УДК 004.032.8 **doi: 10.26906/SUNZ.2021.4.044**

А. С. Янко, А. М. Мартиненко, О. В. Бут

[1] Національний університет «Полтавська політехніка ім. Ю. Кондратюка», Полтава, Україна

МЕТОДИ ВИКОРИСТАННЯ SIMD ІНСТРУКЦІЙ НА X86 СУМІСНИХ ПРОЦЕСОРАХ СТАРШОГО ПОКОЛІННЯ

Анотація. Розглянуто використання векторних SIMD інструкцій на x86 сумісних процесорах для покращення ефективності обчислення та обробки даних. Застосування векторного набору інструкцій дозволяє збільшити кількість операцій виконуваних за такт, при цьому зменшення розгалужень у алгоритмах позитивно влипає на швидкість виконання програми за рахунок меншого навантаження на модуль передбачення умовних переходів у процесорі. До цього часу існує програмне забезпечення, що виконується на x86 архітектурі процесорних ядер, даний факт не завжди дає змогу використовувати новітні векторні інструкції починаючи з SSE4.1. Головним недоліком попередніх реалізацій векторних наборів інструкцій – це відсутність логічних і арифметичних операцій з деякими типами даних, особливо це спостерігається у операціях з цілими числами. Використання особливості бінарної реалізації цілих чисел зі знаком і без знаку, дозволяє компенсувати відсутність логічних операцій для цих типів даних. Експлуатація вироджених та непрямих властивостей деяких інструкцій допомагає, як компенсувати відсутність арифметичних операцій з необхідними типами даних або операцій для цілих чисел іншої розрядності, так і оптимізувати виконання математичних операцій таких, як знаходження суми, різниці, множення та скалярного добутку.

Ключові слова: арифметична операція, векторна інструкція, набір інструкцій процесора, операнд константи, оптимізація процесу обробки даних, паралелізм на рівні інструкцій, паралельне обчислення.

Вступ

Сьогодні практично всі процесори мають кілька блоків для обчислення. Зазвичай центральний процесор містить як арифметичну логічну одиницю (ALU), яка виконує цілочисельні операції, так і одиницю з плаваючою комою (FPU), яка виконує арифметику з плаваючою комою. Ці одиниці, як правило, працюють паралельно, тому ALU може виконувати цілочисельні інструкції в той же час FPU виконує вказівки з плаваючою комою. Деякі центральні процесори є суперскалярними з n-напрямком, що означає, що вони мають n паралельних одиниць виконання (зазвичай ALU) і можуть розпочинати обробку до n інструкцій для кожного циклу. Крім того, багато сучасних архітектур мікропроцесорів пропонують певний тип розширень типу SIMD (одна інструкція, декілька даних) до свого базового набору команд. Ці інструкції SIMD визначені у широких регістра, вони містять 4, 8 або навіть 16 скалярних цілих чисел із плаваючою комою, виконуючи одну операцію над цими значеннями. Таким чином, використання інструкцій SIMD обіцяє потенційне 4-кратне (або 8- або 16-кратне) підвищення продуктивності коду, який може скористатися цим набором інструкцій. Доступні на даний момент розширення SIMD для x86 сумісних процесорів включають SSE, SSE2, SSE3, SSSE3 та більш новітні SSE4.1, SSE4.2, AVX. Оптимізацію послідовності виконання коду можна розділити на:

– паралелізм на рівні інструкцій. Існуючі алгоритми можна оптимізувати шляхом визначення подібних операцій та їх обчислення паралельно. Наприклад, множення в скалярного векторного добутку можна обчислити паралельно, використовуючи одне множення SIMD. Оскільки для більшості обчислень є послідовні аспекти, деякі частини процесу не можна оптимізувати напряму за допомогою паралелізму на рівні інструкцій, наприклад, підсумовування часткових добутків у скалярному векторному добутку.

– паралелізм на рівні даних. SIMD також можна використовувати для паралельної роботи з абсолютно різними даними. Наприклад, типовим використанням може бути обчислення паралельно чотирьох скалярного векторного добутку. Паралелізм на рівні даних є природним оперативним використанням інструкцій SIMD, як це зрозуміло з назви SIMD.[1]

SSE та SSE2 доступні в кожному окремому процесорі сімейства x86 з 64-розрядною підтримкою. На жаль, рівень SSE2, зокрема, також є, мабуть, найбільш не ортогональним набором інструкцій SIMD у сучасному реаліях, де операції або доступні, або недоступні для певних типів даних, особливо там, де задіяні цілі числа. Пізніші версії (особливо починаючи з SSE4.1) заповнюють деякі недоробки та прогалини, але часто при розробці програмного забезпечення доводиться затримуватися на підтримці старих процесорів принаймні ще на кілька років, навіть на процесорах, що підтримують AVX2 набор інструкцій, залишається кілька класичних прогалин. Але не зважаючи на це все одне інтегрування SSE інструкцій при розробці має позитивний вплив на швидкість її виконання. Реалізація тих чи інших алгоритмів з використанням SSE використовується на останніх етапах розробки, оскільки передбачає за собою специфічне рішення, яке залежить від того які дані і як вони використовуються в програмі, але не рідкі випадки, коли такі алгоритми мають дещо спільні шаблони рішення, які часто використовують при програмуванні з використанням SSE інструкцій. Саме про такі найпоширеніші випадки йдеться в даній стаття, в основному увага зосереджена на операціях з цілими числами, але випадки про числа з плаваючою комою також будуть розглянуті.

Аналіз досліджень і публікацій. Питання використання SIMD інструкцій для пришвидшення виконання обчислення існує вже досить давно. Так першим хто почав їх використовувати це суперкомп'ютер ILLIAC IV у 1966 році. З часом зростання технологічного прогресу появи Інтернету та початку ери домашніх комп’ютерів активно почали вико-

ристовуватися задачах як 3D графіка, обробка аудіо та відео. Попит на цей конкретний тип обчислювальної потужності зріс, і виробники мікропроцесорів звернулися до SIMD, щоб задовольнити попит. Перший широко розповсюджений набор інструкцій SIMD для домашній комп'ютерів був з розширеннями MMX від Intel до архітектури x86 у 1996 році, а згодом у 1999 SSE у процесорі Pentium III. Одні з перших публікацій на цю тему були в 1999 році у праці Асі Ельбаза – співрозробника першого кодеку MPEG-2 для Pentium III, його дослідження на той час стосувалися обробки зображень та сигналів. Згодом у 2000 році у своїй праці автори Д. Конте, Ф. Занічеллі та С. Томмесані детально описали обробку зображень з використанням SSE.

У сучасності на таких конференціях як CppCon, GDC практично кожен рік люди діляться своїми дослідженнями та досвідом, щодо використання SIMD інструкцій, наприклад, у 2018 році Боб Стіголл звітував про його дослідження у векторизації процесу декодування тексту, що знаходиться у форматі кодування UTF-8, а пізніше у 2020 року продовження доповіді про дослідження з використанням AVX-512. Актуальність проблеми з неможливістю використання новітніх SIMD на x86 процесорах можна побачити у випадку з компанією CD PROJEKT RED, яка 6 вересня 2021 року зробила оновлення для своєї минулорічної комп'ютерної гри, в якому було видалено використання AVX інструкцій для підтримки більш старих комп'ютерів.

Метою статті є розробка шаблонів використання SIMD інструкцій та їх властивостей для оптимізації обчислення за умов неможливості використання нового набору інструкцій починаючи з SSE4.1 та вище.

Виклад основного матеріалу

Вибірка без розгалуження. Уникнення розгалужень при побудові алгоритму – це природний спосіб виконання SIMD розрахунків. Якщо деяка частина обчислення є умовною, а не виконує еквівалент "if", то більш типово виконувати обчислення для розгалужень "if" та "else", а потім об'єднувати результати на основі умови. Під словом "вибір" мається на увазі операція, яка приймає умову та обидва результати та виконує приблизний еквівалент умов потрійного оператора мови програмування C "cond ? a : b". Спочатку знаходиться результат по обидві сторони вираження a і b. Потім обчислюється умовне вираження, використовуючи порівняння SIMD, яке повертає вектор, що містить бітову маску, в якій всі біти встановлені для ліній, що відповідають умові, та рівні нулю для ліній, що не відповідають умові. Ця операція вибору завжди може бути виконана за допомогою кількох побітових операцій, що добре відомо, але починаючи з SSE 4.1, ми також отримуємо дещо ефективніші варіанти, але менш відомі:

Ціле число:
_mm_or_si128(_mm_and_si128(a, cond);
_mm_andnot_si128(cond, b));
32-розрядний число з плаваючою комою:
_mm_or_ps(_mm_and_ps(a, cond);
_mm_andnot_ps(cond, b));
64-розрядний число з плаваючою комою:
_mm_or_pd(_mm_and_pd(a, cond), _mm_andnot_pd(cond, b))
Ціле число (SSE 4.1+):
_mm_blendv_epi8(a, b, cond).
32-бітне число з плаваючою комою (SSE 4.1+):
_mm_blendv_ps(a, b, cond).
64-бітне число з плаваючою комою (SSE 4.1+):
_mm_blendv_pd(a, b, cond).

Операції andnot не дуже зручні, але вони є найкращим вибором для попередніх версія перед SSE4.1. Порівняння цілих чисел без знаків. SSE, у всіх втіленнях, пропонує точно два типи цілочисельних порівнянь: тест на рівність (PCMPEQt, _mm_cmpeq_T, де t і T означають суфікси різних типів) та тест більше ніж зі знаком (PCMPGTt, _mm_cmpgt_T). Більшість інших типів порівняння можна створити, використовуючи лише логічне заперечення та стандартні ідентичності:

a == b підтримується безпосередньо.
a != b це !(a == b).
a > b (зі знаком) підтримується безпосередньо.
a < b (зі знаком) те саме, що b > a (поміняти місцями a і b).
a >= b (зі знаком) це !(a < b) (що в свою чергу є !(b > a)).
a <= b (зі знаком) це !(a > b).

У цьому списку явно відсутній будь-який тип упорядкованого порівняння без знака. Однак прийом, полягає в тому, щоб задати зміщення для цілого числа, щоб порівняння зі знаком було правильним:

a > b (без знака, 8-біт) те саме, що (a - 0x80) > (b - 0x80) (зі знаком, 8-біт).
a > b (без знака, 16-біт) те саме, що (a - 0x8000)> (b - 0x8000) (зі знаком, 16-біт).
a > b (без знака, 32-біт) те саме, що (a - 0x80000000) > (b - 0x80000000) (зі знаком, 32-біт).

Методи із заміною аргументів, які розглянуті раніше, можна використати для виконання й інших типів порівнянь. Загалом, прийом полягає в тому, щоб додати (або відняти, або виконати XOR – у даному конкретному випадку ці операції призводять до одного результату) мінімальне число зі знаком для відповідного типу у обох операндів порівняння перед його виконанням. Це перетворює найменше можливе ціле число без знака, 0, в найменше можливе ціле число зі знаком для даного типу. Зокрема, при порівнянні з константою це додавання (або віднімання, або XOR) може бути вбудовано в операнд константи, так що порівняння без знаку тільки в кінцевому підсумку виконає на одну операцію більше, ніж порівняння зі знаком (замість двох). Зовсім інший підхід полягає у використанні без знакових цілочисельних інструкцій min/max для побудови порівнянь менших або рівних чи більших або рівних:

a <= b тільки тоді, коли max(a, b) == b
a >= b тільки тоді, коли min(a, b) == b

Позитивною стороною даного процесу є те, що це зменшує порівняння без знаків до без знакового min або max, після чого йде порівняння рівності, яке становить лише дві операції замість трьох. Негативна сторона полягає в тому, що необхідні операції min/max без знака існують лише для 8-біт у SSE2. Варіанти 16-біт і 32-біт були нарешті додані з SSE4.1.

Знаходження min/max цілих чисел. SSE4.1 має повний набір цілих чисел min/max для 8-, 16- та 32-розрядних типів, як знакових, так і без знакових. У SSE2 отримується лише min/max порівняння для 8-бітних цілих чисел без знаку та 16-бітних без знаку. Але комбінуючи методи розглянуті вище можливо вивести рішення. Загальній шаблон буде виглядати як:

max(a, b) == (a > b) ? a : b;

min(a, b) == (a > b) ? b : a;

Тож це лише поєднання операції порівняння та операції вибору. Коли порівняння здійснюється для чисел зі знаком (випадки 8-біт та 32-біт), порівняння відображається на єдину SSE інструкції. Порівняння без знаку (16-біт та 32-біт) можна вирішити, використовуючи шаблон "знакове зміщення і порівняння", який, у свою чергу, дає нам без знакове значення min/max.

Множення. Є багато різних способів забезпечити множення в наборі інструкцій SIMD, і до теперішнього часу SSE випробувала більшість з них в тій чи іншій формі.

Почнемо з історично сформованого першого варіанту: множення 16-бітних чисел. Відповідні інструкції виходять з Pentium MMX і обчислюють нижні і верхні 16 біт знакового 16-бітного × 16-бітного множення. MMX має тільки знакові множення, але SSE також додав інструкцію множення верхньої половини без знакового 16-бітного множення на 16-біт (нижні половини знакових і без знакових множень ідентичні). Ці інструкції підходять, якщо виникає потреба отримати нижню або верхню половину множення. Але також виникає питання, якщо потрібне повне 32-бітове множення векторів 16-бітних значень? Загальним шаблоном рішення у цієї ситуації є обчислення нижньої та верхньої половин, а потім поєднання їх, використовуючи інструкції розпакувати. Отже, для повного множення 16 × 16 → 32-біт (зверніть увагу, що це дає результат в два вектора), ми отримуємо:

```
// або: a*b (16-розрядні смуги), знакові
__m128i lo16 = _mm_mullo_epi16(a, b);
__m128i hi16 = _mm_mulhi_epi16(a, b);

// або: a*b (16-розрядні смуги), без знаку
__m128i lo16 = _mm_mullo_epi16(a, b);
__m128i hi16 = _mm_mulhi_epu16(a, b);

// потім злиття результатів
__m128i res0 = _mm_unpacklo_epi16(lo16, hi16);
// смуги результатів 0..3
__m128i res1 = _mm_unpackhi_epi16(lo16, hi16);
 // смуги результатів 4..7
```

У випадках, коли потребується робота з 32-бітними значеннями виникає потреба у дещо іншому підході. Так існує 32 × 32 → 32-бітне множення (PMULLD/ _mm_mullo_epi32), але ця інструкція була додана тільки з SSE4.1, і в багатьох реалізаціях виконується значно повільніше, ніж інші SSE2 множення. Таким чином, треба розглянути поточну ситуацію і прийняти рішення або не встановлювати такий високий числовий запас, або застосувати більш швидке рішення.

Існує повне 32 × 32 → 64-бітне множення, яке доступне з SSE2 як PMULUDQ/_mm_mul_epu32 (без знаку). SSE4.1 додає знаковий еквівалент PMULDQ/_mm_mul_epi32. Ця інструкція лише обчислюють два добутки (між парними смугами двох джерел) і поміщає їх у 128-бітній результат. Непарні 32-бітні лінії повністю ігноруються, тому, якщо виникає потреба в чотирьох цілочисельних 32 × 32 → 32-біт множеннях, знадобиться принаймні два з цих множень:

```
// res = _mm_mullo_epi32 (a, b) еквівалент з використанням SSE2, через PMULUDQ.

// парні та непарні лінії
__m128i evnp = _mm_mul_epu32(a, b);
__m128i odda = _mm_srli_epi64(a, 32);
__m128i oddb = _mm_srli_epi64(b, 32);
__m128i oddp = _mm_mul_epu32(odda, oddb);

// злиття результатів
__m128i evn_mask = _mm_setr_epi32(-1,0, -1, 0);
__m128i evn_result = _mm_and_si128(evnp,
evn_mask);
__m128i odd_result= _mm_slli_epi64 (evnp, 32);
__m128i res = _mm_or_si128 (evn_result, odd_result);
```

Не є винятком коли використовуються 32-бітні векторні лінії, але в той самий момент числа з умов алгоритму насправді знаходяться в діапазоні 16-бітного цілого числа зі знаком [-32768, 32767]. Одним з можливих рішень в цьому випадку може бути спробувати звузити 32-бітові смуги до 16 біт, а потім використовувати послідовності 16 × 16 → 32, зазначені вище, але існує другий більш оптимальний спосіб вирішення цієї задачі з використанням _mm_madd_epi16, ціль якої перемноження цілих 16-бітних чисел зі знаком створення проміжного 32-бітного результату цілих чисел зі знаком.

```
// a та b мають 32-бітні смуги зі значеннями, які відповідають цілим 16-бітним числам зі знаком.
// видає 32-бітний результат
// res [i] = a [i] * b [i]
// очищає верхні 16 бітів кожної 32-бітної смуги
__m128i bm = _mm_and_si128(b, _mm_set1_epi32
(0xffff));
__m128i res = _mm_madd_epi16(a, bm);
// також можна поміняти ролі a і b вище, коли це зручно.
```

Це набагато коротше, ніж спочатку звуження до 16-біт. На жаль, такий спосіб працює тільки для 16-бітних чисел зі знаком.

Випадок, який залишилося розглянути це множення з використанням 32-бітніх ліній зі значеннями, які вписуються в діапазон 16-бітних цілих чисел без знаку [0, 65536].

```
// a та b мають 32-розрядні смуги зі значеннями, які відповідають цілим 16-бітним числам без знаку.
// тобто a[i] == (uint16) a[i] і те саме для b[i].
//
// видає 32-розрядний результат
// res [i] = a[i] * b[i]

// обчислюємо низькі та високі 16-розрядні добутки
__m128i lop = _mm_mullo_epi16(a, b);
__m128i hip = _mm_mulhi_epu16(a, b);

// злиття результатів
__m128i res = _mm_or_si128(lop,
_mm_slli_epi32(hip16));
```

Додавання по горизонталі та скалярний добуток. SSE3 додає горизонтальні додавання HADDPS (_mm_hadd_ps) та HADDPD (_mm_hadd_pd), а SSE4.1 додає інструкції із множенням DPPS (_mm_dp_ps) та DPPD (_mm_dp_pd). Їх ціль операції над сусідніми парами чисел з плаваючою комою елементів a і b. Ці інструкції мають ряд недоліків, у більшості реалізацій x86 процесорах вони перетворюються на послідовність більш простих операцій (на рівні SSE2), також вони суперечать основній цілі SIMD операцій, замість виконання однотипних операцій над даними одночасно вони оперують над даними лише в межах одного регістру та супроводжується результатом, який ускладнює його використання для повних SIMD операцій, що в подальшому тільки негативно впливає на основну мету досягнути максимальної пропускної спроможності процесора по обробці інструкцій.

Інший вид горизонтальних додавань, та скалярного добутку для цілих чисел. SSE інструкції такого типу, наприклад, такі як PHADDW/PHADDD (_mm_hadd_epi16/_mm_hadd _epi32), які присутні у реалізаціях SSSE3 і у версія, що з'явилися пізніше часто не підходять для рішення задач з причин, які описанні вище або якщо немає можливості використовувати SSSE3 інструкції. У цьому випадку можуть бути корисними PSADBW (_mm_sad_epu8), PMADDWD (_mm_madd_epi16), які належать до реалізації SSE3 та для деяких окремих випадків інструкція PMADDUBSW (_mm_maddubs_epi16), яка доступна з SSSE3. Основна ідея PMADDWD і PMADDUBSW – це скалярний добуток між парами сусідніх ліній. PMADDWD обчислює два добутки 16-бітних цілих чисел зі знаком для кожної пари 16-бітних ліній і видає результат 32-бітне ціле число, що зберігається у 32-бітній результуючій лінії. Їх можна використовувати для обчислення скалярних добутків, але у них також є вироджені властивості які можуть бути корисними:

– _mm_madd_epi16(x, _mm_set1_epi16(1)) підсумовує 16-розрядні парні та непарні смуги по парах, щоб отримати 32-бітні результати.

–_mm_maddubs_epi16(_mm_unpacklo_epi8(a, b), _mm_setr_epi8(1, -1, 1, -1, ..., 1, -1)) є найшвидшим способом обчислення 16-бітної різниці для цілих чисел зі знаком між 8-бітними без знаковими векторами a і b на процесорах, які підтримують SSSE3.

– _mm_sad_epu8 (x, _mm_setzero_si128()) обчислює дві 16-бітові горизонтальні суми груп з восьми ліній 8-бітних цілих чисел без знаку за одну і досить швидку операцію, річ йде про приблизно п'яти циклів на одну інструкцію. Також можна застосувати прийом порівняння розглянутий раніше, але у зворотному порядку. Тобто задати зміщення для виконання операції над числами зі знаком, щоб обчислити суму восьми 8-бітних цілих чисел зі знаком: додати (або відняти, або виконати XOR) _mm_set1_epi8(-128) з потрібним аргументом перед виконанням _mm_sad_epu8() і потім відняти 128×8 з отриманого результату 16-бітної суми.

Висновки

При проектуванні SIMD інструкцій для x86 сумісних процесорів, а сам SEE, та подальшої їх інтеграції було допущено ряд архітектурних рішень які потенціально ускладнюють оптимізацію процесу обробки даних, деякі з цих проблем, такі як, відсутність можливість виконання одної операцій для цілочисельних чисел різної розрядності та їх типу, зі знаком або без, були вирішені у наступних версіях даного набору інструкцій. Не зважаючи на високий рівень підтримки нових інструкцій в сучасних реаліях, деякі продукти потребують повного покриття використовуваних процесорів, тому не завжди є можливість використовувати нові інструкції. Для таких випадків коли потрібно, наприклад, здійснити емуляцію інструкції, було вироблено ряд практик, які показують найкращий результат. При роботі з такими набором інструкцій можливі прийняти більш оптимальні рішень, якщо використовувати їх не тільки по прямому призначені, а знаходити і використовувати їх вироджені властивості.

REFERENCES

1. Christer Ericson, "Real-time Collision Detection" – The Morgan Kaufmann Series, 2004. pp. 543–545.
2. Daniel Kusswurm, "Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX", 2014. – pp. 179-187.
3. David H. Eberly, "GPGPU Programming for Games and Science", 2014. – 93 p.
4. Paul Besl, "A case study comparing AoS (Arrays of Structures) and SoA (Structures of Arrays) data layouts for a compute-intensive loop run on Intel Xeon processors and Intel Xeon Phi product family coprocessors". In: Intel Article 392271, 2015.
5. G. Ren, P. Wu and D. A. Padua, "A preliminary study on the vectorization of multimedia applications for multimedia extensions", in Proc. LCPC 03, 2003. – pp. 420–435.

Received (Надійшла) 27.10.2021
Accepted for publication (Прийнята до друку) 17.11.2021

Methods of using simd instructions on x86 compatible older generation processors

A. Yanko, A. Martynenko, O. But

Abstract. The use of vector SIMD instructions on x86 compatible processors to improve the efficiency of computing and data processing is considered. The use of a vector set of instructions allows you to increase the number of operations performed per clock cycle through the use of parallelism at the instruction level or parallelism at the data level. At the same time, the reduction of branches in the algorithms also adds a positive impact on the speed of program execution due to fewer loads on the module for predicting conditional transitions in the processor. Also it help to optimize the use of cache lines and data transportation between the L1 cache and the CPU register because in modern processors the bus through which data is transported between the L1 cache and the register is at least 128 bits. In some cases this fact can play a significant role in achieving the goal of computational optimization. Additionally, the previously listed factors improve the ability of modern processors to execute instructions out of order. An important factor for the effective use of SIMD instructions is to accurately determine the type, nature of the data and the desired end result of the algorithm because further modification of the algorithm based on the use of SIMD is not flexible and depends entirely on these factors. So far, there is software running on the x86 architecture of processor cores, this fact does not always allow the use of the latest vector instructions starting with SSE4.1. The main disadvantage of previous implementations of vector instruction sets is the lack of logical and arithmetic operations with some types of data, especially in operations with integers. Using the features of binary implementation of sign integers, allow to compensate for the lack of logical operations for these data types. Exploiting the degenerate and indirect properties of some instructions helps both to compensate for the absence of arithmetic operations with the required data types and operations for integers with a different bit depth, likewise to optimize the performance of mathematical operations such as finding the sum, difference, multiplication and scalar product.

Keywords: arithmetic operation, vector instruction, set of processor instructions, operand constants, optimization of data processing, parallelism at the instruction level, parallel calculation.