УДК 681.325+004.4 **DOI:10.15330/itee.2025.2.04**

В. І. ГОЛОТА, канд. техн. наук, В. М. ГРИГА, канд. техн. наук,
Т. Г. БЕНЬКО, А. В. МОРГУН

ВИКОРИСТАННЯ ВЕКТОРНИХ ІНСТРУКЦІЙ МІКРОПРОЦЕСОРА X86-64 В ЗАДАЧАХ ЛІНІЙНОЇ АЛГЕБРИ

Вступ

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням обсягів даних, що підлягають обробці в режимі реального часу. Сфери застосування високопродуктивних обчислень значно розширилися: від класичного наукового моделювання фізичних процесів та прогнозування погоди до завдань штучного інтелекту, глибокого машинного навчання, обробки потокового відео надвисокої роздільної здатності та криптографічного захисту інформації. У всіх згаданих галузях основна частка обчислювального навантаження припадає на операції над векторами та матрицями.

Тривалий час підвищення продуктивності програмного забезпечення досягалося екстенсивним шляхом – за рахунок зростання тактової частоти центральних процесорів. Сучасна парадигма підвищення швидкодії змістилася в бік архітектурного паралелізму. Однією з ключових технологій, що дозволяє ефективно використовувати ресурси центрального процесора є принцип «одна інструкція – множина даних» (Single Instruction, Multiple Data (SIMD)).

Розширення архітектура x86-64 Advanced Vector Extensions (AVX), надало розробникам для використання 256-бітові регістри YMM, що дозволяє виконувати арифметичні операції над вісьмома числами одинарної точності (float) або чотирма числами подвійної точності (double) за один машинний такт [1, 2]. Це теоретично дозволяє збільшити швидкодію програм у 4–8 разів.

Попри те, що сучасні компілятори мов високого рівня (C++, Rust) мають механізми автовекторизації, вони не завжди генерують оптимальний машинний код через складність аналізу залежностей даних та управління пам'яттю [3, 4]. Тому для створення критично важливих бібліотек, драйверів та ядер обчислювальних систем використовується мова асемблера [6, 7]. Безпосередня оптимізація на рівні асемблерних команд дозволяє розробнику повністю контролювати розподіл регістрів, планування конвеєра команд та роботу з кеш-пам'яттю [8].

Основними високопродуктивними мікропроцесорами у настільних і портативних комп'ютерах є серії Core i3/i5/i7/i9, Ultra від Intel та серії Ryzen 3/5/7/9 від AMD. Основними асемблерами для цих мікроконтролерів є MASM, NASM, FASM та GAS. В освітньому середовищі активно використовується NASM асемблер [9-11]. Це зумовлено безоплатними ліцензіями "Simplified BSD License" і "FreeBSD License", підтримкою різних платформ Windows, DOS, Linux, BSD, MAC OS та найновіших векторних інструкцій AVX, AVX2, AVX-512.

Таким чином, на даний час актуальним є використання векторних інструкцій AVX мікропроцесорів x86-64, що дозволить використати апаратний паралелізм в обробці даних та підвищити продуктивність програм, особливо для задач лінійної алгебри.

1 Аналіз сучасного стану та перспективи розвитку векторних обчислень

1.1 Еволюція архітектури процесорів: від SISD до SIMD

Класична архітектура фон Неймана передбачає послідовне виконання команд за принципом «одна інструкція - одиничні дані» (Single Instruction, Single Data, (SISD)). У такій моделі для додавання двох масивів з N елементів процесор змушений виконати N операцій завантаження, N операцій додавання та N операцій збереження результату. Це створює значне навантаження на блок декодування інструкцій та шину даних.

Для подолання цього бар'єру було розроблено технологію SIMD. Суть технології полягає в тому, що одна інструкція ініціює виконання однієї й тієї ж арифметичної дії над цілим вектором значень одночасно. Це дозволяє реалізувати паралелізм на рівні даних, а не на рівні операційної системи, що суттєво зменшує накладні витрати на перемикання контексту [12]. Приклади архітектур SISD та SIMD показано на на рис 1.

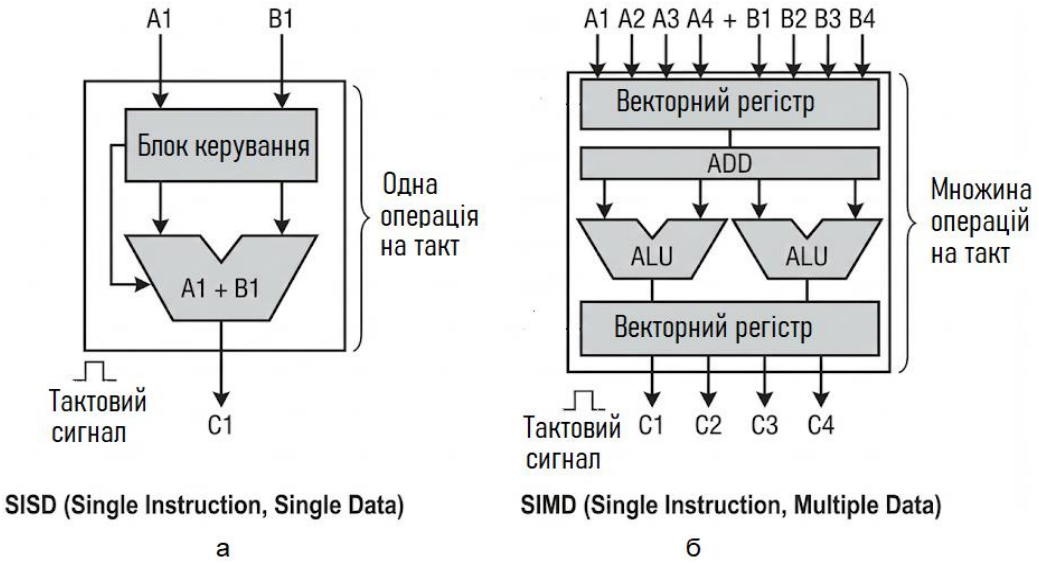

Рис. 1. Архітектури SISD та SIMD: а) додавання операндів (A1)+(B1);
б) додавання векторів (A1,A2,A3,A4) + (B1,B2,B3,B4)

1.2 Огляд розширень системи команд Intel x86-64

Еволюція векторних розширень архітектури x86-64 відбувалася поетапно, кожне нове покоління збільшувало розрядність регістрів та розширювало набір доступних інструкцій.

1. MMX (MultiMedia Extensions, 1997). Перша спроба впровадження SIMD. В MMX використовувалися 64-бітові регістри, які фізично накладалися на регістри співпроцесора (FPU), що унеможливлювало одночасну роботу з дійсними числами та MMX-командами.

2. SSE (Streaming SIMD Extensions, 1999). Наступний крок, що додав 128-бітові регістри XMM (XMM0–XMM15 у 64-бітному режимі). Це дозволило обробляти чотири 32-бітові числа з плаваючою крапкою (float) одночасно. Подальші версії (SSE2, SSE3, SSE4.1, SSE4.2) додали підтримку чисел подвійної точності (double) та розширили функціонал для роботи з цілими числами.

3. AVX (Advanced Vector Extensions, 2011). В AVX розширено регістри до 256 біт (YMM), що дозволило обробляти вже 8 чисел float або 4 числа double за такт. Важливою особливістю стала зміна синтаксису асемблерних команд з двоопераднього на триопарандний.

4. AVX2 (2013). Розширено можливості першої версії AVX, додавши повноцінну підтримку 256-бітових операцій над цілими числами та інструкції FMA (Fused Multiply-Add), які виконують множення та додавання за один крок без втрати точності.

5. AVX-512. Найсучасніше розширення з 512-бітовими регістрами ZMM, яке доступне переважно у серверних процесорах (Xeon) та високопродуктивних настільних комп'ютерах.

1.3 Архітектурні особливості AVX та регістровий файл YMM

Для виконання векторних операцій у наборі команд AVX використовуються регістри YMM. У 64-бітовому режимі роботи процесора (Long Mode) програмісту доступно 16 регістрів: YMM0 – YMM15.

Кожний регістр YMM (рис. 2) має довжину 256 біт (32 байти). Архітектурно молодші 128 біт регістру YMM відповідають регістру XMM (успадкованому від SSE). Це забезпечує зворотну сумісність: команд, що працюють з SSE, автоматично змінюють молодшу частину YMM, залишаючи старшу частину без змін (у деяких режимах) або обнуляючи її.

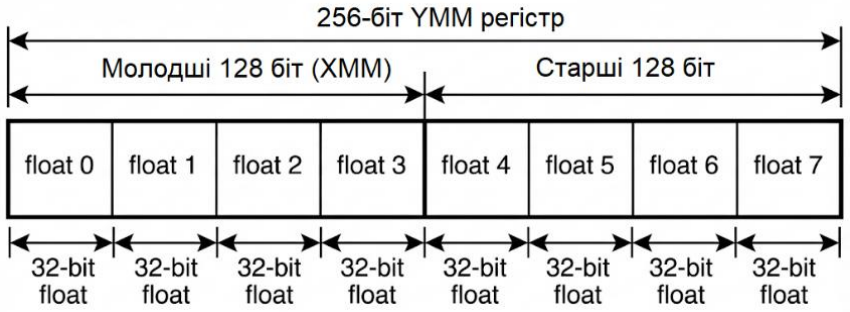

Рис. 2. Структура регістра YMM

YMM не має фіксованого типу, а підтримує запаковані і скалярні формати даних: 32×int8, 16×int16, 8×int32/float, 4×int64/double.

Важливою вимогою при роботі з AVX є вирівнювання даних у пам'яті. Для досягнення максимальної швидкодії адреси початку масивів повинні бути кратними 32 байтам. Використання невирівняних даних (інструкції типу vmovups) може призводити до штрафів продуктивності.

Можливо, найбільш примітним аспектом AVX є використання нового синтаксису команд. Більшість AVX команд використовують формат з двох операндів джерела та одного операнда призначення:

```
InstrMnemonic DesOp, SrcOp1, SrcOp2,
```

де `InstrMnemonic` означає мнемоніку AVX команд, а `DesOp`, `SrcOp1` і `SrcOp2` позначають операнди призначення та джерел відповідно. Майже всі операнди джерела є неруйнівними (тобто не змінюються під час виконання команди), за винятком випадків, коли регістр операнда призначення збігається з одним із регістрів операндів джерела.

Нові команди AVX поділені на наступні підгрупи:

- широкомовна трансляція (Broadcast);
- змішування (Blend);
- переставлення (Permute);
- видобування та вставлення (Extact and Insert);
- пересилання з маскою (Masked move);
- змінний бітовий зсув (Variale bit sift);
- збирання (Gather).

2 Алгоритмічне забезпечення векторних обчислень

2.1 Векторизація базових операцій

Перш ніж переходити до розв’язування задач лінійної алгебри, наприклад систем лінійних рівнянь, необхідно розробити ефективні алгоритми для базових операцій додавання векторів (рис. 3) та множення вектора на скаляр.

Нехай задано два вектори A та B розміром N:

$$A=\{ a_0 , a_1 , \dots , a_{n-1} \},\ B=\{ b_0 , b_1 , \dots , b_{n-1} \}.$$

Операція додавання векторів (C = A + B) у скалярному вигляді (SISD) описується формулою:

$$c_i=a_i+ b_i,\ \text{де } i=0, \dots, N\text{-}1. \quad (1)$$

При використанні розширення AVX з числами одинарної точності (float, 32 біти) у 256-бітовий регістр YMM поміщається 8 елементів. Це дозволяє змінити логіку циклу (1). Індекс i збільшується не на 1, а на 8 за кожний крок.

Тоді алгоритм векторного додавання буде наступним:

1. Завантажити 8 елементів масиву A у регістр `ymm1`.
2. Завантажити 8 елементів масиву B у регістр `ymm2`.
3. Виконати інструкцію векторного додавання `vaddps ymm0, ymm1, ymm2`.
4. Зберегти результат з `ymm0` у пам'ять масиву C.
5. Повторювати для всіх блоків по 8 чисел.

Якщо N не кратна 8, виникає проблема "залишків" масиву. Залишок елементів (від 1 до 7) необхідно обробляти класичним скалярним методом або використовувати маскування.

На рис. 3 показано схему векторного додавання. Вектори A і B розміщені у пам'яті і містять по вісім 32-бітових значень з плаваючою крапкою. Вектори A і B завантажуються у регістри YMM 1 і YMM2 та додаються командою `vaddps ymm0,ymm1,ymm2`. Результат з регістра YMM 0 записується у пам'ять.

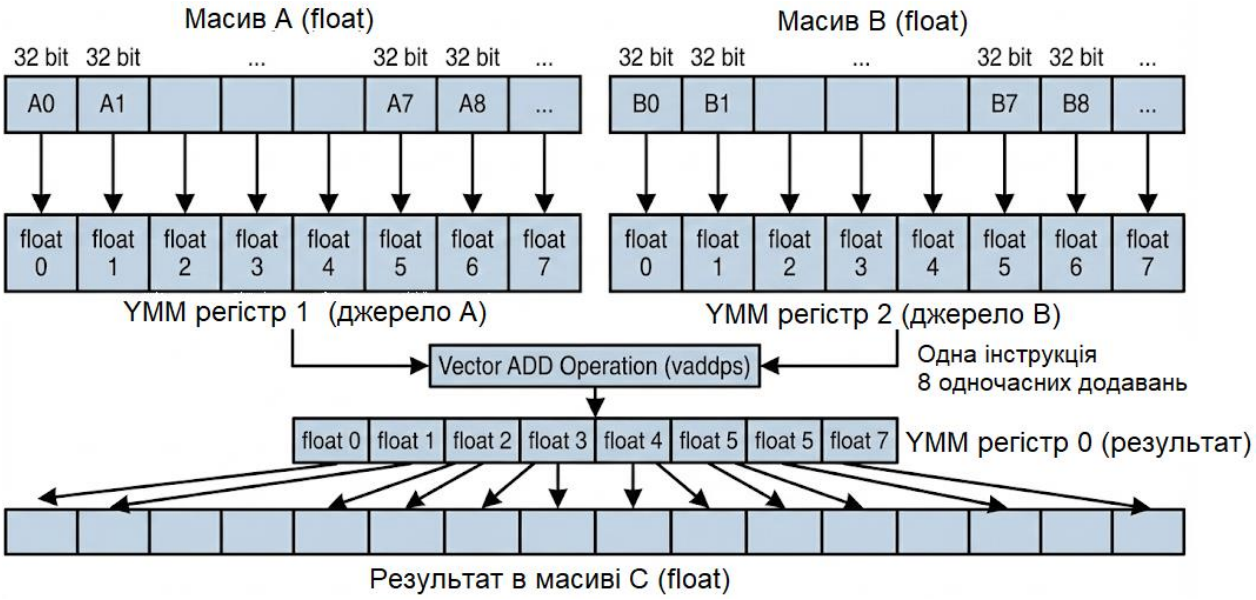

Рис. 3. Схема векторного додавання

2.2 Алгоритмічні особливості методу Жордана-Гауса

Однією з поширених задач лінійної алгебри є розв'язування систем лінійних алгебраїчних рівнянь (СЛАР). Метод Жордана-Гауса – це ефективний прямий алгоритм розв'язування СЛАР, який базується на зведенні розширеної матриці коефіцієнтів до діагонального (одиничного) вигляду за допомогою елементарних перетворень рядків [13-14]. Це вдосконалена версія методу Гаусса, яка відразу дає значення змінних без зворотної підстановки.

Нехай задано СЛАР у матричному вигляді $Ax = B$, де A – матриця коефіцієнтів $N \times N$, B – вектор вільних членів.

Основні етапи методу Жордана-Гауса:

1. Складання розширеної матриці $N\times(N+1)$, шляхом доповнення стовпцем вільних членів стовпці матриці коефіцієнтів.

2. На кожному кроці i вибирається ненульовий елемент a_{ii} (бажано максимальний за модулем), який стає ведучим.

3. Рядок, який містить ведучий елемент, ділиться на цей елемент, щоб зробити ведучий елемент рівним 1.

Алгоритм складається з N ітерацій. На кожному кроці k (де k – номер ведучого рядка) виконуються дві дії:

1.Нормалізація ведучого рядка. Усі елементи k-го рядка діляться на діагональний елемент a_{kk}:

$$a_{kj} \leftarrow \frac{a_{kj}}{a_{kk}}, j = k, \dots, N.$$

2. Обнулення змінних стовпця. Від усіх інших рядків i (де i != k) віднімається ведучий рядок, помножений на відповідний коефіцієнт a_{ik}:

$$a_{ij} \leftarrow a_{ij} - a_{ik} \cdot a_{kj}.$$

Саме друга дія є найбільш ресурсоємною, оскільки вона виконується для всіх рядків матриці. Однак вона ідеально підходить для векторизації. Коефіцієнт a_{ik} є сталим для всього i-го рядка під час однієї операції віднімання.

Послідовність оптимізації обчислень з використання векторних інструкцій AVX:

Для опрацювання i-го рядка:

1. Завантажити елементи поточного i-го рядка (a_{ij}) у вісім комірок регістра YMM.

2. Завантажити коефіцієнт a_{ik} (множник) у вісім комірок іншого регістра YMM інструкцією тиражування Broadcast (рис. 4).

3. Завантажити елементи ведучого рядка (a_{kj}) у вісім комірок іншого регістра YMM.

4. Виконати операція FMA (Fused Multiply-Add) для об’єднаного множення і додавання ($a\cdot b + c$ за один крок) одночасно для всіх 8 елементів поточного рядка.

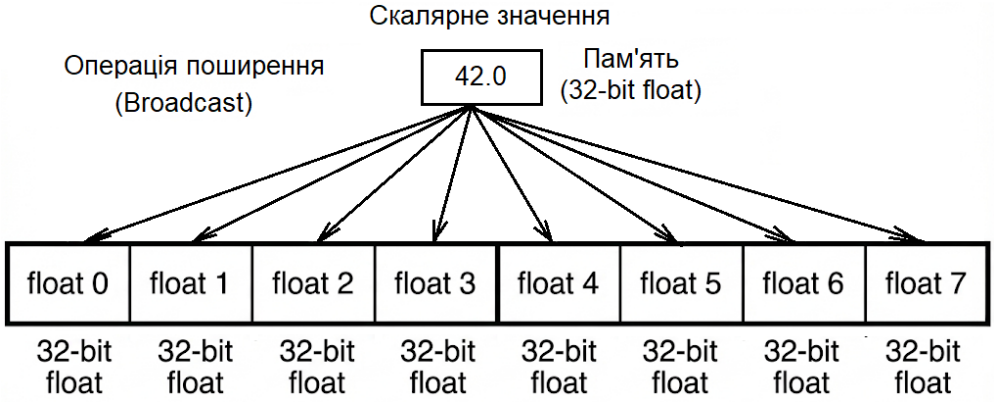

Рис. 4. Принцип роботи команди тиражування (Broadcast)

2.3 Бібліотека на асемблері з використанням інструкцій AVX/AVX2

Для реалізації алгоритму Жордана-Гауса була створена бібліотеки на асемблері NASM з наступними інструкціями процесора. Усі інструкції мають префікс *v*, що дозволяє використовувати три операнди.

1. Інструкції пересилання даних:

• `vmovups ymm1, [mem]` (Move Unaligned Packed Single) – завантаження 256 біт (8 float) з пам'яті в регістр. Використовується версія ups (unaligned), оскільки вона безпечніша, якщо дані не вирівняні на межі 32 байт.

• `vmovups [mem], ymm1` – збереження даних з регістра в пам'ять.

• `vbroadcastss ymm1, [mem]` (Broadcast Scalar Single) – ключова команда для методу Гауса. Вона бере одне 32-бітове число з пам'яті та заповнює ним увесь 256-бітовий регістр.

2. Арифметичні команди:

• `vaddps ymm1, ymm2, ymm3` – додавання запакованих чисел (`ymm1 = ymm2 + ymm3`).

• `vmulps ymm1, ymm2, ymm3` – множення запакованих чисел.

• `vsubps ymm1, ymm2, ymm3` – віднімання запакованих чисел.

• `vdivps ymm1, ymm2, ymm3` – ділення запакованих чисел (використовується при нормалізації рядка).

3. Команди FMA (Fused Multiply-Add) – для AVX2:

• `vfmsub213ps ymm1, ymm2, ymm3` – дія команди `ymm1 = (ymm2 * ymm1) - ymm3`.

Використання FMA дозволяє підвищити точність, оскільки проміжний результат множення не округляється, а зберігається з повною точністю до моменту віднімання.

2.4 Структура даних і алгоритми з використанням векторних операцій

Матриці в пам'яті комп'ютера подані як одновимірний масив, де елементи зберігаються рядками. Доступ до елемента $A[i][j]$ здійснюється за індексом $i\times(N+1)$, де $N+1$ – ширина розширеної матриці.

Алгоритм векторного множення:

1. Початок.
2. Перевірка, чи кількість елементів $N \geq 8$.
3. Якщо так:
   - завантажити 8 чисел з масиву A в YMM0;
   - завантажити 8 чисел з масиву B в YMM1;
   - виконати команду `vmulps ymm0, ymm0, ymm1`;
   - зберегти YMM0;
   - збільшити покажчик на 8 елементів (32 байти);
   - повторити цикл.
4. Якщо ні (або є залишок):
   - обробити елементи по одному класичними командами SSE (скалярно).
5. Кінець.

Блок схема методу Жордана-Гауса показана на рис. 5.

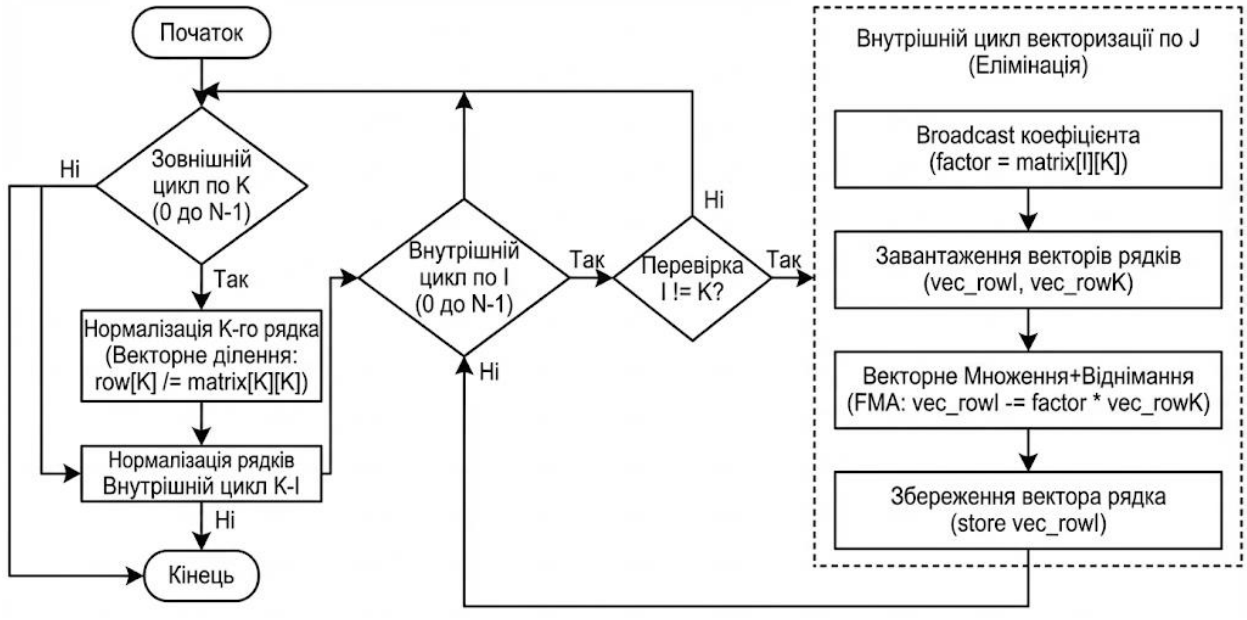

Рис. 5. Блок-схема методу Жордана-Гауса

Алгоритм методу Жордана-Гауса:

1. Зовнішній цикл по *K* (від 0 до *N*-1).
2. Нормалізація *K*-го рядка (векторне ділення).
3. Внутрішній цикл по *I* (від 0 до *N*-1).
4. Перевірка *I* != *K*.
5. Внутрішній цикл векторизації по *J*: Тиражування (Broadcast) коефіцієнта → Завантаження рядка → Множення + Віднімання → Збереження.

3 Реалізація програм та дослідження ефективності бібліотеки

3.1 Структура програм та організація даних

Прийнято векторизовані функції асемблера об'єднувати у бібліотеки, що дозволяє їх використання у мовах програмування високого рівня [15, 16]. Розроблені функції з використанням векторизованих інструкцій AVX об'єднано у бібліотеку на мові асемблера NASM для 64-бітової архітектури Windows (Win64 домовленості викликів). Програмний код структуровано на кілька логічних модулів:

1. Секція даних (`.data` та `.bss`): Відповідає за зберігання глобальних змінних та констант. Ключовим моментом є використання директиви вирівнювання `align 32`. Це критично необхідно для інструкцій AVX, оскільки доступ до невирівняної пам'яті може спричинити виняткову ситуацію (crash) або суттєве падіння продуктивності.
2. Секція коду (`.text`): Містить процедури `VectorAdd`, `VectorMul` та `GaussJordanSolve`.
3. Макроси: Використано макроси препроцесора NASM для стандартизації прологу та епілогу функцій (збереження регістрів `rbp`, `rsp`, `rsi`, `rdi`), що спрощує читальність коду.

Для взаємодії з операційною системою та виведення результатів використано функції Windows API (`WriteConsoleA`, `ExitProcess`) та функції стандартної бібліотеки C (`printf`), підключені через компонувальник.

3.2 Реалізація базових векторних операцій

Нижче показано фрагмент розробленої процедури додавання векторів. Основний цикл обробляє по 8 елементів типу float за ітерацію.

Роздрук 1 – Реалізація векторного додавання (фрагмент)

```
; Вхідні параметри (Win64 convention):
; rcx - покажчик на масив A
; rdx - покажчик на масив B
; r8  - покажчик на масив Result
; r9  - кількість елементів N

global VectorAdd_AVX
VectorAdd_AVX:
    push rbp
    mov rbp, rsp

    ; Основний цикл (обробка по 8 чисел)
    xor rax, rax            ; Обнулення лічильника
.loop_block:
    cmp rax, r9             ; Перевірка кінця масиву
    jge .cleanup            ; Якщо пройшли весь масив - вихід

    ; Завантаження даних (використовується unaligned для безпеки)
    vmovups ymm0, [rcx + rax*4] ; Завантаження 8 float з A
    vmovups ymm1, [rdx + rax*4] ; Завантаження 8 float з B

    ; Векторне додавання
    vaddps ymm0, ymm0, ymm1     ; YMM0 = A + B

    ; Збереження результату
```

```
    vmovups [r8 + rax*4], ymm0

    add rax, 8              ; Крок циклу +8 елементів
    jmp .loop_block

.cleanup:
    ; Тут розміщується код обробки "залишку" масиву (якщо N не кратне 8)
    ; ... (реалізовано через скалярні команди xmm - addss)

    mov rsp, rbp
    pop rbp
    ret
```

3.3 Реалізація ядра методу Жордана-Гауса

Найбільш складним є векторизація внутрішнього циклу, де відбувається віднімання рядків. Для оптимізації використано команду `vbroadcastss`, яка дублює поточний коефіцієнт $K = a_{ik}$ на весь регістр. Це дозволяє уникнути циклічного звернення до пам'яті за одним і тим самим значенням.

Роздрук 2 – Векторизація віднімання рядків

```
; YMM2 містить тиражований коефіцієнт (multiplier)
    ; RDI вказує на поточний рядок (Target Row)
    ; RSI вказує на ведучий рядок  (Pivot Row)

.inner_loop_simd:
    ; Завантаження 8-ми елементів ведучого рядка
    vmovups ymm0, [rsi + rbx*4]

    ; Завантаження 8 елементів рядка, що змінюється
    vmovups ymm1, [rdi + rbx*4]

    ; Обчислення: Row[j] = Row[j] - Multiplier * Pivot[j]
    vmulps ymm0, ymm0, ymm2     ; YMM0 = Pivot * Multiplier
    vsubps ymm1, ymm1, ymm0     ; YMM1 = Target - (Pivot * Mult)

    ; Збереження результату назад у пам'ять
    vmovups [rdi + rbx*4], ymm1

    add rbx, 8                  ; Наступні 8 стовпців
    cmp rbx, r10                ; Перевірка на кінець рядка
    jl .inner_loop_simd
```

3.4 Методика тестування та характеристики тестового стенду

Для оцінки ефективності розроблених алгоритмів проведено серію експериментів. Вимірювання часу виконання здійснювалося за допомогою процесорної інструкції `rdtsc` (Read Time-Stamp Counter), яка повертає кількість тактів процесора з моменту скидання. Це забезпечує найвищу можливу точність вимірювань, нівелюючи похибки системного таймера ОС.

Конфігурація апаратного та програмного забезпечення, на якому проводилося тестування, показана в табл. 1.

3.5 Аналіз результатів експериментів

Було проведено порівняльне тестування двох версій алгоритму розв'язування СЛАР:

1. SISD (Scalar): класична реалізація з використанням одного регістра FPU/XMM для одного числа.

2. SIMD (AVX): оптимізована реалізація з використанням 256-бітових регістрів YMM (8 чисел одночасно).

Тестування проводилося на випадково згенерованих квадратних матрицях розмірністю від 256×256 до 2048×2048. Для кожної розмірності тест запускався 10 разів, після чого

обчислювалося середнє арифметичне значення часу виконання (у мільйонах тактів процесора).

Таблиця 1

Параметр	Значення
Центральний процесор (CPU)	Intel Core i5-11400H @ 2.70GHz
Архітектура	x86-64 (Rocket Lake)
Підтримка інструкцій	MMX, SSE4.2, AVX, AVX2, FMA3
Оперативна пам'ять (RAM)	16 GB DDR4-3200
Операційна система	Windows 11 Home (64-bit)
Середовище розробки	VS Code + NASM 2.15
Режим компіляції	Release (без налагоджувальної інформації)

Результати вимірювань (у тактах процесора) показано в табл. 2.

Таблиця 2

Розмірність матриці (N)	Час виконання SISD (10^6 тактів)	Час виконання AVX, (10^6 тактів)	Прискорення, разів
256	45.2	8.8	5.14
512	385.6	68.9	5.60
1024	3120.5	510.4	6.11
2048	25890.1	4050.8	6.39

Як видно з табл. 2, використання векторних інструкцій забезпечує суттєве прискорення обчислень. Теоретичний максимум прискорення для AVX (обробка 8 чисел float) становить 8 разів. На практиці отримано коефіцієнт у межах 5.1 – 6.4 разів.

Відхилення від теоретичного максимуму показано на рис. 6 і пояснюється такими факторами:

1. Пропускна здатність пам'яті: процесор обчислює дані швидше, ніж встигає отримувати їх з оперативної пам'яті.

2. Накладні витрати: частина часу витрачається на підготовку циклів, завантаження коефіцієнтів та обробку "залишків" масивів, які виконуються скалярно.

3. Кеш-промахи: при великих розмірностях матриць (2048 і більше) дані перестають вміщатися у L2/L3 кеш процесора, що збільшує затримки доступу до даних.

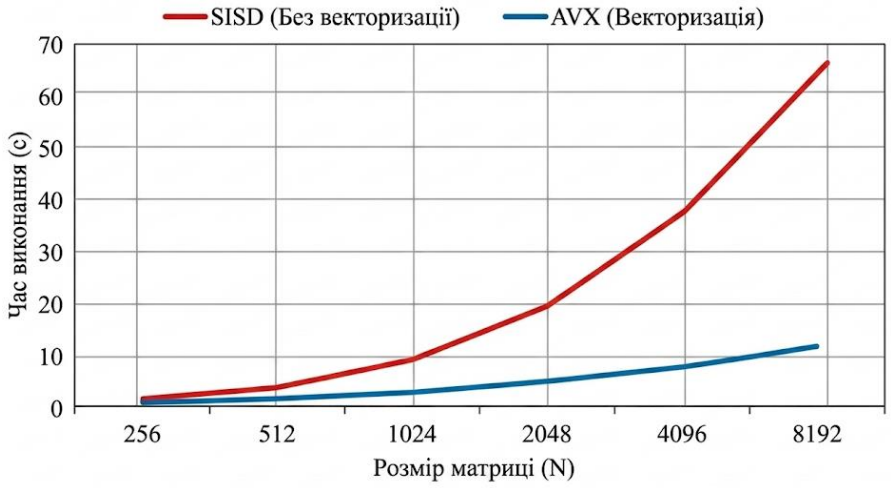

Рис. 6. Залежності часу виконання від розмірів матриць

Висновки

У статті розглянуто актуальну науково-практичну задачу розв'язування систем лінійних алгебраїчних рівнянь з використанням мови асемблера і векторних інструкцій AVX. Показано, що скалярна модель обчислень SISD вичерпала можливості екстенсивного росту продуктивності. Перехід до векторної моделі обчислень SIMD є ключовим фактором для прискорення обробки великих масивів даних. Аналіз системи інструкцій AVX показав, що використання триоперандного синтаксису дозволяє прискорити виконання арифметичних операцій і оптимізувати використання регістрового файлу.

Розроблено алгоритми для базових векторних операцій (додавання, множення) та векторизованого методу Жордана-Гауса. Створено бібліотеку функцій на мові асемблера NASM з використанням інструкцій AVX. Реалізовано метод Жордана-Гауса з використанням розроблених бібліотечних функцій.

Проведено експериментальне дослідження на тестових матрицях розмірністю від 256×256 до 2048×2048. Отримано прискорення обчислень від 5 до 6 разів порівняно з класичною реалізацією. Отримані результати показують доцільність використання асемблерних функцій з векторними інструкціями AVX/AVX2 у критичних до швидкодії ділянках коду.

Список літератури:

1. C. Xie, H. Wu, and J. Zhou, “Vectorization Programming Based on HR DSP Using SIMD,” *Electronics*, vol. 12, no. 13, p. 2922, Jul. 2023. doi: https://doi.org/10.3390/electronics12132922.
2. Intel® Intrinsics Guide: Instruction set. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html?utm_source=chatgpt.com#.
3. Kusswurm D. Modern Parallel Programming with C++ and Assembly Language: X86 SIMD Development Using AVX, AVX2, and AVX-512 / D. Kusswurm. New York : Apress, 2022. 580 p. URL: https://www.oreilly.com/library/view/modern-parallel-programming/9781484279182/.
4. M. Costanzo, E. Rucci, M. Naiouf, and A. D. Giusti, “Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body,” in *2021 XLVII Latin Amer. Comput. Conf. (CLEI)*, Cartago, Costa Rica, Oct. 25–29, 2021. IEEE, 2021. doi: https://doi.org/10.1109/clei53233.2021.9640225.
5. P. E. de Vilhena, O. Lahav, V. Vafeiadis, and A. Raad, “Extending the C/C++ Memory Model with Inline Assembly,” *Proc. ACM Program. Lang.*, vol. 8, OOPSLA2, pp. 1081–1107, Oct. 2024. doi: https://doi.org/10.1145/3689749.
6. M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck, “An Analysis of x86-64 Inline Assembly in C Programs,” in *VEE '18: 14th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, Williamsburg VA USA. New York, NY, USA: ACM, 2018. doi: https://doi.org/10.1145/3186411.3186418.
7. K. Papagiannopoulos, “Low Randomness Masking and Shuffling: An Evaluation Using Mutual Information,” *IACR Trans. Cryptographic Hardware Embedded Syst.*, pp. 524–546, Aug. 2018. doi: https://doi.org/10.46586/tches.v2018.i3.524-546.
8. D. Salomon and I. Levi, “MaskSIMD-lib: on the performance gap of a generic C optimized assembly and wide vector extensions for masked software with an Ascon-p test case,” *J. Cryptographic Eng.*, May 2023. doi: https://doi.org/10.1007/s13389-023-00322-4.
9. NASM - The Netwide Assembler. URL: https://www.nasm.us.
10. “Microsoft Macro Assembler reference.” Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/uk-ua/cpp/assembler/masm/microsoft-macro-assembler-reference?view=msvc-170.
11. GAS - the GNU assembler. URL: https://www.gnu.org/software/binutils/.
12. B. Adeleye and S. M. Jiddah, “Analysis of Parallel Architectures: SIMD, tightly-coupled MIMD, and loosely-coupled MIMD,” *Int. J. Comput. Trends Technol.*, vol. 53, no. 1, pp. 6–8, Nov. 2017. doi: https://doi.org/10.14445/22312803/ijctt-v53p102.
13. L. Bobrowski and C. Boldak, “Stepwise Inversion of Large Matrices with the Gauss-Jordan Vector Transformation,” *J. Advances Math. Comput. Sci.*, pp. 28–39, Jan. 2022. doi: https://doi.org/10.9734/jamcs/2022/v37i130429.
14. P. S. Stanimirović and M. D. Petković, “Gauss–Jordan elimination method for computing outer inverses,” *Appl. Math. Computation*, vol. 219, no. 9, pp. 4667–4679, Jan. 2013. doi: https://doi.org/10.1016/j.amc.2012.10.081.
15. K. Goto and R. A. v. d. Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 1–25, May 2008. doi: https://doi.org/10.1145/1356052.1356053

16.T. B. Schardl, W. S. Moses, and C. E. Leiserson, “Tapir,” in *PPoPP '17: 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Austin Texas USA. New York, NY, USA: ACM, 2017. doi: https://doi.org/10.1145/3018743.3018758.

Надійшла до редколегії 10.03.2025

Відомості про авторів:

Голота Віктор Іванович - кандидат технічних наук, доцент кафедри комп’ютерної інженерії та електроніки, Прикарпатський національний університети імені Василя Стефаника / Vasyl Stefanyk Precarpathian National University, Україна; email: viktor.holota@pnu.edu.ua, ORCID: https://orcid.org/0000-0003-4605-7507.

Грига Володимир Михайлович – канд. техн. наук, доцент, доцент кафедри комп’ютерної інженерії та електроніки, Прикарпатський національний університети імені Василя Стефаника / Vasyl Stefanyk Precarpathian National University, Україна; email: volodymyr.hryha@pnu.edu.ua; ORCID: https://orcid.org/0000-0001-5458-525X

Бенько Тарас Григорович – асистент кафедри комп’ютерної інженерії та електроніки, Прикарпатський національний університет імені Василя Стефаника, Україна / Vasyl Stefanyk Precarpathian National University, Україна; email: taras.benko@pnu.edu.ua; ORCID: https://orcid.org/0000-0002-6310-8743

Моргун Анатолій Володимирович – аспірант кафедри комп’ютерної інженерії та електроніки, Прикарпатський національний університети імені Василя Стефаника / Vasyl Stefanyk Precarpathian National University, Україна; email: morgun.anatoliy@gmail.com; ORCID: https://orcid.org/0009-0002-4765-2885