УДК 004.051
DOI https://doi.org/10.32782/2663-5941/2025.2.2/24

Павловський О. М.
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Платов І. М.
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

ДОСЛІДЖЕННЯ КОДОГЕНЕРАЦІЇ C/C++ КОМПІЛЯТОРА ДЛЯ ВИСОКОПРОДУКТИВНИХ ОБЧИСЛЕНЬ В МІКРОПРОЦЕСОРНИХ СИСТЕМАХ

Стаття присвячена дослідженню кодогенерації C/C++ компілятора високопродуктивних обчислень. У роботі розглядається проблема оптимізації програмного коду підвищення ефективності обчислень в умовах зростаючих вимог до обробки великих обсягів даних. Актуальність теми обумовлена тим, що, попри розвиток обчислювальної техніки, оптимізація коду залишається важливим інструментом досягнення високої продуктивності.

У статті проводиться аналіз можливостей компілятора GCC версії 4.9.2 64-bit на персональному комп'ютері з операційною системою Windows 10 та процесором AMD Ryzen 5600x. Дослідження включає аналіз асемблерних лістингів, згенерованих компілятором, для оцінки якості кодогенерації і оптимізації. Розглядаються різні підходи до оптимізації, включаючи використання параметрів компіляції, що задають різні рівні оптимізації, а також оптимізацію з використанням intrinsic-функцій.

В якості тестової програми використовується проста симуляція руху 1000 тіл під дією сили тяжіння мовою програмування C. Цей приклад дозволяє проаналізувати базовий набір команд, які генерує оптимізуючий компілятор, для оцінки оптимізації коду.

Аналіз асемблерних лістингів демонструє, що компілятор без використання оптимізації генерує код, в якому не використовуються векторні інструкції для обробки даних, а також є надмірне обчислення адрес елементів масиву. Застосування параметрів оптимізації за швидкістю (--Ofast) призводить до використання векторних інструкцій та розгортання циклів, що підвищує продуктивність. Однак, при цьому спостерігається збільшення кількості інструкцій та наявність ділянок коду, що обробляють скалярні дані, що може призвести до неефективного використання кеш-пам'яті.

Для подальшої оптимізації використовуються intrinsic-функції, що дозволяють використовувати розширені асемблерні інструкції (AVX, FMA) та примусове вирівнювання даних. Аналіз асемблерного лістингу модифікованої програми показує значне зменшення кількості інструкцій та ефективнішу роботу з даними.

Зроблено висновок про те, що незважаючи на наявність потужних алгоритмів оптимізації в сучасних компіляторах, ручна оптимізація з використанням intrinsic-функцій може бути більш ефективною. Це наголошує на важливості компетенції програміста в області низькорівневої оптимізації для досягнення максимальної ефективності.

Ключові слова: *компілятор, оптимізація, AVX, FMA, програмування, GCC, асемблер.*

Постановка проблеми. Останнє десятиліття виділяється стрімким зростанням попиту на обробку великої кількості даних. Це пов'язано з мініатюризацією потужної обчислювальної техніки та її інтеграцією в усі галузі людського життя. Поява та розповсюдження рішень на базі штучного інтелекту сприяли експоненційному росту оброблюваної інформації, що в свою чергу викликало потреби в розробках ефективних алгоритмів обробки даних [1, 2].

Наочним прикладом таких рішень можуть бути автоматизовані робототехнічні комплекси, які оснащені великою кількістю різноманітних датчиків, наприклад, датчики дотику, камери, лідари, акустичні сенсори тощо [3].

При розробці таких комплексів користуються комп'ютерним моделюванням [4, 5]. Для аналізу роботи системи в умовах, наближених до реального середовища, використовують складні фізичні моделі, в яких враховується велика кількість різ-

номанітних факторів. Відповідно, зі збільшенням складності систем зростає навантаження на обчислювальні ядра.

Переважна більшість програмного забезпечення для моделювання роботизованих та робототехнічних систем розроблена для операційних систем, які працюють на 32 або 64 розрядних процесорах Intel або AMD. Також доступні рішення для процесорів ARM, однак в даній роботі вони розглядатися не будуть. За останні роки 32 бітна архітектура була витіснена з ринку на користь 64 розрядної. Як було зазначено вище, це пов'язано із зростанням об'єму оброблюваних даних, а як відомо, за один такт роботи процесора 64 розрядна архітектура дозволяє обробити більшу їх кількість [6].

Однак навіть перехід на 64 розрядну архітектуру не вирішив проблему високопродуктивних обчислень. Постійне масштабування та ускладнення систем відбувається швидше, ніж з'являються нові високопродуктивні обчислювачі. Тому важливою та актуальною задачею була і залишається оптимізація програмного коду, оскільки, залежно від навичок програміста, вона дозволяє підвищити ефективність обчислень без заміни обчислювального ядра. Умовно можна розділити оптимізацію на два окремі напрямки: вибір алгоритмів обробки даних та низькорівневу оптимізацію [7]. В першому випадку проблема може бути вирішена розробкою або застосуванням більш ефективних та швидких алгоритмів обробки даних. В другому випадку аналізують згенерований компілятором код на наявність так званого «bottleneck» – це код, який виконується найменш ефективно (повільно), порівняно з усією іншою частиною [8]. Причому це може бути викликано як фізичними обмеженнями архітектури (наприклад, запис/зчитування з диску відбувається повільніше, ніж ті ж операції з оперативною пам'яттю), так і через особливості кодогенератора певного компілятора. Останній фактор пов'язаний зі складністю архітектури, а також з алгоритмами оптимізації, які транслюють код програми високого рівня в машинний код.

Тому важливу роль в ефективності високопродуктивних обчислень грає не лише компетенція програміста, а й інструмент, котрий надає всі можливості для розробки якісних програмних продуктів.

Аналіз останніх досліджень і публікацій. З даної тематики існує декілька досліджень, які розглядають різні аспекти кодогенерації компіляторів та оптимізації програмного коду.

У статті [9] порівнювалися компілятори Microsoft Visual Studio 2017, Embarcadero C++ Builder 10.2 та Open Watcom 1.9 щодо якості кодогенерації для архітектури Intel x86 та підтримки SSE інструкцій. Згідно з результатами, Microsoft Visual Studio 2017 є найкращим вибором для Intel x86, оскільки підтримує SSE інструкції (автоматично та через асемблерні вставки) та обчислює вирази на етапі компіляції. Для багатоплатформної розробки Open Watcom є прийнятним варіантом (за умови відсутності системозалежного коду), оскільки дозволяє створювати виконувані файли для Linux і підтримує SSE через асемблерні вставки, хоча й не автоматично. Компілятор Embarcadero C++ Builder 10.2 не рекомендується для ефективної обробки даних через відсутність автоматичної підтримки SSE та неможливість використання SSE в асемблерних вставках (зокрема через відсутність вирівнювання даних).

При порівнянні компіляторів у роботі [10] автори дослідили, що різні компілятори C++ (GCC, Clang, MSVC) мають різну продуктивність залежно від архітектури процесора та оптимізацій. GCC і Clang показали найкращі результати швидкості виконання з певними прапорами оптимізації. Вибір компілятора та параметрів оптимізації значно впливає на підсумкову продуктивність програм.

У дослідженні [11] були розглянуті компілятори GCC і ICC, а також їхній вплив на продуктивність програм на C++. Аналізувалися різні рівні оптимізації: «-O1», «-O2» і «-O3». Окрім цього, робота охоплювала багатопотокові API, зокрема Pthreads, C++11, OpenMP, Cilk Plus і TBB, з метою визначення найбільш ефективних рішень для паралельного виконання. Порівняння продуктивності показало, що ICC випереджає GCC за швидкістю виконання майже у всіх тестах. Найбільша різниця спостерігалася на процесорі Intel Xeon (Medusa), де ICC демонстрував значно кращі результати. Водночас на AMD Opteron (Hydra) відмінність між GCC та ICC була менш вираженою, що свідчить про меншу ступінь оптимізації ICC під архітектуру AMD. Оптимізація «-O2» виявилася найбільш ефективною для більшості тестів, забезпечуючи оптимальний баланс між швидкістю виконання та розміром коду. Водночас «-O3» не завжди покращувала продуктивність, особливо у випадку GCC. Однак для ICC на Intel Xeon («Medusa») використання «-O3» дало помітне покращення продуктивності, тоді як на AMD («Hydra») різниця між «-O2» і «-O3» була мінімальною. Для перевірки значущості від-

мінностей використовувався t-тест, який показав, що різниця між «-O2» і «-O3» у більшості випадків є статистично незначною. При цьому, на Intel Xeon («Medusa») GCC показав найбільшу різницю у результатах при використанні «-O1», «-O2» та «-O3». Загалом, вибір API впливає на продуктивність більше, ніж рівень оптимізації, що підтверджує важливість правильного підбору багатопотокових технологій.

Аналіз літератури демонструє, що оптимізація коду – це багатогранний процес, який залежить від апаратної архітектури та структури коду. Універсальних рішень не існує – ефективність оптимізації змінюється залежно від платформи та типу обчислень. Крім того, не кожне покращення продуктивності є статистично значущим, тому важливо ретельно аналізувати результати. Оптимізація завжди передбачає пошук балансу між швидкістю, ефективністю використання ресурсів і стабільністю програми.

Постановка завдання. Для 64 бітної архітектури існує велика кількість різноманітних C/C++ компіляторів. Кожен з яких має свої переваги та недоліки. Вибір того чи іншого компілятора залежить перш за все від операційної системи, під яку розроблюється програмне забезпечення, а також від поставлених задач. Найбільш популярним компілятором є GCC, який доступний на багатьох операційних системах (Linux, macOS, Windows, *BSD, Solaris та ін.) та підтримує велику кількість архітектур (x86, x86-64, ARM, PowerPC, MIPS, RISC-V та ін.) [12].

Постає питання якості кодогенерації даного компілятора для високопродуктивних обчислень. Для цього, зазвичай досліджуються асемблерні лістинги, генеровані компілятором, на наявність нераціонального використання машинних інструкцій та наявність «bottleneck».

Отже метою даної роботи є дослідження асемблерних лістингів компілятора GCC для оцінки якості кодогенерації та оптимізації на прикладі вирішення задачі симуляції руху об’єктів.

Виклад основного матеріалу. Для дослідження використовувався компілятор GCC версії 4.9.2 64-bit та операційна система Windows 10 та комп’ютер з процесором AMD Ryzen 5600x з підтримкою інструкцій MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, x86-64, AES, AVX, AVX2, FMA3, SHA [13]. Для оптимізації машинного коду застосуємо стандартний підхід – параметри компіляції, які задають різні рівні оптимізації, а також оптимізацію із використанням intrinsic-функцій [14]. Ці функції дозволяють легко використовувати розширені асемблерні інструкції (SSE, SSE2, SSE3, SSSE3, SSE4, AVX, AVX2, AVX512, FMA), і на відміну від стандартних функцій, транслюються в одну або декілька машинних кодів.

Варто зауважити, що порівняно з безпосереднім використанням асемблерного коду в програмі на мовах C/C++, яке досліджувалось в роботі [9], intrinsic-функції абстрагують деякі операції, що сприяє легшій переносимості програми між різними платформами, а також пришвидшують розробку в цілому. З іншого боку таке рішення обмежує програміста від безпосереднього контролю над операціями, оскільки intrinsic-функції оброблюються компілятором в автоматичному режимі.

В якості тестової програми була реалізована проста симуляція руху 1000 тіл під дією сили тяжіння мовою програмування C. Такий приклад є досить простим для аналізу, але водночас реалізує типову задачу високопродуктивних обчислень, оскільки містить базовий набір команд, що необхідний для аналізу оптимізації коду компілятором. Код програми наведений на рис. 1.

```
#include <math.h>

#define N 1000
#define STEPS 1000
#define DT 0.01

void simulate(double *x, double *v, double *a) {
        int step, i;
    for (step = 0; step < STEPS; step++) {
        for (i = 0; i < N; i++) {
            v[i] += a[i] * DT;
            x[i] += v[i] * DT;
        }
    }
}

int main() {
    double x[N], v[N], a[N];
    int i;
    for (i = 0; i < N; i++) {
        x[i] = 0.0;
        v[i] = 1.0;
        a[i] = -9.81;
    }

    simulate(x, v, a);

    return 0;
}
```

Рис. 1. Код програми симуляції падіння 1000 тіл під дією сили тяжіння

Для початку проаналізуємо кодогенерацію компілятора без використання оптимізацій. Компіляцію програми будемо здійснювати, використовуючи командну строку з відповідними параметрами, а саме:./gcc -march=native -S -masm=intel -o «1000_obj_model_optimiz.s» «1000_obj_model_optimiz.c», де параметр -march=native задає використання усіх можливих інструкцій конкретного процесора. -S вказує компілятору генерувати асемблерний лістинг. -masm=intel асемблерний

лістниг у діалекті Intel. Для стислості, у статті будуть наведені лише ключові фрагменти лістингів. Результат компіляції показаний на рис. 2.

```
simulate:
        push    rbp
        .seh_pushreg    rbp
        mov     rbp, rsp
        .seh_setframe   rbp, 0
        sub     rsp, 16
        .seh_stackalloc 16
        .seh_endprologue
        mov     QWORD PTR 16[rbp], rcx
        mov     QWORD PTR 24[rbp], rdx
        mov     QWORD PTR 32[rbp], r8
        mov     DWORD PTR -4[rbp], 0
        jmp     .L2
.L5:
        mov     DWORD PTR -8[rbp], 0
        jmp     .L3
.L4:
        mov     eax, DWORD PTR -8[rbp]
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR 24[rbp]
        add     rax, rdx
        mov     edx, DWORD PTR -8[rbp]
        movsx   rdx, edx
        lea     rcx, 0[0+rdx*8]
        mov     rdx, QWORD PTR 24[rbp]
        add     rdx, rcx
        vmovsd  xmm1, QWORD PTR [rdx]
        mov     edx, DWORD PTR -8[rbp]
        movsx   rdx, edx
        lea     rcx, 0[0+rdx*8]
        mov     rdx, QWORD PTR 32[rbp]
        add     rdx, rcx
        vmovsd  xmm2, QWORD PTR [rdx]
        vmovsd  xmm0, QWORD PTR .LC0[rip]
        vmulsd  xmm0, xmm2, xmm0
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR [rax], xmm0
        mov     eax, DWORD PTR -8[rbp]
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR 16[rbp]
        add     rax, rdx
        mov     edx, DWORD PTR -8[rbp]
        movsx   rdx, edx
        lea     rcx, 0[0+rdx*8]
        mov     rdx, QWORD PTR 16[rbp]
        add     rdx, rcx
        vmovsd  xmm1, QWORD PTR [rdx]
        mov     edx, DWORD PTR -8[rbp]
        movsx   rdx, edx
        lea     rcx, 0[0+rdx*8]
        mov     rdx, QWORD PTR 24[rbp]
        add     rdx, rcx
        vmovsd  xmm2, QWORD PTR [rdx]
        vmovsd  xmm0, QWORD PTR .LC0[rip]
        vmulsd  xmm0, xmm2, xmm0
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR [rax], xmm0
        inc     DWORD PTR -8[rbp]
.L3:
        cmp     DWORD PTR -8[rbp], 999
        jle     .L4
        inc     DWORD PTR -4[rbp]
.L2:
        cmp     DWORD PTR -4[rbp], 999
        jle     .L5
        add     rsp, 16
        pop     rbp
        ret
        .seh_endproc
```

Рис. 2. Результат компіляції програми без оптимізацій

Аналіз асемблерного лістингу демонструє декілька ключових моментів, на які варто звернути увагу. Перш за все – використання інструкцій «vmovsd», «vmulsd» та «vaddsd». В даному випадку при кожній ітерації циклу оброблюється лише одне значення типу double. Хоча в сучасних процесорах наявні спеціальні інструкції для операцій з векторними інструкціями, так звані SIMD операції, які дозволяють оброблювати одразу декілька значень за одну ітерацію.

Використання інструкції «cdqe» свідчить про приведення типів даних, що збільшує кількість перетворень, тобто збільшує час виконання програми.

Всередині головного циклу для кожної ітерації обрахунок адреси елементу масива виконується декілька разів: зчитування значення індексу зі змінної (інструкція «mov eax, DWORD PTR -8[rbp]»), яка потім приводиться до 64 біт за допомогою інструкції «cdqe». За допомогою інструкції «lea rdx, 0[0+rax*8]» обраховується зміщення, що дорівнює індексу, помноженому на 8 (розмір типу double). Зміщення додається до базової адреси масива («mov rax, QWORD PTR 24[rbp]» та «add rax, rdx»).

Тобто для кожного елементу в циклі адреса обраховується заново, що створює додаткове навантаження, не пов'язане з основними обрахунками. Суттєвий вплив на продуктивність вносить відсутність розгортання циклів. Зовнішній цикл та внутрішній сумарно створюють 10^6 ітерацій.

Тепер дослідимо кодогенерацію із використанням параметрів оптимізації по швидкості. Для цього в GCC є параметр «–Ofast». Повна команда компіляції виглядає наступним чином: cc -march=native -Ofast -S -masm=intel -o « 1000_obj_model_optimiz.s» « 1000_obj_model_optimiz.c». Асемблерний лістинг наведений на рис. 3.

У даному випадку компілятор згенерував код із використанням додаткових інструкцій «vmovups», «vfmadd213pd» та «vfmadd213sd» які призначені для операцій з дробовими числами. Такі інструкції оброблюють одразу декілька елементів за одну ітерацію, що значно скорочує кількість арифметичних операцій.

Застосування інструкцій «prefetcht0» та «prefetchw» зменшує затримки при роботі з пам'яттю, завантажуючи дані в кеш.

Також помітно, що компілятор застосував розгортання циклів, тим самим зменшивши кількість переходів та перевірок умов. Це значно підвищує продуктивність виконання коду на сучасних процесорах.

Але, незважаючи на це, в асемблерному коді спостерігаються недоліки такої оптимізації, а саме: кількість інструкцій значно збільши-

```
simulate:
        push    rbp
        .seh_pushreg    rbp
        mov     rbp, rsp
        .seh_setframe   rbp, 0
        sub     rsp, 16
        mov     DWORD PTR 24[rsp], eax
        lea     eax, [rbx+rbx]
        mov     DWORD PTR 28[rsp], eax
        mov     eax, r15d
        sal     rax, 3
        lea     rsi, [rdi+rax]
        mov     QWORD PTR 32[rsp], rsi
        lea     rsi, [r14+rax]
        add     rax, rbp
        mov     QWORD PTR 40[rsp], rsi
        mov     QWORD PTR 16[rsp], rax
        .p2align 4,,10
.L2:
        cmp     BYTE PTR 15[rsp], 0
        je      .L28
        test    r15d, r15d
        je      .L15
        vmovsd  xmm0, QWORD PTR [r14]
        mov     esi, 1
        vfmadd213sd     xmm0, xmm3, QWORD PTR [rdi]
        vmovsd  QWORD PTR [rdi], xmm0
        vfmadd213sd     xmm0, xmm3, QWORD PTR 0[rbp]
        vmovsd  QWORD PTR 0[rbp], xmm0
.L8:
        mov     r10, QWORD PTR 16[rsp]
        mov     rcx, QWORD PTR 32[rsp]
        xor     edx, edx
        mov     r8, QWORD PTR 40[rsp]
        mov     r11, r10
        mov     r9, rcx
        .p2align 4,,10
.L9:
        vmovups xmm0, XMMWORD PTR [r8]
        prefetcht0      192[rcx]
        prefetcht0      192[r8]
        prefetcht0      192[r10]
        prefetchw       192[r9]
        prefetchw       192[r11]
        add     edx, 4
        add     rcx, 64
        add     r8, 64
        add     r9, 64
        add     r10, 64
        add     r11, 64
        vfmadd213pd     xmm0, xmm2, XMMWORD PTR -64[rcx]
        vmovaps XMMWORD PTR -64[r9], xmm0
        vfmadd213pd     xmm0, xmm2, XMMWORD PTR -64[r10]
        vmovups XMMWORD PTR -64[r11], xmm0
        vmovups xmm0, XMMWORD PTR -48[r8]
        vfmadd213pd     xmm0, xmm2, XMMWORD PTR -48[rcx]
        vmovaps XMMWORD PTR -48[r9], xmm0
        vfmadd213pd     xmm0, xmm2, XMMWORD PTR -48[r10]
        vmovups XMMWORD PTR -48[r11], xmm0
        vmovups xmm0, XMMWORD PTR -32[r8]
        vfmadd213pd     xmm0, xmm2, XMMWORD PTR -32[rcx]
        vmovaps XMMWORD PTR -32[r9], xmm0
        vfmadd213pd     xmm0, xmm2, XMMWORD PTR -32[r10]
        vmovups XMMWORD PTR -32[r11], xmm0
        vmovups xmm0, XMMWORD PTR -16[r8]
        vfmadd213pd     xmm0, xmm2, XMMWORD PTR -16[rcx]
        vmovaps XMMWORD PTR -16[r9], xmm0
        vfmadd213pd     xmm0, xmm2, XMMWORD PTR -16[r10]
        vmovups XMMWORD PTR -16[r11], xmm0
        cmp     edx, 496
        jne     .L9
```

Рис. 3. Асемблерний лістинг, згенерований із використанням оптимізації по швидкості (наведена лише ключова частина функції «simulate»)

лася – це може привести до неефективного використання кеш пам'яті; у лістингу спостерігаються окремі ділянки коду, що обробляють скалярні дані, наприклад, у мітці «.L9» дані оброблюються пакетами, однак, коли залишається невелика кількість елементів, обробка здійснюється скалярно. Інструкція «vmovsd» завантажує один елемент double у xmm-регістр, а «vfmadd213sd» виконує операцію множення та додавання для цього елементу. Така обробка даних може спричиняти додаткові затримки.

Тому далі ми дослідимо роботу компілятора, реалізувавши обчислення за допомогою intrinsic-функцій. Модифікована програма мовою C наведена на рис. 4.

Порівняно з попереднім варіантом, можна помітити зміни як у головній функції main, так і в функції simulate. Для початку розглянемо головну функцію: задавши параметр «__attribute__((aligned(32)))» ми вказуємо компілятору примусово вирівняти дані. Робота з вирівняними даними завжди відбувається швидше, оскільки компілятор генерує код, в якому відсутні додаткові операції читання/запису в пам'ять. Окрім цього вирівнювання сприяє ефективній роботі з кеш пам'яттю – дані потрапляють в кеш блоками і відбувається менше «промахів», коли дані можуть розділятися на декілька блоків, замість одного. Також варто зазначити, що деякі розширені інструкції (наприклад, AVX) потребують обов'язкового вирівнювання даних – це буде розглянуто далі. Стосовно функції simulate, в даному випадку обчислення відбуваються із використанням AVX та FMA інструкцій, а також вони розділені на групи. Цифрою 1 позначені інструкції, котрі обробляють першу групу з чотирьох елементів, 2 – обробка другої групи.

Останні два блока 3 та 4 оновлюють координати для першої та другої групи, відповідно. Також акцентуємо вашу увагу на список параметрів функції simulate, для оптимізації роботи з пам'яттю була використана команда «restrict» (введена в стандарті мови C99), яка сповіщає компілятор про те, що вказівник, що оголошується, адресує область пам'яті, на яку не посилається ніякий інший вказівник. Якщо виразити це простими словами, то компілятор може перевпорядковувати операції читання та запису, використовувати векторні інструкції або виконувати інші перетворення, що базуються на припущенні, що дані не змінюються іншими вказівниками.

Функції «_mm256_load_pd» та «_mm256_store_pd» мінімізують кількість звернень до пам'яті, т. я. працюють з вирівняними даними. Функція «_mm256_fmadd_pd» поєднує множення та додавання в одну операцію, що знижує затримки та підвищує «обчислювальну щільність».

Тепер проаналізуємо асемблерний лістинг, скомпілювавши вищенаведену модифіковану програму наступним чином: gcc -march=native -std=c99 -mavx2 -mfma -S -masm=intel -o «1000_obj_model_optimiz1.s» «1000_obj_model_optimiz1.c». Можна помітити появу нових параметрів компіляції, а саме: «-std=c99» – вказує компілятору, що використовується стандарт мови C99. Це необхідно для використання ключового слова

```
#include <stdio.h>
#include <math.h>
#include <immintrin.h>

#define N 1000
#define STEPS 1000
#define DT 0.01

void simulate(double *restrict x, double *restrict v, const double *restrict a) {
    int step, i;
        __m256d dt_vec = _mm256_set1_pd(DT);
    for (step = 0; step < STEPS; step++) {
        for (i = 0; i < N; i += 8) {
            __m256d v0 = _mm256_load_pd(&v[i]);
            __m256d a0 = _mm256_load_pd(&a[i]);
  1         v0 = _mm256_fmadd_pd(a0, dt_vec, v0); // v0 += a0 * dt_vec
            _mm256_store_pd(&v[i], v0);

            __m256d v1 = _mm256_load_pd(&v[i+4]);
            __m256d a1 = _mm256_load_pd(&a[i+4]);
  2         v1 = _mm256_fmadd_pd(a1, dt_vec, v1); // v1 += a1 * dt_vec
            _mm256_store_pd(&v[i+4], v1);

            __m256d x0 = _mm256_load_pd(&x[i]);
  3         x0 = _mm256_fmadd_pd(v0, dt_vec, x0); // x0 += v0 * dt_vec
            _mm256_store_pd(&x[i], x0);

            __m256d x1 = _mm256_load_pd(&x[i+4]);
  4         x1 = _mm256_fmadd_pd(v1, dt_vec, x1); // x1 += v1 * dt_vec
            _mm256_store_pd(&x[i+4], x1);
        }
    }
}

int main() {
    double x[N] __attribute__((aligned(32)));
    double v[N] __attribute__((aligned(32)));
    double a[N] __attribute__((aligned(32)));
    int i;

    for (i = 0; i < N; i++) {
        x[i] = 0.0;
        v[i] = 1.0;
        a[i] = -9.81;
    }

    simulate(x, v, a);

    return 0;
}
```

Рис. 4. Модифікована програма мовою C із використанням intrinsic-функцій

«restrict». «-mavx2» та «-mfma» вказують компілятору, що в програмі використовуються розширені інструкції (AVX2 та FMA).

Асемблерний лістинг наведено на рис. 5.

```
simulate:
        push    rbx
        .seh_pushreg    rbx
        .seh_endprologue
        vmovaps ymm2, YMMWORD PTR .LC0[rip]
        mov     r9d, 1000
        lea     r11, 416[r8]
        lea     rbx, 416[rdx]
        lea     r10, 416[rcx]
        lea     r8, 8416[rdx]
        .p2align 4,,10
.L2:
        mov     rax, rbx
        mov     rcx, r11
        mov     rdx, r10
        .p2align 4,,10
.L3:
        vmovaps ymm1, YMMWORD PTR -416[rcx]
        vmovaps ymm0, YMMWORD PTR -384[rcx]
        prefetcht0      [rcx]
        prefetcht0      [rax]
        add     rax, 64
        prefetcht0      [rdx]
        add     rcx, 64
        add     rdx, 64
        vfmadd213pd     ymm1, ymm2, YMMWORD PTR -480[rax]
        vfmadd213pd     ymm0, ymm2, YMMWORD PTR -448[rax]
        vmovaps YMMWORD PTR -480[rax], ymm1
        vfmadd213pd     ymm1, ymm2, YMMWORD PTR -480[rdx]
        vmovaps YMMWORD PTR -448[rax], ymm0
        vfmadd213pd     ymm0, ymm2, YMMWORD PTR -448[rdx]
        vmovaps YMMWORD PTR -480[rdx], ymm1
        vmovaps YMMWORD PTR -448[rdx], ymm0
        cmp     rax, r8
        jne     .L3
        dec     r9d
        jne     .L2
        vzeroupper
        pop     rbx
        ret
        .seh_endproc
```

Рис. 5. Асемблерний лістинг модифікованої програми із використанням intrinsic-функцій

Порівнюючи отриманий результат з попередніми лістингами, можна помітити, що кількість інструкцій значно зменшилася. Це говорить про вищезазначену «обчислювальну щільність». В даному випадку робота з даними відбувається більш ефективно, за рахунок завантажень векторів «ymm^0» та «ymm^1», які оновлюються за допомогою команди «vfmadd213pd» (що є прямим наслідком використання «_mm256_fmadd_pd» в C коді). Наявність команди «prefetcht0» свідчить про завантаження даних в кеш, перед їх обробкою – це значно підвищує швидкодію при зверненні до пам'яті. Команди «cmp» + «jne.L3» свідчать про розгортання циклу, тобто зменшена кількість інструкцій для перевірки виходу за межі циклу.

Таким чином, незважаючи на використання тих самих інструкцій AVX/FMA, оптимізована версія забезпечує більш високу продуктивність за рахунок додаткових оптимізацій. Це говорить про те, що хоча сучасні компілятори мають досить потужні оптимізуючі алгоритми, вони не враховують усіх можливих випадків роботи з даними. В деяких випадках, як і у нашому прикладі, ручна оптимізація виявилася більш ефективною, порівняно з результатом оптимізуючого компілятора.

Висновки. Оптимізація програмного коду є важливим завданням для досягнення високої продуктивності в умовах зростаючих вимог до обробки даних. Потреба в ефективних алгоритмах та оптимізованому коді стає критично важливою. Оптимізація дозволяє не лише прискорити виконання програм, а й знизити споживання ресурсів, таких як процесорний час та пам'ять, що особливо важливо для ресурсообмежених систем.

Компілятор GCC надає різні можливості для оптимізації коду, включаючи параметри компіляції та intrinsic-функції. GCC як один з найбільш популярних і широко використовуваних компіляторів пропонує широкий спектр інструментів і технік для оптимізації. Параметри компіляції дозволяють швидко і легко застосовувати різні рівні оптимізації, в той час як intrinsic-функції надають програмісту більш тонкий контроль над машинним кодом, що генерується, дозволяючи використовувати специфічні інструкції процесора.

Аналіз асемблерних лістингів дозволяє оцінити якість кодогенерації та виявити потенційні «вузькі місця» у коді. Асемблерні лістинги, що генеруються компілятором, є низькорівневим уявленням програми, яке дозволяє програмісту побачити, як саме його код буде виконуватися процесором. Аналіз цих лістингів дозволяє виявити неефективні конструкції, надлишкові операції та інші проблеми, які можуть знижувати продуктивність програми.

Використання векторних інструкцій, розгортання циклів та вирівнювання даних є ефективними методами оптимізації. Векторні інструкції дозволяють процесору виконувати ту саму операцію над кількома елементами даних одночасно, що значно прискорює обробку масивів та інших структур даних. Розгортання циклів зменшує накладні витрати управління циклом, виконуючи кілька ітерацій циклу за один прохід. Вирівнювання даних забезпечує більш ефективний доступ до пам'яті, оскільки процесор може завантажувати та зберігати дані блоками, що відповідають розміру його кеш-лінії.

Ручна оптимізація з використанням intrinsic-функцій може забезпечити більшу продуктивність, ніж автоматична оптимізація компілятором. Незважаючи на те, що сучасні компілятори мають потужні алгоритми оптимізації, вони не завжди можуть врахувати всі особливості конкретного завдання або архітектури процесора. Ручна оптимізація, що виконується досвідченим програмістом із використанням intrinsic-функцій, дозволяє досягти максимальної продуктивності, використовуючи специфічні інструкції процесора та застосовуючи техніки оптимізації, які не завжди доступні компілятору.

Список літератури:

1. AI market to grow 39.4 % annually over forecast period. *Zion Market Research*. URL: https://www.zionmarketresearch.com/news/global-artificial-intelligence-market (дата звернення: 19.03.2025).

2. Bubeck S. et al. Sparks of artificial general intelligence: Early experiments with gpt-4 / *ArxivX*. URL: https://arxiv.org/abs/2303.12712 (дата звернення: 20.03.2025).

3. Artificial intelligence and robotics in the manufacturing industry / *Open Innovation Hub for Companies, Startups, and Experts*. URL: https://www.ennomotive.com/artificial-intelligence-robotics/ (дата звернення: 19.03.2025).

4. Robot simulator CoppeliaSim: create, compose, simulate, any robot. *Coppelia Robotics*. URL: https://www.coppeliarobotics.com/ (дата звернення: 19.03.2025).

5. Robotics simulation with Webots. *Cyberbotics*. URL: https://cyberbotics.com/ (дата звернення: 19.03.2025).

6. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf (дата звернення: 19.03.2025).

7. Antoniou A., Lu W.-S. Practical optimization. *Springer* US, New York, NY 2021. URL: https://doi.org/10.1007/978-1-0716-0843-2 (дата звернення: 20.03.2025).

8. What Is Bottlenecking My PC?. URL: https://www.intel.com/content /www/us/en/gaming/resources/what-is-bottlenecking-my-pc.html (дата звернення: 20.03.2025).

9. Платов І. М., Павловський О. М. Порівняння можливостей C/C++ компіляторів для вирішення інженерних. *Вісник інженерної академії України*. 2018. № 4. С. 65–69.

10. Bukie P. T., Udeze C. L., Obono L. O., Edim E. B. Comparative Analysis of Compiler Performances and Program Efficiency. 2019. *Pre prints.org* URL: https://www.preprints.org/manuscript/201909.0322/v1 (дата звернення: 20.03.2025).

11. Machado, R. S., Almeida, R. B., Jardim, A. D. at al. Comparing Performance of C Compilers Optimizations on Different Multicore Architectures. *International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. Campinas, Brazil, 2017. pp. 25-30 doi:10.1109/sbac-padw.2017.13

12. GCC, the GNU compiler collection. GNU project. URL: https://gcc.gnu.org/ (дата звернення: 19.03.2025).
13. Processor Specifications. URL:https://www.amd.com/en/products /specifications/processors. (дата звернення: 25.03.2025).
14. Intel® C++ Compiler Classic Developer Guide and Reference. URL: https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/intrinsics.html (дата звернення: 20.03.2025).

Pavlovskiy O. M., Platov I. M. RESEARCH ON CODE GENERATION OF C/C++ COMPILER FOR HIGH-PERFORMANCE COMPUTING IN MICROPROCESSOR SYSTEMS

The article is devoted to the study of code generation of a C/C++ compiler for high-performance computing. The paper considers the problem of optimizing program code to increase computing efficiency in conditions of growing requirements for processing large amounts of data. The relevance of the topic is due to the fact that, despite the development of computing technology, code optimization remains an important tool for achieving high performance.

The article analyzes the capabilities of the GCC compiler version 4.9.2 64-bit on a personal computer with the Windows 10 operating system and an AMD Ryzen 5600x processor. The study includes an analysis of the assembler listings generated by the compiler to assess the quality of code generation and optimization. Various approaches to optimization are considered, including the use of compilation parameters that specify different levels of optimization, as well as optimization using intrinsic functions.

A simple simulation of the motion of 1000 objects under the influence of gravity in the C programming language is used as a test program. This example allows us to analyze the basic set of instructions generated by the optimizing compiler to evaluate the optimization of the code.

Analysis of the assembly listings demonstrates that the compiler without using optimization generates code that does not use vector instructions for data processing, and there is also excessive calculation of array element addresses. The use of speed optimization parameters (--Ofast) leads to the use of vector instructions and loop unrolling, which increases performance. However, this increases the number of instructions and the presence of code sections that process scalar data, which can lead to inefficient use of cache memory.

For further optimization, intrinsic functions are used that allow the use of extended assembly instructions (AVX, FMA) and forced data alignment. Analysis of the assembly listing of the modified program shows a significant reduction in the number of instructions and more efficient data handling.

Finally, the article concludes that, despite the availability of powerful optimization algorithms in modern compilers, manual optimization using intrinsic functions can be more effective in some cases. This emphasizes the importance of the programmer's competence in low-level optimization to achieve maximum performance in high-performance computing.

Key words: *compiler, optimization, AVX, FMA, programming, GCC, assembler.*