МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

НАРАЛЬНИК БОГДАН ЮРІЙОВИЧ

Допускається до захисту:
В. о. завідувача кафедри
інформаційних технологій,
к. т. н, доцент
_____О.В. Зелінська
«____»_____ 2024 р.

ВИЗНАЧЕННЯ ОПТИМАЛЬНОГО РОЗМІРУ ВХІДНОГО НАБОРУ ДАНИХ ДЛЯ ЕФЕКТИВНОГО ПРИСКОРЕННЯ БАЗОВИХ МАТРИЧНИХ ОПЕРАЦІЙ НА GPU

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна(магістерська) робота

Керівник:
П.К. Ніколюк , професор кафедри
інформаційних технологій
д-р фіз.-мат. наук, професор
_____
(підпис)

Оцінка:_____/ _____/ _____
(бали за шкалою СКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця 2024

АНОТАЦІЯ

Наральник Б.Ю. Визначення оптимального розміру вхідного набору даних для ефективного прискорення базових матричних операцій на GPU.

Спеціальність 122 «Комп’ютерні науки», Донецький національний університет імені Василя Стуса, Вінниця, 2024.

Під час виконання роботи було створено програмний інструмент для визначення оптимальних розмірів вхідних даних, за яких доцільно використовувати GPU для розв’язання задач із матрицями. У тестуванні застосовувалися два процесори (CPU) та два графічні процесори (GPU). Реалізація алгоритмів для CPU була виконана мовою C++, а для GPU – на платформі CUDA. У результаті дослідження з'ясувалося, що неможливо дати універсальну відповідь для всього класу задач із матрицями, а також були вивчені фактори, що впливають на продуктивність роботи на різних типах обчислювальних пристроїв.

Ключові слова: GPU, матричні операції, CUDA, продуктивність, оптимізація, розмір матриць, паралельні обчислення, множення матриць.

91 с., 72 рис., 1 дод., 50 джерел.

ABSTRACT

Naralnik B.Yu. Determining the optimal size of the input data set for effective acceleration of basic matrix operations on GPU.

Specialty 122 "Computer Science", Vasyl Stus Donetsk National University, Vinnytsia, 2024.

During the work, a software tool was created to determine the optimal size of the input data, at which it is advisable to use GPU for solving matrix problems. Two processors (CPU) and two graphics processors (GPU) were used in the testing. The implementation of the algorithms for the CPU was performed in C++, and for the GPU - on the CUDA platform. As a result of the study, it became clear that it is impossible to give a universal answer for the entire class of matrix problems, and the factors affecting the performance of work on different types of computing devices were also studied.

Keywords: GPU, matrix operations, CUDA, performance, optimization, matrix size, parallel computing, matrix multiplication.

91 pp., 72 fig., 1 appendices, 50 sources.

ЗМІСТ

ВСТУП

Сучасні комп’ютери та цифрові пристрої здатні виконувати складні обчислення для широкого спектра задач, завдяки чому щороку вдосконалюються програмні інструменти та реалізуються дедалі складніші алгоритми. Однак це потребує значних ресурсів, тому питання ефективного розподілу й раціонального використання цих ресурсів стає особливо важливим. Основними компонентами для виконання обчислень є центральні процесори (CPU) та графічні процесори (GPU). Кожен із них має свої сильні сторони й оптимізований для розв’язання певних типів задач, що робить грамотний вибір між CPU та GPU ключовою темою в галузі паралельних обчислень.

Для вибору оптимального пристрою враховуються особливості задачі, тип і обсяг даних. У цій роботі акцент зроблено на визначенні мінімального обсягу даних, за якого для задач, орієнтованих на SIMD, доцільніше використовувати GPU, ніж CPU. Хоча GPU традиційно вважаються більш ефективними для задач із високим рівнем паралелізму, вибір не завжди однозначний. Як буде показано в подальших розділах, продуктивність CPU постійно зростає, тоді як GPU мають певні обмеження, зокрема нижчу тактову частоту ядер і затримки, пов’язані з передачею даних між керуючим пристроєм і графічною картою.

Тому ретельний аналіз задач, а також обґрунтований вибір методів і обчислювальних пристроїв є необхідними для досягнення максимальної ефективності.

РОЗДІЛ 1 ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ У СУЧАСНИХ ТЕХНОЛОГІЯХ: ПОРІВНЯННЯ АРХІТЕКТУР CPU ТА GPU

У цьому розділі будуть розглянуті особливості паралельних обчислень у сучасних пристроях, їх значення та застосування в різних задачах. Також буде детально описано архітектурні відмінності між CPU та GPU, а також їх вплив на особливості роботи й продуктивність цих пристроїв.

1.1 Паралелізм як частина архітектури обчислювальних машин

У 1971 році компанія Intel представила перший комерційний мікропроцесор Intel 4004[1]. Це був 4-розрядний процесор із частотою 740 кГц, здатний виконувати близько 92000 операцій за секунду. Мікропроцесори того часу могли виконувати лише одну інструкцію за раз, оскільки використовували послідовну архітектуру.

Обмеження продуктивності таких систем стали очевидними з ростом складності завдань. Це стимулювало пошук нових шляхів збільшення швидкості обробки даних, що призвело до впровадження паралелізму.

У 1965 році Гордон Мур передбачив, що кількість транзисторів, які можна розмістити на інтегральній схемі, буде подвоюватися приблизно кожні 18–24 місяці[2], при цьому вартість одного транзистора зменшуватиметься. Це стало основою так званого "закону Мура", хоча він не є фізичним законом, а радше емпіричним спостереженням. Водночас це покращення було пов’язане не із паралельними обчисленнями, а з принципами, сформульованими Робертом Деннардом у 1974 році[4] . У своїй роботі він показав, що напруга й струм, необхідні для роботи транзистора, зменшуються пропорційно до його розмірів. Це означало, що менші транзистори потребують менше енергії та можуть працювати швидше, завдяки чому зростала тактова частота процесорів і, відповідно, їхня швидкодія.

Однак, хоча закон Мура залишається певною мірою актуальним[5] (рис. 1), закон масштабування Деннарда втратив свою дієвість (рис. 2). Це обмеження

пояснює, чому подальше зростання швидкодії процесорів вимагає альтернативних підходів, зокрема використання паралельних обчислень.

На рис 1 можна побачити, що частота роботи процесорів ПК перестала стрімко зростати приблизно у 2006 році[6]. Подальше зменшення розмірів транзисторів (наноелектроніка) призвело до появи ефекту витоку струму, коли електрони "просочуються" крізь тонкий ізоляційний шар. Це збільшує енергоспоживання та тепловиділення[7].

Крім того, зменшення розмірів транзисторів має фізичні обмеження. Зокрема, цьому процесу заважає ефект тунелювання, який виникає при надмірно малих розмірах транзисторів.

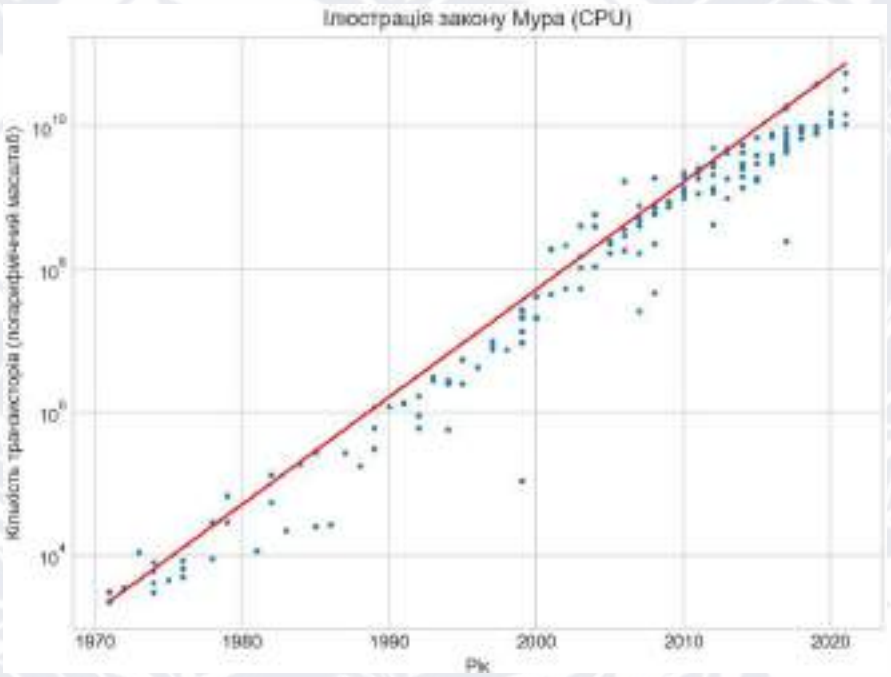

Рис. 1 – Графічна демонстрація закону Мура на основі з джерела [5]

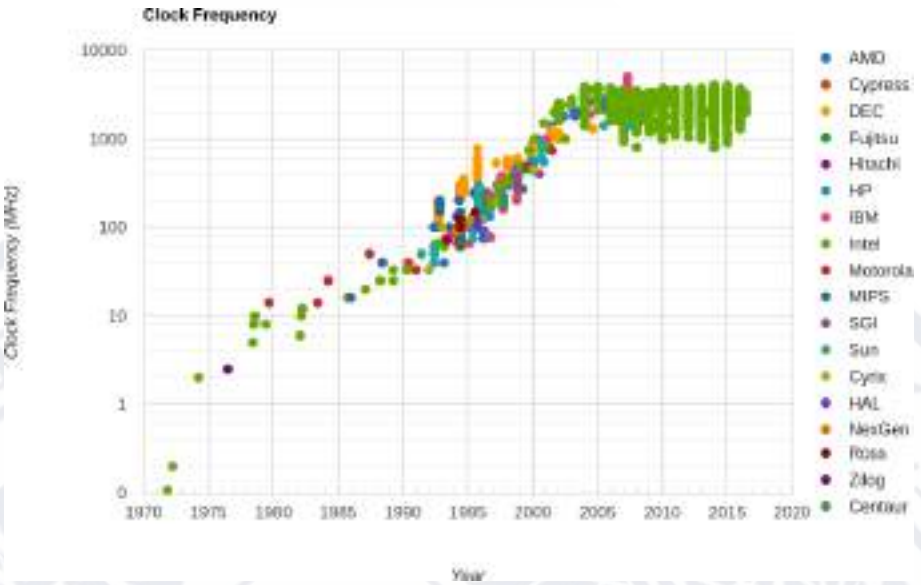

Рис. 2 – Порівняння росту частоти процесорів за 1970-2020 [6]

У 2001 році компанія IBM представила процесор [8], який став першим комерційним процесором із двома обчислювальними ядрами на одному кристалі. Він був орієнтований на сервери та робочі станції, значно підвищивши продуктивність багатозадачних систем. Цей запуск започаткував еру багатоядерних технологій у процесорах.

Тим часом паралелізм став фундаментальною частиною комп'ютерної архітектури — від обмежень ранніх мікропроцесорних реалізацій до складних багатоядерних систем сучасності. Технологічний прогрес відображає більш загальне прагнення людини до швидшого та потужнішого обчислення, яке реалізується в паралелізмі на всіх рівнях, починаючи від настільних комп'ютерів і закінчуючи високопродуктивними обчисленнями та багатоядерними системами[41].

Окрім побутових та мобільних пристроїв, багатоядерні процесори широко використовуються на серверах, що дозволяє запускати декілька служб одночасно. Це також підкреслює, що центральні процесори більше не є єдиними пристроями, оптимізованими для паралельних обчислень.

Крім того, було б помилково стверджувати, що паралельні обчислення розвиваються виключно завдяки збільшенню кількості обчислювальних ядер, оскільки ефективність паралельних обчислень залежить не лише від кількості

ядер, але й від здатності правильно розподіляти завдання між ядрами, а також від оптимізації програмного забезпечення. Збільшення кількості ядер без належної підтримки з боку програмного забезпечення може не призвести до значного покращення продуктивності. Крім того, паралельні обчислення потребують ефективної комунікації між ядрами і швидкої передачі даних, що може стати обмеженням при масштабуванні. Тому важливими аспектами є також архітектурні рішення, алгоритми та програмні оптимізації, що дозволяють реалізувати ефективне паралельне виконання.

Паралелізм є невід'ємною частиною всіх рівнів сучасної архітектури комп’ютерів, включаючи мікроархітектуру процесора [9]. Раніше для виконання програм процесори проходили через чотири основні етапи:

1. Вибірка інструкції (Fetch) – процесор отримує інструкцію з пам'яті.
2. Декодування інструкції (Decode) – процесор розшифровує отриману інструкцію, щоб зрозуміти, яку операцію потрібно виконати.
3. Виконання (Execute) – процесор виконує операцію, яку вказує інструкція (наприклад, арифметичну чи логічну операцію).
4. Запис результату (Write-back) – результат виконання операції записується назад у реєстри або пам'ять.

Ці етапи були основою класичної архітектури процесорів, яка застосовувалася для одноетапного виконання інструкцій. Вони стали базою для створення більш складних архітектур, таких як конвеєрна обробка, що дозволяє виконувати інструкції паралельно на різних етапах.

Затримки на етапі пошуку даних, що виникали на другому кроці, призводили до значних затримок у виконанні програм. Це сприяло розвитку досліджень, спрямованих на зменшення таких затримок і, як результат, на підвищення ефективності програм. Проте з часом головною метою стало створення процесорів, здатних виконувати кілька інструкцій одночасно.

Такі процесори могли б виявляти та використовувати паралелізм, властивий виконанню інструкцій, що дозволяло б досягти ще більшої швидкості виконання програм, незалежно від частоти процесора чи обсягу пам'яті.

Таким чином, з часом різноманітні паралельні рішення поступово переросли в сучасні обчислювальні системи, які можна класифікувати за таксономією Флінна. Ця класифікація, запропонована ще в 1966 році, й до сьогодні залишається актуальною [10]. Таксономія Флінна (або класифікація архітектур обчислювальних систем) була запропонована Джоном Флінном у 1966 році і є одним із найважливіших способів класифікації комп'ютерних архітектур на основі кількості потоків інструкцій та даних, які система може обробляти одночасно. Флінн визначив чотири основні категорії:

1) Один потік інструкцій, один потік даних (Single Instruction, Single Data – SISD): Це класична архітектура для одноядерних процесорів, де кожен процесор виконує одну інструкцію за раз, обробляючи один потік даних.

2) Один потік інструкцій, кілька потоків даних. (Single Instruction, Multiple Data – SIMD): У цій архітектурі всі обчислювальні одиниці виконують одну й ту саму інструкцію одночасно, але над різними наборами даних. Це дозволяє ефективно виконувати операції над великими масивами даних, як у випадку з обробкою зображень або науковими обчисленнями.

3) Кілька потоків інструкцій, один потік даних (Multiple Instructions, Single Data – MISD): У цій архітектурі кілька процесорів можуть одночасно виконувати різні інструкції над одним потоком даних. Це дуже рідко використовувана архітектура, і на практиці такі системи рідко застосовуються.

4) Кілька потоків інструкцій, кілька потоків даних (Multiple Instructions, Multiple Data – MIMD): Ця архітектура підтримує виконання різних інструкцій на різних наборах даних одночасно, що дозволяє обробляти різні завдання паралельно. Це найбільш гнучка архітектура, яка є основою сучасних багатоядерних та мультипроцесорних систем.

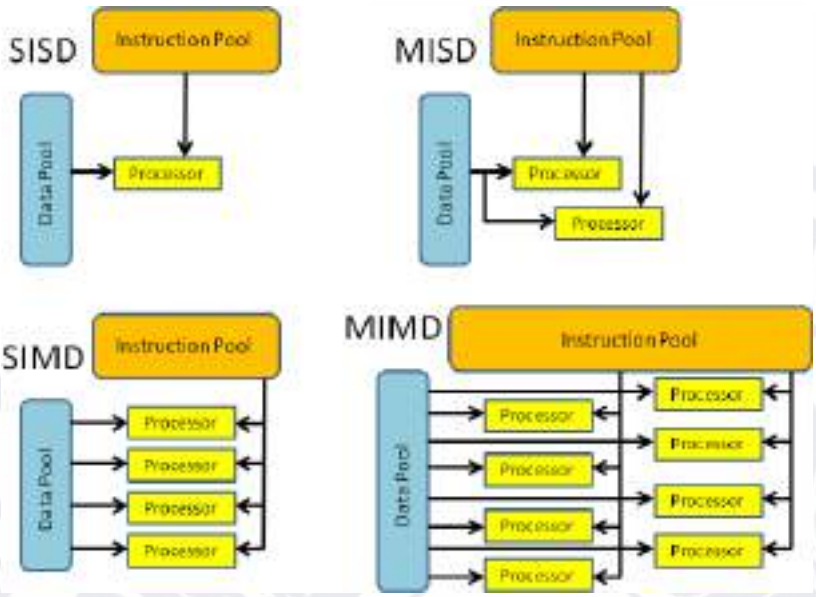

Рис. 3 – Класифікація Флінна [11]

Згодом було додано дві ще підкатегорії, що стосуються MIMD-машин (рис. 4).

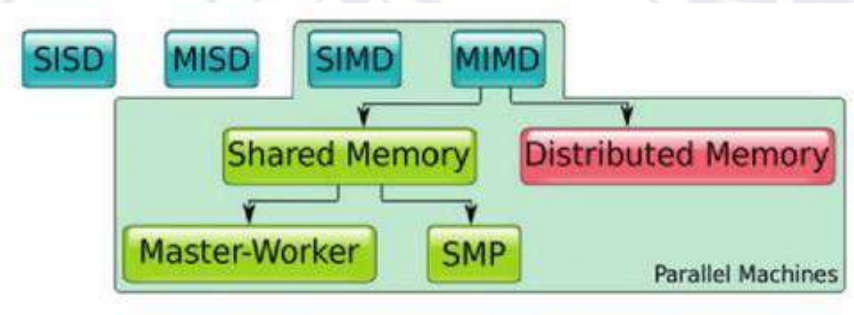

Рис. 4 – Розширена таксономія Флінна з [12]

а) MIMD з спільною пам'яттю (shared-memory MIMD): архітектура машин, яка передбачає наявність загального простору спільної пам'яті. Спільна пам’ять спрощує обмін даними між центральними процесорами з мінімальними накладними витратами, але водночас є потенційним "вузьким місцем" (ботлнеком), що обмежує масштабованість системи. Це можна вирішити, розподіливши пам’ять між процесорами, так, щоб кожен з них мав власну локальну частину пам’яті[3].

Таким чином, центральний процесор отримує швидший доступ до своєї пам’яті, водночас зберігаючи можливість доступу до пам’яті інших процесорів,

але з більшими затримками. Така схема не змінює адресації, оскільки адресний простір залишається спільним для всіх процесорів.

б) MIMD з розподіленою пам'яттю (або shared-nothing MIMD): процесори в такій архітектурі обмінюються даними через повідомлення. Хоча цей спосіб комунікації має високу вартість, такі системи здатні масштабуватися майже без обмежень, за винятком факторів розміру та споживання енергії.

Машини зі спільною пам’яттю також можна поділити на підкатегорії master-worker та симетричні багатопроцесорні платформи[31].

У симетричних багатопроцесорних платформах всі процесори рівноправні і можуть виконувати будь-яку програму, включаючи системне та користувацьке програмне забезпечення. У системі master-worker певні процесори призначені для виконання спеціалізованих завдань, і їх можна розглядати як співпроцесори. Системи з графічними процесорами (GPU) також можуть потрапляти в цю категорію, хоча більшість високопродуктивних платформ GPU мають окремі області пам'яті для CPU і GPU[42]. В таких випадках вони не є платформами зі спільною пам'яттю, незважаючи на сучасні дослідження, що спрямовані на вдосконалення процесу передачі даних між цими двома областями пам'яті.

Таким чином, розвиток архітектур пам'яті в обчислювальних системах забезпечує нові можливості для виконання паралельних обчислень і вдосконалення масштабованості в різних застосуваннях.

1.2 Порівняння архітектур центральних та графічних процесорів

У цій роботі буде розглянуто потенціал паралельних обчислень для центральних і графічних процесорів. Для цього важливо розглянути принципи їхньої роботи. В загальному схему ядра центрального процесора представлено на рис. 5.

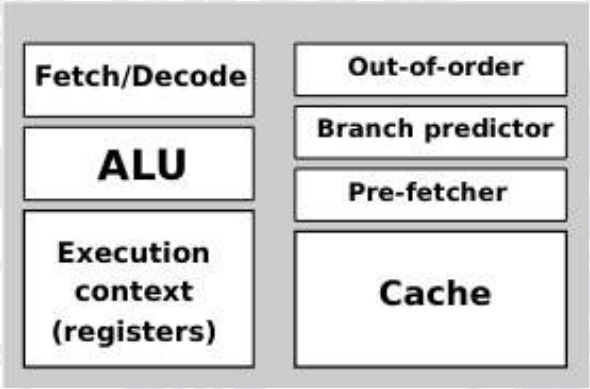

Рис. 5 – Схема ядра процесора з [9]

Типове ядро центрального процесора (CPU) включає в себе кілька основних компонентів, серед яких: **арифметико-логічний пристрій (ALU, Arithmetic Logic Unit), контролер (Control Unit), реєстри (registers), кеш пам'ять (cache)** [32,43].

Арифметико-логічний пристрій виконує арифметичні та логічні операції, такі як додавання, віднімання, множення, ділення, порівняння значень тощо.

Контролер координує роботу всіх інших компонентів процесора, інтерпретує інструкції з пам'яті та визначає порядок їх виконання.

Реєстри це швидкі вбудовані сховища даних, які використовуються для тимчасового зберігання інтермедійних результатів та операндів, необхідних для виконання інструкцій.

Кеш пам'ять це невелика, але дуже швидка пам'ять, що знаходиться безпосередньо всередині або дуже близько до ядра процесора, призначена для зберігання часто використовуваних даних та інструкцій, щоб зменшити затримки при доступі до основної пам'яті.

Для покращення продуктивності при виконанні в один потік такі процесори з одним ядром використовують технології позачергового виконання інструкцій (коли інструкції виконуються в порядку їх готовності, а не по черзі) та передбачення переходів (коли визначається, чи буде умовний перехід виконано, щоб уникнути затримок).

Проте, навіть за цих умов, всі виконуючі одиниці процесора не можуть функціонувати без інструкцій та операндів, що зберігаються в основній пам'яті (див. рис. 6).

Передача інструкцій та операндів між процесором і основною пам'яттю є ресурсоємним і часозатратним процесом, тому для підвищення ефективності використовуються кеш-пам'яті. Кеші працюють за принципами просторової та тимчасової локальності[44]. **Арифметико-логічний пристрій** (АЛП) і логіка вибірки та декодування працюють швидко і з низьким енергоспоживанням, не потребуючи значних ресурсів. Однак побудова кешу вимагає великої кількості транзисторів, що робить його дорогим і однією з найбільш енергозатратних частин центрального процесора.

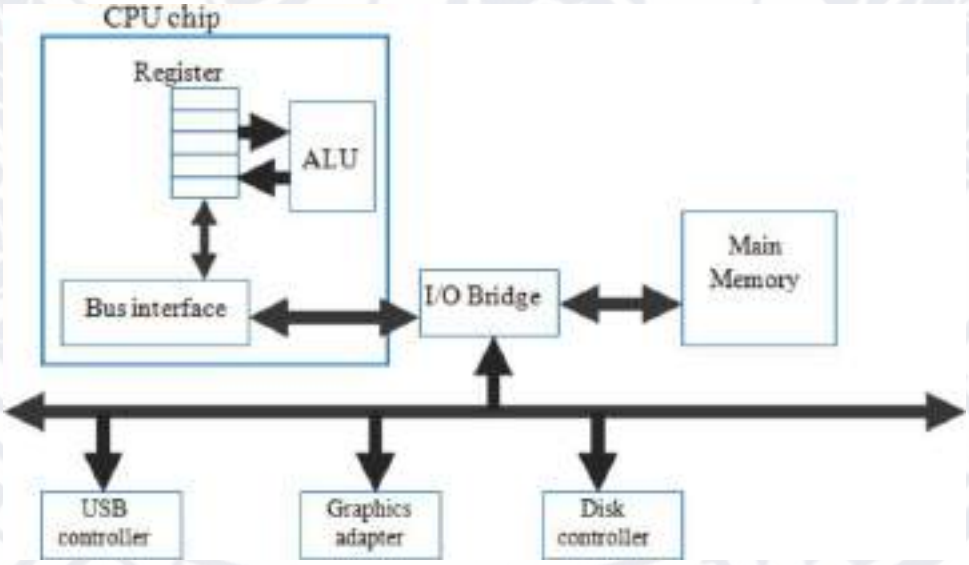

Рис. 6 – Схема центрального процесора і пам’яті [13]

З підвищенням швидкодії процесорів завдяки багатоядерним архітектурам, більш поширеною на сьогодні є схема, зображена на рис. 7. Паралельно з розвитком центральних процесорів, область графічних обчислень зазнала

значних змін. Наприкінці 80-х і на початку 90-х років, зростання популярності операційних систем з графічним інтерфейсом призвело до появи нового типу процесорів[40].

У той час компанія Silicon Graphics активно пропагувала використання тривимірної графіки в таких сферах, як наука, військова справа та візуальні ефекти для кіно.

В 1992 році вона також випустила бібліотеку OpenGL, що зробила 3D-графіку доступною для широкого кола користувачів[2].

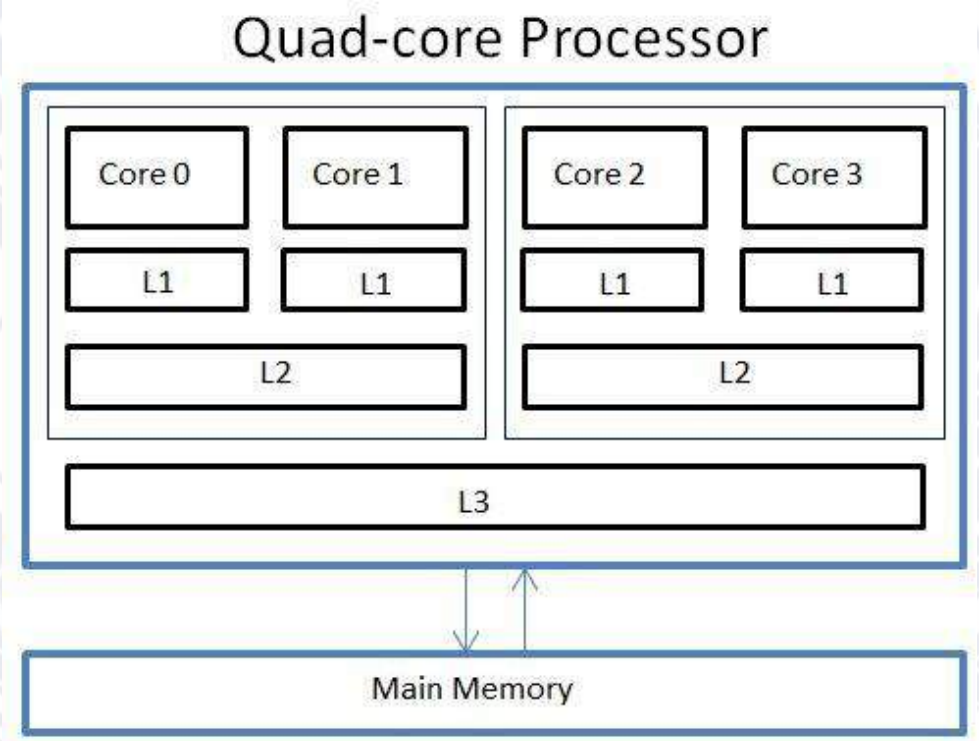

Рис. 7 – Схема центрального процесора і пам’яті [14]

Ця технологія мала стати платформонезалежним інструментом для розробки програм із 3D-графікою. З часом OpenGL почала використовуватись не лише в обраних сферах, але й для створення програм для широкого кола користувачів. OpenGL спочатку була розроблена для роботи з 3D графікою, але з часом її можливості були розширені і для 2D графіки, що дозволило використовувати цю технологію не лише для розробки ігор та графічних додатків, а й для більш загальних застосувань, наприклад, в інтерфейсах користувача (UI) і візуалізації даних. У 1999 році NVIDIA представила графічний пристрій GeForce 256, який можна вважати першим GPU, з мікросхемою для

обчислення трансформацій (створення двовимірного зображення) і освітлення (зміна кольору поверхонь залежно від освітлення сцени) [15].

Основна відмінність між процесором і графічним процесором полягає в тому, що центральний процесор має до 32 ядер, оптимізованих для послідовної обробки, тоді як графічний процесор має кілька або навіть тисячі ядер для великомасштабної паралельної обробки завдань. [39].

Ці ядра значно відрізняються від ядер у центральних процесорах загального призначення, оскільки вони не мають таких складних механізмів, як кеші, передвибірка (cache pre-fetcher), логіка позачергового виконання або передбачення переходів. Якщо прибрати ці елементи зі схеми ядра, наведеної на рис. 5, то можна отримати більш спрощену архітектуру, що ілюструється на рис. 8.

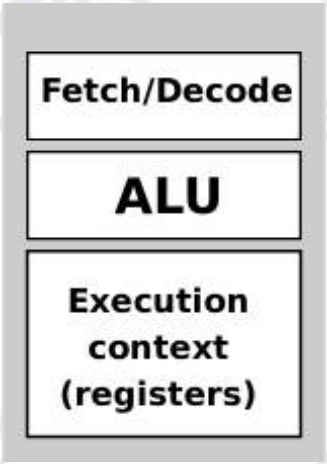

Рис. 8 – Скорочена схема процесора [9]

Тим не менш, це не впливає на виконання деяких задач. Як показано в [9], код “add_vectors" для двох векторів на мові C, наведений у рис. 9, зрештою перетворюється на наступний асемблерний код, як показано в рис. 10.

```
void vectorAdd( float vecA, float *vecB, float *vecC ) {
    int tid = 0;
    while (tid < 128) {
        vecC[tid] = vecA[tid] + vecB[tid];
        tid += 1;
    }
}
```

Рис. 9 – Код операції доданку векторів на мові програмування С

```
    add r2,r0,r0; tid = 0
    add r3,r0,r0
    add r4,r0,r0
L1:
    lfp f1,r3(vecA) ;
    lfp f2,r3(vecB) ;
    addf fl,f1,f2 ;
    sfp f1,r3(vecC) ;
    addi. r3,r3, #4
    addi r2,r2, #1 ;
    slti r4,r2, #128 ;
    bne r4,L1 ;
```

Рис. 10 – Код операції доданку векторів на асемблері

Якщо регістр r0 є нульовим регістром, це очистить регістри r2 і r3, які зберігають лічильник `tid' і зсув для векторів `vecA' і `vecB' відповідно. Потім у циклі числа завантажуються з векторів, додаються та записуються як результат у `vecC`[37].

Лічильник `tid` збільшується, і якщо кількість виконань не перевищує 128, цикл повторюється. Тому, навіть без деяких елементів архітектури,

спрощене ядро здатне виконати цей код. Крім того, за допомогою паралелізму даних, цю задачу можна реалізувати, наприклад, на двох ідентичних спрощених ядрах, кожне з яких виконує однаковий набір інструкцій, але з відповідним розподілом масиву даних. Кожному з ядер буде призначено по 64 елементи для обробки, що дозволяє значно прискорити виконання операції[38].

Як видно з рисунка, інструкції залишаються тими ж, зміни стосуються лише лічильників та розподілу частин масиву даних. Цей підхід дозволяє паралельно обробляти більше даних, де кожен контекст виконання відповідає за свою ділянку, що веде до зменшення часу виконання задачі та покращення загальної продуктивності системи.

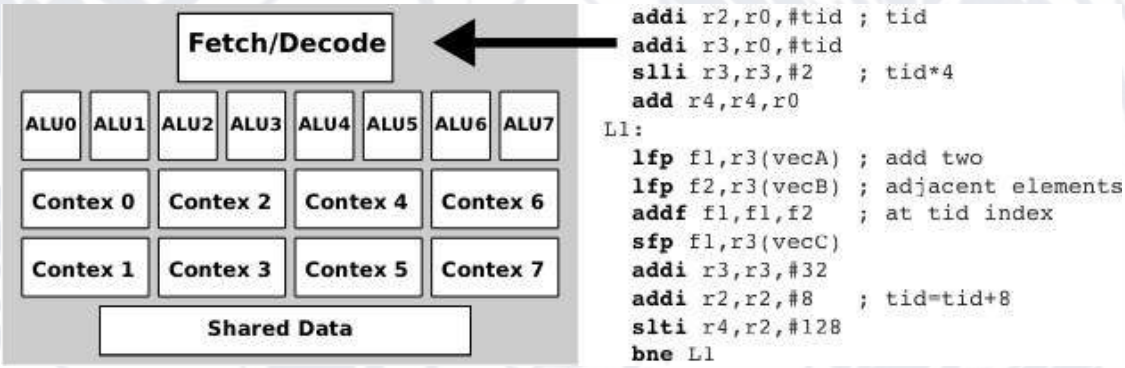

Рис. 11 – Збільшення АЛП та викликів для виконання

Цю архітектуру можна періодично прискорювати, додаючи по 8 елементів з кожного вектора за один і той же потік інструкцій — одна й та сама інструкція виконується для кожного потоку, але над різними наборами даних. Для цього кожен потік повинен мати унікальний ідентифікатор (ID), щоб визначити, з якою частиною даних він працює[33]. Таким чином, компілятор для коду на мові С здатний транслювати код з лістингу 1 у відповідні інструкції, зображені на рис. 11, а також реалізувати логіку, що включає ідентифікацію потоків. Оскільки кожен арифметико-логічний пристрій має власний набір регістрів кожен пристрій працює з унікальним значенням tid, яке зберігається в регістрі r2. Така ж логіка застосовується і для інших інструкцій.

Позначимо ядро, зображене на рис. 8, як обчислювальну одиницю, а АЛП — як елемент обробки.

Подальше підвищення ефективності обчислень можна досягти шляхом збільшення кількості обчислювальних одиниць, що дозволить одночасно виконувати більше операцій та паралельно обробляти більші обсяги даних. Це дає змогу розподіляти навантаження між різними обчислювальними пристроями, що, у свою чергу, збільшує продуктивність системи.

Рис. 12 ілюструє GPU з 16 такими обчислювальними одиниць. Такий пристрій здатний виконати операцію додавання вектора з 256 елементів за одну ітерацію. Кожна обчислювальна одиниця виконує однаковий код, представлений на цьому рисунку, що дозволяє паралельно обробляти усі елементи вектора за мінімальний час. Кожен потік має унікальний ID в діапазоні від 0 до 127. Перші дві інструкції завантажують ідентифікатор потоку в регістр r3 і обчислюють га скільки він міг зміститися у пам'яті, і, при цьому, множачи його на 4, щоб врахувати тип даних з плаваючою точкою.

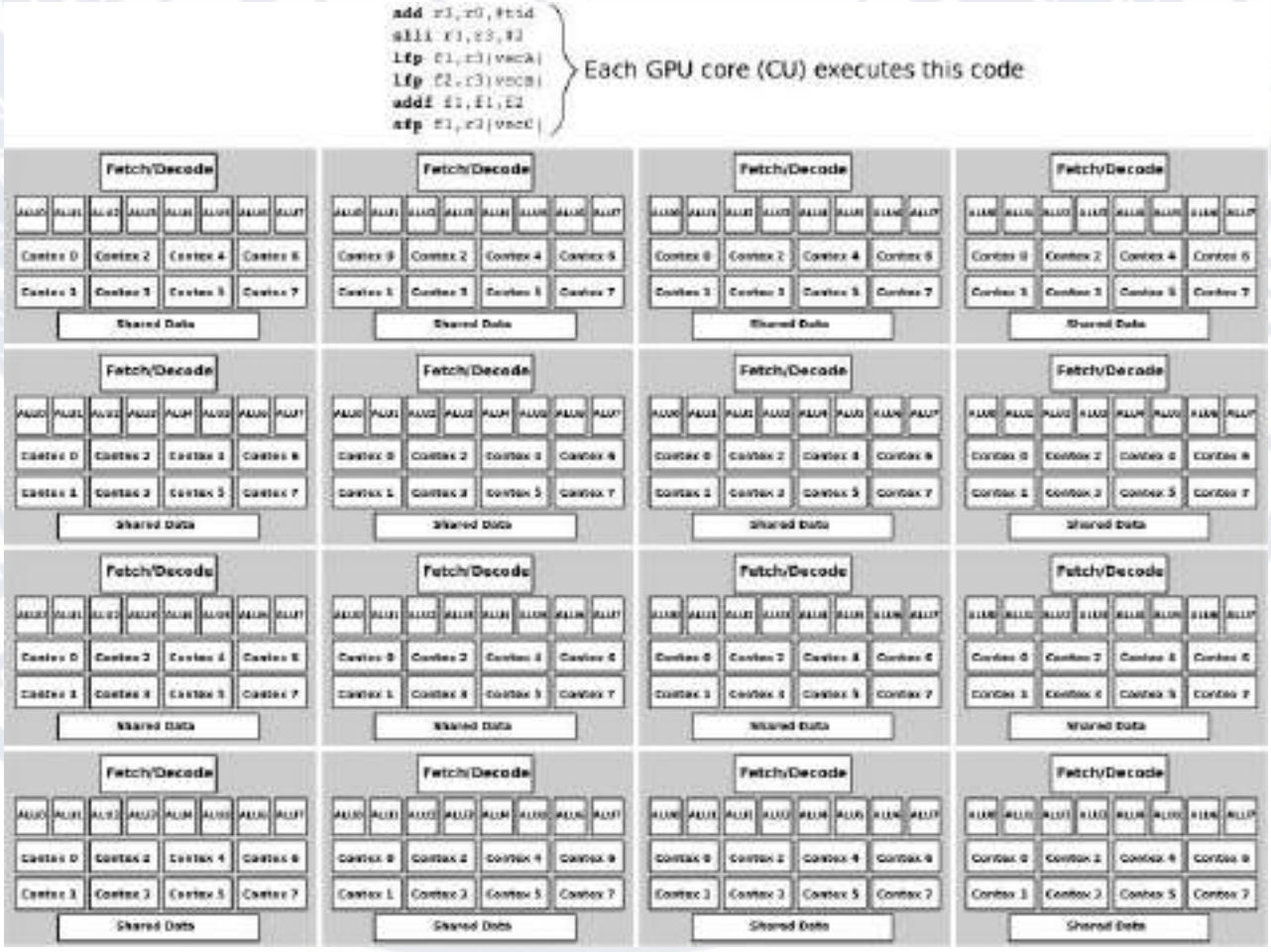

Рис. 12 – 16 обчислювальних одиниць

Однак, хоча спочатку графічні процесори використовувались переважно для графічних обчислень, з виходом серії GeForce 3 у 2001 році почали з'являтись ідеї використовувати графічні карти для розрахунків поза цією сферою. Такий розвиток відбувся завдяки впровадженню стандарту DirectX 8.0 в цій серії пристроїв, який вимагав від графічних чіпів підтримки програмованих вершинних і піксельних шейдерів.

Це дозволило розробникам вперше мати можливість контролювати, які саме обчислення виконуються на GPU, відкриваючи нові можливості для його використання у широкому колі задач. Дослідники почали зацікавлюватися можливістю використовувати графічні процесора для більшого спектра задач, хоча на той момент це не вдалося в повній мірі реалізувати через суттєві обмеження[35].На той час єдиним способом проведення обчислень можна було через використання графічних API на кшталт OpenGl або DirectX[46].

Довільні дані для обчислень подавались через кольори, координати тощо. Однак структура даних, що відповідала за кольори та текстури, мала серйозні обмеження щодо числових значень. Додатково, на той час не існувало інструментів для дебагінгу, що значно ускладнювало процес розробки. Ці та інші обмеження обмежували широке використання GPU для неграфічних обчислень. З появою платформи від NVIDIA у 2006 році значно змінився підхід до використання графічних процесорів для обчислень. До цього часу GPU в основному використовувалися для графічного рендерингу, і виконання обчислень, не пов'язаних з графікою, обмежувалося спеціальними методами або апаратними можливостями[36]. Однак CUDA дозволила значно розширити можливості відеокарт, надаючи програмістам доступ до потужних обчислювальних ресурсів GPU для загальних паралельних обчислень. Ця платформа зняла обмеження, накладені традиційними графічними API, і дозволила використовувати GPU для широкого спектру задач, включаючи наукові обчислення, машинне навчання, моделювання та інші високопродуктивні обчислення.

1.3 Висновки до розділу

Паралелізм є ключовим елементом архітектури сучасних обчислювальних пристроїв, оскільки він дозволяє значно збільшити продуктивність, ефективно використовуючи ресурси для одночасного виконання численних операцій. Це стало можливим завдяки розвитку багатоядерних процесорів та спеціалізованих обчислювальних одиниць, які здатні паралельно обробляти великі обсяги даних.

Паралельно з розвитком загальних обчислювальних процесорів виник окремий тип пристроїв, орієнтованих на графічні обчислення — графічні процесори (GPU). Завдяки своїй SIMD-природі (Single Instruction, Multiple Data) ці процесори можуть одночасно обробляти тисячі даних, що робить їх надзвичайно ефективними для задач, що вимагають високої паралельності, таких як рендеринг графіки, наукові обчислення та машинне навчання.

Із розвитком стандартів і програмних інструментів, а особливо завдяки запуску платформи CUDA для GPU від NVIDIA, стало можливим значне розширення можливостей паралельних обчислень. CUDA зняла обмеження, накладені традиційними графічними API, дозволивши використовувати потужність GPU для виконання загальних обчислень. Це стало важливим кроком у розвитку паралельних обчислень, відкривши нові горизонти для використання GPU в широкому спектрі додатків[50].

РОЗДІЛ 2 ЕФЕКТИВНСТЬ РОБОТИ CPU ТА GPU І АНАЛІЗ ІНШИХ РОБІТ

Як було зазначено в попередньому розділі, графічні та центральні процесори спеціалізуються на виконанні різних типів задач. Наразі графічні процесори широко застосовуються для обчислень у різних галузях, таких як машинне навчання, аналітика даних, фінансове моделювання, медичні дослідження та обробка відео. Завдяки високій продуктивності та здатності до паралельної обробки великих обсягів даних, GPU стали незамінним інструментом у задачах, які потребують інтенсивних обчислень. Їхнє використання дозволяє скорочувати час обробки, підвищувати точність моделей та відкривати нові можливості для наукових і комерційних досліджень[16].

У цьому підрозділі ми розглянемо кілька таких робіт і проаналізуємо їхні висновки. Ці результати допоможуть зрозуміти, як відеокарти поводяться при вирішенні різних задач з різними обсягами даних, а також сформувати очікування для майбутніх тестів.

2.1 Огляд підходів і методі машинного навчання

Як вже згадувалось, використання GPU для машинного навчання стало дуже популярним. Існує безліч досліджень, що вивчають цю тему. Один з таких прикладів наведено у роботі [17], де розглядається використання CPU та GPU для обчислень у сфері глибокого навчання. У цьому дослідженні аналізується задача класифікації веб-сторінок на 23 категорії з використанням рекурентної нейронної мережі (RNN). У такій мережі зв'язки між вузлами формують орієнтований у часі граф, що дозволяє враховувати залежності між даними послідовності та ефективно обробляти текстову інформацію веб-сторінок.. Для проведення тестів було використано хмарну інфраструктуру. Дослідники застосовували процесор Intel Xeon Gold 6126 з дванадцятьма ядрами, а завдяки хмарним можливостям, вони змогли залучити необхідну кількість ядер із декількох процесорів Intel Xeon, де частота процесора складала 1350 та 2665 МГц. Крім того, доступ до хмарного сервісу надав можливість використовувати GPU NVIDIA Tesla K80, що має 2496 ядер, працюючих на частоті 562 МГц.

Порівняльні характеристики можливостей пристроїв для цього дослідження були наведені у таблиці на рис. 13.

Під час дослідження було відзначено досить значний приріст у швидкодії під час класифікації веб-сторінок. Досягнути цього можна було завдяки підвищеній частоті ядер самого процесора, так і за рахунок збільшення їх самої кількості. Результати, були отримані під час тренування нейронної мережі протягом 3 повних ітерацій через датасет при різних розмірах кількості екземплярів для тренування. Загальна кількість екземплярів складає 1 211 976 одиниць.

TABLE II. HARDWARE FEATURES OF DEVICES

Features	Devices		
	PC	*CPU Server*	*GPU Server*
The number of Cores	2	16/24	2496
Core Clock	2,3 GHz	1,3GHz/2,6 GHz	562 MHz
Boost Clock	--	3,7 GHz	875 MHz
Ram	8 GB	24 GB	61 GB
vRam (GPU)	1,5 GB	--	12 GB
Storage	128 GB	2 TB	110 GB

PC: Personal Computer

Рис. 13 – Характеристики пристроїв

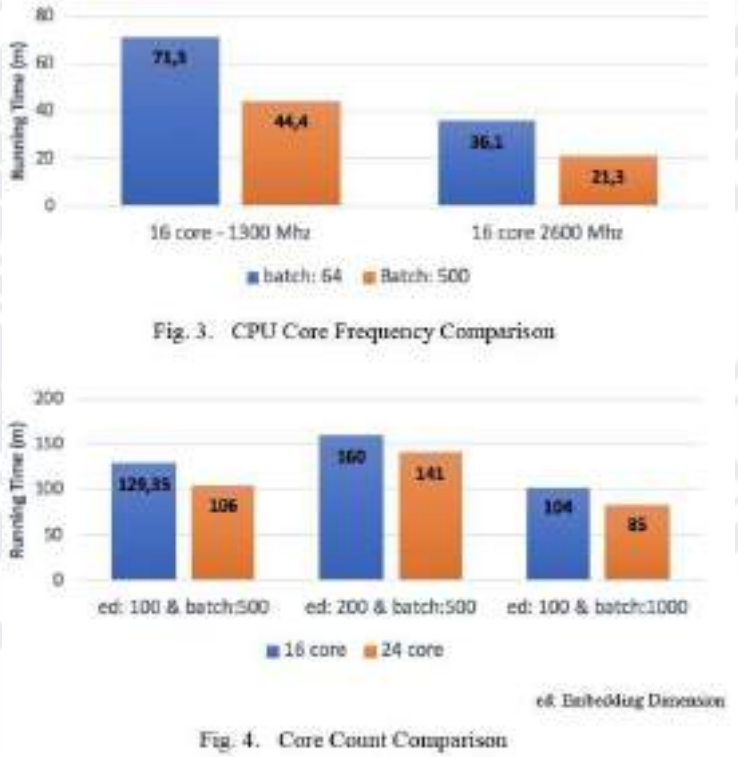

Рис. 14 – Робота процесора на різній частоті і кількості ядер

Порівнюючи продуктивність процесорів і відеоадаптерів, автори цієї статті представили результати, які зображені на рис. 15 Тестування проводились із використанням графічної карти Tesla K80 та 16-ти ядерним Intel Xeon CPU з тактовою частотою 2601 MHz.

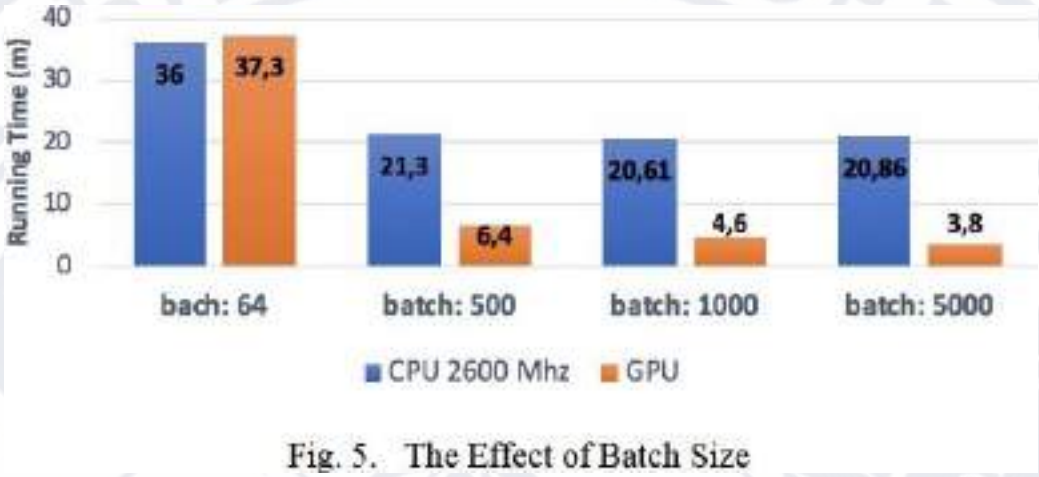

Fig. 5. The Effect of Batch Size

Рис. 15 – Ефективність роботи процесора і відеоадаптера

Одним з важливих спостережень є те із збільшенням кількості одиниць даних, що обробляються за одну ітерацію, ефективність роботи графічної карти значно зростає.

2.2 Огляд досліджень у галузі еволюційних алгоритмів

Іншим напрямком штучного інтелекту, де використовуються графічні процесори для обчислень, є еволюційні алгоритми. Розглянемо роботу , присвячену тестуванням CPU і GPU на еволюційному алгоритмі. Множина вибірки з кожною новою ітерацією формується шляхом комбінування найкращих екземплярів (особин) з попередньої ітерації та поточної популяції за визначеними правилами і умовами.

Для проведення бенчмарку використовувалися центральний процесор Intel i5-10600K з частотою 4 ГГц і 4 ядра, а також графічна карта NVIDIA GeForce GTX 1080 з Cuda 2560 ядрами, графічний процесор працював на чатоті 1607 МГц. Для еволюційного алгоритму NSGA-II було створено дві реалізації: одна на мові програмування C#, а інша з використанням CUDA C.

Обидві версії побудовані відповідно до псевдокоду алгоритму, представленого в рис 15, і проілюстровані на рис. 16 та 17 [18].

```
generationsCount = 0
currentPopulation = GenerateRandomIndividuals(N)
EvaluateFitness(currentPopulation)
NonDominatedSort(currentPopulation)
while(generationsCount < X)
    while(newPopulation.count < N)
        parent1, parent2 = Select(currentPopulation)
        child1, child2 = Crossover(parent1, parent2)
        Mutate(child1, child2)
        newPopulation.Add(child1, child2)
    nextGeneration = currentPopulation + newGeneration
    NonDominatedSort(nextGeneration)
    currentGeneration = SelectBestN(newGeneration)
    generationsCount = generationsCount + 1
```

Рис 16 – Код еволюційного алгоритму NSGA-II

C#	CUDA C++

`int split = rnd.Next(p1.Genome.Length);`
 	
`c1 = p1.DeepCopy();`
`c2 = p2.DeepCopy();`
 	
`for (int i = 0; i < split; i++)`
`{`
`c2.Genome[i] = p1.Genome[i];`
`c1.Genome[i] = p2.Genome[i];`
`}`
	`int split(curnd(genomeCount));`
	
	`c1 = p1;`
	`c2 = p2;`
	
	`for (int i = 0; i < split; i++)`
	`{`
	`c2.Genome[i] = p1.Genome[i];`
	`c1.Genome[i] = p2.Genome[i];`
	`}`

Рис. 17 – Реалізація на різних мовах програмуваннях спліту у NSGA-II [18]

C#	CUDA C++

while (newGeneration.Count < populationSize)
{
IIndividual<T> parent1 = selector.Select(currentGeneration);
IIndividual<T> parent2 = selector.Select(currentGeneration);
 	
IIndividual<T> child1;
IIndividual<T> child2;
crossover.Crossover(parent1, parent2, out child1, out child2);
 	
mutator.Mutate(child1);
mutator.Mutate(child2);
 	
child1.Objectives = evaluator.CalculateObjectives(child1);
child2.Objectives = evaluator.CalculateObjectives(child2);
 	
newGeneration.Add(child1);
newGeneration.Add(child2);
}
 	
newGeneration.AddRange(currentGeneration);
 	
newGeneration = sorter.GetBestIndividuals(newGeneration, populationSize);
	i = 0;
	while(i <popCount)
	{
	int p1 = TurnamentSelection(pop);
	int p2 = TurnamentSelection(pop);
	
	OnePointSplitCrossover(pop[p1], pop[p2], c1, c2);
	
	GaussianMutation(c1, mutationChance);
	GaussianMutation(c2, mutationChance);
	
	ZDT1(c1);
	ZDT1(c2);
	
	newPop[i++] = c1;
	newPop[i++] = c2;
	}
	
	j = 0;
	while(j <popCount)
	newPop[i++] = pop[j++];
	
	GetBestIndividuals(newPop, pop);

Рис. 18 – Реалізація на різних мовах програмуваннях спліту у NSGA-II двома способами [18]

Рисунок 17 ілюструє реалізацію функції з рисунка 18. Функція OnePointSplitCrossover виконує поділ двох особин із популяції шляхом випадкового поділу їхніх характерних векторів (геномів) на дві частини та заміни їх один одним. Якщо коротко описати роботу еволюційного алгоритму, то спочатку генерується вихідна популяція, що складається з випадково вибраних характерних значень (геномів), які лежать у прийнятному просторі розв’язку задачі.

Потім на кожній ітерації виконуються наступні кроки:

1) Дві особини випадковим чином вибираються з популяції, які є «батьками» (на основі значення цільової функції);

2) Використовуючи функцію схрещування (OnePointSplitCrossover), ці батьки генерують двох нових особин, «нащадків»;

3) З певною ймовірністю в геномах нових особин відбуваються мутації, що означає випадкові зміни деяких характеристик.

Після виконання алгоритму з кількістю потоків до 40 000 на кожному з пристроїв були отримані результати, представлені на рис. 19. Як видно, зростання кількості потоків призводить до зниження ефективності CPU, тоді як ефективність GPU значно покращується. Збільшений графік на рис. 20 демонструє, що перехід на користь GPU починається вже при 3000 потоках. Аналогічний експеримент із розміром геному 2 замість 20 (в результаті обробляється 100 характеристик, а не 1000) показав інші результати, які можна побачити на рис. 21.

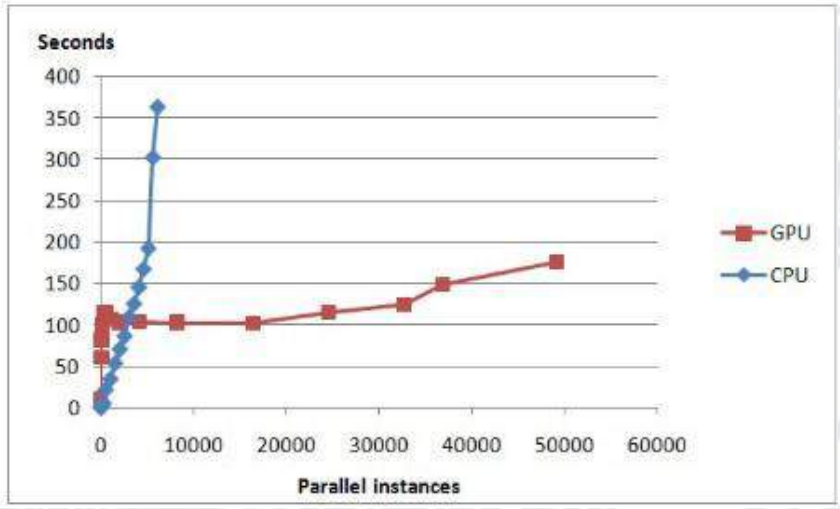

Рис. 19 – Виконання еволюційного алгоритму за допомогою багатопотокової обробки. [18]

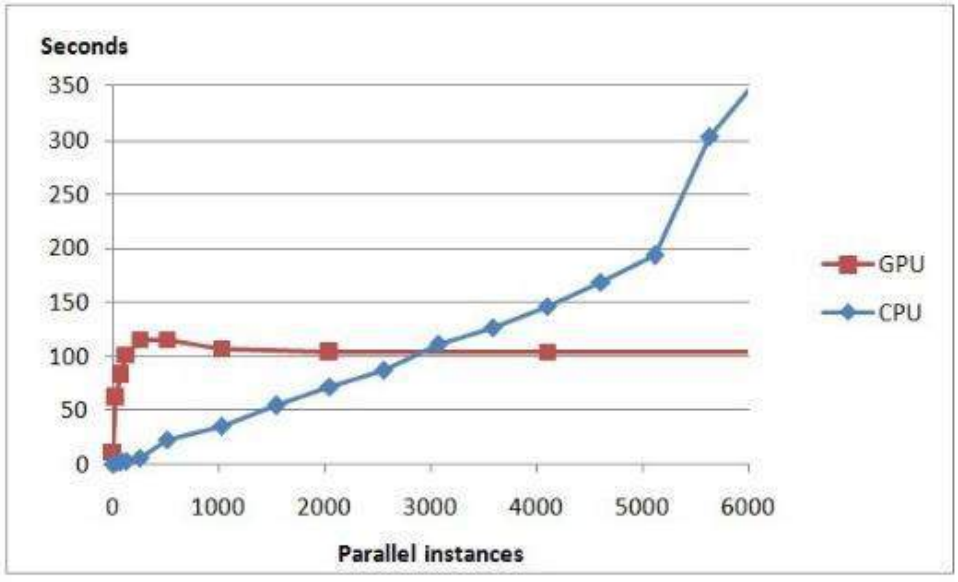

Рис. 20 – Приблидеий вигляд [18]

На рис. 21 видно, що GPU почав демонструвати перевагу раніше, приблизно при 1700 екземплярах відтворення генетичного алгоритму. У результаті, автори публікації зробили висновок, що ефективність GPU значно знижується при простоюванні, а управління пам'яттю та копіювання даних між графічною картою та іншими компонентами вимагає додаткових ресурсів. Спостереження показали, що при обробці невеликої кількості даних та обмеженій кількості потоків центральні процесори забезпечують вищу ефективність.

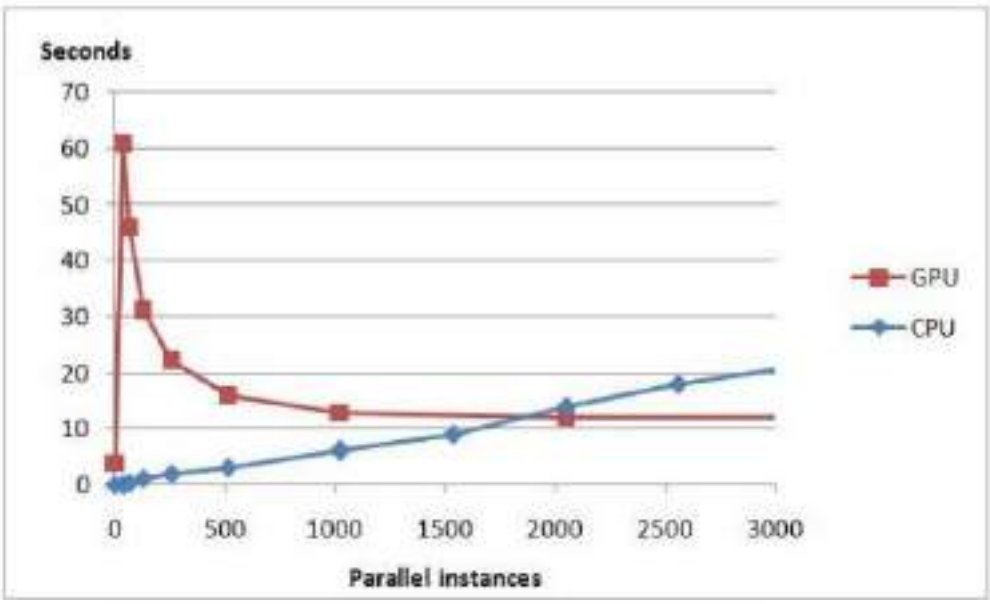

Рис. 21 – Результат з використанням геному меншого розміру [18]

2.3 Огляд підходів і методів в області медіаданих

Роботою, яка порівнює ефективність центральних та графічних процесорів, є дослідження [19], що аналізує це питання через тести з використанням графічного формату JPEG 2000. Застосування відеокарт для обробки кодеків, зокрема для відео, є окремою галуззю в сфері роботи з GPU, причому карти NVIDIA навіть оснащені спеціальними апаратними прискорювачами для таких задач. Саме тому ми обираємо роботу [19] як приклад для дослідження. JPEG 2000 — це метод стиснення даних, який потребує значних обчислювальних ресурсів для виконання різних операцій, як показано на рис. 22. До таких операцій належать перетворення (наприклад, дискретне косинусне перетворення — CT, дискретне вейвлет-перетворення — DWT), квантування, кодування (EBCOT), оптимізації (PCRD) та інші етапи, що забезпечують високу якість стиснення.

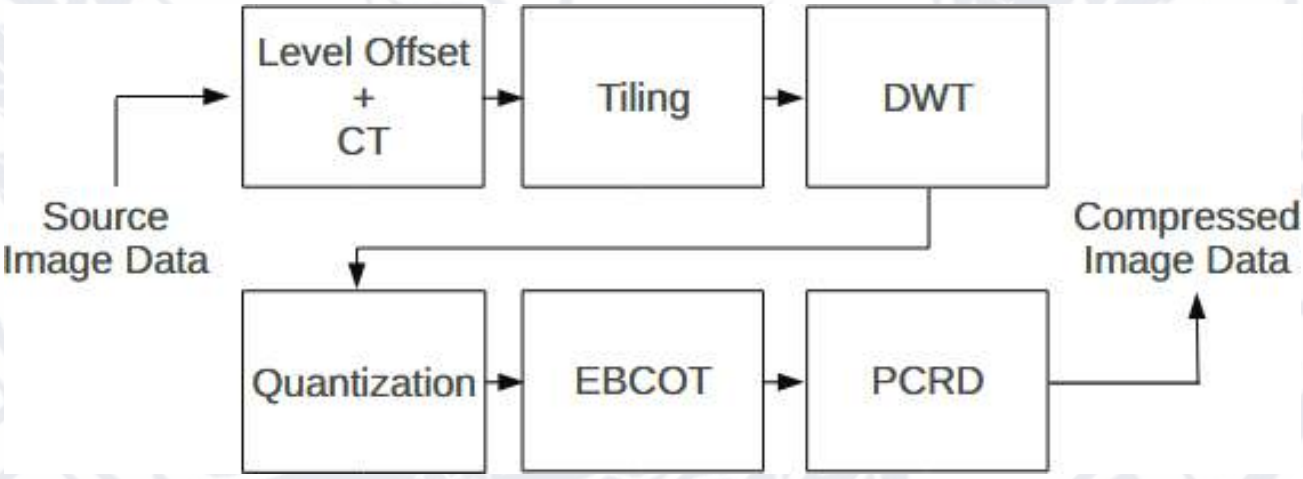

Рис. 22 – Наглядне відображення кодування у формат JPEG 2000 [19]

Дослідники реалізували цей кодек на мові CUDA C, щоб мати можливість проводити експерименти на GPU, а для тестування на CPU вони скористалися готовими бібліотеками OpenJPEG, комерційною бібліотекою Kakadu. Під час проведення експерименту використовувалися такі пристрої: Intel Xeon X5570 (8 ядер), Intel Xeon E5-2640 (6 ядер), а також GPU: NVIDIA GTX580 (752 ядра, 780 Мгц).

Результати порівняльних тестів для центрального і графічного процесора наведені на рис. 23. Також є дані про виконання на одному ядрі для кожної моделі центрального процесора з використанням різних бібліотек, що представлені на рис. 24.

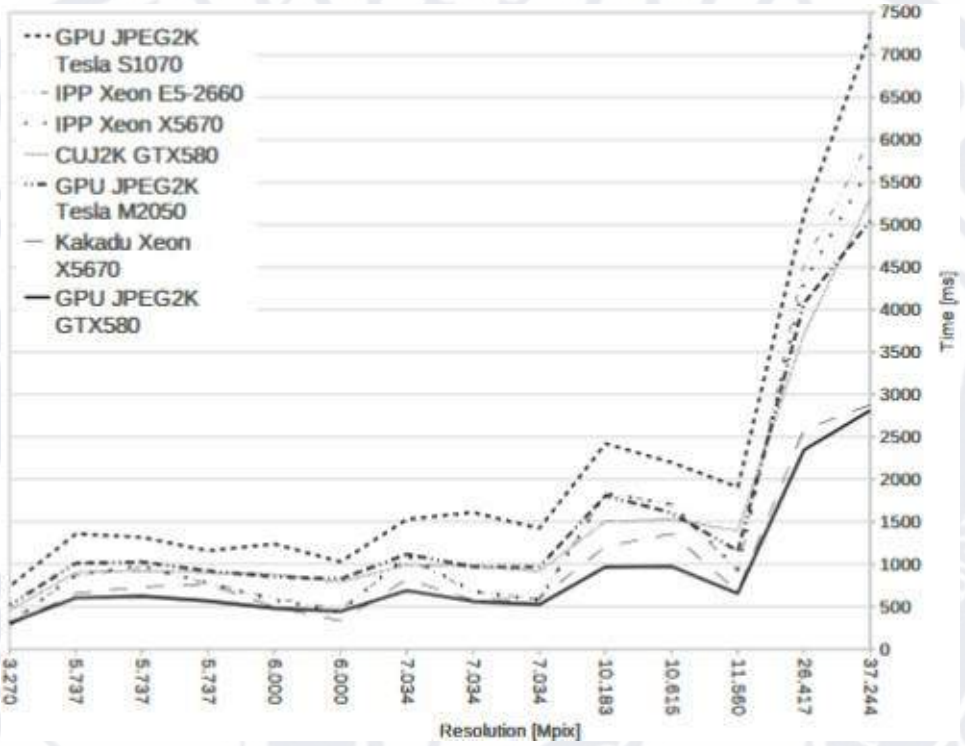

Рис. 23 – Результати тестування алгоритмі JPEG 2000 [19]

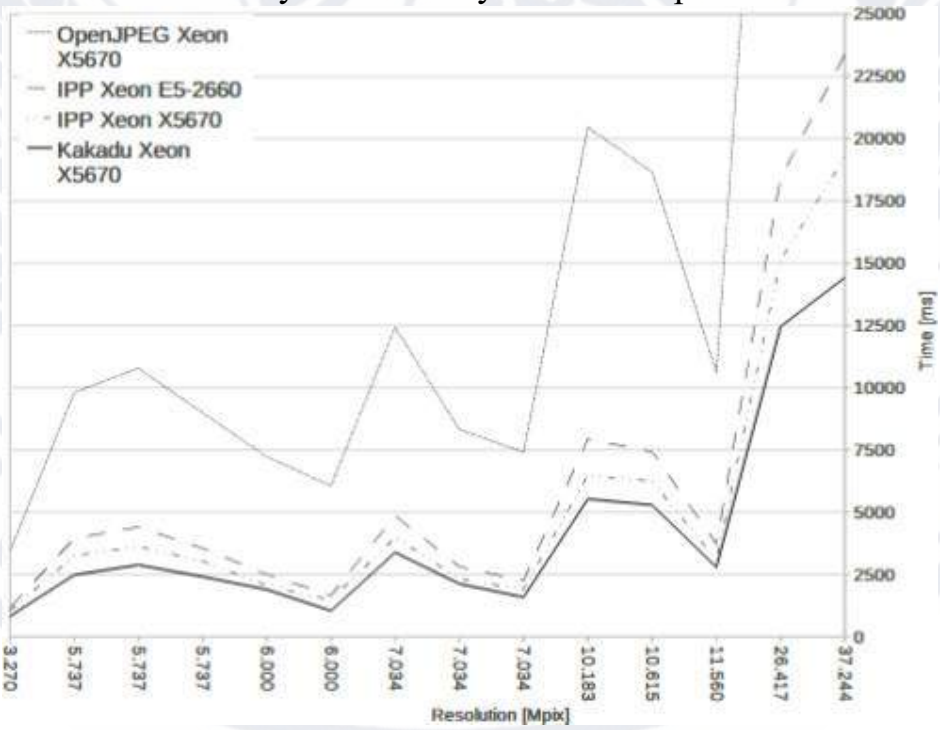

Рис. 24 – Результат з використанням одного ядра процесора[19]

Також можна відзначити з графіку на рис. 24, що якісно написаний програмний код має вплив на висновки тестування.

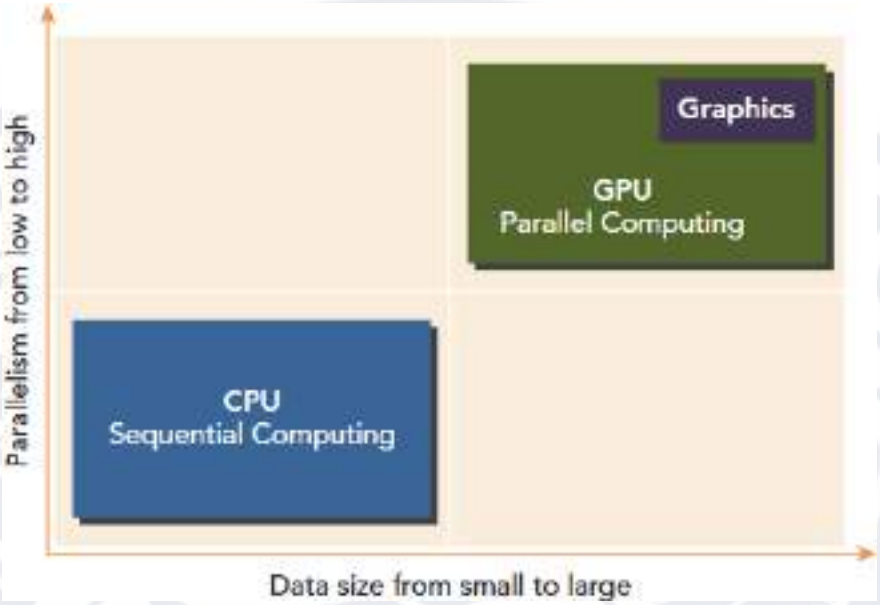

Рис. 25 – Використання GPU і CPU відповідно до задач [20]

Якщо використовуються готові бібліотеки, важливо враховувати оптимізації, закладені їх розробниками, а також здатність цих бібліотек адаптуватися до особливостей апаратної архітектури.

З огляду на аналіз розглянутих робіт, метою цієї бакалаврської роботи є визначення оптимального розміру вхідного набору даних для ефективного прискорення обчислень на GPU. У рамках дослідження будуть розв'язуватися такі задачі:

1. Обчислення скалярного добутку векторів.
2. Множення матриць.
3. Додавання матриць.
4. Додавання матриць із коефіцієнтом.
5. Ділення матриці на число.

Ці задачі дозволять оцінити продуктивність GPU при різних обсягах даних та вибрати оптимальні підходи для кожної з них.

2.4 Висновки до розділу

Аналізуючи раніше описані роботи, можна зробити висновок, що успішність заміни CPU на GPU для певних завдань залежить від таких факторів, як розмір вхідних даних, складність виконуваних операцій, а також здатність конкретного завдання бути ефективно паралелізованим. Окрім того, важливо враховувати специфіку апаратного забезпечення, таку як кількість обчислювальних ядер, об'єм пам'яті, пропускну здатність та архітектурні особливості процесорів.

Таким чином, для майбутніх експериментів з задачами на матрицях можна припустити, що ефективність результатів буде варіюватися залежно від пристроїв, їх новизни та апаратних можливостей. Зокрема, новітні графічні процесори з більшою кількістю ядер та покращеною архітектурою зможуть забезпечити кращу продуктивність для операцій, які вимагають великої паралельної обробки даних. Однак, для кожного конкретного завдання важливо оптимізувати не тільки розмір вхідних даних, а й структуру обчислень для досягнення максимального ефекту. Це включає в себе вибір відповідних алгоритмів, що зменшують час обробки та ефективно використовують ресурси GPU, а також правильну організацію пам'яті для мінімізації затримок у доступі до даних.

Крім того, важливо враховувати, що деякі задачі, що можуть добре працювати на CPU, не обов'язково будуть ефективно виконуватись на GPU, особливо якщо завдання має низький рівень паралелізму або великі вимоги до обробки послідовних операцій. Тому для успішного впровадження GPU необхідно ретельно підходити до вибору задач і оптимізації програмного забезпечення.

РОЗДІЛ 3 ВИБІР ПРОГРАМНИХ ІНСТРУМЕНТІВ

Цей розділ спрямований на аналіз програмних інструментів, що використовуються для досліджень на центральних і графічних процесорах. Увага буде приділена основним особливостям кожної платформи, які відповідають ключовому критерію — забезпеченню написання максимально ефективного коду. Це є важливим аспектом, який буде врахований у подальших тестах.

3.1 Огляд мови для проведення тестування на CPU

Для проведення тестів на центральному процесорі буде розроблено програму мовою C++. Вибір цієї мови обумовлений її високою продуктивністю та широкими можливостями стандартної бібліотеки `std`. Як компільована мова, C++ має ключову перевагу: її код виконується швидше за інтерпретований. Це пояснюється тим, що компілятор одразу генерує виконуваний файл із машинними інструкціями, тоді як інтерпретатори опрацьовують код пострічково, що уповільнює виконання[47].

Згідно з результатами бенчмарків, мови C та C++ демонструють одні з найкращих показників продуктивності на різних типах задач. Наприклад, на рис. 26 показано результати тестів на побудову bitmap-зображення множини Мандельброта розміром 1600^2 елементів. Дані взято з проекту The Computer Language Benchmarks Game. Для побудови графіка було використано найкращі показники для кожної мови програмування, включно з оптимізаціями та можливостями паралелізації, доступними для відповідної мови.

Іншим прикладом є дослідження [22], яке стосується біоінформатики. У рамках тесту був реалізований алгоритм вирівнювання послідовностей — методу, що використовується для порівняння біологічних ділянок з метою виявлення схожих послідовностей. Алгоритм був протестований на двох послідовностях ДНК. Тести проводилися на платформах Linux та Windows. Результати експерименту представлені на рис. 27.

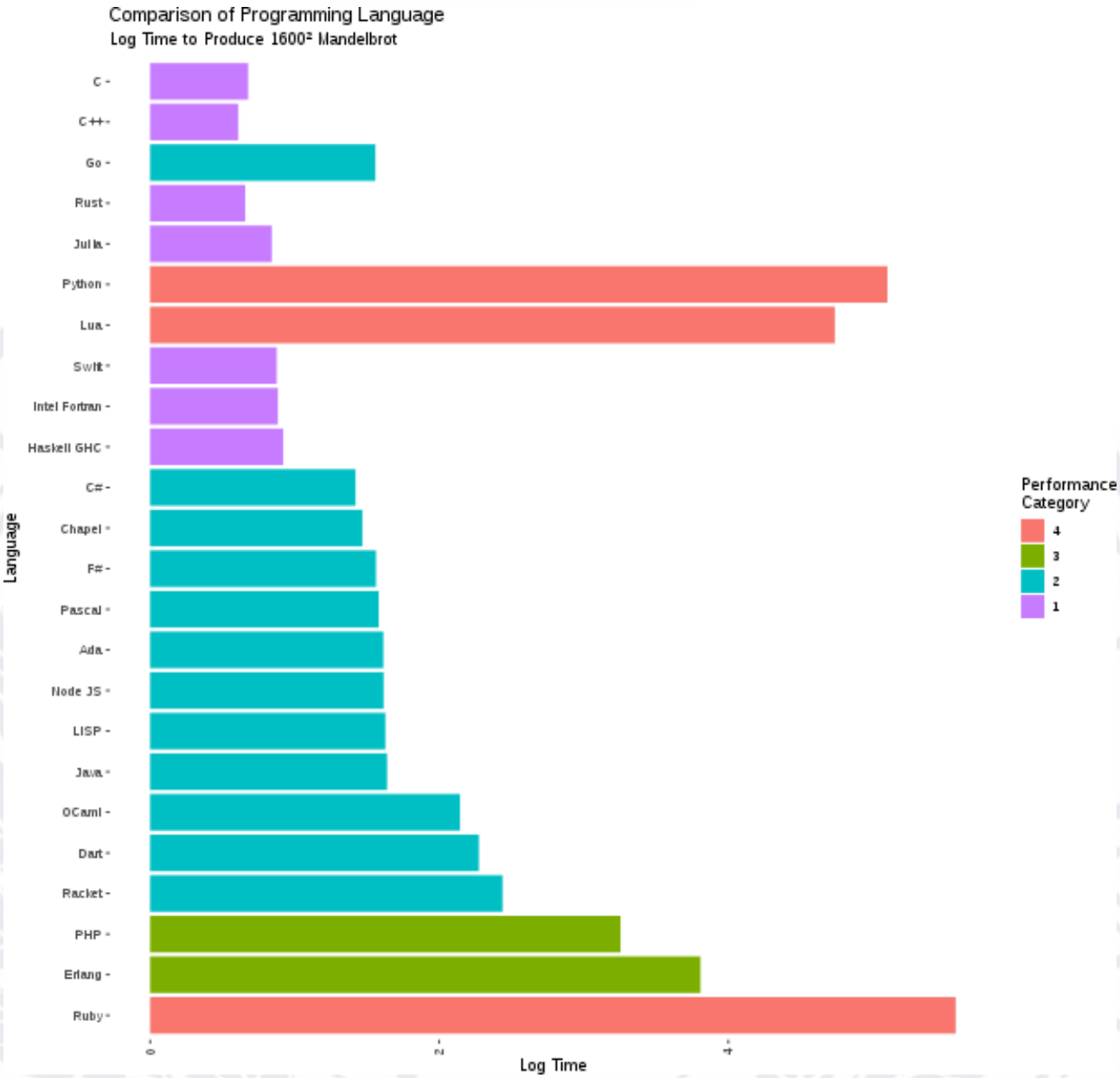

Рис. 26 – Результати проведення тестування [21]

На рис. 26, до групи високопродуктивних також входять мови, зокрема Rust і Haskell[48]. Проте вибір C++ був обґрунтований кількома факторами. Зокрема, C++ надає можливість безпосередньо взаємодіяти з пам'яттю машини та управляти нею на низькому рівні, що дозволяє досягати більшої ефективності у використанні ресурсів. Це важливо для оптимізації продуктивності, особливо в обчислювальних задачах, що потребують високої швидкості обробки даних. Окрім цього, C++ має значно багатший набір інструментів у своїй стандартній бібліотеці, особливо у сфері паралелізму та синхронізації, що є ключовим для задач, пов’язаних з оптимізацією обчислень.

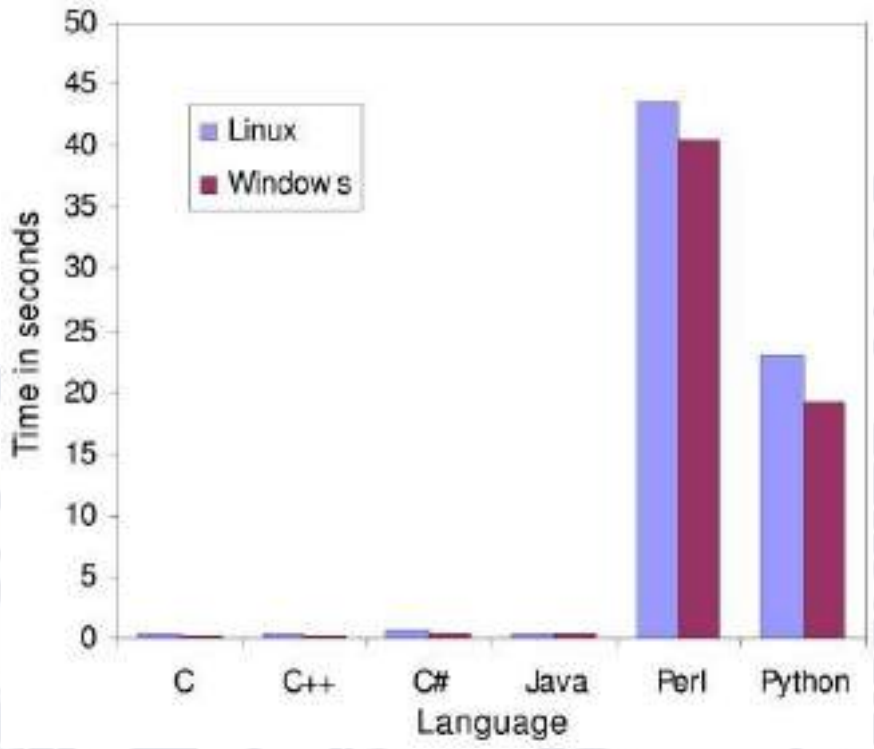

Рис. 27 – Результати згідно із [22]

Головним інструментом стандартної бібліотеки C++ для реалізації багатопоточності є `std::thread`. Цей клас надає платформонезалежний інтерфейс, що спрощує роботу з потоками, використовуючи системні виклики операційної системи.

У середовищі операційних систем потік (thread) є найменшою одиницею виконання, яку також називають [49]. Потоки існують виключно в межах процесів і мають такі ресурси:

- власний програмний лічильник,
- стек,
- набір регістрів (див. рис. 28).

Основними перевагами потоків є:

- швидке переключення контексту,
- легка взаємодія між потоками.

Будь-яка програма на C++ за замовчуванням має хоча б один потік — main(). Для створення додаткових потоків програміст ініціалізує об’єкти класу `std::thread`, у яких вказується функція, що має виконуватися в потоці, та її аргументи.

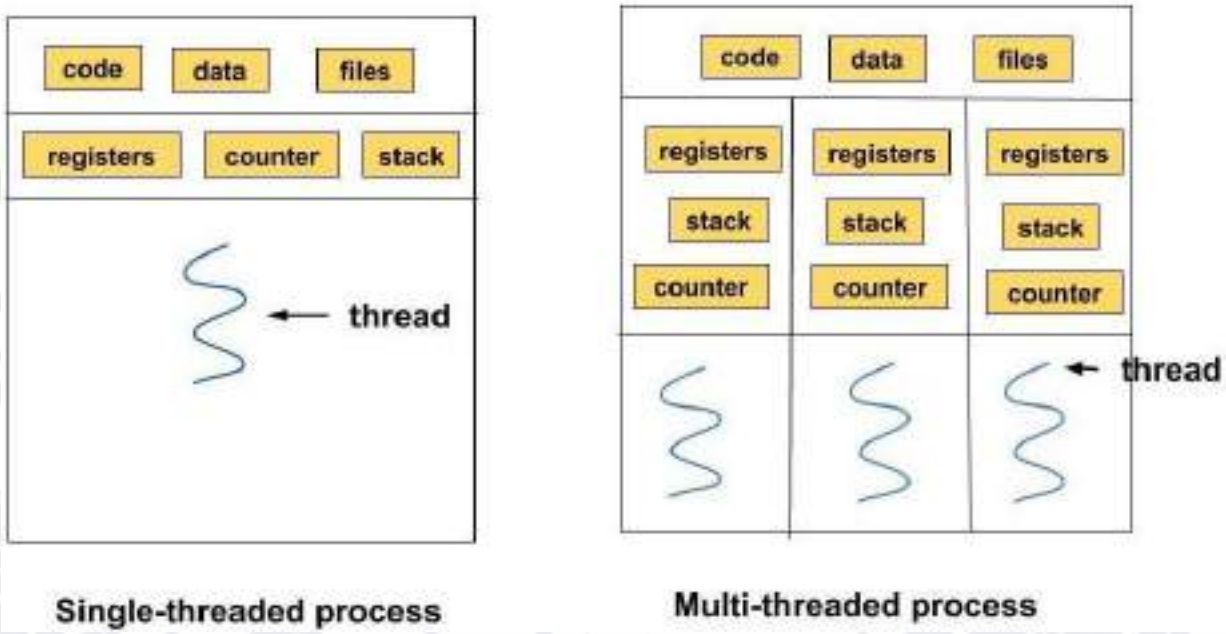

Рис. 28 – Схема роботи потоків [23]

Для очікування завершення роботи потоку в C++ використовується метод `join()`, який може бути викликаний для потоку лише один раз. Якщо ж потрібно залишити потік працювати у фоновому режимі без подальшого прямого зв’язку, застосовується метод `detach()`.

Виклик `detach()` від’єднує потік від об’єкта `std::thread`, передаючи контроль над ним бібліотеці C++ Runtime Library. У такому випадку:

- Потік продовжує виконуватись у фоновому режимі.
- Очікування завершення його роботи більше не передбачається.
- Об’єкт `std::thread`, що посилався на потік, стає недійсним, і виклик `join()` стає неможливим.

C++ Runtime Library відповідає за правильне вивільнення ресурсів, пов’язаних із від’єднаним потоком, після завершення його роботи. Потоки, які працюють у цьому режимі, часто називають , за аналогією з процесами-демонами в UNIX. Як і демони, такі потоки виконують фонові завдання без взаємодії з користувачем.

Розглянуті вище концепції є основними принципами, пов'язаними зі створенням і управлінням потоками в C++. Ці знання будуть застосовані в практичній частині роботи для реалізації та виконання обчислювальних задач на центральному процесорі.

3.2 Платформа тестів на GPU

CUDA — це паралельна обчислювальна платформа та програмна модель, розроблена компанією NVIDIA для обчислень на графічних процесорах (GPU). Вона дозволяє програмістам використовувати потужність GPU для загальних обчислень, не обмежуючись лише графічними задачами..

Модель програмування CUDA пропонує розробникам абстрактний інтерфейс для роботи з графічним процесором. У цій моделі основними поняттями є хост і пристрій (device). Хостом виступає центральний процесор системи, а пам'ять, якою він користується, називається “хостовою пам’яттю.

Реалізація гетерогенної архітектури використовує різні типи процесорів для виконання завдань. Такий підхід дозволяє оптимально розподіляти задачі між центральним і графічним процесорами відповідно до їх особливостей. Схематичне зображення архітектури представлено на рис. 29.

Рис. 29 – Архітектура [21]

Щоб запустити програму на CUDA потрібно виконати три основні етапи:

1. Копіювання вхідних даних з пам'яті хоста в пам'ять пристрою (графічного процесора).

2. Завантаження програми на GPU та виконання інструкцій на графічному процесорі.

3. Копіювання результатів з пам'яті пристрою назад у пам'ять хоста.

Схематичне зображення цього процесу наведено на рис. 30.

Програма виконується на графічному процесорі, де паралельна частина обчислюється N разів, кожного разу на окремому потоці CUDA. Це дає можливість графічному процесору реалізувати модель потокових обчислень (stream computing model)[34], за якої вхідні та вихідні дані організовані в потоки, що складаються з однакових елементів, здатних оброблятися незалежно один від одного. Програма пишеться на розширеній версії мови C, призначеній для графічних процесорів NVIDIA — CUDA C, яка дозволяє ефективно використовувати паралельні обчислювальні можливості GPU. Для компіляції коду використовується компілятор nvcc, що спеціалізується на перетворенні CUDA C в код, здатний виконуватися на графічних процесорах.

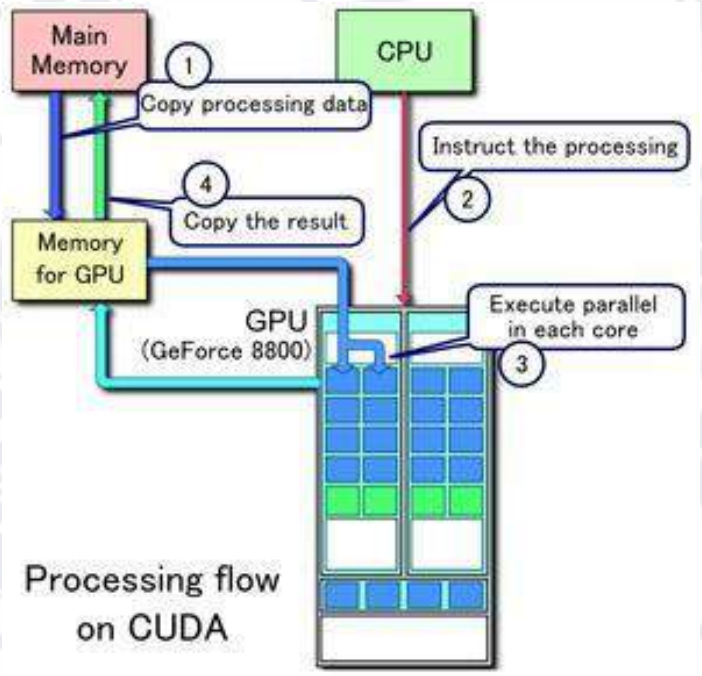

Рис. 30 – Демонстрація роботи алгоритму CUDA[24]

У моделі CUDA кожному кернелу відповідає апаратне ядро, що дозволяє виконувати обчислення на кількох потоках одночасно. Потоки організовуються в блоки, а блоки в свою чергу — у решітку (grid).

Потокові процесори в графічному процесорі (GPU) об'єднуються в групи, які утворюють потокові мультипроцесори (SM, Streaming Multiprocessors), кожен з яких здатний обробляти декілька потоків одночасно.

Для потоків і блоків в CUDA існують вбудовані тривимірні змінні, які дозволяють зручно організовувати та адресувати потоки. Зокрема, змінна threadIdx є тривимірною змінною, яка містить індекси потоку в межах його блоку. Зважаючи на ці апаратні особливості, в коді використовуються абстракції (рис. 31). Виконувані одиниці організовані в ієрархію потоків, блоків і сіток — thread, block, grid. Ця змінна дозволяє кожному потоку визначити своє положення в блоці і таким чином працювати з унікальними наборами даних. Аналогічно, змінна blockIdx визначає індекс блоку в межах решітки блоків, а blockDim і gridDim дозволяють отримати розміри відповідно блоку та решітки, що дає змогу точно управляти кількістю потоків і блоків для оптимальної організації паралельних обчислень.

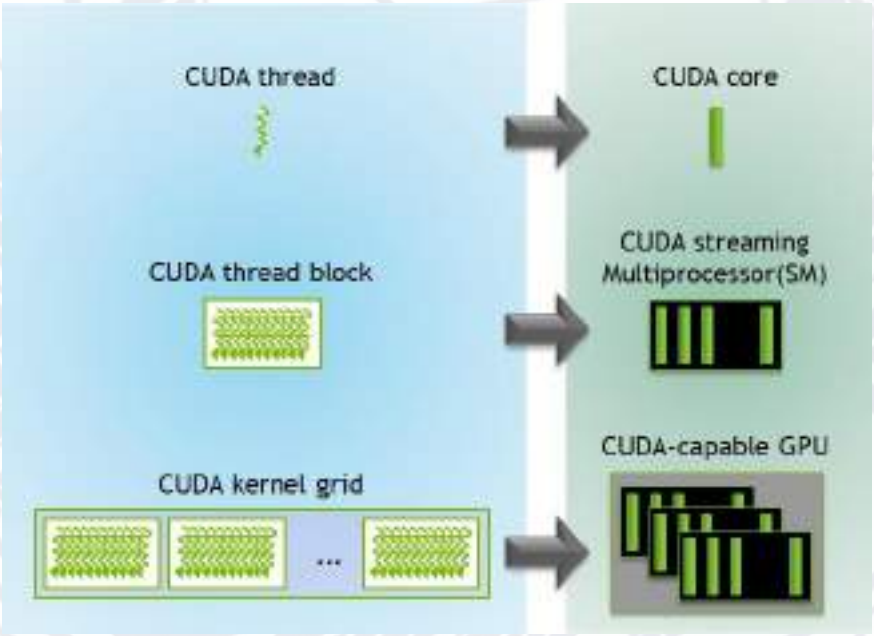

Рис. 31 – Абстрактне відображення CUDA [25]

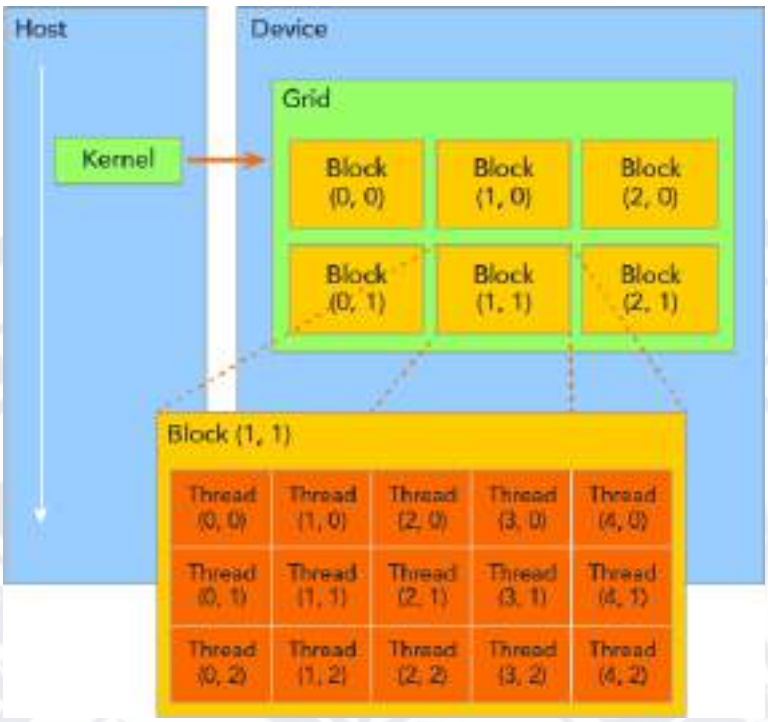

Рис. 32 – Відображення схематично структури потоків [21]

Організація пам'яті в графічному процесорі, включає кілька рівнів пам'яті з різними властивостями доступу та ефективності:

1. Регістри – приватні для кожного потоку. Це означає, що регістри, призначені для одного потоку, не доступні іншим потокам. Кількість регістрів обмежена, і компілятор CUDA самостійно приймає рішення щодо їх використання для кожного потоку.

2. L1/спільна пам'ять (shared memory, SMEM) – це швидка пам'ять, яка є вбудованою у кожен потоковий мультипроцесор. Вона використовується як кеш L1 і спільна пам'ять для потоків в одному блоці CUDA. Усі потоки, що належать до одного блоку, можуть спільно використовувати цю пам'ять. Блоки, що працюють на одному потоковому мультипроцесорі, можуть доступати спільний ресурс фізичної пам'яті, наданий цим мультипроцесором.

3. Read-only пам'ять – доступна для читання і має кеш для інструкцій, постійну пам'ять, текстурну пам'ять і кеш для кернел-коду. Вона є доступною для всіх потоків у програмі, але тільки для читання.

4. L2-кеш – загальний для всіх потокових мультипроцесорів. Це означає, що потоки з різних блоків можуть звертатися до цього кешу, що дозволяє

забезпечити більш ефективний доступ до даних.

5. Глобальна пам'ять – це основна пам'ять GPU (DRAM), яка використовується для зберігання даних, що обробляються на GPU. Ця пам'ять має більшу затримку доступу порівняно з іншими рівнями пам'яті, але здатна зберігати великі обсяги даних.

Для графічних карт, що підтримують CUDA, існує поняття обчислювальних можливостей(Compute Capability). Compute Capability використовується для визначення можливостей GPU на етапі розробки програм, оскільки різні версії архітектури можуть мати різні функціональні можливості. Наприклад, деякі старіші моделі GPU можуть не підтримувати новітні функції, такі як певні оптимізації пам'яті або обчислення з плаваючою точкою в кількох точках одночасно [45].

Обчислювальна здатність позначається як X.Y, де:

- X — основна версія, що відображає основні зміни та функціональні можливості;
- Y — вторинний номер, який вказує на покращення, нові функції та оптимізації, що з'являються в кожній новій версії архітектури.

Наприклад:

- NVIDIA 360M (2010 рік випуску) має обчислювальну здатність 1.0,
- NVIDIA GTX 750 (2014 рік випуску) має обчислювальну здатність 5.0
- NVIDIA Quadro RTX (2018 рік випуску) має обчислювальну здатність 7.5.

Цей номер обчислювальних можливостей допомагає визначити, які специфічні можливості доступні для програм, що виконуються на різних поколіннях графічних процесорів.

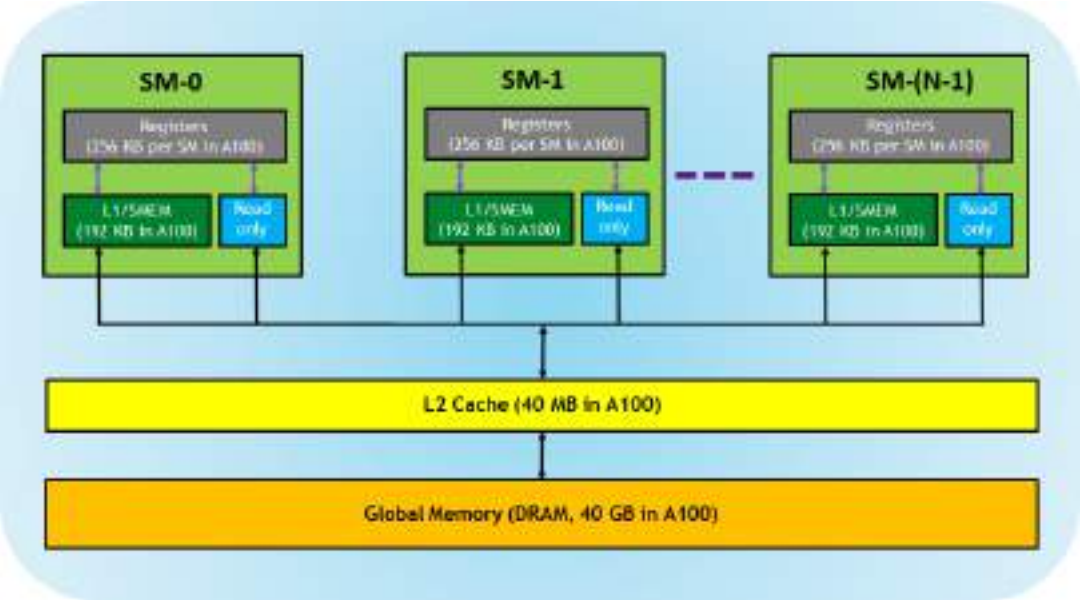

Рис. 33 – Схематичне відображення будови пам’яті графічного процесора

Таким чином, платформа CUDA є потужним інструментом для виконання обчислень загального призначення, що не пов'язані з графікою. Вона має специфічну мову програмування, компілятор та набір правил, орієнтованих на досягнення максимальної ефективності при роботі з графічними процесорами. Випуск CUDA став важливим етапом у розвитку паралельних обчислень і значно підвищив ефективність роботи в таких задачах, як обробка великих даних, машинне навчання, наукові обчислення, обробка зображень і відео, фінансова аналітика та інші. Завдяки CUDA, програмісти отримали доступ до потужних обчислювальних ресурсів графічних процесорів, що дозволяє знижувати час виконання складних алгоритмів.

Сьогодні CUDA є найсучаснішим інструментом для реалізації неграфічних обчислень на відеокартах. Платформа постійно вдосконалюється, інтегруючи нові технології, бібліотеки та інструменти для підтримки найрізноманітніших напрямків обчислень. Вона має значний вплив на розвиток штучного інтелекту, глибинного навчання та інших галузей, де потрібні потужні паралельні обчислення. CUDA також підтримує інші важливі аспекти, як оптимізація пам'яті та зручна інтеграція з іншими програмними середовищами, що робить її необхідним інструментом для сучасних високопродуктивних обчислень.

3.3. Висновки до розділу

Мова C++ була обрана для цієї роботи завдяки кільком важливим перевагам. Вона є однією з найпопулярніших мов програмування, що поєднує високу продуктивність та гнучкість, що робить її ідеальним вибором для розробки високопродуктивних обчислювальних систем. C++ дозволяє програмістам здійснювати детальне управління пам'яттю та обчислювальними ресурсами, надаючи можливість оптимізувати код для досягнення максимальної швидкості виконання. Крім того, C++ має велику кількість бібліотек для роботи з паралельними обчисленнями, що дозволяє значно спростити розробку багатопотокових програм. Однією з таких бібліотек є std::thread, що забезпечує зручний інтерфейс для створення та керування потоками, що особливо важливо для задач, які можуть бути ефективно розподілені між кількома процесорами або ядрами.

Для роботи з графічними процесорами (GPU), яка є ключовою частиною цього дослідження, була обрана платформа CUDA. CUDA (Compute Unified Device Architecture) — це спеціалізоване середовище розробки від NVIDIA, яке дозволяє програмістам використовувати обчислювальну потужність графічних процесорів для задач загального призначення, що не пов'язані безпосередньо з графікою. CUDA дозволяє виконувати складні паралельні обчислення, що істотно прискорює виконання багатьох типів обчислень, таких як обробка великих обсягів даних, математичні моделювання, машинне навчання та інші ресурсоємні завдання.

Поєднання C++ та CUDA дає значні переваги. C++ забезпечує високу швидкість виконання програм, а також детальне управління пам'яттю та процесами, що важливо для роботи з великими обсягами даних та високопродуктивними системами. CUDA в свою чергу додає можливість розподіленого обчислення, оскільки GPU можуть обробляти тисячі потоків одночасно, що значно прискорює виконання паралельних обчислень.

РОЗДІЛ 4 ТЕСТУВАННЯ І ПЕРЕВІРКА ЕФЕКТИВНОСТІ ВИКОРИСТАНИХ ПРИСТРОЇВ

4.1 Опис поставленого завдання та використаних пристроїв

У цьому розділі ставиться за мету порівняти ефективність виконання різних обчислювальних завдань на графічних та центральних процесорах за допомогою таких операцій:

1. обчислення скалярного добутку векторів;
2. множення матриць;
3. додавання матриць;
4. додавання матриць з коефіцієнтом (kA + B);
5. ділення матриці на число.

Як було перевірено раніше, кількість потоків та розмір даних мають значний вплив на продуктивність. У рамках цієї роботи буде проведено серію тестів для визначення оптимального розміру даних для кожного з пристроїв. Вимірювання часу виконання будуть здійснені для різних розмірів векторів, починаючи з 256 елементів і до 81 920 000, а для матриць – від 402 елементів до 90 642. Ці експерименти допоможуть з'ясувати, на якому рівні розмірів даних кожен пристрій демонструє найкращу ефективність, а також допоможуть побудувати картину порівняння продуктивності центрального і графічного процесора для обчислювальних задач різного масштабу.

Вкажемо характеристики пристроїв для тестів, у якості центральних процесорів будуть використовуватись дві моделі: Intel Core i7-8700K (рис. 34) та Intel Core i5-7300HQ (рис. 35). Другий процесор представлений у персональному ноутбуці.

CPU Specifications

Total Cores	6
Total Threads	12
Max Turbo Frequency	5.10 GHz
Intel® Thermal Velocity Boost Frequency	5.10 GHz
Intel® Turbo Boost Max Technology 3.0 Frequency ‡	4.90 GHz
Processor Base Frequency	2.70 GHz

Рис. 34 – Загальна характеристика i7-8700K [26]

CPU Specifications

Total Cores	2
Total Threads	4
Max Turbo Frequency	3.10 GHz
Intel® Turbo Boost Technology 2.0 Frequency‡	3.10 GHz
Processor Base Frequency	2.50 GHz

Рис. 35 – Загальна характеристика i5-7300HQ [27]

GPU були обрані NVIDIA GeForce GTX 940, NVIDIA Tesla P100 та NVIDIA RTX 3080 Ti (у персональному ноутбуці, у Google Colab та на персональному ПК відповідно). Характеристики наведені на рис. 36, 37 та 38.

Frequency	
Graphics clock:	795 MHz
GPU boost:	2.0
Memory specifications	
Memory size:	2048 MB
Memory type:	GDDR5
Memory clock:	1253 MHz
Memory clock (effective):	5012 MHz
Memory interface width:	64-bit
Memory bandwidth:	40.10 GB/s
Cores / Texture	
CUDA:	5.0
CUDA cores:	384
ROPs:	8
Texture units:	24
RAMDACs:	400 MHz

Рис. 36 – Загальна характеристика GeForce GTX 940 [28]

Frequency	
Base clock:	1189 MHz
Boost clock:	1328 MHz
Memory specifications	
Memory size:	16 GB
Memory type:	HBM2
Memory clock:	715 MHz
Memory clock (effective):	1430 MHz
Memory interface width:	4096-bit
Memory bandwidth:	732.16 GB/s
Cores / Texture	
CUDA:	6.1
CUDA cores:	3584
ROPs:	96
Texture units:	224

Рис. 37 – Загальна характеристика GeForce Tesla P100[29]

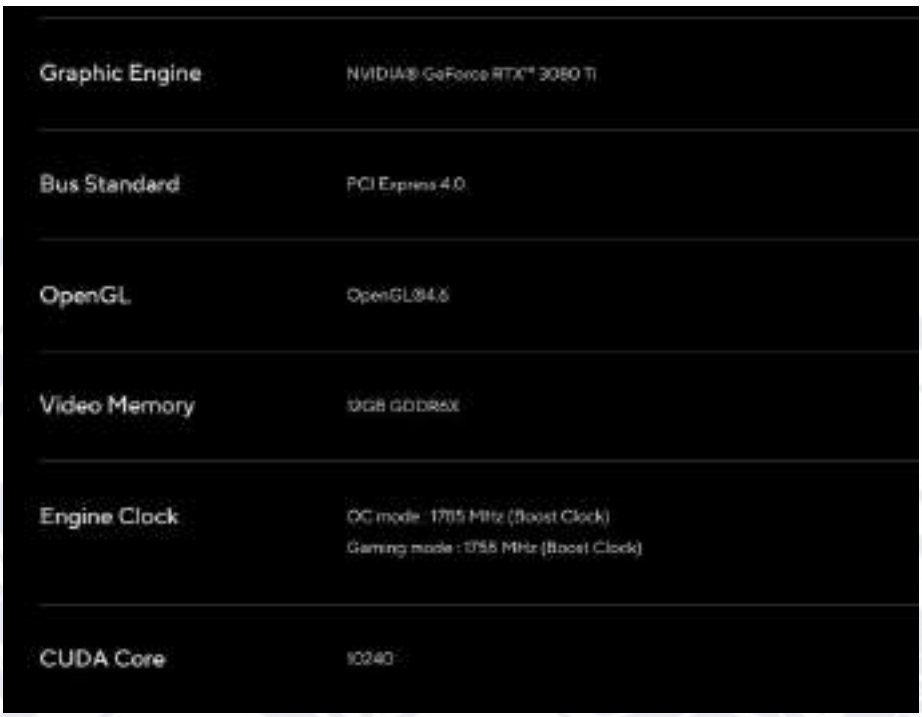

Рис. 38 – Загальна характеристика GeForce RTX 3080 [30]

Розглянемо детальніше представлені задачі. Як для центрального процесора, так і для графічного, матриці будуть представлені у вигляді одномірних масивів відповідного типу даних, що дозволить використовувати просту індексацію через зсуви, відповідно до розміру матриці (у даному випадку матриці є квадратними). Це дозволить швидше виконувати алокацію та адресацію, оскільки буде використовуватися одна неперервна ділянка пам'яті. Для обробки матриць у рамках цієї роботи не передбачено використання додаткових інструментів синхронізації між потоками, оскільки передбачається повний паралелізм даних для кожного окремого елемента матриці. Іншими словами, кожен потік буде працювати над окремими елементами, що дозволяє здійснити ефективне і незалежне обчислення для кожного елемента без необхідності синхронізації між потоками.

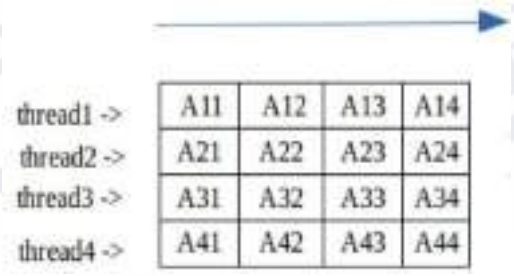

Рис. 39 – Схематичний зображення алгоритму обходу матриці

На рис. 39 представлений схематичний обхід матриці. Схематичний обхід матриці — це спосіб організації доступу до елементів матриці для виконання обчислень, що дозволяє ефективно розподіляти роботу між різними потоками або обчислювальними одиницями. У випадку паралельних обчислень, таких як CUDA, матриця може бути поділена на блоки, де кожен блок містить певний діапазон елементів, і кожен потік відповідає за обробку одного або кількох елементів в межах свого блоку.

У випадку двовимірної матриці, кожен потік може бути призначений для обробки конкретного елемента, що визначається його унікальними індексами. Використовуються вбудовані змінні, такі як blockIdx і threadIdx, щоб розподілити обчислення між потоками та блоками. Наприклад:

- blockIdx відповідає за індекс блоку, що дозволяє кожному блоку обробляти певну частину матриці.
- threadIdx дає унікальний ідентифікатор потоку всередині блоку, що дозволяє кожному потоку працювати з конкретними елементами цієї частини.

Такий підхід дозволяє ефективно розподіляти навантаження і максимально використовувати ресурси обчислювальних пристроїв, таких як GPU, забезпечуючи високий рівень паралелізму.

4.2 Оптимальна кількість потоків під час тестування на CPU

Спочатку визначимо оптимальний режим роботи центрального процесора (CPU). Для цього запишемо час виконання задач з різною кількістю потоків на різних процесорах. На графіках будуть значення, отримані після 5 запусків для кожної конфігурації, і лише після цього значення буде усереднюватися. Всі подальші графіки також базуватимуться на середніх значеннях з 10 запусків. Час вимірюватиметься в мілісекундах, а параметр size позначатиме розмір одного виміру квадратної матриці (якщо загально описати кількість елементів матриці, то вона дорівнює $size^2$)

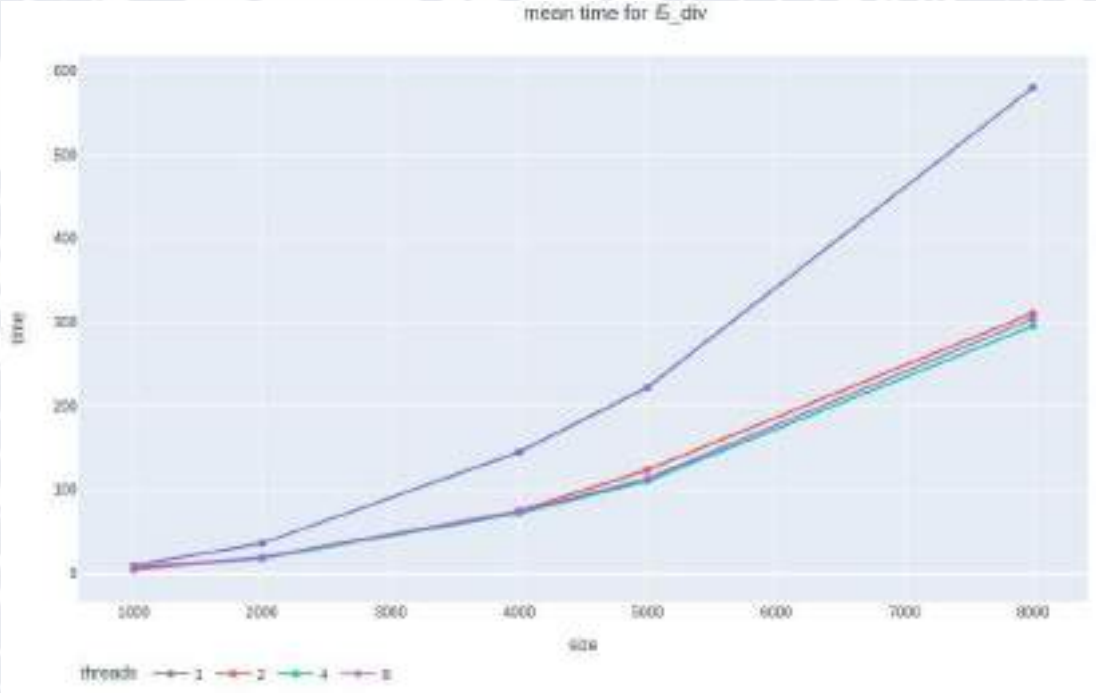

Рис. 40 – Графік виконання ділення матриці на число на i5-7300HQ

На рис. 40 можна побачити, що найкращу ефективність програма показала при використанні 4 потоків. При цьому, реальна ефективність з використанням 2 та 8 потоків також була близькою до найбільш оптимального результату.

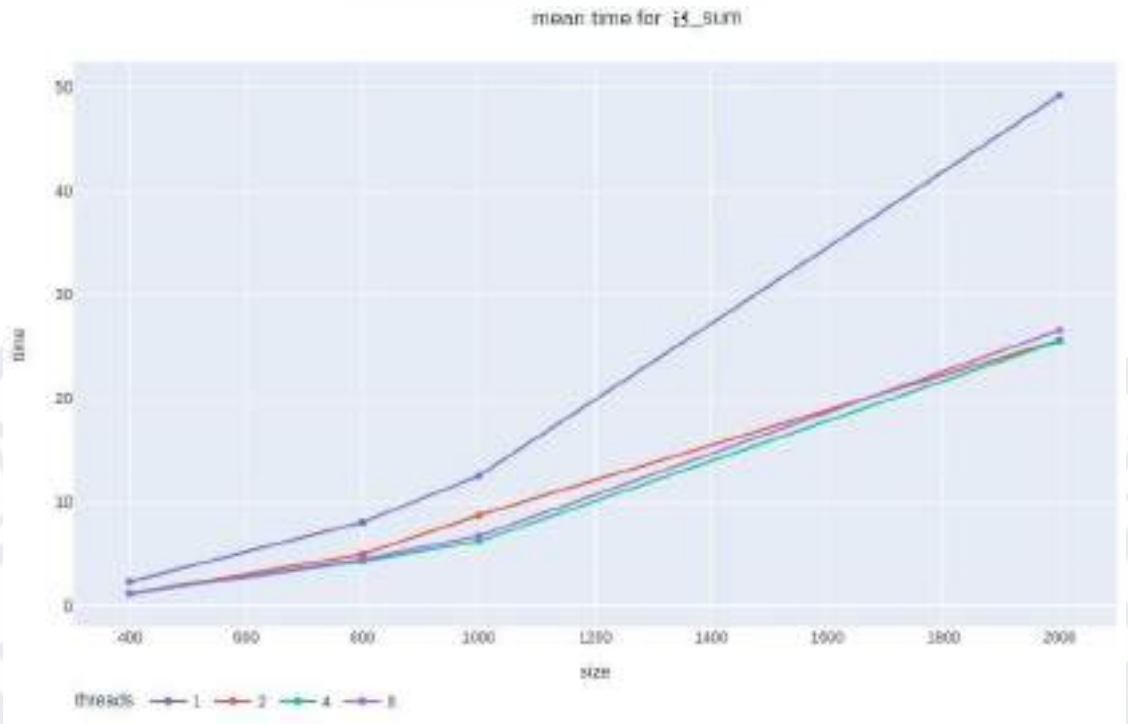

Рис. 41 – Графік виконання додавання матриць на i5-7300HQ

На рис. 41 ситуація повністю повторюється до рис. 40.

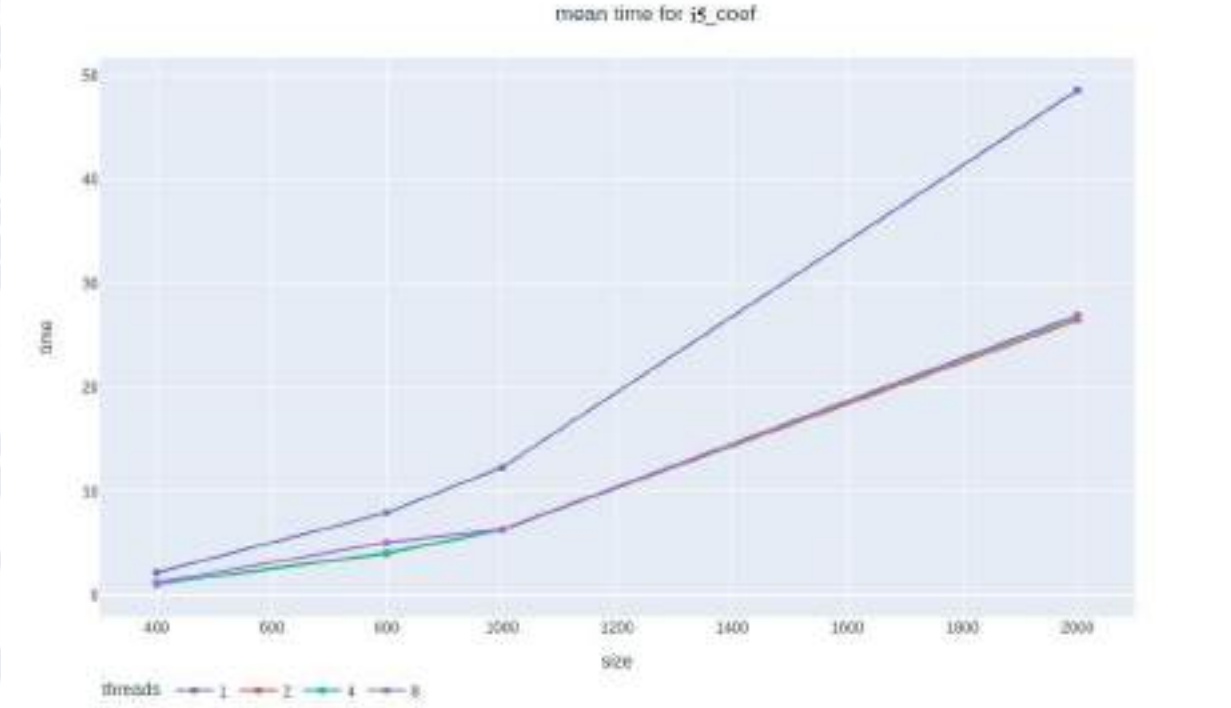

Рис. 42 – Графік виконання додавання матриць з використанням коефіцієнта на i5-7300HQ

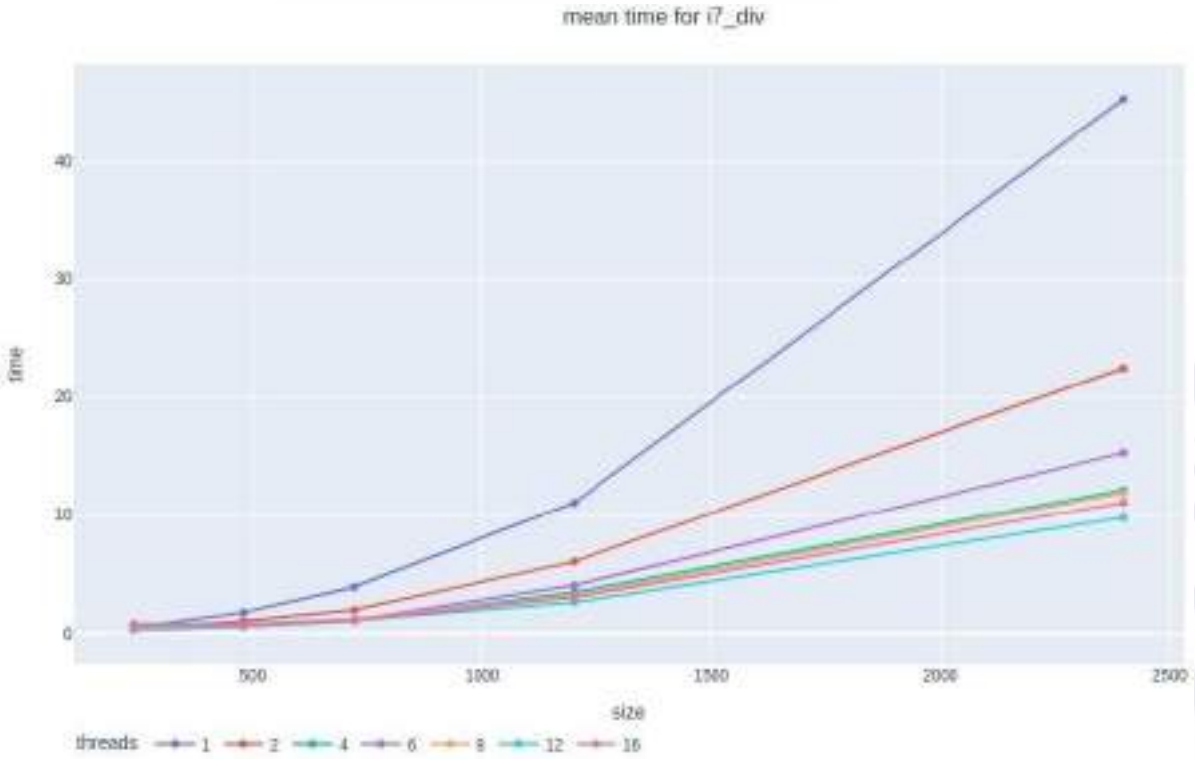

Рис. 43 – Графік виконання ділення матриці на число на i7-8700K

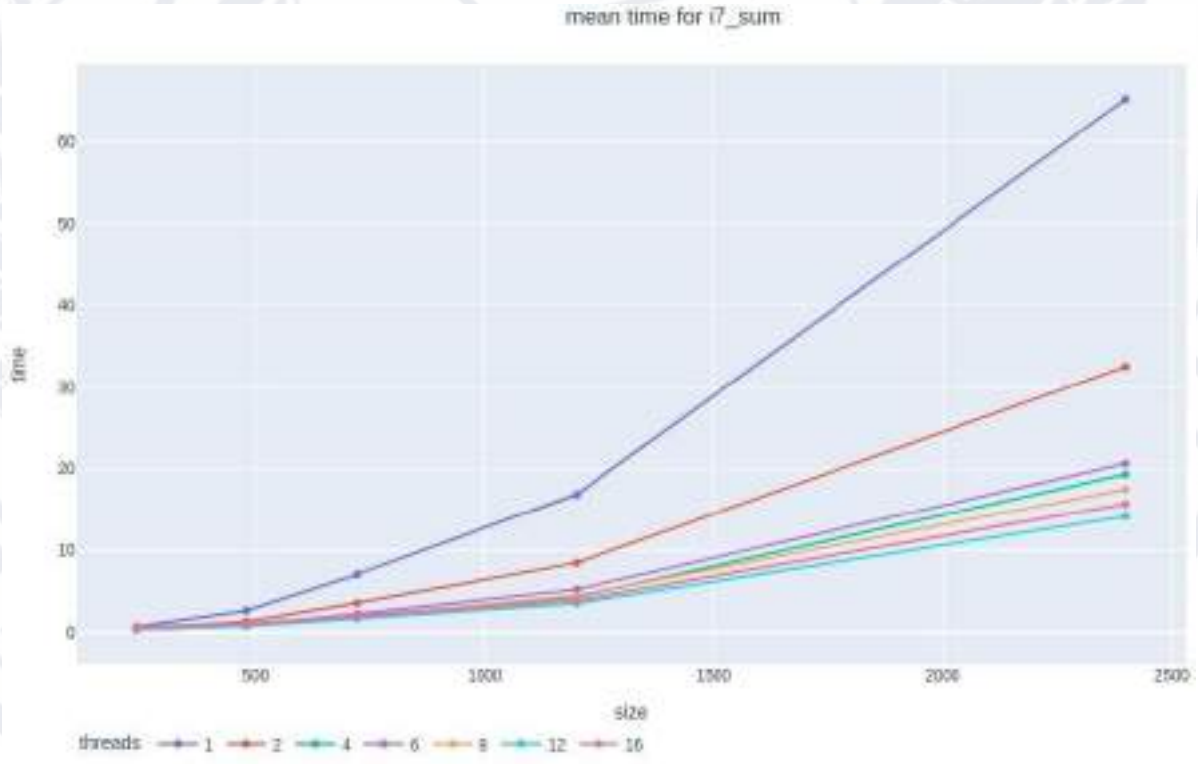

Рис. 44 – Графік виконання додавання матриць на i7-8700K

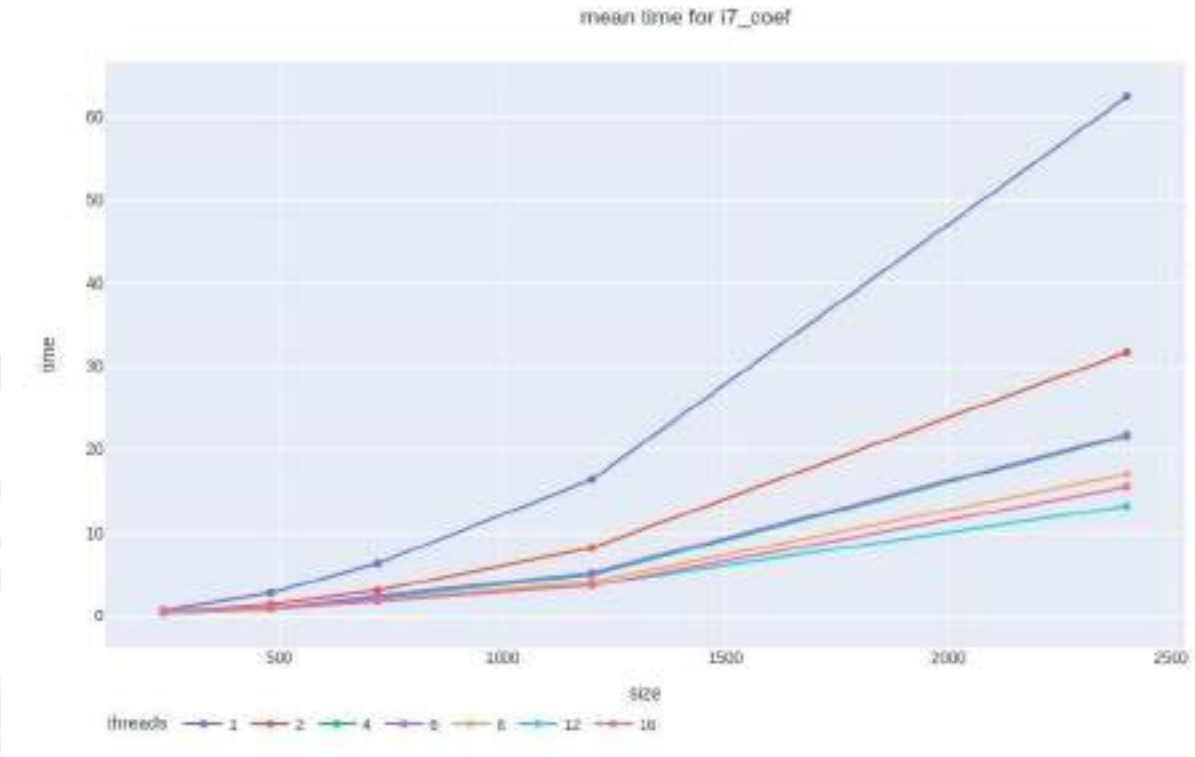

Рис. 45 – Графік виконання додавання матриць з коефіцієнтом на i7-8700K

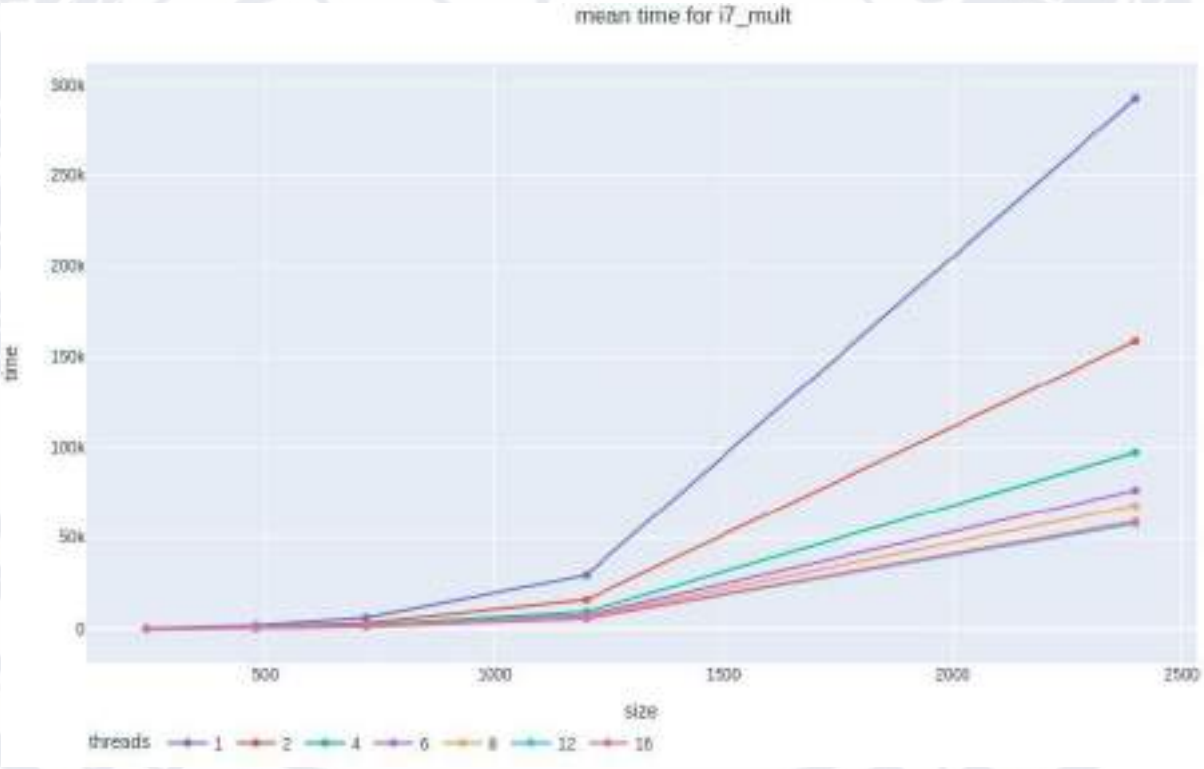

Рис. 46 – Графік виконання множення матриць на i7-8700K

З рис. 43 – 46 можна побачити, де представлені результати тестів на процесорі i7, видно, що оптимальним є використання 12 потоків. Додаткові потоки, які перевищують кількість, передбачену архітектурою, призводять до

зайвих витрат часу на їх створення, що знижує ефективність. Тести на процесорі i5 показують, що найкращий результат зазвичай досягається при використанні 4 потоків, хоча результати з 2 та 8 потоками не суттєво відрізняються від оптимального часу.

Розглянемо задачу скалярного добутку векторів. Розрахунок виконується за формулою $\vec{a} * \vec{b} = a_1 * b_1 + a_2 * b_2 + \cdots + a_n * b_n$. Для оптимізації цього процесу при виконанні паралельних обчислень можна використовувати шаблонний клас std::atomic та метод fetch_add. Це дозволяє зберігати результат у загальній змінній без блокування інших потоків, що значно підвищує ефективність.

Між std::mutex та std::atomic був обраний std::atomic, оскільки використання м'ютекса для синхронізації потоків призводить до більших затримок через необхідність блокування та розблокування доступу до змінної. Натомість атомарні операції забезпечують безпечний доступ до спільної змінної без блокування потоків, що дозволяє значно знизити час виконання та покращити ефективність паралельних обчислень при обчисленні скалярного добутку векторів.

Між std::mutex та std::atomic був обраний std::atomic, оскільки блокування за допомогою м'ютекса вимагає додаткових витрат часу на очікування та скасування блокувань, що призводить до значних затримок у виконанні. В той же час, атомарні операції, такі як fetch_add, дозволяють безпечний доступ до спільних змінних без необхідності блокувати інші потоки. Це дозволяє значно підвищити ефективність обчислень, зменшивши час на синхронізацію та забезпечивши більш швидке виконання паралельних операцій.

Реалізація потоку скалярного добутку відображена на рис. 4.15.

```
void vec_sum(std::vector<float> *vec1, std::vector<float> *vec2, std::atomic<float> &res, int start, int end)
{
    for (int i = start; i < end; ++i)
    {
        float interm = vec1->at(i) * vec2->at(i);
        res.fetch_add(interm, std::memory_order_relaxed);
    }
}
```

Рис. 47 – Реалізація потоку скалярного добутку

Результати задачі скалярного добутку на кожному з процесорів представлені на рис. 48 і рис. 49. Час в мілісекундах, розмірність вектора відображує розмір вектора який досліджується.

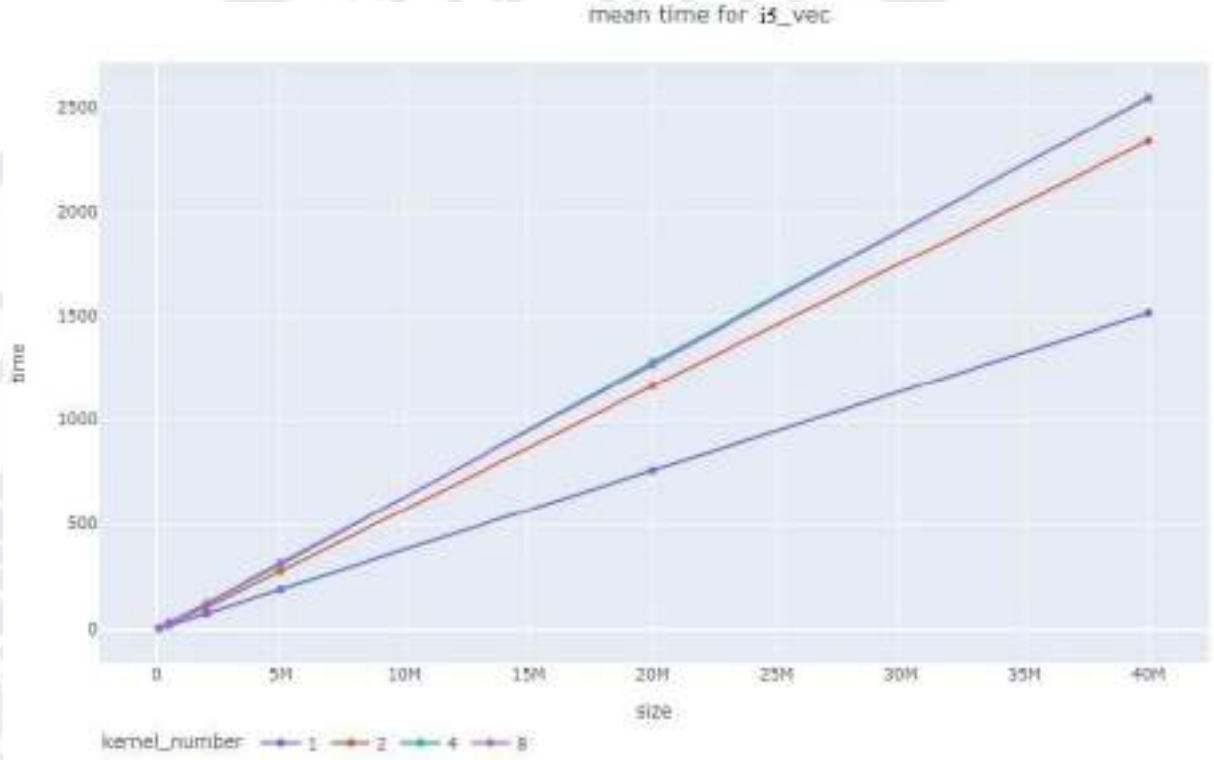

Рис. 48 – Графік виконання скалярного добутку векторів на i5-7300HQ

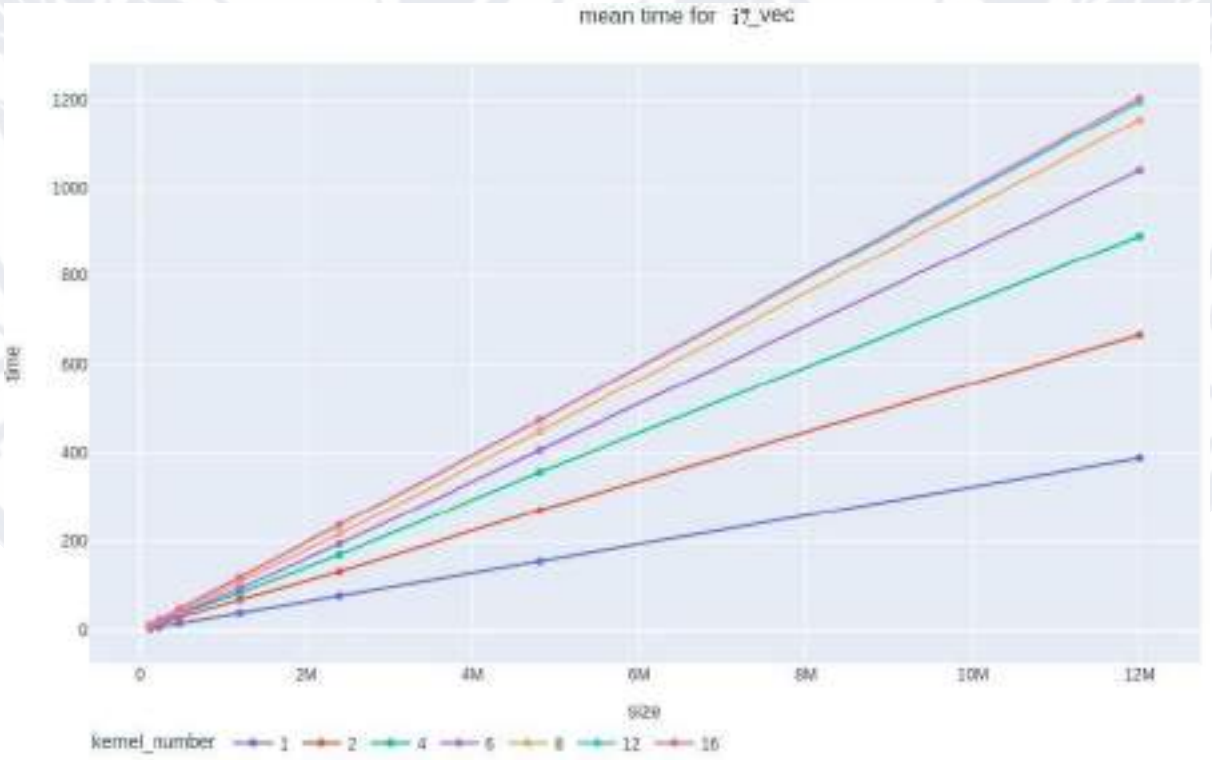

Рис. 49 – Графік виконання скалярного добутку векторів на i7-8700K

З отриманих результатів видно, що найкращий час виконання програми коли використовується лише один потік процесора.

З тестуваннння можна зробити висновок, що навіть атомарні операції не завжди сприяють прискоренню, а в даному випадку вони можуть навіть сповільнювати роботу. Причина цього полягає в тому, що після обробки кожної пари елементів векторів відбувається атомарне додавання результату до загальної змінної, що зберігає поточний підсумок скалярного добутку. Атомарні операції гарантують, що кожен потік безпечно оновлює значення змінної без потреби в блокуванні інших потоків. Це дозволяє мінімізувати накладні витрати на синхронізацію, у порівнянні з використанням м'ютексів, і значно збільшити ефективність паралельного виконання операції.

Для вирішення цієї проблеми була переписана функція, що виконується всередині кожного потоку, оскільки обробка елементів векторів потребувала ефективної синхронізації при паралельному обчисленні скалярного добутку. Для цього була застосована атомарна операція додавання, що дозволяє кожному потоку безпечно вносити свій результат в загальну змінну без блокування інших потоків. Така реалізація представлена на рис. 50

Тепер, після впровадження варіанту методу багатопоточного обчислення, кожен потік обробляє окрему пару елементів векторів, що дозволяє значно скоротити час виконання операції скалярного добутку. Результати представлені на рис. 51 та рис. 52.

Рис. 50 – Функція щоб разрахувати скалярний добуток

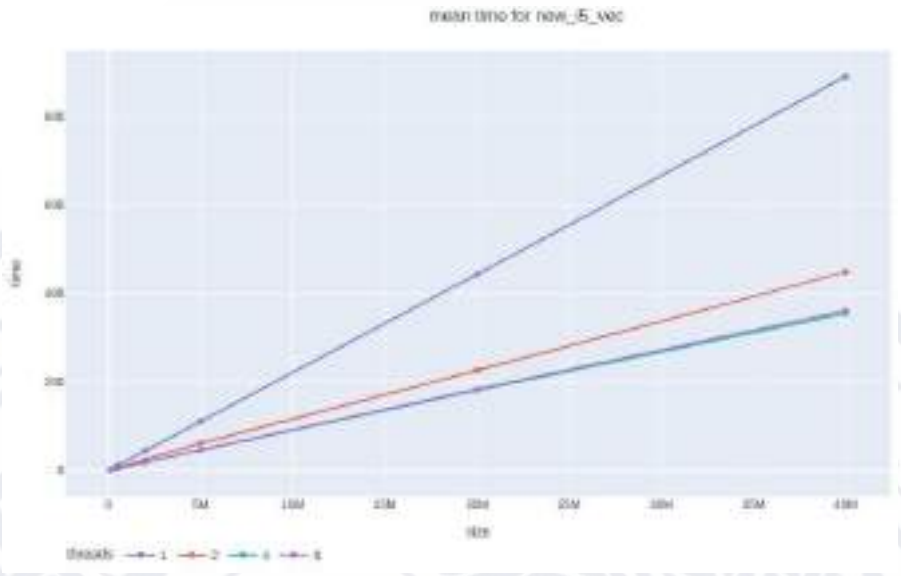

Рис. 51 – Графік виконання скалярного добутку на i5-7300HQ

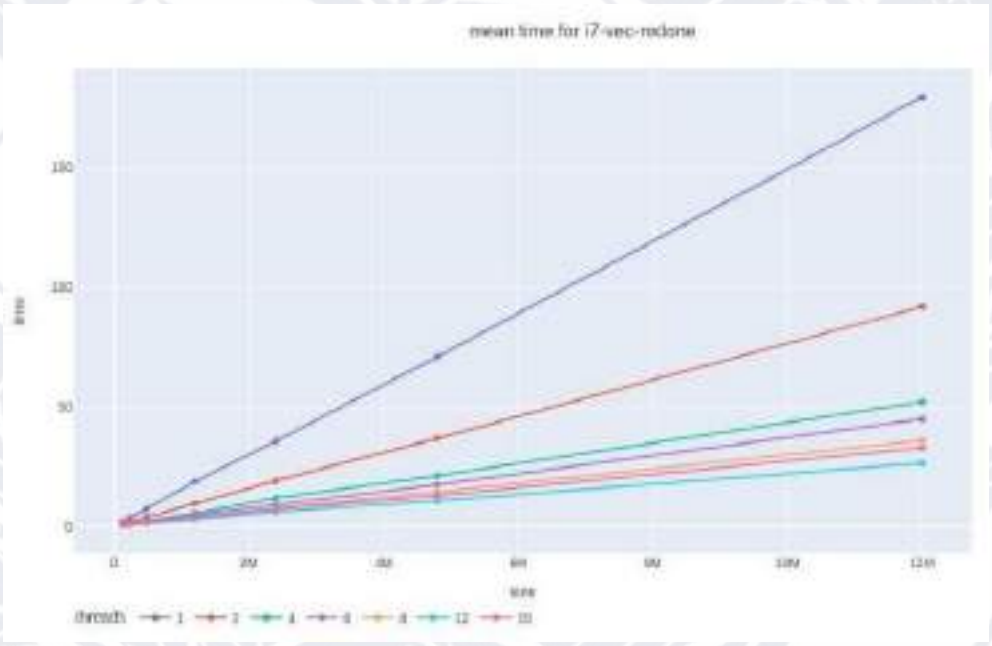

Рис. 52 – Графік виконання скалярного добутку на i7-8700K

Тепер ці результати повністю узгоджуються з висновками, отриманими під час обчислень над матрицями, і показують помітне поліпшення ефективності завдяки використанню багатопоточності. Враховуючи це, для порівняння продуктивності центрального та графічних процесорів, тести на процесорі i7 будуть виконуватись з 12 потоками, що дозволить максимально використовувати потужність багатоядерного процесора, а на процесорі i5 – з 4 та 8 потоками для оцінки впливу різних конфігурацій потоків на продуктивність.

4.3 Проведення тестувань

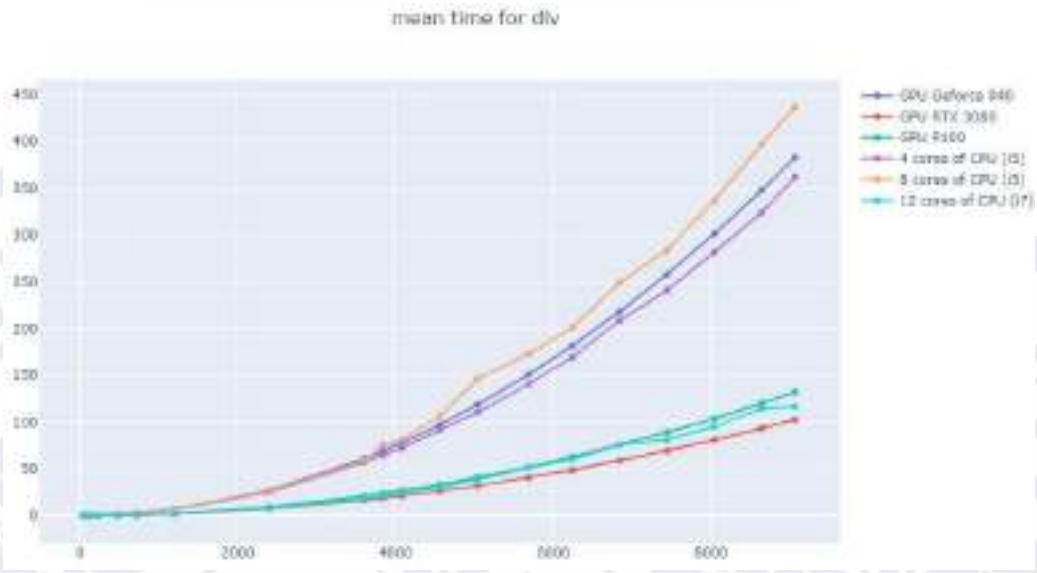

Рис. 53 – Графік виконання ділення матриці на різних CPU та GPU

На цьому рисунку можна побачити, що найкращі результати демонструє графічний процесор RTX 3080. Найгірші результати спостерігаються у випадку використання i5-7300HQ на 8 потоках. 4 потоки процесора i5-7300HQ показують кращий результат, ніж GeForce 940, а процесор Intel i7 працює ефективніше, ніж як Intel i5.

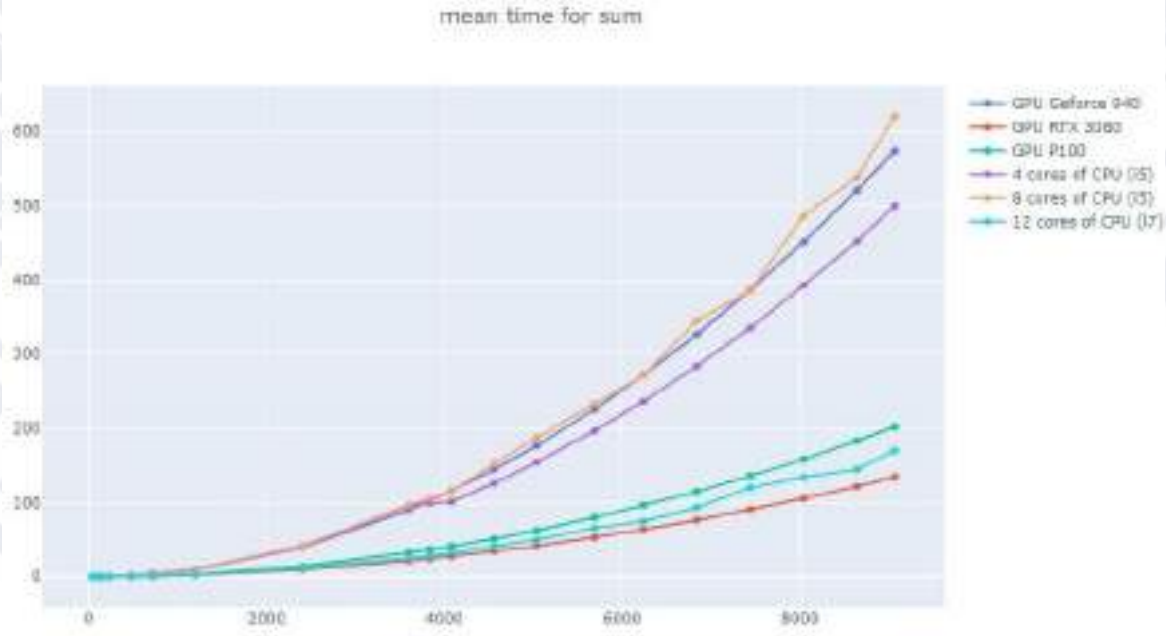

Рис. 54 – Графік виконання додавання матриць на різних CPU та GPU

Ці результати повністю відповідають тим, що представлені на рисунку 53.

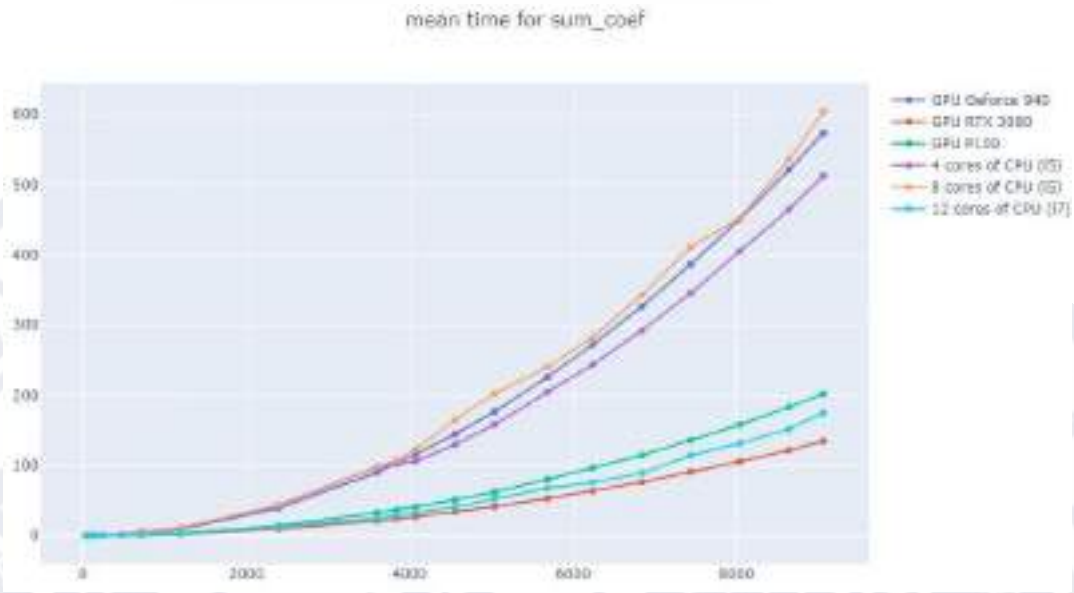

Рис. 55 – Графік виконання додавання матриць з коефіцієнтом на різних CPU та GPU

На рисунку 55 можна побачити, що ефективность пристроїв виявляються однаковими. Розглянемо результати операції додавання матриць, як без коефіцієнта, так і з ним, щоб зрозуміти, чи додає цей коефіцієнт значущу різницю в часі. Порівняємо середні значення часу виконання операцій додавання матриць без коефіцієнта та з коефіцієнтом для різних пристроїв.

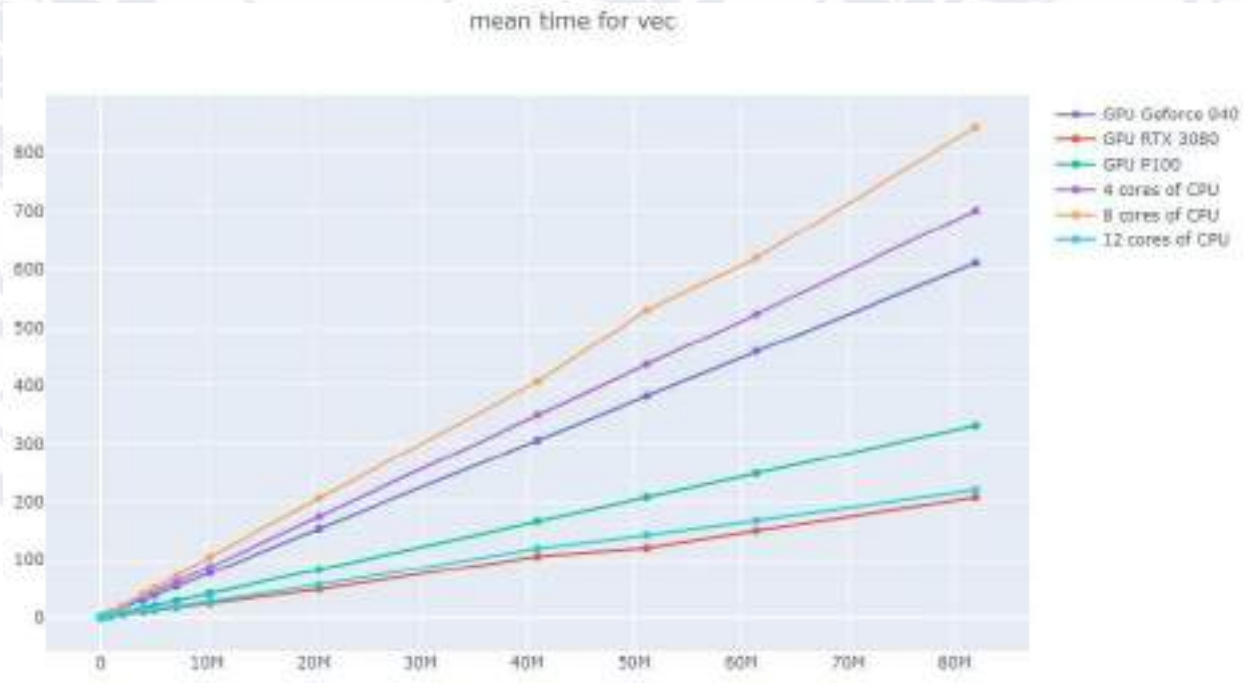

Рис. 56 – Графік виконання скалярного добутку двох векторів

З даних, наведених у додатку, видно, що помітної різниці в часі виконання операцій між цими двома варіантами не спостерігається.

На рисунку 56 видно, що тепер GeForce 940 демонструє кращі результати, ніж Intel i5. Для інших пристроїв результати залишилися незмінними. Варто зазначити, що всередині CUDA-потоків також використовувалась паралельна обробка для ефективного виконання операцій. Кожен потік обробляє певну частину завдання, що дозволяє максимізувати використання обчислювальних ресурсів графічного процесора. Це забезпечує значне прискорення обчислень порівняно з традиційними методами, де кожен процесорний потік виконує одну операцію, знижуючи час виконання завдань та підвищуючи загальну ефективність паралельних обчислень.

Проте, незважаючи на використання атомарних операцій, проблем зі сповільненням роботи не спостерігається, що свідчить про ефективність паралельної обробки даних на GPU.

Розглянувши рис. 57, можна помітити, що виконання множення матриць значно відрізняється від інших випадків.

У цьому випадку центральні процесори показують погані результати, тоді як всі графічні процесори працюють на оптимальному рівні. Це можна пояснити тим, що складність алгоритму множення матриць у потоках C++ має порядок $O(n^3)$, тоді як у потоках CUDA — $O(n)$.

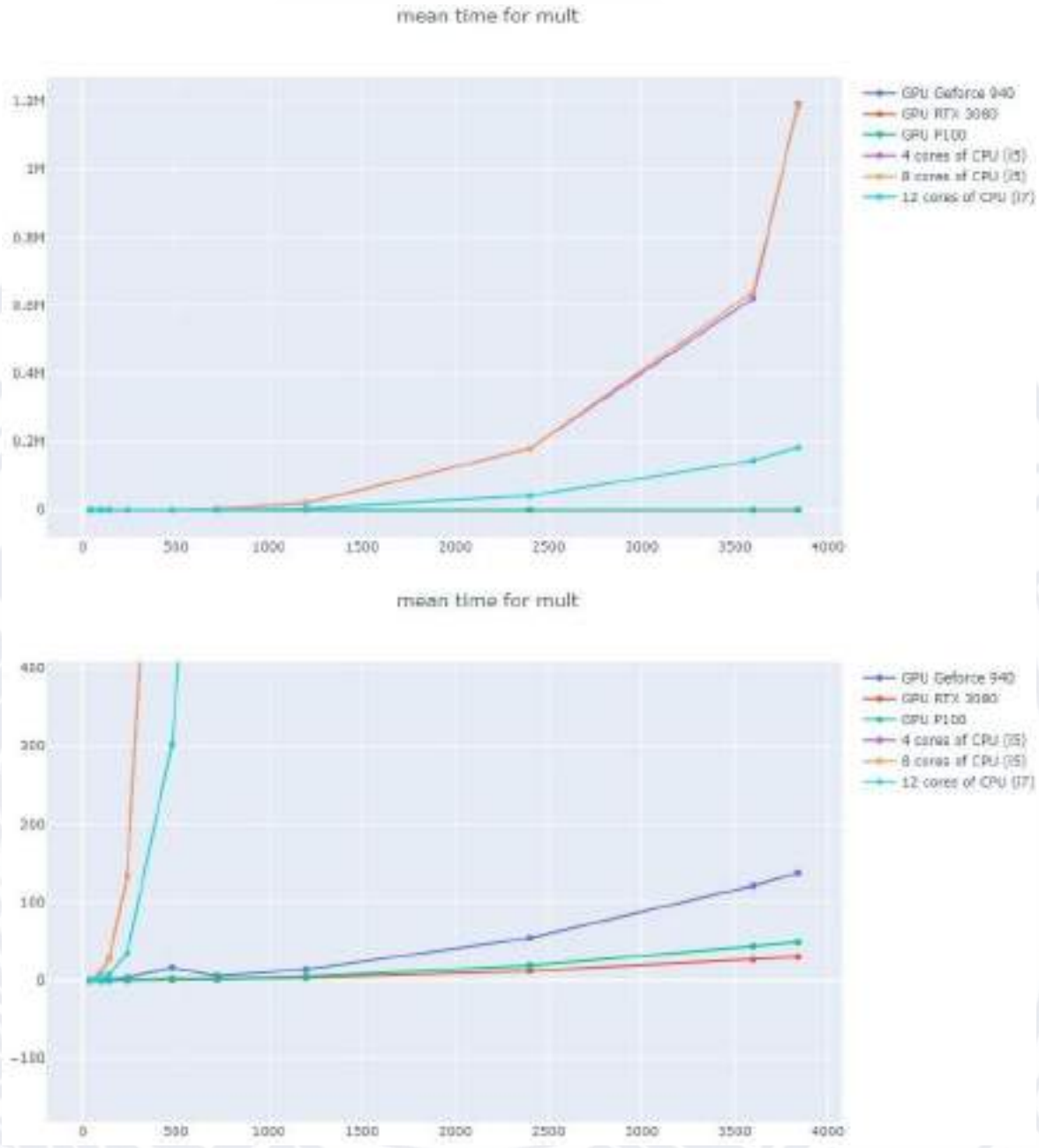

Рис. 57 – Графік виконання множення двох матриць

Це відбувається через те, що в C++ необхідно проходити трьома вкладеними циклами: перший цикл обирає рядки матриці, другий цикл проходить по стовпцях, а третій — по елементах кожного стовпця. У CUDA цей процес спрощується завдяки двовимірній індексації, що дозволяє усунути два перші цикли і значно прискорити операцію. Тому для множення матриць стандартним способом найкраще використовувати CUDA.

Аналізуючи результати з інших графіків, можна зробити висновок, що CPU не завжди поступається GPU в плані ефективності виконання обчислень.

У деяких випадках центральний процесор може показати конкурентоспроможні результати, особливо при менших розмірах оброблюваних даних або завданнях, де паралелізм не є критичним. Це свідчить про те, що вибір між CPU та GPU залежить від характеру завдання, обсягу даних і ефективності використання ресурсів кожного з процесорів.

Також варто зазначити, що при виконанні тестів на 8 потоках з більшими обсягами даних для Intel i5, оптимальна кількість потоків для цього процесора все ж склала 4, як і зазначено в технічній специфікації. Яка ж основна причина різниці в результатах між графічними процесорами з найкращим та найгіршим результатами? Чому GPU може показувати гірші результати, ніж CPU? Щоб краще розуміти це питання, перевіримо швидкості кожної операції окремо на прикладі задачі скалярного добутку (рис. 58).

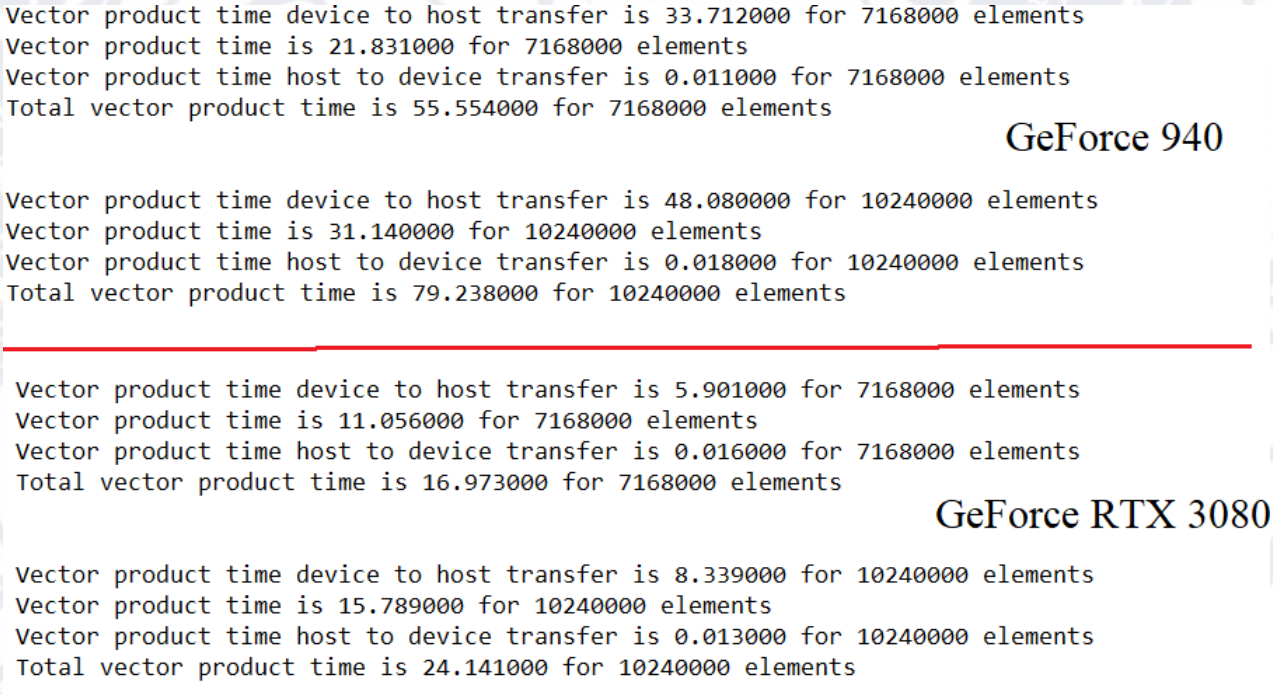

Рис. 58 – Порівняння швидкості виконання для кожної окремої операції

Операції на GPU можуть виконуватись значно швидше, однак важливо відзначити, що для GeForce 940 значна частина часу вимірювань витрачається саме на передачу даних. Іншими словами, при виборі GPU замість CPU, необхідно переконатися, що складність обчислень у потоках є достатньою, щоб виправдати витрати часу на передачу даних.

Спробуємо продемонструвати це на прикладах, де GeForce показував найгірші результати. Спробуємо продемонструвати це на прикладах, де GeForce показував найгірші результати. Розглянемо виконання задач на таких пристроях, як Intel i5 з 4 потоками, Intel i7 з 2 потоками, GeForce 940 та Tesla P100.

Аналізуючи продуктивність кожного пристрою, можна побачити, що у певних випадках GeForce 940 не показує перевагу над процесорами, особливо при виконанні менш об'ємних завдань або задач, де не вдається ефективно використати всі можливості паралельного обчислення. У той же час, Tesla P100, яка спеціалізується на високопродуктивних обчисленнях, показує значне прискорення в порівнянні з іншими пристроями завдяки своїм потужним обчислювальним можливостям і архітектурі, оптимізованій для паралельної обробки великих обсягів даних.

Замість виконання однієї операції в потоці (наприклад, додавання чи ділення), будемо повторювати її 2, 5, 10 разів. Оскільки, як було зазначено раніше, різниця в часі виконання для двох версій може бути значною, важливо враховувати не лише апаратні характеристики пристроїв, але й особливості самих завдань. Наприклад, деякі операції можуть бути більш ефективно виконані на процесорах, де паралелізм обмежений, в той час як інші, більш масштабні обчислення, можуть значно виграти при використанні графічних процесорів.

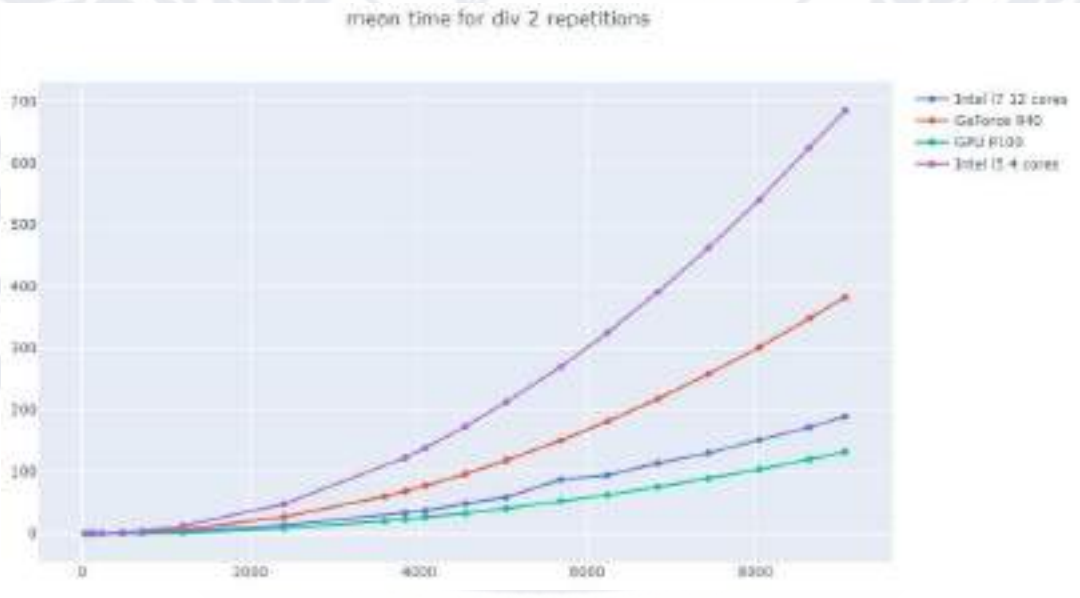

Рис. 59 – Графік виконання із повторенням операції 2 рази

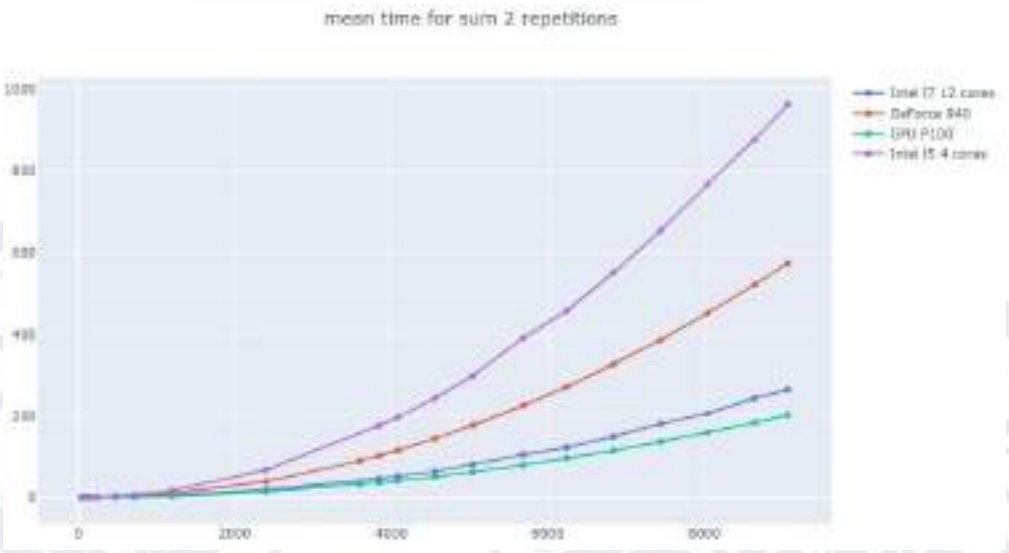

Рис. 60 – Графік виконання із повторенням операції 2 рази всередині потоку

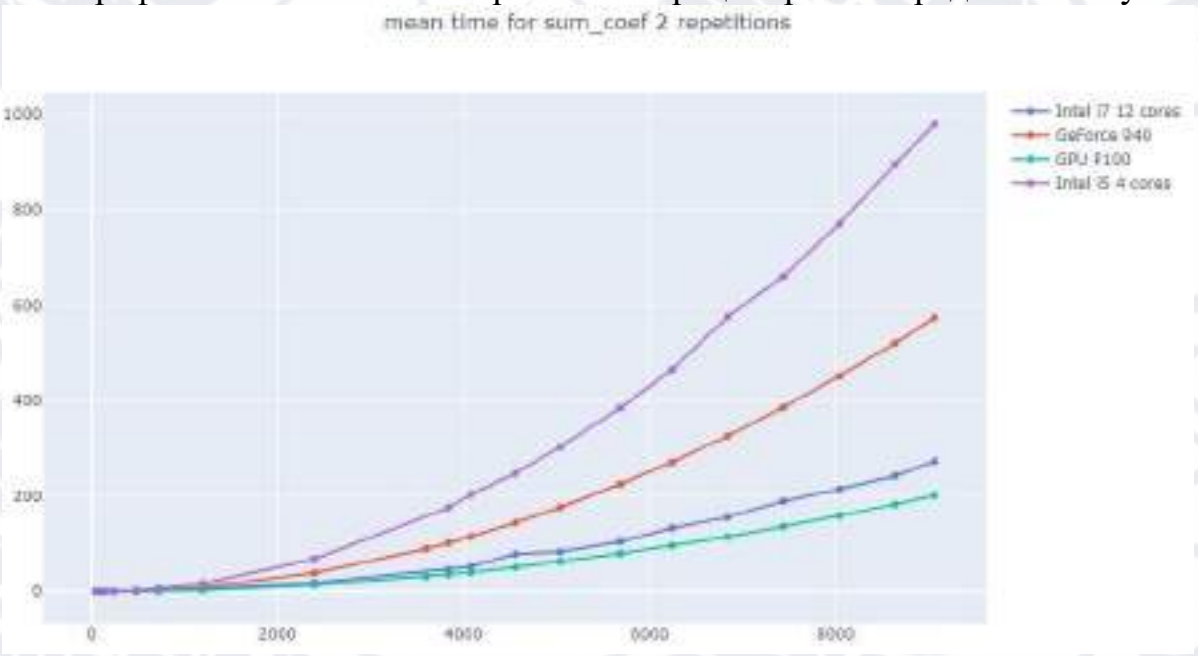

Рис. 61 – Графік виконання із коефецієнтом із повторенням операції 2 рази

Тепер, на графіках рис. 59 – 61, можна побачити, що GTX 940 обраховує швидше за Intel i5-7300HQ. Також ситуація з використанням Nvidia P100 та i7-8700K змінилася, і результати стали більш конкурентоспроможними.

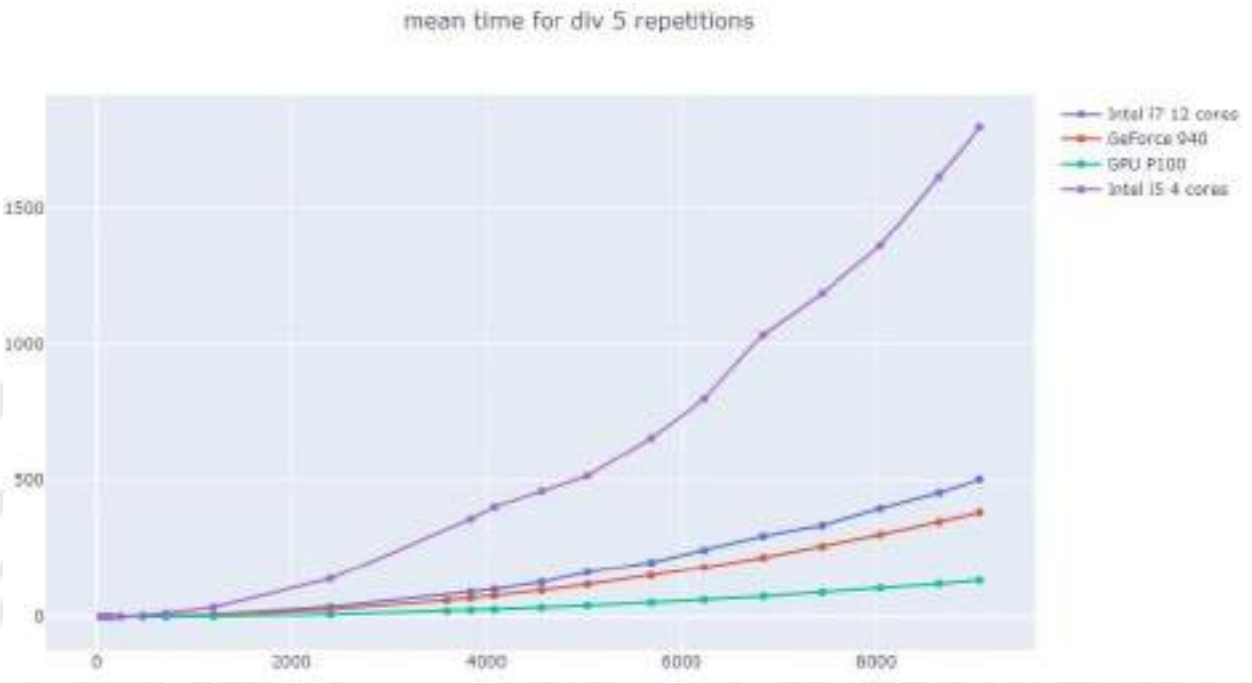

Рис. 62 – Графік виконання із повторенням операції 5 разів у середині потоку

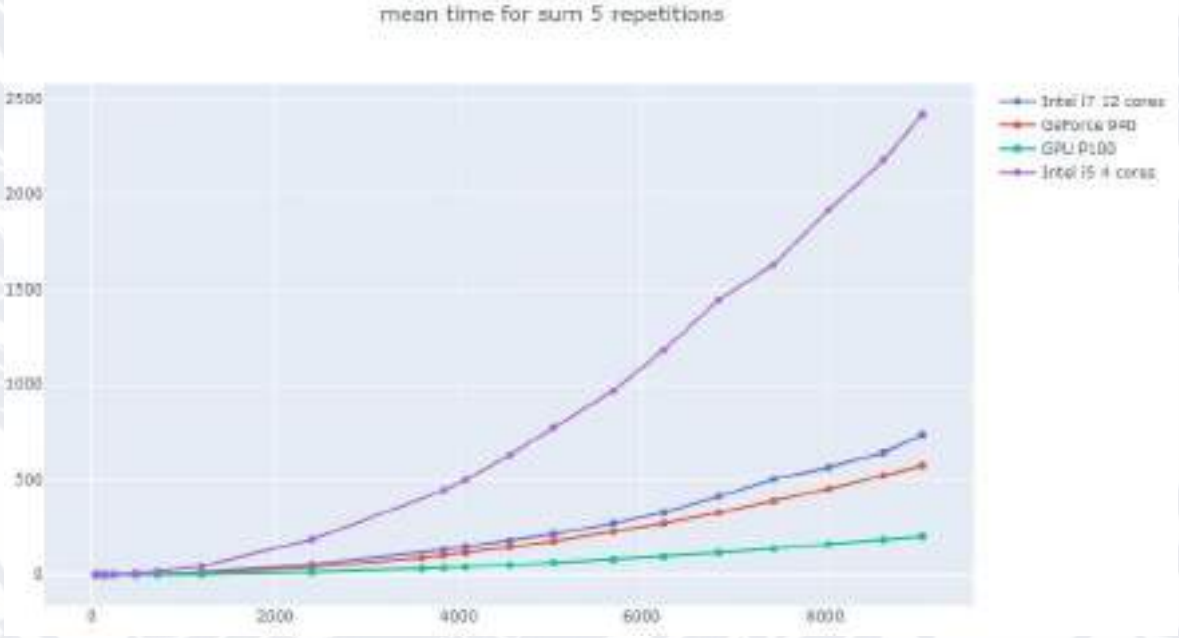

Рис. 63 – Графік виконання із повторенням операції 5 разів у середині потоку

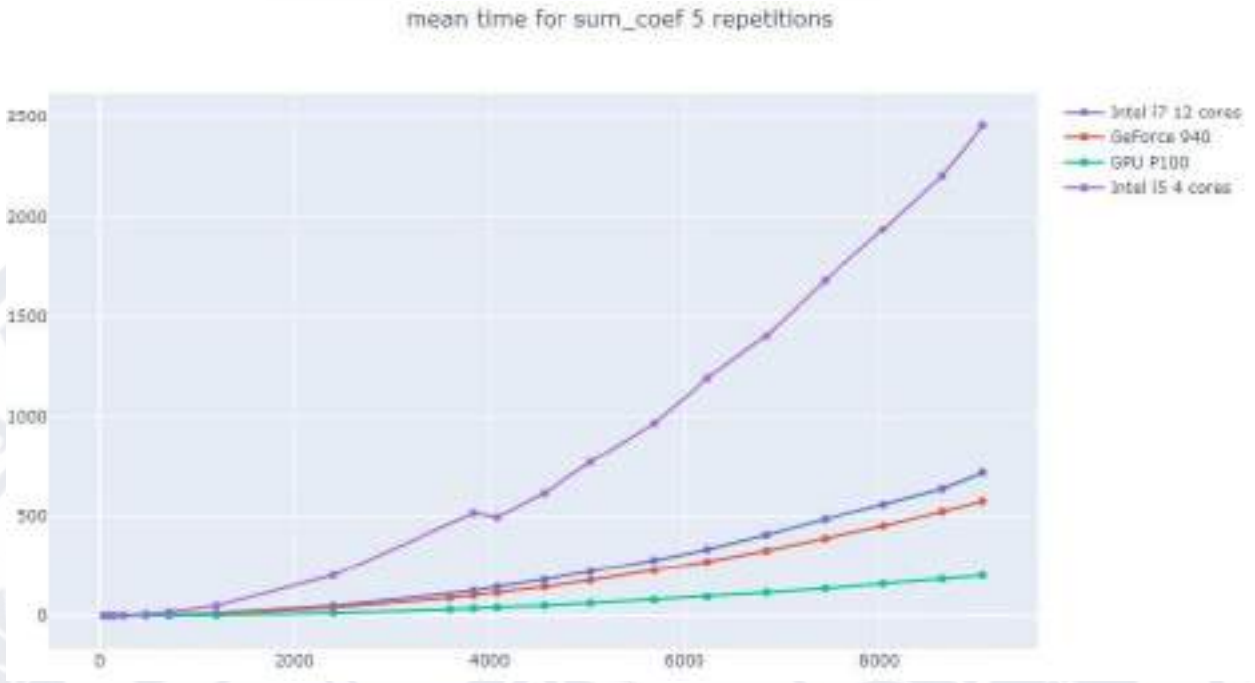

Рис. 64 – Графік виконання із коефецієнтом із повторенням операції 5 разів у середині потоку

Як можна побачити з графіків 62 – 64, ситуація змінилася порівняно з випадком з 2 потоками: всі GPU тепер показують кращі результати, ніж CPU.

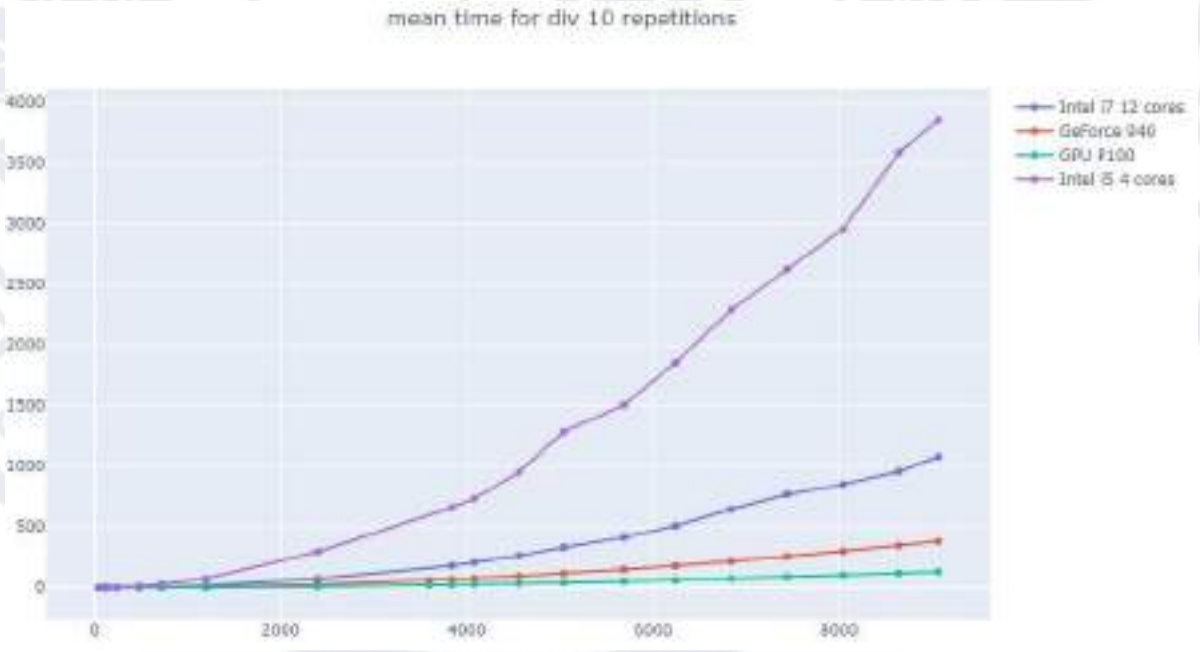

Рис. 65 – Ділення матриць із повторенням операції 10 разів всередині потоку

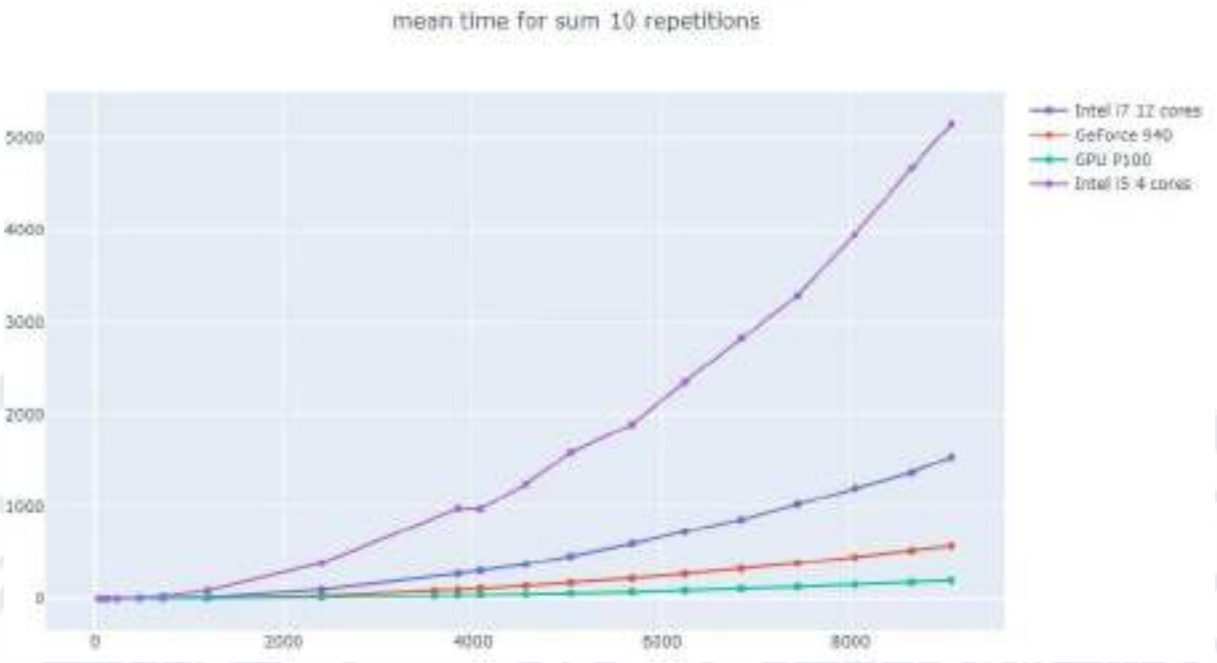

Рис. 66 – Додавання матриць із повторенням операції 10 разів всередині потоку

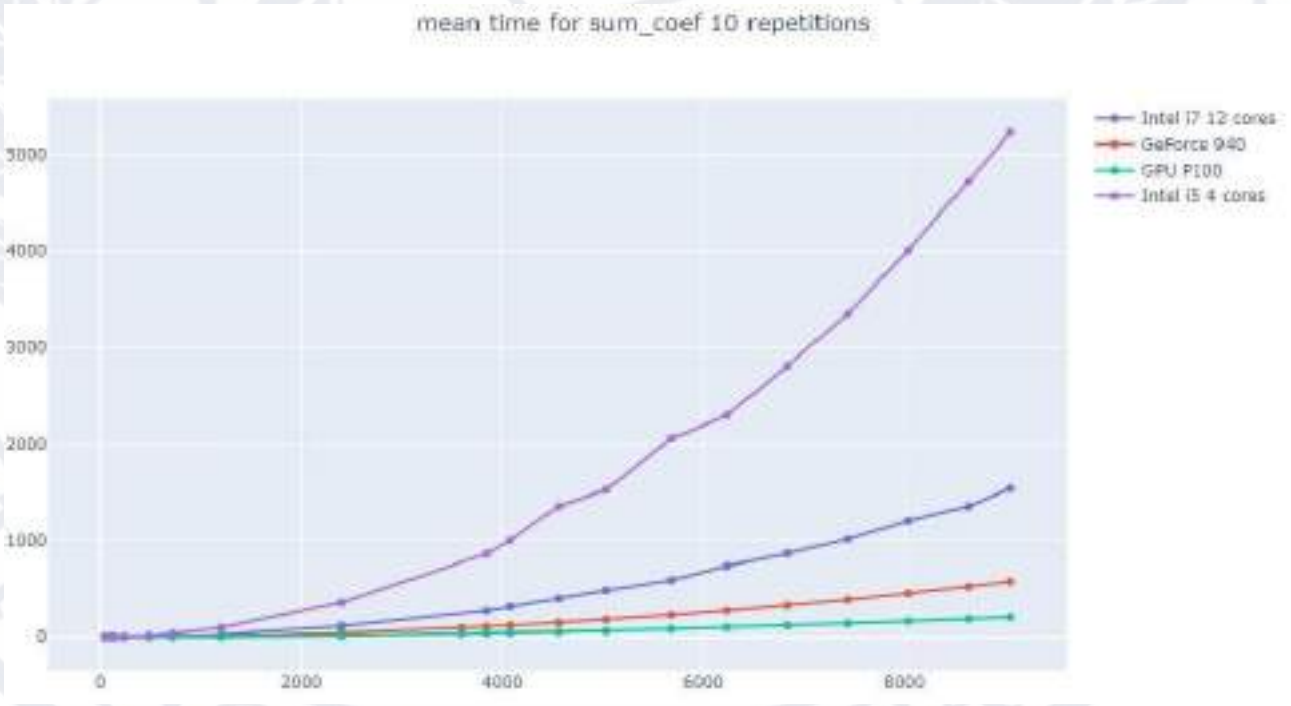

Рис. 67 – Додавання матриць із коефецієнтом із повторенням операції 10 разів всередині потоку

Як показують графіки 65 – 67, в таких умовах CPU демонструють гірші результати, ніж всі GPU.

Це дозволяє зробити висновок, що перевага GPU над CPU в основному залежить від співвідношення між витратами на трансфер даних та складністю обчислень, які виконуються в потоках. Коли в потоках проводиться достатня кількість операцій, GPU показують суттєву перевагу.

Отже, питання оптимальних розмірів даних, при яких варто використовувати GPU, є ключовим для досягнення максимальної ефективності обчислень. Для малих обсягів даних перевагу може мати CPU, оскільки запуск на GPU може не виправдати витрат на перенесення даних та налаштування обчислень.

А зараз давайте переглянемо графіки з виконанням початкових задач і перевіримо, чи спостерігаються значні зміни в продуктивності пристроїв при менших розмірах даних (див. графіки 68 – 72).

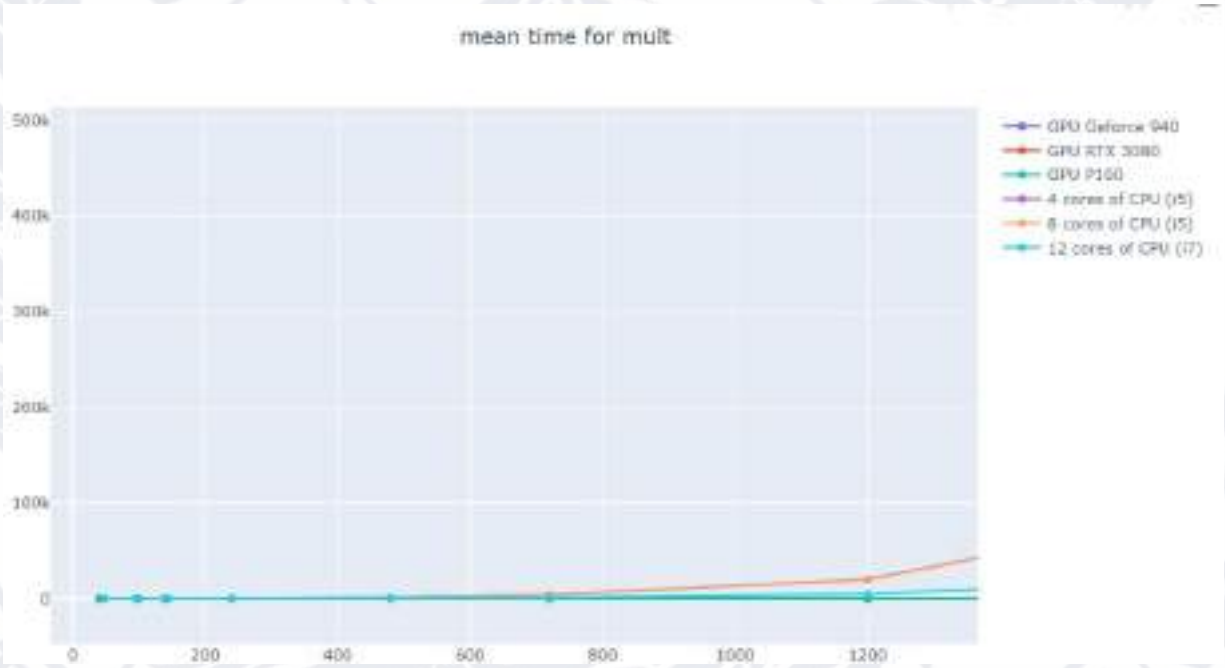

Рис. 68 – Графік виконання множення матриць у найближчому масштабі

Подивившись на графік 68 можна побачиити, що в загальному не відбуваєтсья будь-яких змін чи різких переходів у зміні ефективності пристроїв в залежності від вибраних розмірності даних.

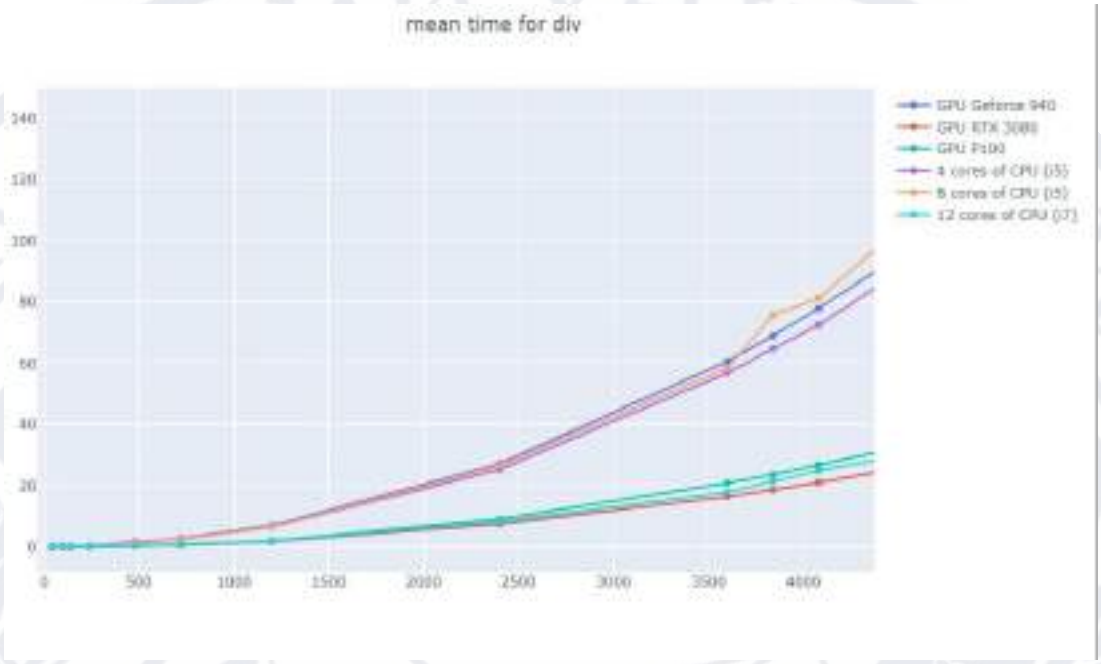

Рис. 69 – Графік виконання ділення матриці на число у найближчому масштабі

Загалом, картина ефективності залишається стабільною: рішення на CPU починає показувати помітно гірші результати, коли розмір матриці розпочинається з 200×200. Водночас, під час обчислення матриць 1000×1000 результати GPU залишаються подібними, що свідчить про сталість продуктивності і ефективності графічних процесорів навіть при значному збільшенні розміру відібраних даних.

На рисунку 69 також немає чітко вираженої тенденції, однак розподіл на дві явні групи за працездатністю стає помітним при розмірності близько 700x700. Це вказує на те, що при збільшенні обсягу даних починають проявлятися різні показники ефективності, де для деяких розмірів завдання обробка стає більш оптимізованою, а для інших — менш ефективною, залежно від характеристик використовуваних пристроїв чи алгоритмів..

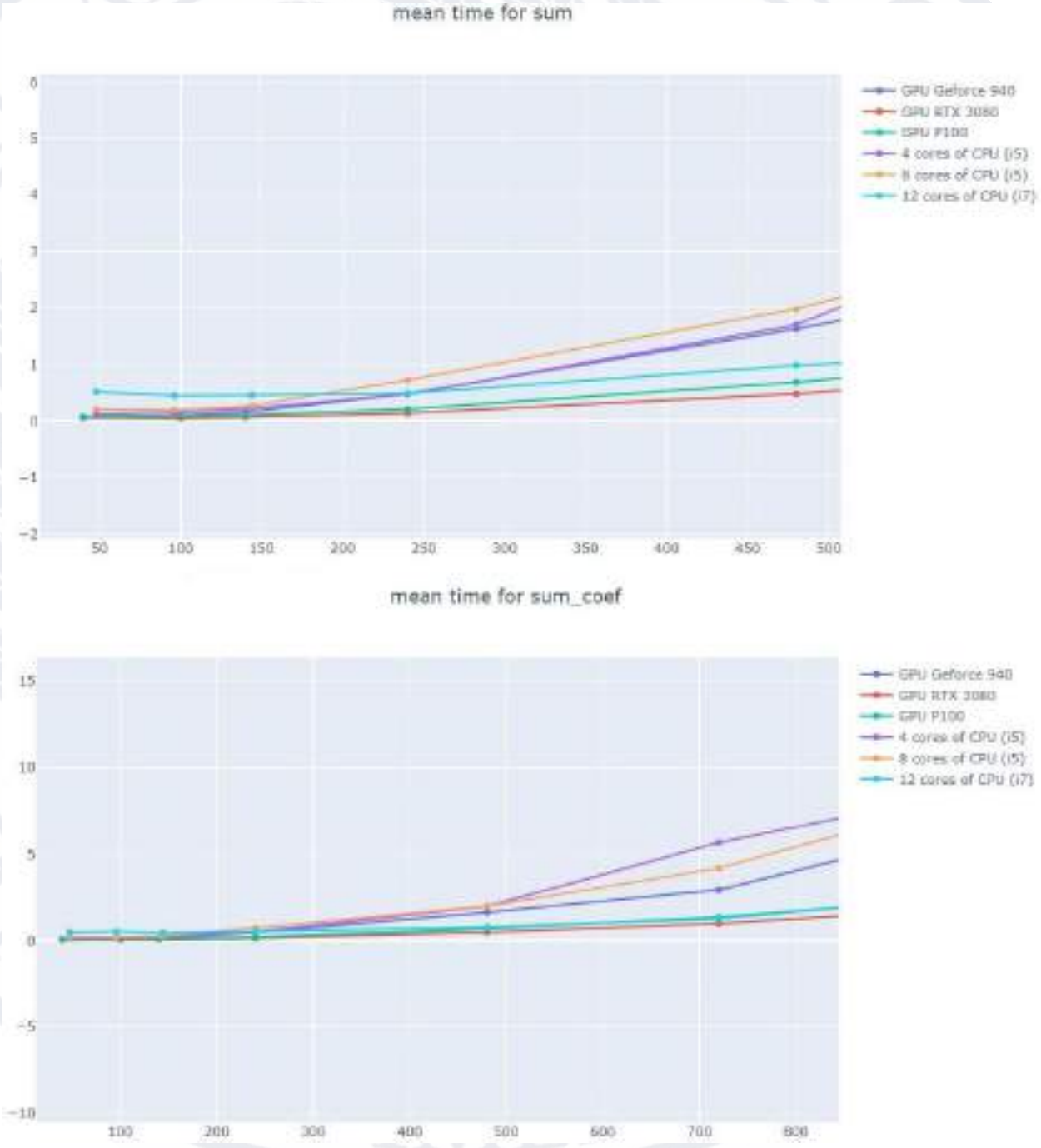

Рис. 70 – Графік виконання додавань

На графіку 70 представлені дві практично ідентичні групи результатів. Однак, при розмірності близько 200 елементів, можна помітити певні зміни в тенденціях. Це вказує на те, що для малих розмірів даних продуктивність залишається стабільною, але при досягненні певного порогу починають проявлятися відмінності в ефективності обробки, що може бути пов'язано з оптимізацією алгоритмів чи ресурсами, доступними на різних пристроях..

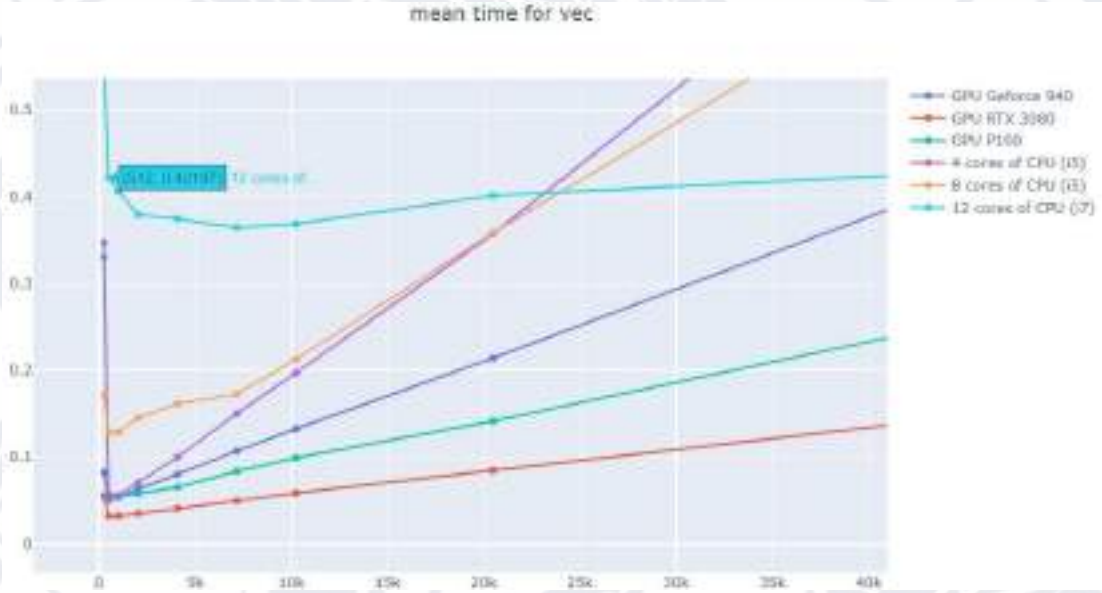

Рис. 71 – Графік виконання додавань скалярного добутку векторів у найближчому масштабі

Для задачі скалярного добутку векторів характерним є те, що на початку, при малих розмірах (256 елементів), ефективність обробки на CPU та GPU майже однакова. З початком роботи з 512 елементами, GPU починає показувати кращі результати, оскільки здатен ефективніше обробляти більші обсяги даних завдяки паралельному виконанню. Однак, при розмірі даних близько 50 000 елементів Intel i7 починає демонструвати значні затримки, оскільки CPU не здатен ефективно обробляти таку кількість даних порівняно з GPU. Це підтверджує, що для великих обсягів даних перевага використання GPU стає очевидною.

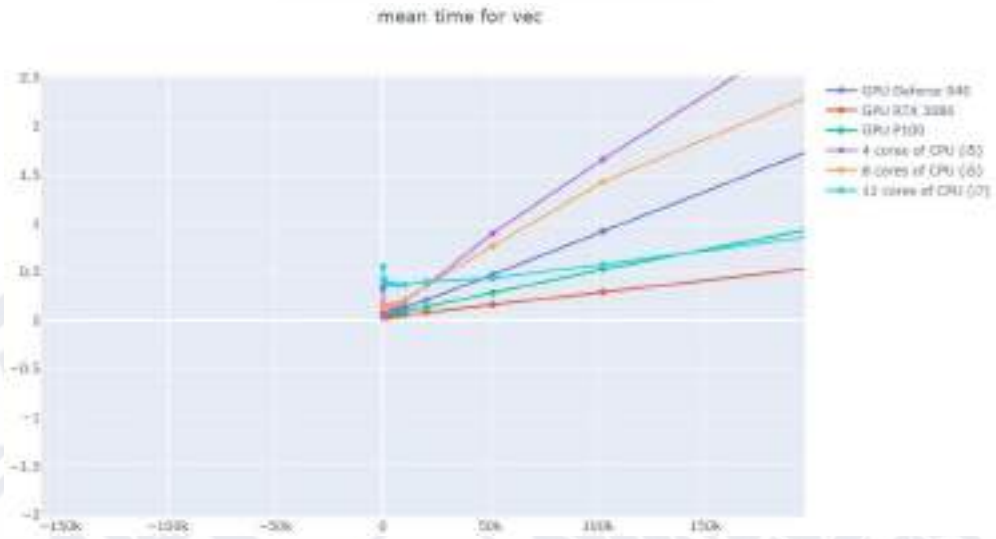

Рис. 72 – Графік виконання додавань скалярного добутку векторів у найближчому масштабі

4.4 Висновок до розділу

У цьому розділі були проведені тести з використанням різних центральних і графічних процесорів. Результати показали, що застосування GPU не завжди є ефективнішим для всіх SIMD-задач, оскільки для деяких малих розмірів даних або специфічних задач центральний процесор може працювати ефективніше. Проте, в більшості випадків, зокрема при великих обсягах даних, графічний процесор демонстрував значно кращі результати. Тому RTX 3080 стабільно показувала найкращі показники продуктивності в порівнянні з іншими пристроями, особливо при виконанні паралельних обчислень на великих масивах даних.

Важливими факторами, що визначають ефективність, є швидкість передачі даних між процесором і пам'яттю, а також здатність обробляти великий обсяг даних одночасно. Для GPU це включає в себе високу пропускну здатність пам'яті та ефективну організацію обробки даних у паралельному режимі. У свою чергу, на CPU ефективність часто обмежується кількістю ядер і частотою їх роботи, що робить його менш ефективним для завдань, що потребують обробки великих обсягів даних за короткий проміжок часу.

ВИСНОВКИ

Метою цієї дипломної роботи було дослідити, при яких розмірах вхідних даних використання GPU є оптимальним для вирішення задач SIMD. Зокрема, робота фокусується на виявленні порогових значень розміру даних, при яких GPU може значно перевищити ефективність CPU. Завдяки паралельній архітектурі графічних процесорів, вони можуть обробляти велику кількість елементів одночасно, що дозволяє досягти значного прискорення для певних обчислювальних завдань, зокрема таких, як множення матриць, скалярний добуток векторів та інші операції, характерні для SIMD-обчислень. Тестування на різних розмірах даних допомогло з'ясувати, на яких етапах GPU починає демонструвати переваги перед традиційними центральними процесорами.

.Для проведення експериментів були використані процесори Intel Core i5 7300HQ та Intel i7-8700K, а також тестування на платформі CUDA здійснювалося за допомогою GPU NVIDIA GeForce GTX 940, NVIDIA Tesla P100 та RTX 3080. Результати досліджень показали, що відеокарта RTX 3080 стабільно показувала найкращі результати серед усіх пристроїв. Проте також було виявлено, що не завжди GPU демонструють кращу продуктивність за CPU.

Зокрема, для задачі множення матриць всі GPU продемонстрували високу ефективність, що пояснюється тим, що вбудована 2D-індексація потоків дозволяє ефективно розподіляти обчислення на численні потоки, кожен з яких обробляє свою частину матриць. Це дозволяє одночасно виконувати операції множення для великої кількості елементів матриць, максимально використовуючи потенціал паралельної архітектури графічного процесора. Такий підхід значно скорочує час виконання операції, порівняно з CPU, де кожен потік зазвичай обробляє лише одну операцію за раз, без такої ефективної паралелізації.

Однак для інших задач спостерігалися випадки, коли GPU працювали гірше за CPU. 12-потоковий Intel i7-8700K показував кращі результати, ніж NVIDIA Tesla P100 та NVIDIAвав GeForce 940. У 3 з 5 випадків GeForce 940 демонструвала гірші результати порівняно з 4-потоковим Intel i5, що призводило до найбільших затримок у виконанні.

Подальший аналіз часу передачі даних між хостом і пристроєм, а також чистого часу виконання обчислень показав, що для відеокарт з нижчою продуктивністю основним чинником уповільнення були саме витрати часу на переміщення пам'яті. Вартість цих переміщень не компенсувалася часом виконання самих операцій, що й обумовлювало гірші результати таких GPU.

Для перевірки цієї тези в задачі додавання та ділення матриці на число була внесена зміна: базова операція додавання або ділення виконувалася кілька разів підряд. Після проведення цього тесту одразу стали помітні суттєві зміни, а саме у ефективності роботи пристроїв. Зокрема, в разі використання 2 і 5 потоків, Tesla P100 показала кращий результат, ніж Intel i7, а GeForce 940 впоралась краще за Intel i5.

Проте, Intel i7 все одно виконав задачу за менший час, ніж Intel i5. У випадку з 10 потоками ситуація змінилася так, що всі GPU стали швидшими за всі CPU.

Таким чином, питання оптимального розміру даних для роботи з GPU можна вважати другорядним. Чистий час обробки на GPU значно менший за час обробки на CPU, тому збільшення складності операцій не має такого критичного впливу на час роботи GPU, як на центральний процесор.

Підсумовуючи, новітні GPU, завдяки вдосконаленим апаратним можливостям передачі даних, ймовірно, виявляться більш ефективними в порівнянні з попередніми моделями. Це зумовлено низкою факторів, серед яких підвищена пропускна здатність пам'яті, кращі алгоритми управління потоками та покращена архітектура, що дозволяє ефективніше обробляти велику кількість одночасних обчислень. Крім того, сучасні GPU забезпечують швидший доступ до даних, завдяки чому зменшується час затримки між виконанням операцій, що особливо важливо при роботі з великими наборами даних або складними обчислювальними задачами.

ДЖЕРЕЛА

1. The Story of the Intel® 4004 [Веб-сайт] // Intel. – Режим доступу: https://www.intel.co.uk/content/www/uk/en/history/museum-story-of-intel-4004.html

2. Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming : Addison-Wesley / Edward Kandrot, Jason Sanders. – [Б. м. : б. в.]. – 424 с.

3. Moore G. E. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. [Веб-сайт] / Gordon E. Moore // IEEE Solid-State Circuits Society Newsletter. – 2006.–Т. 11,№ 3. – С. 33–35. – Режим доступу: https://doi.org/10.1109/n-ssc.2006.4785860 – .

4. Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions [Веб-сайт] / R. H. Dennard [та ін.] // Proceedings of the IEEE. – 1999. – Т. 87, № 4. –С. 668–678. – Режим доступу: https://doi.org/10.1109/jproc.1999.752522 – .

5. Contributors to Wikimedia projects. Transistor count - Wikipedia [Веб-сайт] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. – Режим доступу: https://en.wikipedia.org/wiki/Transistor_count#Microprocessors – .

6. CPU DB - Looking At 40 Years of Processor Improvements | A complete database of processors for researchers and hobbyists alike. [Веб-сайт] // cpudb.stanford.edu.– Режим доступу: http://cpudb.stanford.edu/visualize/clock_frequency – .

7. Bohr M. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper [Веб-сайт] / Mark Bohr // IEEE Solid-State Circuits Newsletter. – 2007. Т. 12, № 1. – С. 11–13. – Режим доступу: https://doi.org/10.1109/n-ssc.2007.4785534 – .

8. Contributors to Wikimedia projects. Multi-core processor - Wikipedia [Веб-сайт] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. – Режим доступу: https://en.wikipedia.org/wiki/Multi-core_processor – .

9. Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms / Roman Trobec [та ін.]. – [Б. м.] : Springer, 2018. – 268 с.

10. Barlas G. Multicore and GPU Programming: An Integrated Approach / Gerassimos Barlas. – [Б. м.] : Elsevier Science & Technology Books, 2014. – 698 с.

11. Ilg M. Projectile Monte-Carlo Trajectory Analysis Using a Graphics Processing Unit [Електронний ресурс] / Mark Ilg, Jonathan Rogers. –Режим доступу: https://www.researchgate.net/publication/268011284_Projectile_Monte-Carlo_Trajectory_Analysis_Using_a_Graphics_Processing_Unit

12. Contributors to Wikimedia projects. Non-uniform memory access - Wikipedia [Веб-сайт] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. – Режим доступу: https://en.wikipedia.org/wiki/Non- uniform_memory_access – .

13. Neelap A. K. Master's thesis in Electrical Engineering. Performance analysis of GPGPU and CPU On AES Encryption

14. A communication-aware solution framework for mapping AUTOSAR runnables on multi-core systems [Веб-сайт] / Hamid Reza Faragardi [та ін.]. – 2015. – Режим доступу: https://www.researchgate.net/publication/282053815_A_communication-aware_solution_framework_for_mapping_AUTOSAR_runnables_on_multi-core_systems. – .

15. Singer G. History of the Modern Graphics Processor, Part 2 [Веб-сайт] / Graham Singer // TechSpot. – Режим доступу: https://www.techspot.com/article/653-history-of-the-gpu-part-2 – .

16. Samel B. GPU Computing and Its Applications [Веб-сайт] / Bhavana Samel, Shubhrata Mahajan, A. Ingole // International Research Journal of Engineering and Technology (IRJET). – 2016. – Т. 3, № 4. – Режим доступу: https://www.irjet.net/archives/V3/i4/IRJET-V3I4357.pdf. – .

17. Syberfeldt A. A Comparative Evaluation of the GPU vs The CPU for Parallelization of Evolutionary Algorithms Through Multiple Independent Runs [Веб-сайт] / Anna Syberfeldt, Tom Ekblom // International Journal of Information Technology and Computer Scien.– 2017. –Режим доступу: https://www.researchgate.net/publication/318320729_A_Comparative_Evaluation_of_the_GPU_vs_The_CPU_for_Parallelization_of_Evolutionary_Algorithms_Through_Mu ltiple_Independent_Runs. – .

18. Benchmarking Data and Compute Intensive Applications on Modern CPU and GPU Architectures [Веб-сайт] / Miłosz Ciżnicki [та ін.] // Procedia Computer Science. – 2012. – Т. 9. – С. 1900–1909. – Режим доступу: https://doi.org/10.1016/j.procs.2012.04.208 – .

19. Cheng J. Professional CUDA C Programming / John Cheng, Max Grossman, Ty McKercher. – [Б. м.] : Wiley & Sons, Incorporated, John, 2014. – 528 с.

20. File:Barplot language speeds (Benchmarks Game Mandelbrot).svg - Wikimedia Commons [Веб-сайт] // Wikimedia Commons. – Режим доступу: https://commons.wikimedia.org/wiki/File:Barplot_language_speeds_(Benchmarks_Gam e_Mandelbrot).svg – .

21. Fourment M. A comparison of common programming languages used in bioinformatics [Веб-сайт] / Mathieu Fourment, Michael R. Gillings // BMC Bioinformatics. – 2008. – Т. 9, № 1. – Режим доступу: https://doi.org/10.1186/1471-2105-9-82 – .

22. What is a Thread in OS and what are the differences between a Process and a Thread? [Веб-сайт] // AfterAcademy | Platform for learning coding & software development. – Режим доступу: https://afteracademy.com/blog/what-is-a- thread-in-os-and-what-are-the-differences-between-a-process-and-a-thread – .

23. CUDA Refresher: The CUDA Programming Model. [Веб-сайт] // Edge AI and Vision Alliance – Режим доступу: https://www.edge-ai- vision.com/2020/08/cuda-refresher-the-cuda-programming-model/ –

24. Parallel Computing for Data Science (EN 600.420/620) [Веб-сайт] //–Режим доступу: http://parallel.cs.jhu.edu/assets/slides/lec17.1.gpu.pdf –

25. Product Specifications [Веб-сайт] // Intel® Product Specifications. – Режим доступу: https://ark.intel.com/content/www/us/en/ark/products/201897/intel- core-i78700K-processor-12m-cache-up-to-5-10-ghz.html . – .

26. Product Specifications [Веб-сайт] // Intel® Product Specifications. – Режим доступу: https://ark.intel.com/content/www/us/en/ark/products/95443/intel- core-i57300hq-processor-3m-cache-up-to-3-10-ghz.html

– .

27. NVIDIA GeForce 940 (GDDR5) specs - GPUZoo [Веб-сайт] // https://www.gpuzoo.com/. – Режим доступу: https://www.gpuzoo.com/GPU-NVIDIA/GeForce_940MX_GDDR5.html. – .

28. NVIDIA Tesla P100 16 GB specs - GPUZoo [Веб-сайт] // https://www.gpuzoo.com. – Режим доступу: https://www.gpuzoo.com/GPU-NVIDIA/Tesla_P100_16_GB.html. – .

29. ASUS TUF Gaming GeForce RTX 3080 Ti OC Edition 12GB GDDR6X | Graphics Card - Tech Specs [Веб-сайт] // ASUS Deutschland. – Режим доступу: https://www.asus.com/us/Motherboards-Components/Graphics-Cards/TUF-Gaming/TUF-RTX3080TI-O12G-GAMING/techspec/ – .

30. Addition and Subtraction of Matrix using pthreads - GeeksforGeeks. [Веб-сайт] // GeeksforGeeks – Режим доступу: https://www.geeksforgeeks.org/addition-subtraction-matrix-using-pthreads/ – .

31. E. K. Jason Sanders, CUDA by Example: An Introduction to General-Purpose GPU Programming, 2014: Addison-Wesley Professional.

32. E. K. Jason Sanders, CUDA by Example: An Introduction to General-Purpose GPU Programming, 2010: Addison-Wesley Professional.

33 M. R. M. Mark D. Hill, «Amdahl's Law in the Multicore Era,» Computer, № 41, pp. 33-38, 15 July 2018

34. Chapman B., Jost G., Van Der Pas R. Using OpenMP: Portable Shared Memory Parallel Programming. Cambridge, MA: The MIT Press, 2017. 384 p.

35. D. Exterman, «CUDA vs OpenCL: Which to Use for GPU Programming,» Incredibuild blog, 2021. [Веб-сайт], URL: https://www.incredibuild.com/blog/cuda-vs-opencl-which-to-use-for-gpu-programming

36. Г. С. Стребков, «Комплексне дослідження методів програмної спеціалізації для графічних процесорів,» 2020. [Веб-сайт], URL: https://openarchive.nure.ua/server/api/core/bitstreams/012e65e6-12a4-429f-b000-3ac3ab822bb1/content

37. M. R. M. Mark D. Hill, «Amdahl's Law in the Multicore Era,» Computer, № 41, pp. 33-38, 15 July 2018.

38. «Wikipedia: Bus (computing)». [Веб-сайт], URL: https://en.wikipedia.org/wiki/Bus_(computing)

39. D. Exterman, «CUDA vs OpenCL: Which to Use for GPU Programming,» Incredibuild blog, 2021. [Веб-сайт], URL: https://www.incredibuild.com/blog/cuda-vs-opencl-which-to-use-for-gpu-programming

40. Robey R., Zamora Y. Parallel and High Performance Computing. New York, NY: Manning Publications, 2021. 704 с.

41. Foster I. Designing and Building Parallel Programs. London: Pearson Plc, 2019. 408 с.

42. Tuomanen B. Hands-On GPU Programming with Python and CUDA: Explore high-performance parallel computing with CUDA. Birmingham, UK: Packt Publishing Ltd, 2018. 310 с

43. Pacheco P., Malensek M. An Introduction to Parallel Programming. Burlington, MA: Morgan Kaufmann Publishers, 2020. 496 с

44. 6 Van Der Pas R., Stotzer E., Terboven C. Using OpenMP-The Next Step: Affinity, Accelerators, Tasking, and SIMD. Cambridge, MA: The MIT Press, 2017. 392 с

45. Barlas G. Multicore and GPU Programming: An Integrated Approach. Burlington, MA: Morgan Kaufmann Publishers, 2014. 698 с

46. 8 Quinn M. Parallel Programming in C with MPI and OpenMP. New York, NY: McGraw Hill, 2003. 544 с

47. Pacheco P. An Introduction to Parallel Programming. Burlington, MA: Morgan Kaufmann Publishers, 2021. 392 с

48. Sanders J., Kandrot E. CUDA by Example: An Introduction to GeneralPurpose GPU Programming. Boston, MA: Addison-Wesley, 2020. 320 с

49. Cheng J., Grossman M., McKercher T. Professional CUDA C Programming. Birmingham: Wrox Press, 2014. 528 с

50. CUDA Tutorial. [Веб-сайт], URL: https://www.tutorialspoint.com/cuda/index.htm

ДОДАТКИ

1 .Код тестування на CPU

```
#include <iostream>
#include <thread>
#include <chrono>
#include <vector>
#include <atomic>
#include <random>
#include <ctime>
#include <fstream>
#include <functional>

#define MIN_VALUE -1
#define MAX_VALUE 1

using namespace std;
using std::chrono::high_resolution_clock;
using std::chrono::duration;

template <typename T>
class Matrix {
public:
    Matrix(int size, bool init = true) : size_(size) {
        data_ = new T[size_ * size_];
        if (init) {
            generate_random_data();
        } else {
            std::fill(data_, data_ + size_ * size_, 0);
        }
    }

    ~Matrix() {
        delete[] data_;
    }
```

```
    void display() const {
        for (int i = 0; i < size_; ++i) {
            for (int j = 0; j < size_; ++j) {
                std::cout << data_[i * size_ + j] << "\t";
            }
            std::cout << "\n";
        }
    }

    void set_value(int i, int j, T value) {
        data_[i * size_ + j] = value;
    }

    T get_value(int i, int j) const {
        return (i < size_ && j < size_) ? data_[i * size_ + j] : 0;
    }

    void clear() {
        std::fill(data_, data_ + size_ * size_, 0);
    }

private:
    void generate_random_data() {
        std::default_random_engine generator(static_cast<unsigned int>(time(nullptr)));
        std::uniform_int_distribution<int> distribution(MIN_VALUE, MAX_VALUE);

        for (int i = 0; i < size_ * size_; ++i) {
            data_[i] = distribution(generator);
        }
    }

    int size_;
    T* data_;
};
```

```
template <typename T>
void matrix_operation(int start, int end, int size, Matrix<T>& mat1, Matrix<T>& mat2,
function<void(int, int, int, Matrix<T>&, Matrix<T>&)> operation) {
    for (int i = start; i < end; ++i) {
        for (int j = 0; j < size; ++j) {
            operation(i, j, size, mat1, mat2);
        }
    }
}

template <typename T>
void sum_matrices(int i, int j, int size, Matrix<T>& mat1, Matrix<T>& mat2) {
    mat2.set_value(i, j, mat1.get_value(i, j) + mat2.get_value(i, j));
}

template <typename T>
void multiply_matrices(int i, int j, int size, Matrix<T>& mat1, Matrix<T>& mat2) {
    int result = 0;
    for (int k = 0; k < size; ++k) {
        result += mat1.get_value(i, k) * mat2.get_value(k, j);
    }
    mat2.set_value(i, j, result);
}

template <typename T>
void divide_matrix(int i, int j, int size, Matrix<T>& mat1, Matrix<T>& mat2, int divisor) {
    mat2.set_value(i, j, mat1.get_value(i, j) / divisor);
}

template <typename T>
void perform_operations_on_matrix(const std::vector<int>& sizes, const std::vector<int>&
num_threads, std::ofstream& output_file, function<void(int, int, int, Matrix<T>&,
Matrix<T>&)> operation, const std::string& operation_name) {
    std::vector<std::thread> threads;
```

```
    for (int size : sizes) {
        Matrix<T> mat1(size, true);
        Matrix<T> mat2(size, true);

        for (int threads_count : num_threads) {
            auto start_time = high_resolution_clock::now();
            for (int i = 0; i < threads_count; ++i) {
                int start = (size / threads_count) * i;
                int end = (size / threads_count) * (i + 1);
                threads.push_back(std::thread(matrix_operation<T>, start, end, size,
std::ref(mat1), std::ref(mat2), operation));
            }

            for (auto& th : threads) {
                th.join();
            }

            auto end_time = high_resolution_clock::now();
            duration<double, std::milli> elapsed_time = end_time - start_time;
            output_file << size << " " << elapsed_time.count() << " " << threads_count << "\n";

            threads.clear();
        }
    }
}

template <typename T>
void vector_operations(const std::vector<int>& sizes, const std::vector<int>& num_threads)
{
    std::vector<std::thread> threads;
    std::ofstream result_file("output/vector_product_results.txt", std::ios::app);
    std::default_random_engine generator(static_cast<unsigned int>(time(nullptr)));

    for (int size : sizes) {
        std::vector<T> vec1(size), vec2(size);
```

```
    std::uniform_real_distribution<T> distribution(MIN_VALUE, MAX_VALUE);

    for (int i = 0; i < size; ++i) {
        vec1[i] = distribution(generator);
        vec2[i] = distribution(generator);
    }

    for (int threads_count : num_threads) {
        std::atomic<T> result(0);
        auto start_time = high_resolution_clock::now();

        for (int i = 0; i < threads_count; ++i) {
            int start = (size / threads_count) * i;
            int end = (size / threads_count) * (i + 1);
            threads.push_back(std::thread([&vec1, &vec2, &result, start, end]() {
                T temp_result = 0;
                for (int i = start; i < end; ++i) {
                    temp_result += vec1[i] * vec2[i];
                }
                result.fetch_add(temp_result, std::memory_order_relaxed);
            }));
        }

        for (auto& th : threads) {
            th.join();
        }

        auto end_time = high_resolution_clock::now();
        duration<double, std::milli> elapsed_time = end_time - start_time;
        result_file << size << " " << elapsed_time.count() << " " << threads_count << "\n";

        threads.clear();
    }
}
```

```
    result_file.close();
}

int main() {
    std::vector<int> matrix_sizes = {400, 800, 1000, 2000};
    std::vector<int> thread_counts = {1, 2, 4, 8}
    // File for results
    std::ofstream sum_file("output/matrix_sum_results.txt", std::ios::app);
    std::ofstream multiply_file("output/matrix_multiply_results.txt", std::ios::app);
    std::ofstream divide_file("output/matrix_divide_results.txt", std::ios::app);
    // Perform matrix operations
    perform_operations_on_matrix<int>(matrix_sizes, thread_counts, sum_file,
sum_matrices<int>, "Matrix Sum");
    perform_operations_on_matrix<int>(matrix_sizes, thread_counts, multiply_file,
multiply_matrices<int>, "Matrix Multiply");
    perform_operations_on_matrix<int>(matrix_sizes, thread_counts, divide_file,
divide_matrix<int>, "Matrix Divide");
    // Perform vector operations
    vector_operations<float>(matrix_sizes, thread_counts);

    return 0;
}
```

2. Тестування на GPU

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/time.h>
#include <cuda_runtime.h>

#define LOW -1
#define HIGH 1
#define MAX_THREADS 1024
#define THREADXY 20

// CUDA kernel for vector product
```

```
__global__ void vec_product_GPU(float *a, float *b, float *res, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < N) {
        atomicAdd(res, a[index] * b[index]);
    }
}

// CUDA kernel for matrix division
__global__ void matrix_divide_GPU(float *a, int d, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N)
        a[row * N + col] /= d;
}

// CUDA kernel for matrix multiplication
__global__ void matrix_mult_GPU(int *a, int *b, int *c, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        c[row * N + col] = 0;
        for (int k = 0; k < N; k++) {
            c[row * N + col] += a[row * N + k] * b[k * N + col];
        }
    }
}

// CUDA kernel for matrix addition
__global__ void matrix_add_GPU(int *a, int *b, int N)
{
```

```
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N)
        a[row * N + col] += b[row * N + col];
}

// CUDA kernel for matrix addition with a coefficient
__global__ void matrix_add_coef_GPU(int *a, int *b, int coef, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N)
        a[row * N + col] = a[row * N + col] * coef + b[row * N + col];
}

// Function to print a float array
void print_arr(float *a, int N)
{
    for (int i = 0; i < N; ++i)
        printf("%.6f\t", a[i]);
    printf("\n");
}

// Function to print an integer matrix
void print_matrix(int *a, int N)
{
    for (int row = 0; row < N; ++row) {
        for (int col = 0; col < N; ++col) {
            printf("%d\t", a[row * N + col]);
        }
        printf("\n");
    }
}
```

```
// Function to perform scalar product and log performance times
void scalar_product(void)
{
   float *h_a, *h_b, *d_a, *d_b, *h_product, *d_product;
   int list[] = {256, 512, 1024, 2048, 4096, 7168, 10240, 20480, 51200, 102400, 307200, 512000,
1024000, 2048000,
              4096000, 5120000, 7168000, 10240000, 20480000, 40960000, 51200000, 61440000,
81920000};
   struct timeval t1, t2, t1_htod, t2_htod, t1_dtoh, t2_dtoh;

   FILE *vec_ptr = fopen("improved_geforce/gpu_vec.txt", "a");

   for (int j = 0; j < sizeof(list) / sizeof(list[0]); ++j) {
      int N = list[j];
      h_a = (float *)malloc(sizeof(float) * N);
      h_b = (float *)malloc(sizeof(float) * N);
      h_product = (float *)malloc(sizeof(float));

      for (int i = 0; i < N; ++i) {
         h_a[i] = LOW + static_cast<float>(rand()) * static_cast<float>(HIGH - LOW) /
RAND_MAX;
         h_b[i] = LOW + static_cast<float>(rand()) * static_cast<float>(HIGH - LOW) /
RAND_MAX;
      }

      cudaMalloc((void **)&d_a, sizeof(float) * N);
      cudaMalloc((void **)&d_b, sizeof(float) * N);
      cudaMalloc((void **)&d_product, sizeof(float));

      gettimeofday(&t1_htod, 0);
      cudaMemcpy(d_a, h_a, sizeof(float) * N, cudaMemcpyHostToDevice);
      cudaMemcpy(d_b, h_b, sizeof(float) * N, cudaMemcpyHostToDevice);
      cudaMemcpy(d_product, h_product, sizeof(float), cudaMemcpyHostToDevice);
      gettimeofday(&t2_htod, 0);
```

```
        int grids = (N + MAX_THREADS - 1) / MAX_THREADS;
        int num_threads_per_block = (N < MAX_THREADS) ? N : MAX_THREADS;

        gettimeofday(&t1, 0);
        vec_product_GPU<<<grids, num_threads_per_block>>>(d_a, d_b, d_product, N);
        cudaDeviceSynchronize();
        gettimeofday(&t2, 0);

        gettimeofday(&t1_dtoh, 0);
        cudaMemcpy(h_product, d_product, sizeof(float), cudaMemcpyDeviceToHost);
        gettimeofday(&t2_dtoh, 0);

        cudaFree(d_a);
        cudaFree(d_b);
        cudaFree(d_product);

        double htod_time = (1000000.0 * (t2_htod.tv_sec - t1_htod.tv_sec) + t2_htod.tv_usec -
t1_htod.tv_usec) / 1000.0;
        double time = (1000000.0 * (t2.tv_sec - t1.tv_sec) + t2.tv_usec - t1.tv_usec) / 1000.0;
        double dtoh_time = (1000000.0 * (t2_dtoh.tv_sec - t1_dtoh.tv_sec) + t2_dtoh.tv_usec -
t1_dtoh.tv_usec) / 1000.0;

        double total = htod_time + time + dtoh_time;
        printf("Total vector product time is %lf for %d elements\n\n\n", total, N);
        fprintf(vec_ptr, "%d %lf\n", N, total);

        free(h_a);
        free(h_b);
        free(h_product);
    }

    fclose(vec_ptr);
}
```

```
// Function to perform matrix operations
void matrix_operations(void)
{
    int *h_a, *h_b, *d_a, *d_b, *h_product, *d_product;
    int list[] = {40, 100, 140, 240, 480, 720, 1200, 2400, 3600, 3840, 4080, 4560, 5040, 5688, 6240,
6840, 7440, 8040, 8640, 9064};
    struct timeval htod_1, htod_2, htod_3, htod_4, htod_5, dtoh_1, dtoh_2, dtoh_3, dtoh_4, t1, t2, t3,
t4, t5;
    int coef = rand() % 100 + 1;

    FILE *sum_ptr = fopen("improved_geforce/gpu_sum.txt", "a");
    FILE *coef_ptr = fopen("improved_geforce/gpu_sum_coef.txt", "a");
    FILE *div_ptr = fopen("improved_geforce/gpu_div.txt", "a");
    FILE *mult_ptr = fopen("improved_geforce/gpu_mult.txt", "a");

    for (int j = 0; j < sizeof(list) / sizeof(list[0]); ++j) {
        int N = list[j];
        long m_size = N * N;
        h_a = (int *)malloc(sizeof(int) * m_size);
        h_b = (int *)malloc(sizeof(int) * m_size);
        h_product = (int *)malloc(sizeof(int) * m_size);

        for (int i = 0; i < m_size; ++i) {
            h_a[i] = rand() % 10000;
            h_b[i] = rand() % 10000;
            h_product[i] = 0;
        }

        cudaMalloc(&d_a, m_size * sizeof(int));
        cudaMalloc(&d_b, m_size * sizeof(int));
        cudaMalloc(&d_product, m_size * sizeof(int));

        gettimeofday(&htod_1, 0);
        cudaMemcpy(d_a, h_a, sizeof(int) * m_size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, h_b, sizeof(int) * m_size, cudaMemcpyHostToDevice);
```

```
        cudaMemcpy(d_product, h_product, sizeof(int) * m_size, cudaMemcpyHostToDevice);
        gettimeofday(&htod_2, 0);
        gettimeofday(&t1, 0);
        matrix_add_GPU<<<dim3(ceil(N / 16.0), ceil(N / 16.0)), dim3(16, 16)>>>(d_a, d_b, N);
        cudaDeviceSynchronize();
        gettimeofday(&t2, 0);

        gettimeofday(&dtoh_1, 0);
        cudaMemcpy(h_product, d_product, sizeof(int) * m_size, cudaMemcpyDeviceToHost);
        gettimeofday(&dtoh_2, 0);

        double htod_time = (1000000.0 * (htod_2.tv_sec - htod_1.tv_sec) + htod_2.tv_usec -
htod_1.tv_usec) / 1000.0;
        double time = (1000000.0 * (t2.tv_sec - t1.tv_sec) + t2.tv_usec - t1.tv_usec) / 1000.0;
        double dtoh_time = (1000000.0 * (dtoh_2.tv_sec - dtoh_1.tv_sec) + dtoh_2.tv_usec -
dtoh_1.tv_usec) / 1000.0;

        printf("Matrix addition time device to host transfer is %lf for %d elements\n", htod_time,
m_size);
        printf("Matrix addition time is %lf for %d elements\n", time, m_size);
        printf("Matrix addition time host to device transfer is %lf for %d elements\n", dtoh_time,
m_size);

        double total_time = htod_time + time + dtoh_time;
        printf("Total matrix addition time is %lf for %d elements\n", total_time, m_size);
        fprintf(sum_ptr, "%d %lf\n", m_size, total_time);

        // Coefficient multiplication
        matrix_add_coef_GPU<<<dim3(ceil(N / THREADXY), ceil(N / THREADXY)),
dim3(THREADXY, THREADXY)>>>(d_a, d_b, coef, N);
        cudaDeviceSynchronize();

        gettimeofday(&t3, 0);
        gettimeofday(&t4, 0);
        cudaMemcpy(h_product, d_product, sizeof(int) * m_size, cudaMemcpyDeviceToHost);
```

```
        gettimeofday(&t5, 0);

        double coef_time = (1000000.0 * (t4.tv_sec - t3.tv_sec) + t4.tv_usec - t3.tv_usec) / 1000.0;
        double coef_dtoh_time = (1000000.0 * (t5.tv_sec - t4.tv_sec) + t5.tv_usec - t4.tv_usec) /
1000.0;

        free(h_a);
        free(h_b);
        free(h_product);
    }

    fclose(sum_ptr);
    fclose(coef_ptr);
    fclose(div_ptr);
    fclose(mult_ptr);
}

int main()
{
    scalar_product();
    matrix_operations();
    return 0;
}
```

Наукова публікація

УДК 004.72

Наральник Б.Ю., студент 2 курсу ОС Магістр
Спеціальності 122 «Комп'ютерні науки»
Ніколюк П. К., д-р фіз.-мат. наук, професор,
професор кафедри комп'ютерних технологій

ОБРОБКА ДАНИХ У ХМАРНИХ СЕРЕДОВИЩАХ: ВИКЛИКИ ТА ПЕРСПЕКТИВИ

Донецький національний університет імені Василя Стуса, м. Вінниця

Сучасна цифрова епоха супроводжується вибуховим зростанням обсягів даних, які необхідно зберігати, обробляти та аналізувати. Згідно зі звітами провідних аналітичних компаній, до 2025 року обсяг глобального цифрового контенту сягне понад 180 зетабайтів, що створює нові виклики для традиційних систем зберігання та обробки даних[1]. У цьому контексті хмарні технології виступають одним із ключових інструментів, які дозволяють ефективно вирішувати завдання масштабування, продуктивності та економічної ефективності.

Чому це актуально сьогодні?[2,3]

1. ***Зростання ролі хмарних рішень***
   Багато організацій, включаючи міжнародні корпорації, державні установи та стартапи, переходять на хмарні платформи, такі як AWS, Google Cloud чи Azure, щоб скоротити витрати на локальну інфраструктуру та забезпечити гнучкість своїх операцій.

2. **Складність традиційної обробки даних**
   У зв'язку з необхідністю обробляти неструктуровані дані, такі як відео, текст та дані IoT, традиційні підходи більше не відповідають сучасним потребам. Хмарні платформи забезпечують адаптивність та високу обчислювальну потужність для аналізу таких даних у реальному часі.

3. **Інтеграція з іншими інноваціями**
   Хмарні рішення слугують базою для розвитку технологій штучного інтелекту (AI), машинного навчання (ML), великих даних (Big Data) та Інтернету речей (IoT). Вони дозволяють об'єднати та обробляти дані з різних джерел, створюючи нові можливості для бізнесу та науки.

4. **Економічний та соціальний вплив**
   Хмарні технології не лише змінюють підходи до обробки даних, а й сприяють розвитку економіки, відкриваючи нові робочі місця та вдосконалюючи якість послуг. Вони дозволяють компаніям будь-якого

масштабу конкурувати на глобальному ринку, використовуючи передові інструменти аналізу даних.

Основні виклики обробки даних у хмарних середовищах[2,3]

1. **Безпека та конфіденційність даних**
   Хмарні платформи стають мішенню для кіберзлочинців, оскільки в них зберігаються великі обсяги чутливої інформації. Основні загрози включають:

   - **Несанкціонований доступ до даних**: у разі порушення безпеки зловмисники можуть отримати доступ до конфіденційної інформації, включаючи персональні дані користувачів, фінансову інформацію або корпоративні секрети.
   - **Атаки типу DDoS**: такі атаки можуть паралізувати роботу хмарних сервісів, створюючи перебої у доступі до даних.
   - **Ризики внутрішньої загрози**: працівники компаній чи адміністратори хмарної платформи можуть випадково чи навмисно спричинити витік даних.

Захист даних потребує впровадження багаторівневої безпеки, використання шифрування, контроль доступу та регулярний аудит системи.

2. **Затримки та продуктивність мережі**
   Одним із головних викликів є затримки у передачі даних між клієнтом і хмарним центром обробки. Це особливо критично для систем реального часу, таких як відеоконференції, онлайн-ігри або системи моніторингу IoT. Причини цього включають:

   - **Обмеження пропускної здатності мережі**: швидкість передачі даних може бути недостатньою для великих обсягів інформації.
   - **Віддаленість дата-центрів**: чим далі розташований дата-центр, тим більший час потрібен для передачі даних.
   - **Залежність від мережевого обладнання**: нестабільність чи застарілість мережевої інфраструктури можуть суттєво вплинути на продуктивність.

Для вирішення цих проблем впроваджуються технології оптимізації трафіку, розподілу даних між кількома дата-центрами та використання локальних гібридних хмар.

3. **Вартість обслуговування**
   Незважаючи на те, що хмарні платформи дозволяють знизити початкові витрати, у довгостроковій перспективі можуть виникати значні витрати, пов’язані з такими аспектами:

   - **Обробка даних**: складні операції обробки, особливо аналітичні задачі, можуть бути дорогими через високу потужність обчислювальних ресурсів.

   Оптимізація витрат включає правильний вибір тарифного плану, використання автоматизованих інструментів для моніторингу витрат і аналізу ефективності використання ресурсів.

Перспективи розвитку хмарних технологій[2,3]

1. **Підвищення продуктивності**
   - Впровадження квантових обчислень та вдосконалених алгоритмів дозволить значно прискорити обробку великих обсягів даних.
2. **Посилення безпеки**
   - Інтеграція штучного інтелекту для моніторингу загроз і блокчейн-технологій для забезпечення прозорості та безпеки операцій з даними.
3. **Розвиток гібридних хмар**
   - Поєднання публічних та приватних хмар забезпечить гнучкість у керуванні даними, балансуючи між доступністю та конфіденційністю.
4. **Екологічна стійкість**
   - Використання енергоефективних дата-центрів і технологій зменшення енергоспоживання сприятиме зниженню впливу на довкілля.

Список літератури

1. *Volume of data/information created: веб-сайт. URL: https://www.statista.com/statistics/871513/worldwide-data-created/*
2. *Що таке хмарні технології і їх використання: веб-сайт. URL: https://biznestrendy.com.ua/shcho-take-khmarni-tekhnolohii-i-ikh-vykorystannia/*
3. *Хмарні технології 2024 – що це таке та які хмари найкращі? Веб-сайт. URL: https://ucloud.ua/hmarni-tehnologiyi-shho-cze-take/*

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

_____

_____

Повністю вказується ПІБ та статус (освітня (освітньо-наукова) програма – для здобувачів вищої освіти, назва кваліфікаційної роботи)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)