;
#define SIMD_BEGIN(instructions, ...) \
    { ::simd_verify = "`SIMDVERIFY`" instructions "`" __VA_ARGS__ "`DMIS"; }
#define SIMD_END() { \
        atomic_thread_fence(memory_order_release); \
        ::simd_verify = "`SIMDVERIFY`_``DMIS"; \
    }
```

Рисунок 3 — Фрагмент коду SIMD-verify.h

При позначенні ділянки *SIMD_BEGIN("AVX", "add_numbers") .. SIMD_END()* у секцію *.rodata* бінарного файла додаються 2 рядки:

«`SIMDVERIFY`AVX`add_numbers`DMIS» та *«`SIMDVERIFY`_``DMIS»*, як зображено на рис. 4.

```
Contents of section .rodata:
 2000 01000200 00247449 6053494d 44564552  .....$tI`SIMDVER
 2010 49465960 41565860 6164645f 6e756d62  IFY`AVX`add_numb
 2020 65727360 444d4953 00605349 4d445645  ers`DMIS.`SIMDVE
 2030 52494659 605f6060 444d4953 0073756d  RIFY`_``DMIS.sum
```

Рисунок 4 — Вміст секції *.rodata*

У самому коді додається по дві інструкції (сумарно 28 байт), що записують у глобальну змінну, як показано на рис. 5. У архітектурі amd64 використовується відносна адресація, тому замість «*lea rax,[0x2008]*» згенерувався відносний запис «*[rip+0xeb3]*». Це, а також те, що замість RAX може бути будь-який інший регістр загального призначення, ускладнює пошук позначених ділянок: потрібно знайти всі інструкції виду «48 8d [05|0d|15|1d|35|3d] …», після яких іде число, рівне *(marker_address-RIP-instruction_size)*, де RIP — адреса виконуваного коду, різна для кожної інструкції.

```
114e:  48 8d 05 b3 0e 00 00   lea   rax,[rip+0xeb3]          # 2008 =114e+7+0eb3
1155:  48 89 05 d4 2e 00 00   mov   QWORD PTR [rip+0x2ed4],rax  # 4030 =1155+7+2ed4
                              ...(ділянка)...
11a1:  48 8d 05 81 0e 00 00   lea   rax,[rip+0xe81]          # 2029 =11a1+7+0e81
11b3:  48 89 05 76 2e 00 00   mov   QWORD PTR [rip+0x2e76],rax  # 4030 =11b3+7+2e76
```

Рисунок 5 — Інструкції, що позначають ділянку

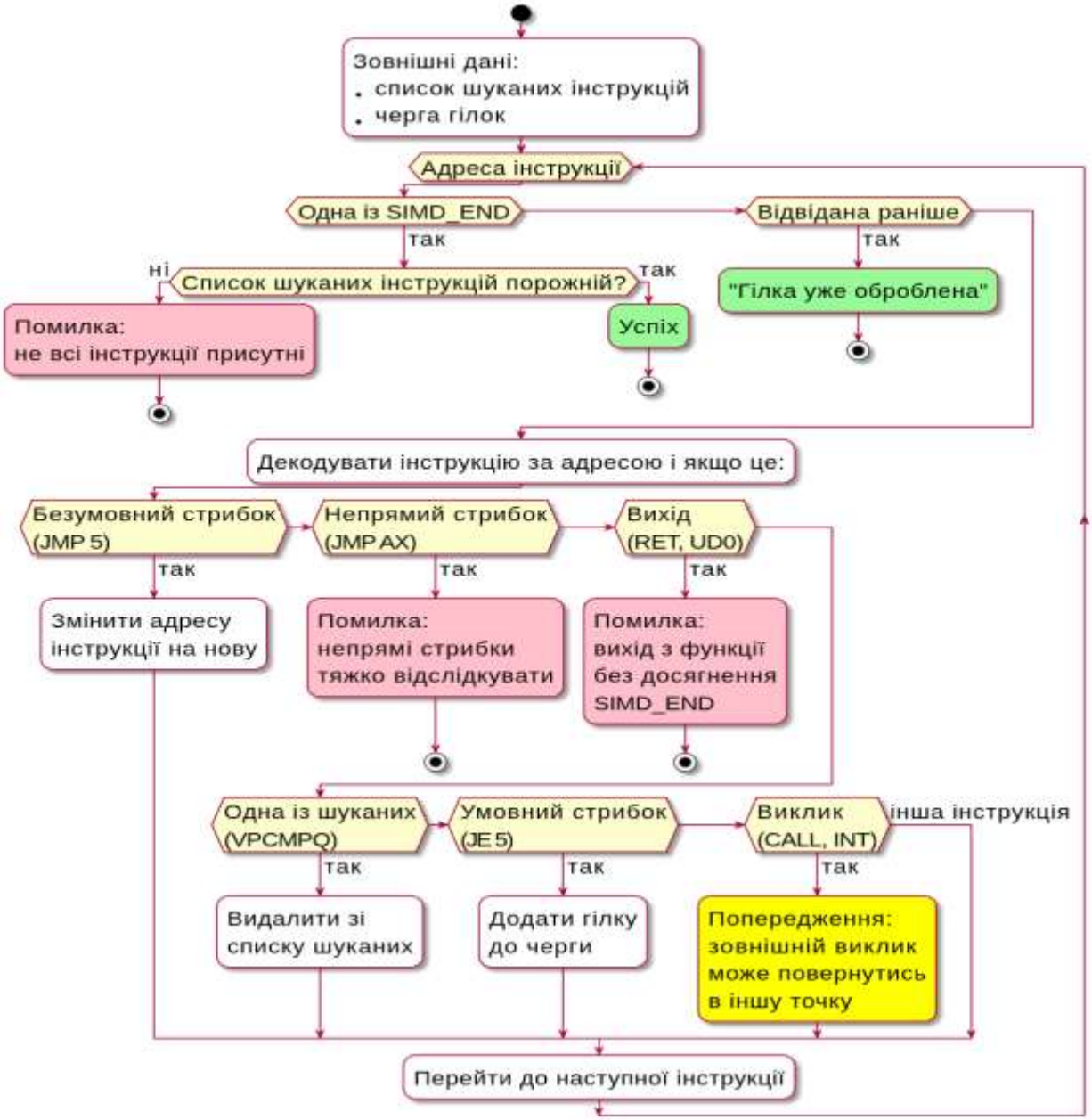

Рисунок 6 — Алгоритм аналізу фрагмента коду для векторизації

Для запобігання видалення компілятором цих інструкцій під час оптимізації глобальна змінна позначена як *volatile*, а для запобігання переміщенню (порушивши межі ділянки) використовується функція *atomic_thread_fence()*.

Наступним етапом після визначення адрес маркерів є встановлення відношень між ними. Алгоритм передбачає почергове виконання інструкцій, а також перевірку умов для визначення можливих шляхів виконання (рис. 6).

У випадку, якщо не всі гілки, що починаються з *SIMD_BEGIN()*, завершуються *SIMD_END()*, генерується помилка. У випадку явного використання інструкцій «раннього виходу» в межах ділянки (наприклад, «*if (error) return;*»), користувач може змінити цю поведінку шляхом зміни конфігурації.

5. Експериментальні результати

У табл. 1 наведено порівняння основних параметрів запропонованої методики з існуючими інструментами згідно з критеріями, визначеними вище в цій роботі.

Таблиця 1 — Порівняння інструментів аналізу інструкцій

	Деталізація	Inline	Залежності	Задання вимог	Автоматизованість
SIMD-verify (прототип)	Рядок	+	Мінімум: GLibC/MSVC	Сімейства та/або інструкції	+
#pragma loop	Рядок	+	C++	Ширина вектора	+
grep *.s	Модуль або функція	–	Мінімум	Інструкції	Часткова
llvm-mca	Рядок	–	LLVM	Сімейства та/або інструкції	–
Godbolt	Функція	–	Браузер	Інструкції	–

6. Висновки

Запропонований підхід дозволяє автоматизувати перевірку факту векторизації коду з точністю до рядка і працює навіть при використанні агресивної оптимізації, включаючи вбудовування функцій.

Подальше удосконалення запропонованої методики буде проводитись у напрямах:

– розширення підтримки архітектур систем інструкцій (ARM64, RISC-V, i86) і контейнерів (Mach-O);

– інтеграції з іншими мовами системного програмування: Rust, Swift;

– мінімізації run-time слідів (наразі це дві базові інструкції на початку ділянки і дві в кінці контролю);

– вдосконалення API для підтримки більш гнучких запитів (наприклад, «ділянка повинна містити 10 інструкцій із набору AVX512F, крім VPCMP*, і не повинна містити жодної інструкції PAVG*»).

СПИСОК ДЖЕРЕЛ

1. Intel, Intel Advanced Vector Extensions 10 Architecture Specification, rev 1.0, order no. 355989-001US, July 2023. URL: https://cdrdv2-public.intel.com/784267/355989-intel-avx10-spec.pdf.
2. Introduction to ARM NEON SIMD Intrinisics. URL: https://masterchef2209.wordpress.com/2020/08/12/introduction-to-arm-neon-simd-intrinisics-guide-for-simde-neon-impls/.

3. Coyle D., Hampton L. 21st century progress in computing. *Telecommunications Policy*. 2024. N 48 (1). P.102649. URL: https://doi.org/10.1016/j.telpol.2023.102649.
4. Burg D., Ausubel J.H. Moore's Law revisited through Intel chip density. *PLoS ONE*. 2021. N 16 (8). e0256245. DOI: https://doi.org/10.1371/journal.pone.0256245.
5. Siso S., Armour W., Thiyagalingam J. Evaluating auto-vectorizing compilers through objective withdrawal of useful information. *ACM Transactions on Architecture and Code Optimization (TACO)*. 2019. Vol. 16, N 4. P. 1–23. DOI: https://doi.org/10.1145/3356842.
6. Liu B., Laird A., Tsang W.H., Mahjour B., Dehnavi M.M. Combining Run-Time Checks and Compile-Time Analysis to Improve Control Flow Auto-Vectorization. *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT '22)*. Association for Computing Machinery. New York, NY, USA, 2023. P. 439–450. DOI: https://doi.org/10.1145/3559009.3569663.
7. Taneja J., Laird A., Yan C., Musuvathi M., Lahiri S.K. Llm-vectorizer: Llm-based verified loop vectorizer. *Proc. of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 2025. March. P. 137–149. DOI: https://doi.org/10.48550/arXiv.2406.04693.
8. Zheng Z., Wu K., Cheng L., Li L., Rocha R.C., Liu T., Wei W., Zeng J., Zhang X., Gao Y. VecTrans: Enhancing Compiler Auto-Vectorization through LLM-Assisted Code Transformations. 2025. arXiv preprint. DOI: https://doi.org/10.48550/arXiv.2503.19449.
9. Богдюк М.О., Коваленко О.Є. Перевірка векторизації інструкцій. *Теоретичні та прикладні аспекти розробки комп'ютерних систем 2025 (TAACSD-2025)*: VII Всеукр. наук.-практ. інтернет конф. студентів і аспірантів. Київ: Національний університет біоресурсів і природокористування України, 2025. URL: https://drive.google.com/file/d/179dsQND4mvTZLH4_MR-0gpSwwKJ35u6Y/view?usp=sharing.
10. Rust. A language empowering everyone to build reliable and efficient software. URL: https://rust-lang.org/.
11. Crate goblin. URL: https://docs.rs/goblin/latest/goblin/.
12. Crate iced_x86. URL: https://docs.rs/iced-x86/latest/iced_x86/.

Стаття надійшла до редакції 03.12.2025 / прийнята до друку 12.02.2026