doi: 10.32620/oikit.2025.104.09

УДК 004.415

О. В. Тіхвиський, О. В. Каратанов

Огляд сучасних методів оптимізації продуктивності комп’ютерних програм

Національний аерокосмічний університет «Харківський авіаційний інститут», Харків, Україна.

У статті здійснено комплексний аналіз сучасних підходів до оптимізації продуктивності комп’ютерних програм в умовах безперервного ускладнення обчислювальних задач і збільшення обсягів оброблюваних даних. Розглянуто основні апаратні й програмні чинники, що визначають ефективність виконання програм, включаючи архітектурні особливості центральних та графічних процесорів, організацію багаторівневої пам’яті, підтримку векторних обчислень та механізми паралельної обробки даних. Особливу увагу приділено впливу вибору алгоритмів і структур даних на швидкодію програмного забезпечення, враховуючи аспекти локальності пам’яті, уникнення промахів кешу та мінімізації обсягів передавання даних між різними рівнями ієрархії пам’яті. Проаналізовано типові проблеми, пов’язані з накладними витратами на створення та синхронізацію потоків, а також із затримками виконання через розгалуження у коді. Розглянуто принципи ефективного використання багатоядерних процесорів і гетерогенних платформ, де оптимальний розподіл обчислень між CPU та GPU є критичним для досягнення високої продуктивності. Підкреслюється важливість обґрунтованого вибору оптимізаційних стратегій залежно від специфіки прикладного завдання, особливостей апаратного середовища та характеру обчислювального навантаження. У роботі наголошується на необхідності системного підходу до оптимізації, що передбачає не лише використання окремих прийомів, але й їх комплексну інтеграцію з урахуванням взаємного впливу на кінцеву продуктивність. Також акцентовано на важливості емпіричного оцінювання результатів оптимізації, що базується на реальних вимірюваннях часу виконання, профілюванні продуктивності та аналізі використання ресурсів. Сформульовано узагальнені рекомендації щодо побудови високопродуктивних програмних систем, спрямовані на мінімізацію затримок доступу до пам’яті, оптимізацію обробки умовних переходів, раціональне використання можливостей апаратних прискорювачів та забезпечення ефективного паралелізму. Представлені результати можуть бути корисними як для розробників програмного забезпечення, що працюють над підвищенням продуктивності прикладних систем, так і для дослідників у галузі високопродуктивних обчислень, сприяючи розвитку нових методик оптимізації в умовах еволюції обчислювальних архітектур.

Ключові слова: оптимізація; продуктивність; багатопотоковість багатопоточність; кеш процесора; передбачення розгалужень; конвеєризація; попереднє вибирання; SIMD.

Вступ

У світі, де технології швидко розвиваються, а очікування користувачів зростають, оптимізація стає не просто бажаною, а критично необхідною. Раніше, поки працював Закон Мура, сформульований у 1965 році, було достатньо просто зачекати два роки і ваше залізо ставало вдвічі швидшим. Проте з досягненням нанометрових технологічних норм цей підхід перестав бути ефективним [1], оскільки масштабування Деннарда, яке передбачає збереження постійної щільності потужності при зменшенні розмірів транзисторів, більше не підтримувало темпи розвитку, передбачені законом Мура, що показано на Рисунку 1. Також зростання обсягів даних, які потрібно обробляти в реальному часі, ставить перед програмами нові завдання, а забезпечення високої швидкості

вимагає від розробників стратегічного мислення та глибоких технічних знань.

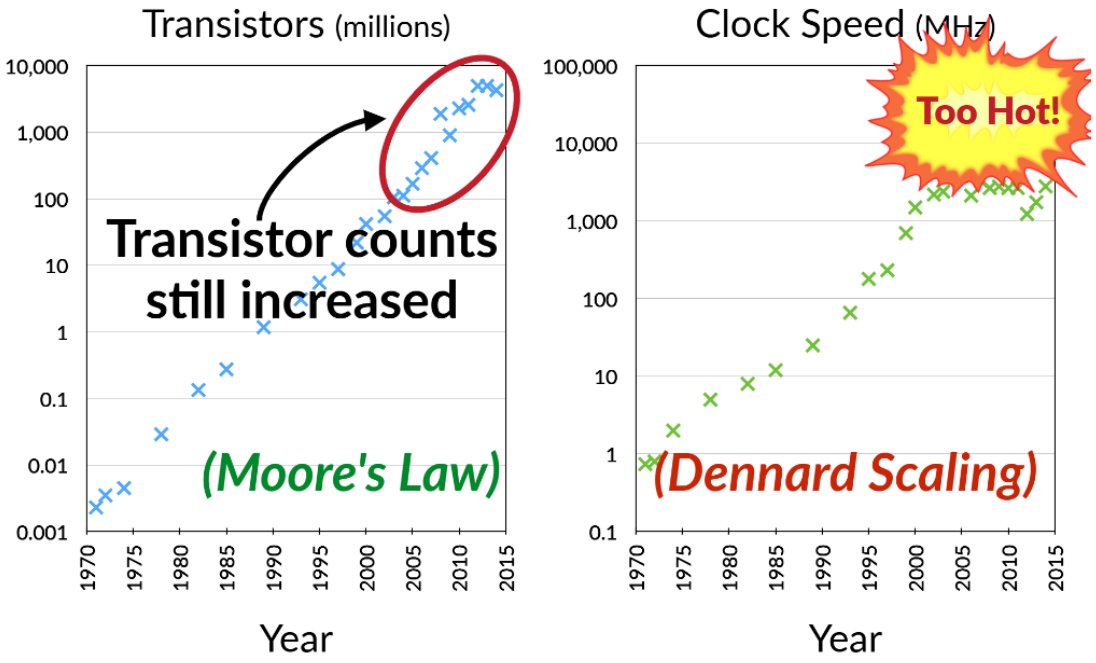

Рис. 1. Невідповідність масштабування Деннарда закону Мура [2]

Ще однією проблемою являється оцінка ефективності алгоритмів, що обмежується виключно аналізом їхньої часової складності, як це зроблено, наприклад, в статті Big-O Time Complexity Analysis Of Algorithm [3]. Така оцінка є необхідною, але недостатньою для всебічного розуміння їхньої практичної цінності. Теоретична складність, виражена за допомогою нотації Ландау, фокусується на поведінці алгоритму при необмежено великих обсягах вхідних даних, ігноруючи значущі константні множники та члени нижчого порядку, які можуть суттєво впливати на продуктивність при реалістичних розмірах задач. Крім того, модель обчислень, що лежить в основі аналізу складності, часто є ідеалізованою та не враховує архітектурні особливості сучасних обчислювальних систем, такі як ієрархія пам'яті, паралелізм та кешування. Відтак, для повної оцінки алгоритму необхідно доповнювати теоретичний аналіз емпіричними дослідженнями його продуктивності на репрезентативних наборах даних та враховувати контекст конкретного застосування, включаючи вимоги до реального часу, доступні ресурси та складність імплементації. Саме тому важливо проаналізувати існуючи підходи до оптимізації комп’ютерних програм, та можливості їх поєднання задля отримання ще кращої продуктивності.

1. Мета роботи

Метою даного дослідження є здійснення комплексного аналізу сучасних підходів до підвищення продуктивності комп’ютерних програм в умовах постійного розвитку апаратного забезпечення та зростання вимог до ефективності обробки даних. Дослідження спрямоване на виявлення основних чинників, що визначають продуктивність програмних систем, оцінку загальних закономірностей впливу архітектурних та програмних особливостей на швидкодію, а також на узагальнення принципів оптимізації, придатних для

широкого спектра обчислювальних завдань. Крім того, робота має на меті обґрунтувати важливість системного підходу до оптимізації, що передбачає врахування взаємодії між різними рівнями апаратного та програмного середовищ, а також сформулювати загальні рекомендації щодо підвищення ефективності програмного забезпечення шляхом об'єднання різних стратегій оптимізації. Особливу увагу приділено питанням оцінювання доцільності застосування оптимізаційних заходів залежно від характеристик завдань і ресурсних обмежень середовища виконання. Таким чином, дослідження спрямоване на поглиблення теоретичних основ оптимізації продуктивності та на підтримку розробників у прийнятті обґрунтованих рішень щодо покращення ефективності програмних систем.

2. Продуктивність за допомогою процесора

Коли стало зрозуміло, що традиційне масштабування більше не працює, індустрія почала шукати нові шляхи для підвищення продуктивності. Основними напрямками стали:

1. Багатоядерність. Замість збільшення тактової частоти виробники почали додавати більше ядер у процесори. Це дозволяє виконувати кілька задач одночасно, покращуючи загальну продуктивність у багатозадачних середовищах. Однак ефективність багатоядерних процесорів залежить від здатності програмного забезпечення до паралельної обробки.

2. SIMD (Single Instruction, Multiple Data). SIMD дозволяє одній інструкції одночасно виконувати операції над кількома наборами даних. Це особливо ефективно в задачах, пов'язаних з обробкою графіки, штучним інтелектом та числовими розрахунками. Сучасні процесори з підтримкою AVX або SSE, забезпечують високопродуктивну обробку великих масивів даних завдяки SIMD.

3. Конвеєризація (pipelining) — це техніка підвищення продуктивності процесора шляхом розбиття виконання інструкцій на кілька стадій (етапів), які виконуються одночасно. Замість того щоб виконувати кожну інструкцію від початку до кінця перед переходом до наступної, процесор розбиває виконання на кілька етапів, і коли одна інструкція переходить на наступний етап, наступна інструкція може зайняти її місце в попередньому етапі. Таким чином, кілька інструкцій обробляються одночасно, що суттєво підвищує продуктивність.

4. Передбачення розгалужень (Branch Prediction) — це механізм, який намагається передбачити результат умовних переходів у коді (наприклад, if-else або циклів) ще до того, як їх фактичний результат буде відомий. Успішне передбачення дозволяє уникати простоїв у конвеєрі процесора (pipeline), що значно підвищує продуктивність. Хибне передбачення (branch misprediction) викликає очищення конвеєра (pipeline flush), що призводить до значних затримок. Наприклад, у процесорі з глибоким конвеєром це може означати втрату 10–20 циклів. Для кодів із великою кількістю умовних переходів і непередбачуваними патернами розгалужень (наприклад, залежність від даних) хибні передбачення можуть значно знижувати продуктивність.

5. Попереднє вибирання **(**Prefetching) — це техніка підвищення продуктивності процесора, яка передбачає завантаження даних або інструкцій у кеш-пам’ять до того, як вони стануть потрібними. Це допомагає зменшити затримки при доступі до основної пам’яті та покращує ефективність виконання програм.

3. Продуктивність за допомогою пам’яті

У сучасних обчислювальних системах продуктивність програм залежить не лише від потужності процесора, але й від ефективності використання пам'яті. Основною перешкодою для досягнення високої продуктивності є різниця у швидкості доступу до різних рівнів пам'яті: кешу, оперативної пам'яті (RAM) і дискових систем. Оптимальне використання кешу процесора та управління доступом до пам'яті є критично важливими для забезпечення швидкої роботи програм. Таблиця 1 демонструє величезну різницю у швидкості доступу до різних рівнів пам'яті та операцій [4], підкреслюючи критичну важливість ефективного використання кешу для досягнення високої продуктивності. Затримки зростають у сотні й тисячі разів при переході від кешу до оперативної пам'яті, диска або мережі.

Таблиця 1

Різниця у швидкості доступу до різних видів пам’яті

Операція	Час (нс)	Приблизне відношення
Звернення до L1 кешу	0.5	
Один цикл на процесорі 3 ГГц	1	
Неправильне передбачення переходу	5	10x L1 кеш
Звернення до L2 кешу	7	14x L1 кеш
Звернення до основної пам'яті	100	20x L2, 200x L1
Випадкове читання 4 КБ з SSD*	150	
Послідовне читання 1 МБ з пам'яті	250	
Послідовне читання 1 МБ з SSD*	1,000,000	4x пам'ять
Послідовне читання 1 МБ з диска	20,000,000	80x пам'ять, 20x SSD

Коли дані, необхідні процесору, не знаходяться в кеші, відбувається кеш-промах. Це призводить до значного зниження продуктивності, оскільки процесор змушений чекати на завантаження даних з оперативної пам'яті [5]. Вирівнювання структур даних у пам'яті відповідно до розміру кеш-ліній дозволяє уникати зайвих кеш-промахів і прискорює доступ до даних.

Згідно з Хеннесі та Паттерсоном [6], причини промахів кешу можна класифікувати наступним чином:

1. Обов'язкові промахи виникають при першому зверненні до певної області пам'яті.
2. Промахи через справжнє спільне використання виникають на ядрі X внаслідок запису з ядра Y, що робить недійними дані, необхідні ядру X.
3. Промахи через хибне спільне використання виникають на ядрі X внаслідок запису з ядра Y в іншу частину кеш-лінії, необхідну ядру X.
4. Конфліктні промахи виникають через те, що ядро використовує за короткий проміжок часу занадто багато різних елементів даних, які потрапляють в один і той самий набір асоціативності.
5. Промахи через обмеження місткості виникають через те, що робочий набір програмного забезпечення більший за загальну місткість кешу.

4. Продуктивність за допомогою графічного процесора

Зростання популярності гетерогенних обчислювальних платформ, що поєднують традиційні центральні процесори (CPU) з потужними прискорювачами, такими як графічні процесори (GPU), відкриває значні можливості для оптимізації продуктивності програм. Використання GPU дозволяє перенести обчислювально інтенсивні частини коду на пристрій, що краще підходить для масового паралелізму даних, потенційно досягаючи значного прискорення.

Сучасні програмні моделі, зокрема OpenMP та OpenACC, надають засоби для написання коду, орієнтованого на прискорювачі, абстрагуючись від специфіки конкретної архітектури. Такі моделі дозволяють розробникам за допомогою директив вказувати, які частини коду, наприклад, паралельні цикли, можуть бути виконані на прискорювачі. Задача генерації специфічного для пристрою коду, такого як SIMT для GPU або векторизованого коду для CPU, оптимізації та налаштування параметрів перекладається на компілятор та середовище виконання.

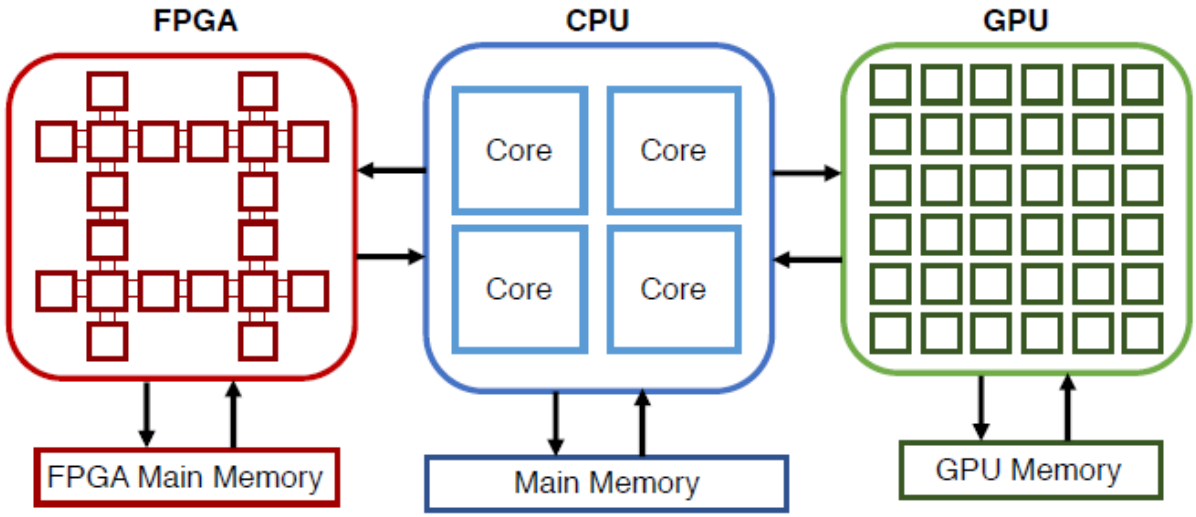

Рис. 2. Приклад топології гетерогенної системи

Ключовою можливістю оптимізації в гетерогенних системах є автоматичний вибір найбільш ефективного обчислювального пристрою (CPU чи GPU) для виконання конкретного обчислювального ядра [7]. Оскільки не всі завдання отримують виграш від виконання на GPU, а продуктивність залежить від характеристик завдання та архітектури, виникає потреба в механізмах аналізу прибутковості офлоадингу. Майбутні стандарти, як OpenMP 5.0, планують запровадити директиви, що дозволять компілятору або середовищу виконання самостійно обирати оптимальний пристрій.

5. Практичне застосування

Правильний вибір структури даних безпосередньо впливає на продуктивність, використання пам’яті та швидкість обчислень. У статті Скотта Мейерса [8] показано, що звичне використання std::set може бути неоптимальним. Хоча std::set гарантує O(log n) для пошуку, вставки та видалення, його продуктивність може бути гіршою через погану кеш-локальність

і високі накладні витрати (Рисунок 3). Натомість відсортований std::vector може працювати швидше завдяки ефективному використанню кешу процесора, що робить бінарний пошук на ньому більш продуктивним, ніж аналогічний пошук у std::set. Якщо ж немає потреби в підтримці порядку елементів, ще швидшим рішенням може бути std::unordered_set, який у середньому забезпечує доступ за O(1). Таким чином, вибір структури даних має базуватися не на звичних рішеннях, а на конкретних вимогах до швидкості пошуку, частоти змін і ефективного використання пам’яті.

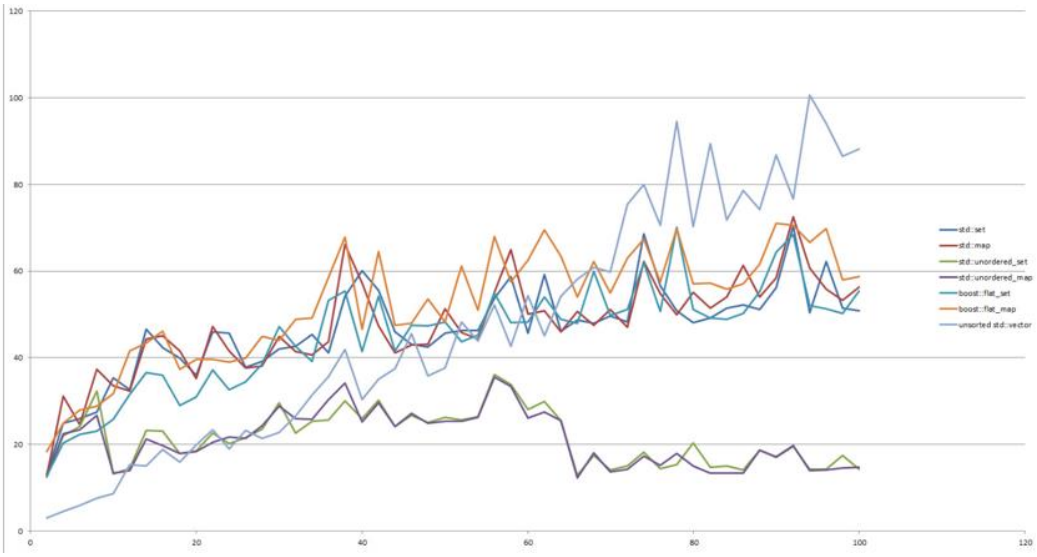

Рис. 3. Порівняння швидкості пошуку в структурах даних std та boost

Стаття Поля-Вірака Кхуонга та Пета Моріна [9] шукає найкращий спосіб зберігання даних у масиві в RAM для швидкого пошуку наступника. Розглядаються різні структури: відсортований масив, макет Ейцінгера, B-дерева та ван Емде Боаса. Автори доходять висновка, що для малих масивів найкращим є відсортований масив та бінарний пошук, а для великих — несподівано найшвидшим виявляється макет Ейцінгера (Рисунок 4). Це стає можливим, тому що найшвидші реалізації на C++ використовують дві ключові оптимізації процесора: відсутність розгалужень та попередню вибірку. Звичайні if/else в пошуку викликають інструкції переходу.

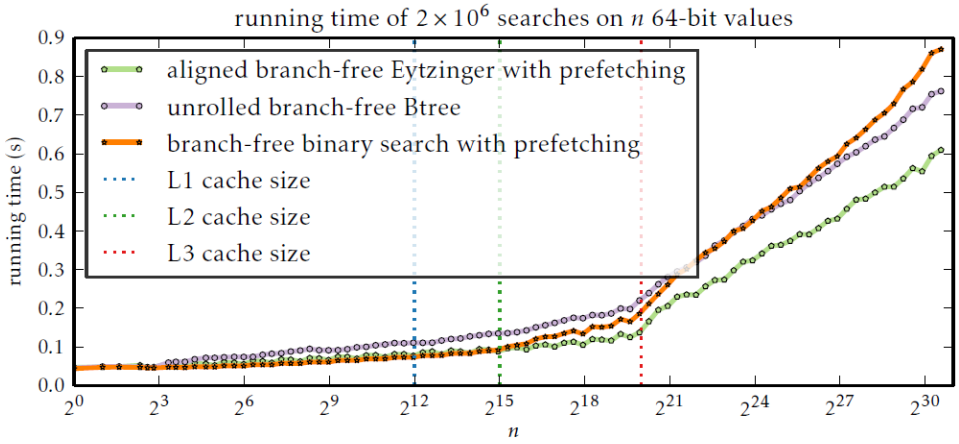

Рис. 4. Порівняння швидкості пошуку в масиві, макеті Ейцінгера та B-дереві

Якщо процесор неправильно передбачає результат порівняння (branch misprediction), виникає велика затримка. Інструкція CMOV виконує пересилання даних залежно від результату порівняння, але без зміни потоку виконання (без стрибка). Це уникає штрафів за помилкове передбачення. Для уникнення кеш промахів в свою чергу використовується інструкція PREFETCH, яке дає команду процесору завантажити потрібні дані з RAM у кеш заздалегідь. Якщо дані встигають завантажитись до моменту використання, затримка приховується. Це особливо добре працює для структури Ейцінгера, де адреси "дітей" передбачувані, але також може застосовуватись і для звичайного масиву, що показано на Рисунку 5.

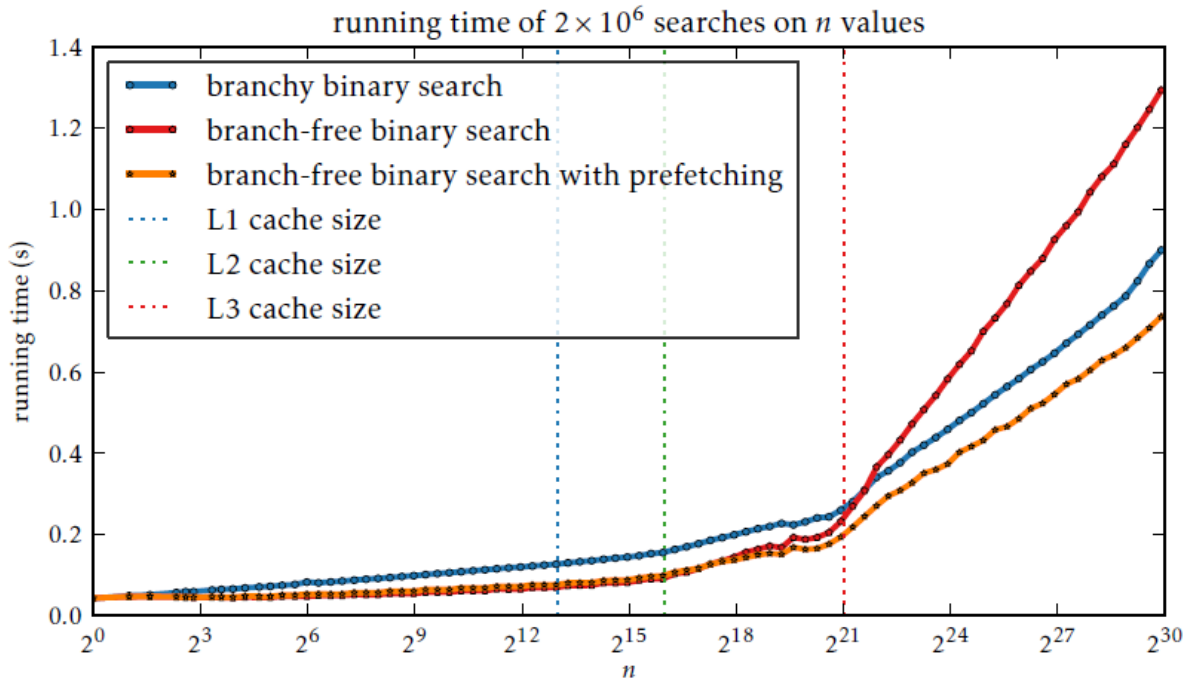

Рис. 5. Вплив відсутності розгалужень та попередньої вибірки на швидкості пошуку в відсортованому масиві

Розвиток багатоядерних процесорів зробив багатопотоковість надзвичайно привабливою технологією для підвищення продуктивності програмного забезпечення. Основна перевага полягає в можливості розділити складну задачу на менші частини, які виконуються паралельно на різних ядрах процесора. Це дозволяє значно прискорити обчислювально інтенсивні операції та ефективніше використовувати доступні апаратні ресурси, адже замість одного ядра працює одразу декілька. Однак, як показує аналіз, зокрема наведений у статті Гуан Джена [10] щодо продуктивності різних моделей паралельного програмування для C++, використання багатьох потоків не завжди є виправданим. Головна причина криється в накладних витратах, пов'язаних із створенням, синхронізацією та управлінням потоками. Ці операції самі по собі потребують часу та ресурсів.

Якщо обчислювальна задача занадто мала, ці накладні витрати можуть нівелювати весь потенційний виграш від паралелізму або навіть призвести до сповільнення (Рисунок 6). Наочним прикладом є множення матриць розміром 8x8. Згідно з дослідженням, для такої невеликої задачі час, витрачений на

управління потоками, виявився значно більшим за час, зекономлений на паралельних обчисленнях. У результаті всі багатопотокові реалізації працювали суттєво повільніше, ніж звичайний послідовний код. Це свідчить про те, що лише при досягненні певного, досить значного, обсягу обчислень (як у випадку матриць 1000x1000 і більше) виграш від паралельного виконання починає впевнено переважати над накладними витратами.

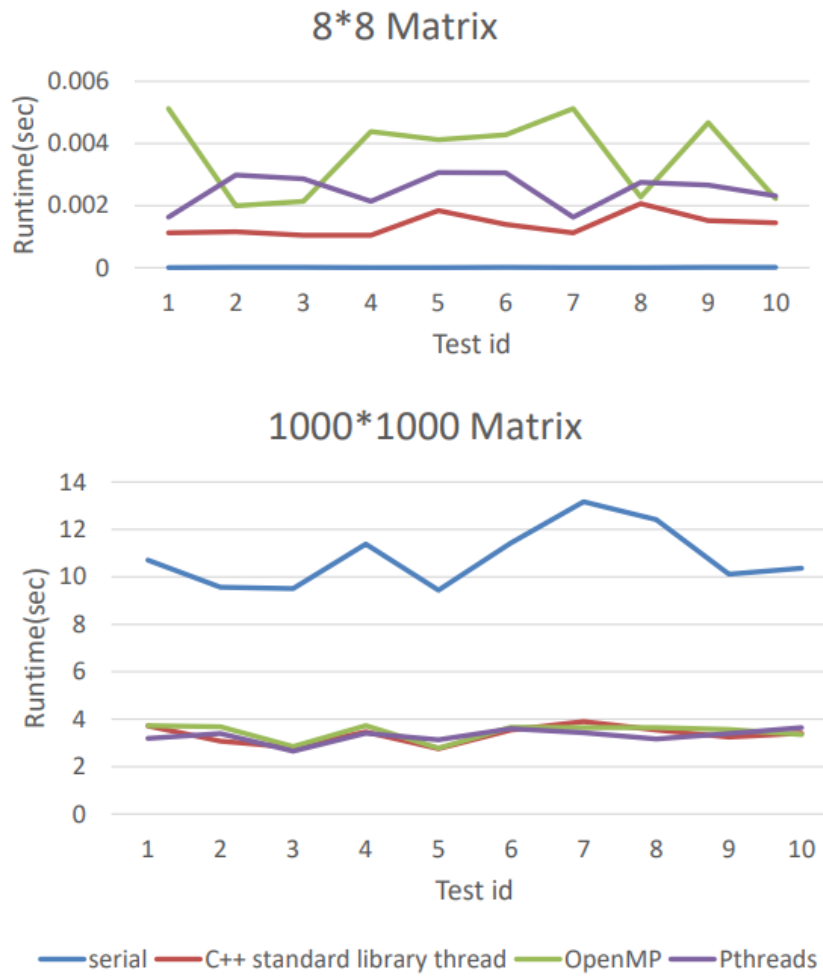

Рис. 6. Порівняння продуктивності багатопоточного множення матриць різного розміру

Виконання однієї інструкції для кількох значень одночасно значно скорочує кількість тактів процесора, необхідних для виконання операцій. Це особливо ефективно в матричних обчисленнях, цифровій обробці сигналів (DSP), комп'ютерному зорі та фізичних симуляціях. Багато сучасних процесорів мають SIMD-розширення (SSE, AVX, NEON), але без відповідного коду вони простоюють або не використовуються на повну потужність [11]. Використання SIMD дозволяє максимально задіяти доступні обчислювальні блоки процесора. SIMD працює з блоками даних, що часто розташовані послідовно в пам'яті. Це підвищує кеш-ефективність, зменшуючи частоту звернень до оперативної пам'яті, що є відносно повільною операцією. Класичний код із великою кількістю умовних операторів (if-else) може викликати проблеми з передбаченням переходів у процесорному конвеєрі. SIMD допомагає мінімізувати такі розгалуження, виконуючи обчислення на всіх елементах одночасно. SIMD

ідеально поєднується з багатопотоковістю (SIMD + Multi-threading), дозволяючи не лише розпаралелювати обчислення між ядрами процесора, а й ефективно обробляти кожен потік всередині ядра за допомогою векторних інструкцій. На Рисунку 7 можна побачити можливий приріст ефективності від використання SIMD.

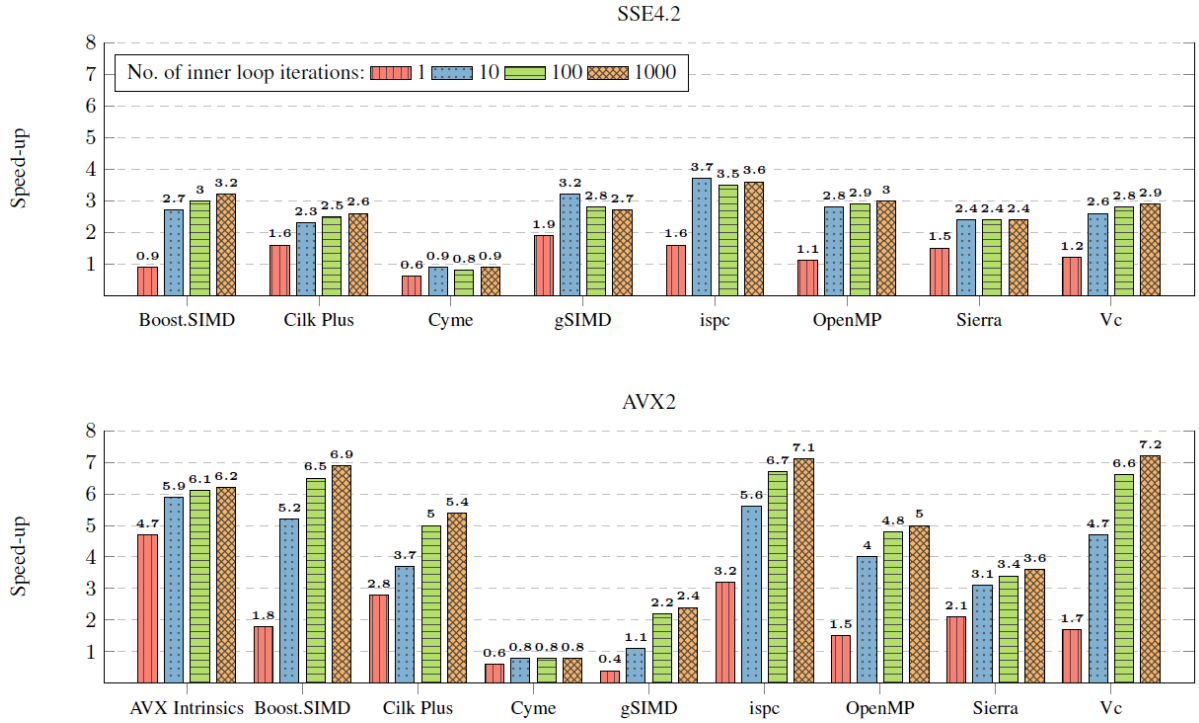

Рис. 7. Значне зростання продуктивності при використанні SIMD [12]

Висновок

Для досягнення максимальної оптимізації коду необхідно враховувати кілька ключових аспектів:

1. Розмір та об’єм оброблюваних даних, що дозволяє ефективніше використовувати пам’ять і вибирати оптимальні структури даних.
2. Зменшення кількості кеш-промахів, наприклад, працюючи з даними, що розташовані послідовно в пам’яті, та використовуючи підходи, орієнтовані на локальність доступу.
3. Зведення до мінімуму розгалуження в коді, оскільки непередбачувані умовні переходи можуть негативно впливати на продуктивність процесора через порушення конвеєра виконання команд.
4. Використання багатопоточності не гарантує покращення продуктивності, але за відповідних умов може в рази скоротити час виконання
5. Використання можливостей SIMD для виконання однотипних операцій над векторами даних, що дозволяє значно прискорити обчислення внаслідок паралельної обробки.
6. Використання графічних процесорів надає значний потенціал для прискорення обчислень у сучасних гетерогенних системах, особливо для завдань, що добре розпаралелюються. Однак, як показує аналіз, ефективність перенесення на графічний процесор не є універсальною і сильно залежить від конкретного обчислювального ядра, покоління архітектури процесорів, а також

від накладних витрат, зокрема на передачу даних.

7. Кожна оптимізація вимагає нових вимірів продуктивності, тому що не можна спиратись лише на теоретичну складність алгоритму O(n).

Список літератури

1. Theis T. N., Wong H. S. P. The end of Moore's law: A new beginning for information technology. 2017.

2. Berger E. Performance Matters [Електронний ресурс]. – 2020. Режим доступу: https://github.com/CppCon/CppCon2020/blob/main/Presentations/performance_matters/performance_matters__emery_berger__cppcon_2020.pdf

3. Swapnil Phalke, Yogita Vaidya, Shilpa Metkar. Big-O Time Complexity Analysis Of Algorithm. – 2022.

4. Carruth, C. Efficiency with Algorithms, Performance with Data Structures [Електронний ресурс]. 2014. Режим доступу: https://github.com/CppCon/CppCon2014/blob/master/Presentations/Efficiency%20with%20Algorithms%2C%20Performance%20with%20Data%20Structures/Efficiency%20with%20Algorithms%2C%20Performance%20with%20Data%20Structures%20-%20Chandler%20Carruth%20-%20CppCon%202014.pdf

5. Pesterev, A., Zeldovich, N., Morris, R. T. Locating Cache Performance Bottlenecks Using Data Profiling. – 2010.

6. Hennessy, J. L., Patterson, D. A. Computer Architecture: A Quantitative Approach. 3rd ed. – 2002.

7. Chikin, A., Amaral, J. N., Ali, K., Tiotto, E. Toward an Analytical Performance Model to Select between GPU and CPU Executio. – 2019.

8. Meyers S. Should you be using something instead of what you should use instead? [Електронний ресурс]. – 2015. Режим доступу: https://scottmeyers.blogspot.com/2015/09/should-you-be-using-something-instead.html

9. Khuong, P.-V., Morin, P. Array Layouts For Comparison-based Searching. – 2017.

10. Zeng, G. Performance analysis of parallel programming models for C++. – 2023.

11. Mölle,r R. Design of a low-level template SIMD library. – 2016.

12. Pohl, A., Cosenza, B., Alvarez Mesa, M., Chi, C. C., Juurlink B. An Evaluation of Current SIMD Programming Models for C++. – 2020.

References

1. Theis T. N., Wong H. S. P. The end of Moore's law: A new beginning for information technology. 2017.

2. Berger E. Performance Matters [Elektronnij resurs]. – 2020. Rezhim dostupu: https://github.com/CppCon/CppCon2020/blob/main/Presentations/performance_matters/performance_matters__emery_berger__cppcon_2020.pdf

3. Swapnil Phalke, Yogita Vaidya, Shilpa Metkar. Big-O Time Complexity Analysis Of Algorithm. – 2022.

4. Carruth, C. Efficiency with Algorithms, Performance with Data Structures [Elektronnij resurs]. 2014. Rezhim dostupu: https://github.com/CppCon/CppCon2014/blob/master/Presentations/Efficiency%20with%20Algorithms%2C%20Performance%20with%20Data%20Structures/Efficiency%20with%20Algorithms%2C%20Performance%20with%20Data%20Structures%20-%20Chandler%20Carruth%20-

%20CppCon%202014.pdf

5. Pesterev, A., Zeldovich, N., Morris, R. T. Locating Cache Performance Bottlenecks Using Data Profiling. – 2010.

6. Hennessy, J. L., Patterson, D. A. Computer Architecture: A Quantitative Approach. 3rd ed. – 2002.

7. Chikin, A., Amaral, J. N., Ali, K., Tiotto, E. Toward an Analytical Performance Model to Select between GPU and CPU Executio. – 2019.

8. Meyers S. Should you be using something instead of what you should use instead? [Elektronnij resurs]. – 2015. Rezhim dostupu: https://scottmeyers.blogspot.com/2015/09/should-you-be-using-something-instead.html

9. Khuong, P.-V., Morin, P. Array Layouts For Comparison-based Searching. – 2017.

10. Zeng, G. Performance analysis of parallel programming models for C++. – 2023.

11. Molle,r R. Design of a low-level template SIMD library. – 2016.

12. Pohl, A., Cosenza, B., Alvarez Mesa, M., Chi, C. C., Juurlink B. An Evaluation of Current SIMD Programming Models for C++. – 2020.

Надійшла до редакції 13.05.2025, розглянута на редколегії 13.05.2025.

A Survey of Contemporary Methods for Performance Optimization of Computer Programs

This article provides a comprehensive analysis of modern approaches to optimizing the performance of computer programs in the context of the increasing complexity of computational tasks and the growing volume of data to be processed. The main hardware and software factors influencing the efficiency of program execution are considered, including the architectural features of central and graphics processors, the organization of multi-level memory systems, support for vectorized computations, and mechanisms for parallel data processing. Special attention is given to the impact of algorithm and data structure selection on the performance of software, taking into account memory locality, cache miss avoidance, and the minimization of data transfer volumes across different levels of the memory hierarchy. Typical challenges related to overhead from thread creation and synchronization, as well as delays caused by branching in code execution, are analyzed. Principles for the effective use of multicore processors and heterogeneous platforms are discussed, highlighting the critical role of optimal task distribution between CPUs and GPUs for achieving high performance. The article emphasizes the necessity of a systematic approach to optimization, involving not only the application of individual techniques but also their comprehensive integration, considering their combined effect on overall performance. The importance of empirical evaluation of optimization results through real execution time measurements, performance profiling, and resource utilization analysis is underscored. General recommendations are formulated for the development of high-performance software systems, aimed at minimizing memory access delays, optimizing conditional branch handling, efficiently utilizing hardware accelerators, and ensuring effective parallelism. The findings presented may be valuable both for practicing software developers working on application performance improvement and for researchers in the field of high-performance computing, contributing to the advancement of new optimization methodologies in the evolving

landscape of computational architectures.

Keywords: optimization, performance, multithreading, processor caches, branch prediction, pipelining, prefetching, SIMD.

Відомості про авторів:

Каратанов Олександр Володимирович – канд. техн. наук, доцент, доцент каф. № 105 «Інформаційні технології проектування», Національний аерокосмічний університет «Харківський авіаційний інститут», Харків, Україна, ORCID: 0000-0002-6451-7901.

Тіхвинський Олексій Вадимович – аспірант каф. № 105 «Інформаційні технології проектування», Національний аерокосмічний університет «Харківський авіаційний інститут», Харків, Україна, ORCID: 0009-0001-6474-2041.

About the Authors:

Oleksandr Karatanov – PhD, Associate Professor, Associate Professor of Department «Information Technology Design», National Aerospace University "Kharkiv Aviation Institute", Kharkiv, Ukraine, ORCID: 0000-0002-6451-7901.

Oleksii Tikhvynskyi – Postgraduate of Department «Information Technology Design», National Aerospace University "Kharkiv Aviation Institute", Kharkiv, Ukraine, ORCID: 0009-0001-6474-2041.