НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проектування

До захисту допущено:

Завідувач кафедри

_____ Вадим МУХІН

«___»_____2022 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою

“Інтелектуальні сервіс-орієнтовані розподілені обчислювання”

зі спеціальності 122 "Комп'ютерні науки"

на тему: «Пошук оптимального розміру вхідного набору даних для ефективного прискорення базових матричних операцій на GPU»

Виконала:
студентка IV курсу, групи ДА-81
Харитонова Дарія Сергіївна _____

Керівник:
ас. Яременко Вадим Сергійович
_____

Консультант з економіки:
доцент, к.е.н. Рощина Надія Василівна _____

Рецензент:
д.т.н., доц. Недашківська Н. І. _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2022

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра системного проектування

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 «Комп'ютерні науки та інформаційні технології»

Освітньо-професійна програма «Системне проектування сервісів»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ В. Є. Мухін

«___»_____ 2022 р.

ЗАВДАННЯ

на дипломну роботу студенту

Харитоновій Дарії Сергієвні

1. Тема роботи «Пошук оптимального розміру вхідного набору даних для ефективного прискорення базових матричних операцій на GPU», керівник роботи ас. Яременко Вадим Сергійович, затверджені наказом по університету від «06» червня 2022 р. №906-с

2. Термін подання студентом роботи: 10.06.2022

3. Вихідні дані до роботи

Операційна система Windows 10/Gnu Linux, GPU NVIDIA з підтримкою CUDA, статті та книги за напрямком роботи.

4. Зміст роботи

- Проаналізувати інформацію про застосування графічних та центральних процесорів
- Дослідити платформу CUDA
- Розробити програмний продукт для порівняння ефективностей CPU і GPU
- Зробити порівняльну характеристику отриманих результатів

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

- Презентація до захисту роботи.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економіка	доцент, к.е.н. Рощина Надія Василівна		

7. Дата видачі завдання 02.12.2021

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	02.12.2021	виконано
2	Аналіз завдання	01.04.2022	виконано
3	Огляд літератури про пристрої	05.04.2022	виконано
4	Дослідження платформи CUDA	08.04.2022	виконано
5	Розробка реалізації на CPU	19.04.2022	виконано
6	Розробка реалізації на GPU	27.04.2022	виконано
7	Оформлення результатів роботи	05.04.2022	виконано
8	Оформлення дипломної роботи	10.04.2022	виконано
9	Отримання допуску до захисту та подача роботи в ДЕК	15.06.2022	виконано

Студент Д.С. Харитонова

Керівник В.С. Яременко

АНОТАЦІЯ

Дипломна робота: 108 с., 6 табл., 2 дод., 32 джерел

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

На сервері кафедри системного проектування наявні графічні процесори для паралельних обчислень. Відповідно, у ході роботи було розроблено програмний інструментарій для дослідження оптимальних значень розмірів вхідних даних, при яких є сенс використовувати GPU для обраних задач з матрицями. Для тестів було використано 2 CPU і 3 GPU. Реалізація вирішення задач на CPU була імплементована на C++, в той час як для GPU використовувалась платформа CUDA. У ході роботи було встановлено, що однозначної відповіді для класу задач з матрицями не існує, та були досліджені аспекти, від яких залежить продуктивність роботи із задачами на тому чи іншому пристрої.

ABSTRACT

Thesis: 108 p., 6 tables, 2 appendces, 32 sources.

FINDING THE OPTIMAL SIZE OF THE INPUT DATA SET FOR EFFICIENT ACCELERATION OF BASIC MATRIX OPERATIONS ON THE GPU

Graphics processors for parallel computing are available on the server of the CAD department. Accordingly, software tools were developed to study the optimal values of the size of the input data with which it makes sense to use the GPU for selected tasks with matrices. 2 CPUs and 3 GPUs were used for the tests. The problem solution with CPU was implemented in C ++, while the GPU used the CUDA platform. In the course of the work it was established that there is no unambiguous answer for the class of problems with matrices. The aspects on which the productivity of working with tasks on a particular device depends are investigated as a result of the aforementioned conclusion.

ЗМІСТ

ВСТУП

Сучасні комп’ютери та цифрові пристрої здатні виконувати потужні обчислення для різноманітних задач. Щороку покращується рівень програмного забезпечення, виконуються все складніші завдання. Усі ці речі вимагають певних ресурсів, через що далеко не останнім питанням стає ефективність їх розподілення та використання. Основними пристроями для безпосередніх обчислень є центральні та графічні процесори. Кожен з цих двох пристроїв орієнтований на вирішування певних типів задач, і грамотне використання CPU та GPU є важливою темою у області паралельних обчислень. Для вирішення питання доцільності використання того чи іншого пристрою до уваги беруться специфіка задачі, тип наявних даних, їх кількість. GPU. У даній роботі особлива увага буде приділятись останньому питанню, а саме дослідженню того, з якою кількістю даних необхідно працювати, аби для певного класу задач (а саме для SIMD-орієнтованих) було ефективніше використовувати GPU замість CPU. Хоч GPU загально визнані як пристрій для роботи саме у випадках із можливим паралелізмом даних, відповідь про використання того чи іншого процесора не завжди є очевидною. Як буде показано в майбутніх розділах, ефективність CPU продовжує зростати, а у GPU є такі недоліки як менша тактова частота ядер та затримки при передачі даних з управляючого пристрою до графічної карти. Саме тому важливо аналізувати поставлені задачі та ретельно підбирати методи та пристрої, за допомогою яких вони будуть вирішуватись.

На кафедрі системного проєктування наявне обладнання для SIMD-обчислень, а саме відеокарти. Необхідно розуміти доцільність їх використання для різних задач. Дана дипломна робота досліджує це питання для задач з матрицями.

РОЗДІЛ 1. ПАРАЛЕЛІЗМ У СУЧАСНИХ ПРИСТРОЯХ. ОСОБЛИВОСТІ АРХІТЕКТУРИ CPU І GPU

Даний розділ розгляне особливості паралелізму в сучасних пристроях, його роль та використання в різноманітних обчисленнях. Буде описана різниця між CPU і GPU на рівні архітектури та які особливості роботи з цього випливають.

1.1. Паралелізм як невід'ємна частина архітектури сучасних машин

Починаючи з 1971 року, – року створення першого мікропроцесору для комерційного ринку [1], – швидкодія центральних процесорів значно покращилась. У 1980-х роках їхня тактова частота складала близько 1 Мгц, а 40 років потому цей показник сягає 4 Ггц для персональних комп’ютерів [2]. Цей процес відбувався стабільно за рахунок збільшення кількості транзисторів на кристалі. Такий розвиток донедавна вдало описувався за допомогою емпірично виведеного закона Мура та законом масштабування Деннарда. Мур передбачив у 1965 році, що кількість транзисторів на чіпах буде подвоюватись кожні 2 роки [3]. Завдяки цьому покращувалась швидкодія, але це пояснювалось не паралелізмом, а тим, що у 1974 році описав Роберт Деннард у спільно написаній роботі [4]. Було встановлено, що напруга та струм пропорційні до лінійних розмірів транзисторів. Це означало, що чим менше транзистор, тим менша потрібна напруга живлення. Крім того, було вивелено, що чим менше транзистор за розміром, тим швидше він може переключатись. Отже, частота роботи процесора збільшується, і, відповідно, швидкість обчислень. Та поки закон Мура ще доволі працює на практиці [5] (рис. 1.1), закон Деннарда втратив актуальність (рис. 1.2).

З цього графіку можна побачити, що частота роботи процесорів ПК перестала різко зростати близько 2006 року. Це пояснюється тим, що у свій час Деннард не врахував витоки струму, які за 30 років зменшення розмірів мікросхем почали складати помітну проблему, що в результаті призводить до перегрівання схеми і виведення процесору з ладу [7]. Окрім того, зменшення розмірів транзисторів

також має свої межі і, в результаті, цьому процесу стане на заваді ефект тунелювання. Ці фактори змусили виробників шукати альтернативні способи підвищення швидкодії.

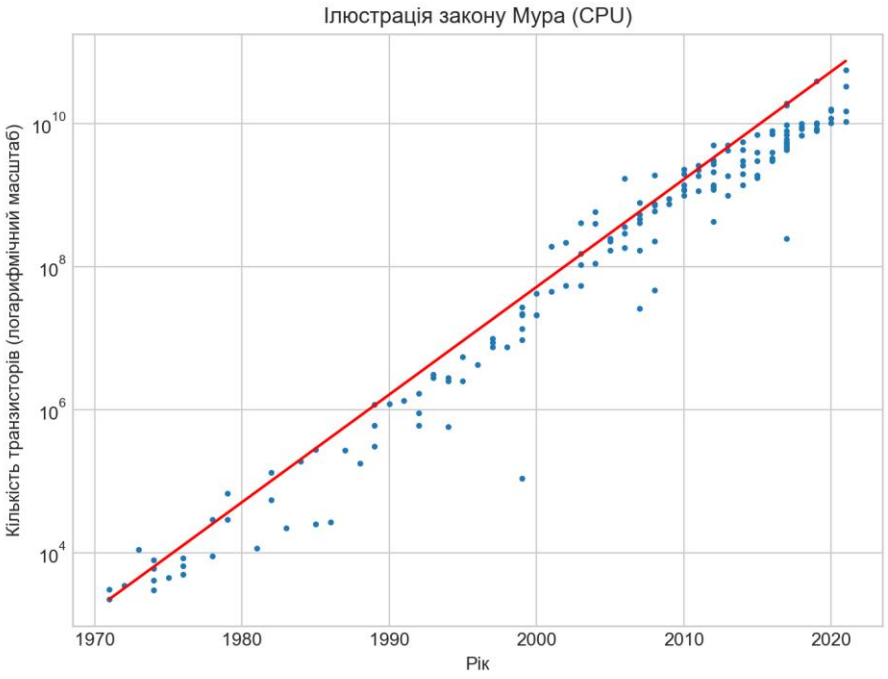

Рис. 1.1 – Ілюстрація закону Мура на основі даних з [5]

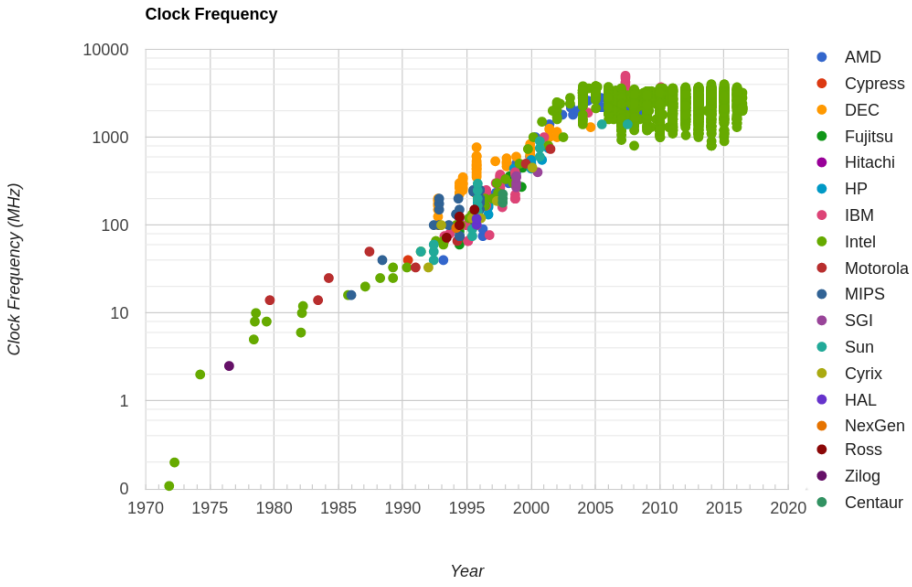

Рис. 1.2 – Розвиток частоти центральних процесорів [6]

У 2001 році на ринку з’явились процесори із двома обчислювальними ядрами [8], а у наступні роки почали з’являтись пристрої, що мають 3, 4, 6 та 8 ядер. На сьогоднішній день на ринку практично відсутні комп’ютери з процесорами, що працюють на одному ядрі, а споживачам пропонуються чіпи з 16-ма ядрами.

Це є ілюстрацією того, що паралелізм надійно зайняв своє місце у обчисювальних пристроях, як в комерційних продуктах, так і в машинах для вузьконапрямлених обрахунків. Зараз майже будь-який сучасний комп’ютер, планшет чи смартфон містить у собі процесор з кількома ядрами, кожне з яких здатне виконувати власний потік інструкцій. Якщо ці потоки розроблені таким чином, що ядра можуть співпрацювати під час виконання програми, її виконання може значно пришвидшитись. Окрім таких домашніх чи кишенькових пристроїв, багатоядерні процесори зустрічаються на серверах, і такі сервери здатні запускати більше однієї служби паралельно. І останнім, але не менш важливим прикладом є той факт, що навіть комп’ютери для широкого кола користувачів можуть виконувати сотні або навіть тисячі потоків інструкцій паралельно завдяки графічним процесорам. Такий паралелізм необхідний для підтримки графічної анімації. Це також вказує на те, що центральні процесори не є єдиним пристроєм, який на сьогоднішній день адаптований для вирішення паралельних задач.

Крім того, було би хибно стверджувати, що паралельні обчислення розвиваються виключно завдяки збільшенню обчислювальних ядер. Паралелізм присутній на всіх рівнях сучасної архітектури комп’ютерів, і закладений навіть в мікроархітектурі процесора [9]. В минулому, для виконання програм процесори виконували послідовність із чотирьох кроків:

1) читання та декодування інструкції;

2) пошук даних, необхідних для виконання інструкції;

3) виконання інструкції;

4) запис (виведення) результату.

Дані, які надходили на другому кроці, спричиняли тривалі затримки. Це стало причиною появ досліджень з теми зменшення цих затримок і, як наслідок, підвищення ефективності програм. Тим не менш, з роками, головною метою стало

створення процесора, здатного виконувати кілька інструкцій одночасно. Такий процесор міг би виявляти та використовувати паралелізм, властивий виконанню інструкцій. Незалежно від частоти процесора та пам’яті, ці процесори забезпечували би ще більшу швидкість виконання програми.

Таким чином, з роками, різноманітні паралельні рішення поступово розвинулись у сучасні системи. Їх можна класифікувати за таксономією Флінна. Цей розподіл пристроїв на категорії з’явився ще у 1966 році, проте досі є актуальним [10]. Схема розподілу наведена на рис. 1.3. Належність машини до тої чи іншої категорії визначається тим, скільки елементів даних вона може обробляти одночасно і скільки різних інструкцій вона може виконувати одночасно. Можливі варіанти обидвох цих критеріїв – один або декілька, а це означає, що їхня комбінація може призвести до чотирьох можливих результатів:

1) Єдина інструкція, єдиний елемент даних (Single Instruction, Single Data – SISD): машина виконує одну команду за раз, оперуючи одним елементом даних. Більшість сучасних процесорів не підпадають під цю категорію. У наш час навіть мікроконтролери випускаються в багатоядерних конфігураціях, а кожне їхнє ядро можна розглядати як SISD-машину.

2) Єдина інструкція, кілька елементів даних (Single Instruction, Multiple Data – SIMD): усі ядра в будь-який час виконують один і той же потік інструкцій, кожний з яких працює з різними потоками даних. SIMD імплементована у більшості сучасних комп’ютерів.

3) Кілька інструкцій, єдиний елемент даних (Multiple Instructions, Single Data – MISD): у цій архітектурі кожне ядро працює з одним і тим же потоком даних, використовуючи різні потоки інструкцій. За звичайних обставин це не має сенсу. Однак, коли в системі потрібна відмовостійкість (наприклад, військові чи аерокосмічні пристрої), дані можуть оброблятися кількома машинами, і рішення примаються за принципом більшості.

4) Кілька інструкцій, кілька елементів даних (Multiple Instructions, Multiple Data – MIMD): кілька ядер працюють з кількома потоками даних, кожне з яких виконує різні інструкції. Архітектура MIMD також може включати в собі підкомпоненти з архітектурою SIMD.

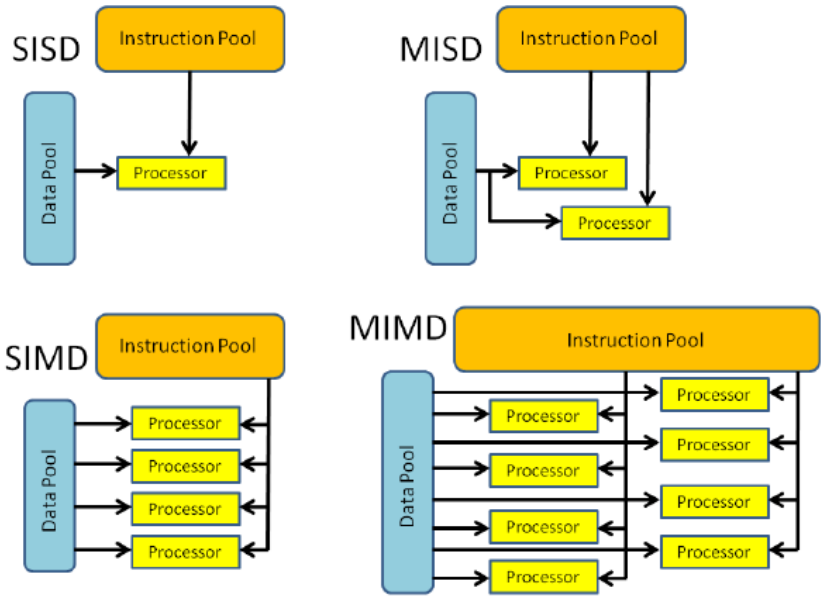

Рис. 1.3 – Таксономія Флінна [11]

З роками ця класифікація була покращена, і були додані дві підкатегорії, які особливо стосуються MIMD-машин (рис. 1.4).

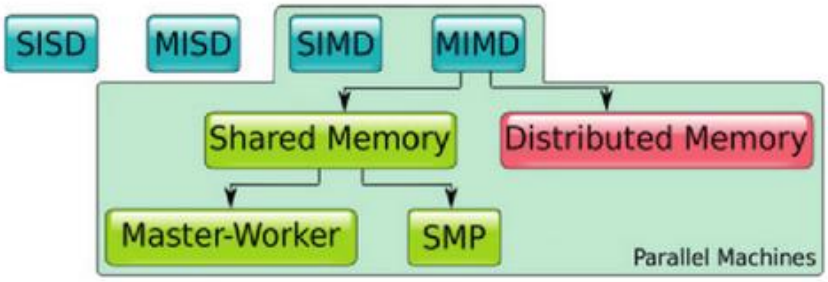

Рис. 1.4 – Розширена таксономія Флінна з [10]

а) MIMD зі спільною пам’яттю (shared-memory MIMD): архітектура машин, що має загальнодоступний простір спільної пам'яті. Спільна пам’ять спрощує всі транзакції, які відбуваються між центральним процесором з найменшими накладними витратами, але це також «вузьке місце» (ботлнек), яке обмежує масштабованість системи. Цю проблему можна вирішити, розділивши пам'ять між процесорами так, щоб кожен з них «володів» її частиною. Таким чином, центральний процесор має швидший доступ до своєї локальної пам’яті і все ще має доступ до нелокальної пам’яті інших процесорів, хоча й повільніше. Цей поділ не впливає на адресацію, адже адресний простір є спільним для всіх. Такий підхід відомий як NUMA (Non-Uniform Memory Access) [12], та дозволяє масштабувати машини до кількох десятків центральних процесорів.

б) MIMD з розподіленою пам’яттю (або shared-nothing MIMD): процесори, які складають дану машину, комунікують між собою за допомогою повідомлень. Таке спілкування є затратним, проте такі машини можуть масштабуватися практично без верхнього ліміту, окрім обмежень по розміру та споживанню енергії.

Машини зі спільною пам’яттю можна також поділити на підкатегорії master-worker та симетричні багатопроцесорні платформи. В останньому, всі процесори, що приймають участь, є «рівними» та здатні виконувати будь-яку програму у системі, включно із системним та користувацьким програмним забезпеченням. В машині типу master-worker деякі процесори відведені під виконання спеціальних програм, тобто їх можна розглядати як співпроцесори. Системи з графічними процесорами можуть бути віднесені до цієї категорії, хоч і більшість високопродуктивних GPU-платформ мають окремі простори пам’яті для CPU та GPU. В такому випадку вони не є платформами зі спільною пам’яттю, попри усі сучасні дослідження, спрямовані на покращення процесу переносу даних між двома областями пам’яті.

1.2. Порівняння архітектур центральних та графічних процесорів

У даній роботі можливості паралелізму будуть досліджені для двох типів пристроїв — центральних та графічних процесорів. Для цього необхідно розглянути принципи роботи кожного з них. Спрощена схема ядра центрального процесора наведена на рис. 1.5.

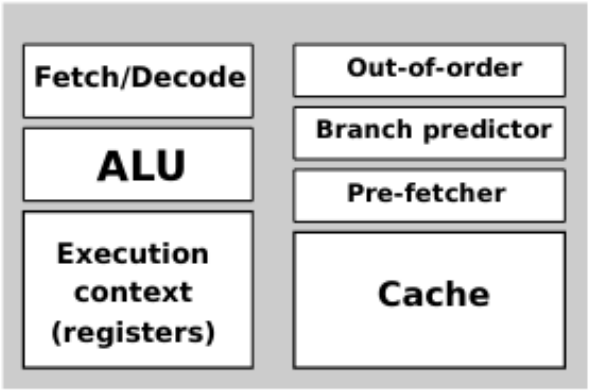

Рис. 1.5 – Схема ядра центрального процесора з [9]

Типове ядро CPU складається з логіки вибірки та декодування інструкції, арифметико-логічного пристрою (ALU) та контексту виконання. Логіка вибірки-декодування відповідає за отримання інструкцій з пам'яті, їх декодування, щоб підготувати операнди та вибрати необхідну операцію в АЛП. Контекст виконання включає в себе лічильник команд (аби визначати їх послідовність), вказівник на стек, регістр стану (для зберігання флагів виконання) та регістри загального призначення. Такий процесор із єдиним ядром, у якого один АЛП та контекст виконання можуть виконувати одну інструкцію з потоку інструкцій за раз, тобто, фактично, є SISD-пристроєм. Задля підвищення продуктивності виконання в один потік такі центральні процесори з один ядром вдаються до позачергового виконання (тобто в порядку готовності, а не слідування) інструкцій та передбачення переходів (визначається виконання чи невиконання умовного переходу) для уникнення затримок. Тим не менш, усі виконуючі одиниці не мають сенсу без інструкцій та операндів, які зберігаються в основній пам’яті (рис.1.6). Передача інструкцій та операндів з та до основної пам’яті вимагає значної

кількості ресурсів та часу, саме тому існує таке поняття як кеш. Кеші працюють за принципом або просторової локальності, або локальності у часі, та є ефективними у випадку, коли потік інструкцій повторюється багато разів (наприклад, програмні цикли) або коли доступ до даних відбувається за близькими адресами. АЛП та логіка вибірки-декодування працюють швидко, не є затратними в плані потужності та не потребують багатьох hardware-ресурсів. На противагу цьому, необхідно багато транзисторів для того, аби сконструювати кеш, що робить їх дуже дорогими. Це також одна з найбільш енергозатратних частин у CPU загального призначення.

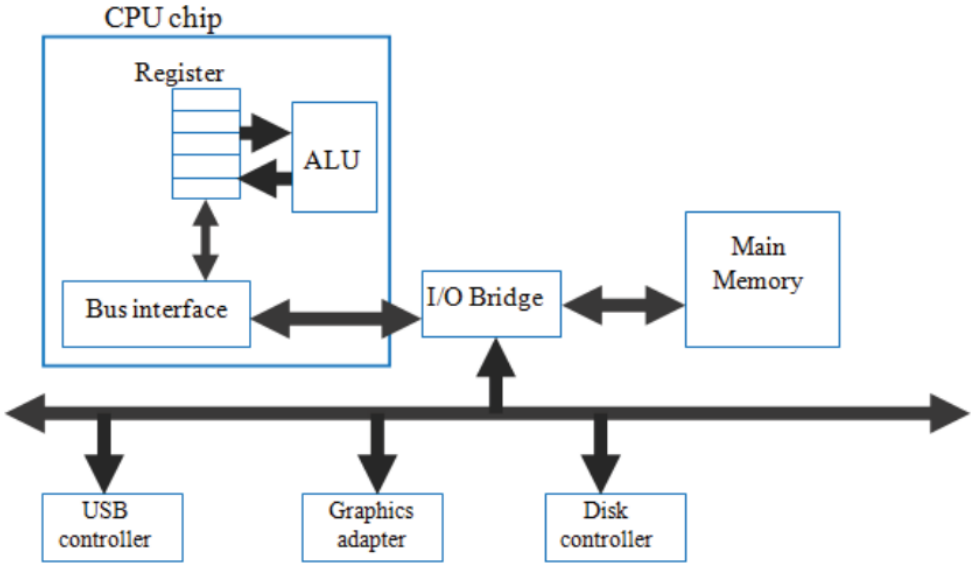

Рис. 1.6 – Центральний процесор та основна пам’ять [13]

В умовах збільшення продуктивності центральних процесорів за рахунок багатоядерної архітектури, більш типовою на сьогоднішній день буде схема з рис.1.7.

У той час, поки центральні процесори проходили свій шлях розвитку, область графічних обчислень також мала в собі значні зміни. Наприкінці 80-х та на початку 90-х років популярність операційних систем із графічною складовою дала поштовх, аби на ринку з'явився новий тип процесора. Крім того, в той же час компанія Silicon Graphics популяризовувала використання тривимірної графіки у

різноманітних сферах, як-от наукова, військова, а також візуальні ефекти для кіно, а у 1992 випустила бібліотеку на широкий загал під назвою OpenGL [2].

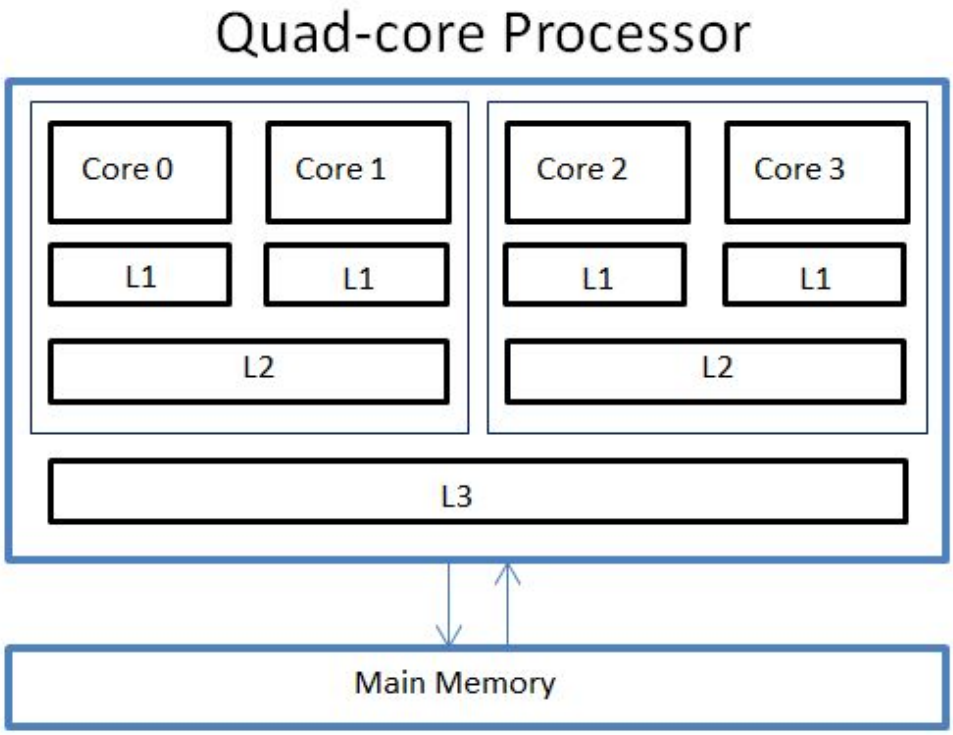

Рис. 1.7 – Центральний процесор та основна пам’ять [14]

Вона мала слугувати як платформонезалежний інструмент для створення програм із 3D-графікою. Із часом ця технологія стала використовуватись і в розробці програм для широкого кола користувачів, а не лише у кількох обраних сферах. У середині 90-х років почали з'являтись та отримували велику популярність комп'ютерні ігри із 3D-графікою, а такі компанії, як NVIDIA, ATI technologies та 3dfx interactive випустили доступні графічні прискорювачі. У 1999 році був випущений графічний пристрій NVIDIA GeForce 256, який можна назвати першим графічним процесором завдяки наявності чіпу обрахунків трансформацій (створення двовимірного вигляду тривимірної сцени) та освітлення (зміни кольору поверхонь в залежності від інформації про освітлення сцени) [15].

Принципова різниця між GPU і CPU полягає в тому, що GPU має в собі сотні або навіть тисячі обчислювальних ядер. Ці ядра не є ідентичними тим, які

використовуються в центральних процесорах загального призначення. Якщо забрати зі схеми ядра на рис. 5 кеші, механізм передвибірки (cache pre-fetcher), логіку позачергового виконання та передбачення переходів, отримаємо дизайн на рис. 1.8.

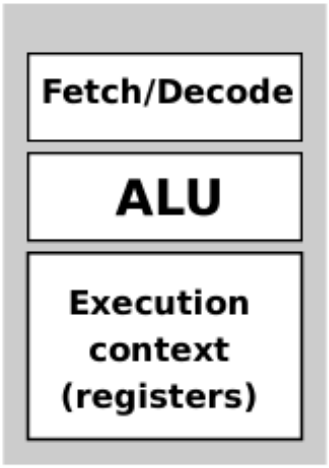

Рис. 1.8 – Схема спрощеного ядра центрального процесора [9]

Тим не менш, на виконання певних задач це не впливає. Згідно із [9], код для додавання двох векторів на мові C, що виглядає як на лістингу 1.1, орієнтовно транслюється в асемблерний код як на лістингу 1.2.

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

Лістинг 1.1 – Код для додавання векторів на мові C

```
    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 ;
```

Лістинг 1.2 – Код для додавання векторів на асемблері

Припускаючи, що r0 це нульовий регістр, дані інструкції спершу очищують регістри r2 та r3, у яких зберігається лічильник tid та зміщення у векторах vecA та vecB відповідно. Далі у циклі завантажуються у регістри числа з векторів, додаються, зберігається результат у vecC. Збільшуємо лічильник tid та, за умови що ця послідовність виконалась не більше 128 разів, повторюємо цикл. Отже, спрощене ядро здатне виконати даний код, попри те, що деякі елементи архітектури не є присутніми. Крім того, використовуючи паралелізм даних, можливо виконати цю функцію за допомогою, наприклад, двох таких ідентичних спрощених ядер із ідентичним набором інструкцій з поправкою на коректне розбиття масиву даних для обробки, розподіляючи на кожне ядро вже по 64 елементи.

Наступним кроком із підвищення ефективності є збільшення кількості АЛП та контекстів виконання, як на рис. 1.9. Як можна побачити з рисунку, інструкції лишаються незмінними, зміна відбувається щодо лічильника та ділянок масиву даних.

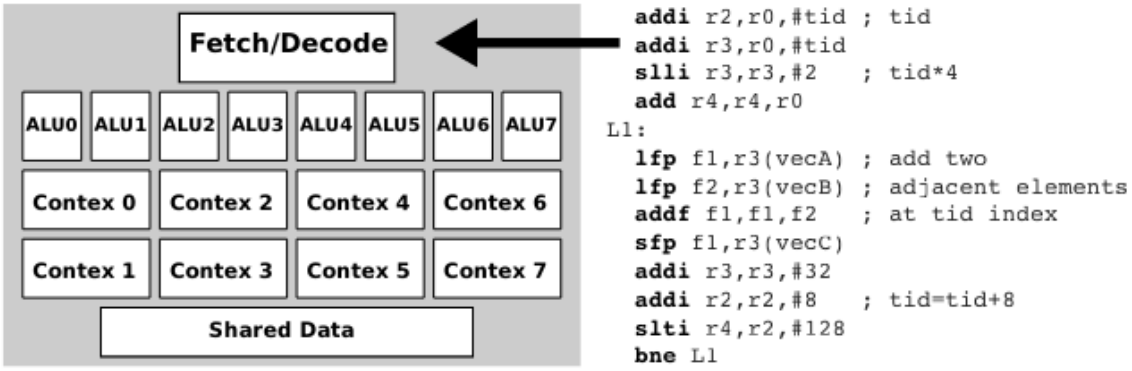

Рис. 1.9 – Збільшення кількості АЛП та контекстів виконання

Із такою архітектурою можливо додавати по 8 елементів з кожного вектора, користуючись тим самим потоком інструкцій – та сама інструкція виконується в кожному потоці, але на різних даних. Кожен потік тепер повинен мати власний номер (ID), аби позначити, над якою ділянкою даних працювати. Таким чином, компілятор для коду на мові С повинен мати змогу транслювати код з лістингу 1 у інструкції, наведені на рис.1.9, а також імплементувати щойно описану логіку із номерами потоків. Оскільки кожен АЛП має свій набір регістрів (контекст виконання), кожен з цих пристроїв працює з власним tid у r2. Аналогічно виконується робота і з рештою інструкцій.

Назвемо ядро з рис. 1.8 обчислювальнлю одиницею, а АЛП — елементом обробки. Таким чином, підсумовуючи, такі обчислювальна одиниці є процесорами загального призначення із модифікованою архітектурою, якп підтримує SIMD-паралелізм завдяки чисельним АЛП та контекстам виконання. Ця модифікована архітектура не підтримує branch prediction та має менше пам'яті кешу, аніж в CPU. Ще більших покращень у ефективності обчислень можна досягнути за допомогою збільшення кількості вже самих обчислювальних одиниць. Рис. 1.10 зображає GPU із 16 такими обчислювальними одиницями. Такий пристрій виконає додавання вектора на 128 елементів за один цикл інструкцій. Кожна обчислювальна одиниця виконує код, вказаний на цьому ж рисунку. Кожен потік

має власний ID від 0 до 127. Перші дві інструкції завантажують в регістр r3 ID потоку tid та розраховують зсув по ділянці пам'яті, помножуючи на 4, аби врахувати те, що це тип даних із плаваючою точкою. Тепер кожен потік має для себе розрахований індекс потрібного елемента. Далі відбувається аналогічне додавання елементів векторів vecA та vecB і зберігання результату у vecC. Оскільки виконання відбувається у 128 потоків, другий цикл виконання інструкцій не потрібен.

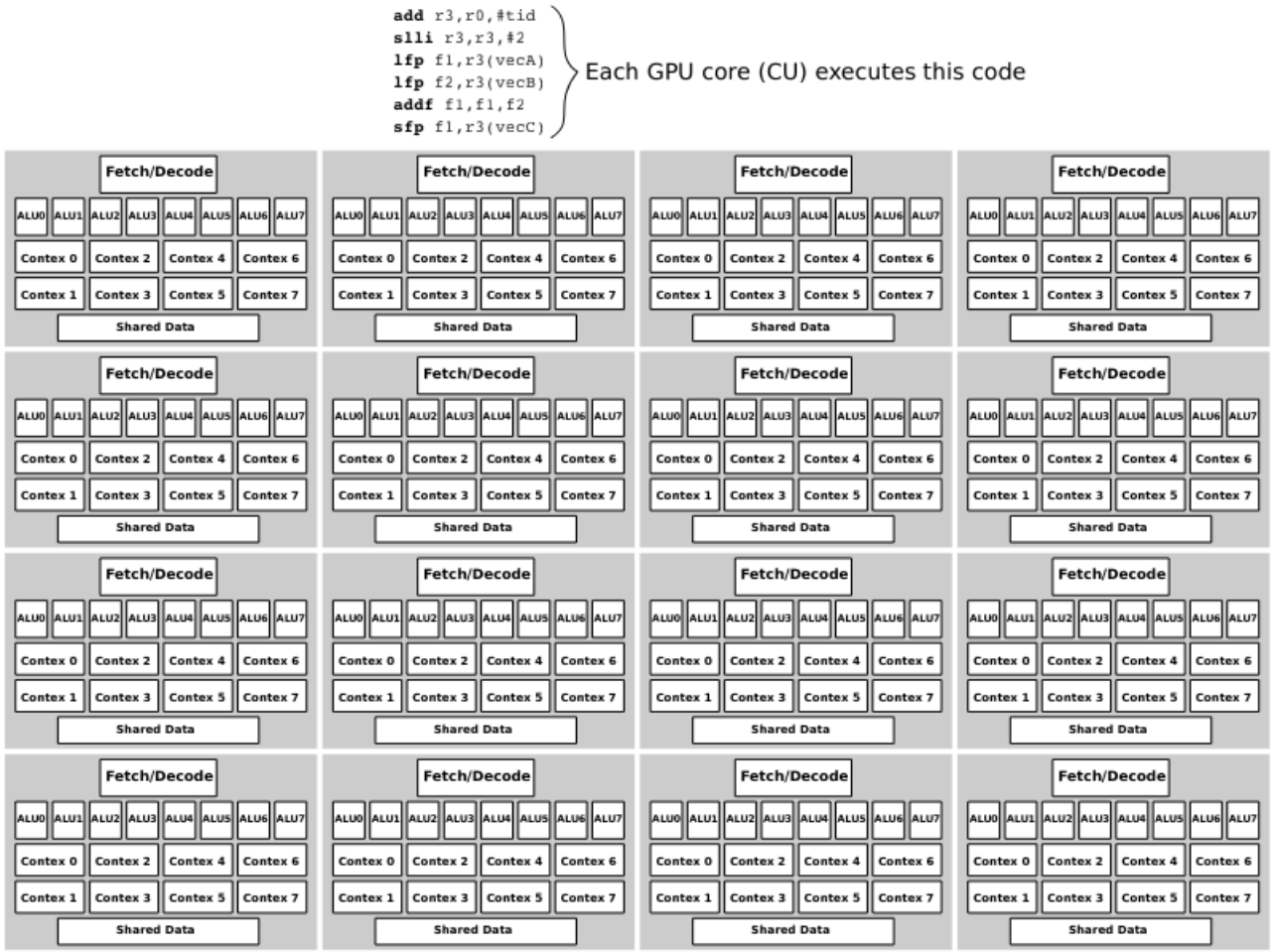

Рис. 1.10 – 16 обчислювальних одиниць

Тим не менш, подібні напрацювання, як походить з назви, використовувалась для графічних розрахунків. Проте у 2001, із випуском серії GeForce 3, розробники почали розглядати графічні карти як інструменти для розрахунків не лише із цією

метою. Такий напрям роботи із GPU почав з'являтись завдяки імплементації стандарту DirectX 8.0 у вищезгаданій серії пристроїв, який вимагав, аби графічні чіпи підтримували обробку програмованих вершинних та піксельних шейдерів. Таким чином, розробники вперше змогли контролювати, які саме розрахунки вони проводили на GPU. Дослідники зацікавились можливістю використовувати графічні чіпи для більш широкого кола задач, хоч на той час були серйозні обмеження. Єдиним способом провести обрахунки було використанням графічних API, таких як OpenGL або DirectX, тобто пристрій мав розглядати вхідні дані як дані для рендерингу. Довільні дані для обрахунків подавались як кольори, координати і т.д. Але структури, що відповідали за кольори та текстури, мали обмеження щодо своїх чисельних значень. Крім того, інструментів для дебагінгу на цій платформі не було. Ці та інші обмеження не сприяли широкому використанню GPU для неграфічних обрахунків. Проте ця ситуація змінилась із представленням на загал платформи CUDA від NVIDIA в 2006 році [15]. CUDA довзолила проводити різноманітні неграфічні розрахунки на відеокартах NVIDIA без обмежень графічних API, і є значним досягненням у області паралельних обчислень. Саме на цій платформі будуть виконуватись завдання обчислень на GPU із даної роботи.

1.3. Висновки до розділу 1

Зі змісту даного розділу було встановлено, що паралелізм є невід'ємним компонентом архітектури сучасних обчилювальних пристроїв. Ефективність обчислень на центральних процесорах спершу збільшувалась завдяки підвищенню тактової частоти роботи через зменшення розмірів транзисторів, та з часом цей підхід проявив власні недоліки. Через це розробники вдались до апаратного паралелізму через підвищення кількості обчилювальних ядер в пристроях. Паралельно до цього виник окремий вид процесорів, в першу чергу орієнтований на графічні обчислення. Завдяки SIMD-природі таких обчислень архітектура графічних процесорів включає в себе велику кількість ядер нижчої потужності,

ніж ядра CPU. Із розвитком стандартів та програмних інструментів, а особливо із випуском платформи CUDA для GPU від NVIDIA, відеокарти стало можливо використовувати для обчислень загального призначення, а не тільки для рендерингу графіки. Завдяки цьому графічні процесори отримали велику популярність у широкого кола користувачів та використовуються там, де центральні процесори показують низку ефективність роботи на SIMD-задачах із великою кількістю потоків. З цієї ж причини на кафедрі системного проектування для вирішення різноманітних задач, наприклад, розв’язку великих систем рівнянь, використовують графічні карти NVIDIA. Саме для визначення найоптимальніших сценаріїв користування цими відеокартами для певних задач і проводиться дослідження у даній роботі.

РОЗДІЛ 2 ПОРІВНЯННЯ ЕФЕКТИВНОСТІ CPU ТА GPU: АНАЛІЗ ІСНУЮЧИХ РОБІТ. ПОСТАНОВКА ЗАДАЧІ

Як було зазначено у попередньому розділі, графічні та центральні процесори орієнтовані під певні типи задач. Сьогодні графічні процесори широко використовуються для обчислень в таких сферах як машинне навчання, аналітика даних, фінтех, комп'ютерний зір, біоінформатика та інші [16], а порівняльні дослідження швидкодії обох пристроїв стали важливою темою у області паралельних обчислень. Розглянемо кілька робіт цього напрямку та проаналізуємо їх висновки. З цих висновків буде складене враження про те, як схильні поводити себе відеокарти при використанні для тих чи інших задач та при відповідних розмірах даних, та що очікувати у майбутніх тестах.

2.1 Огляд роботи з області машинного навчання

Як вже було відмічено, використання GPU для машинного навчання отримало велику популярність. Існує багато напрацювань, що досліджують це питання. Наведемо як приклад одну таку працю. У [17] розглядається питання використання CPU і GPU для обчислень глибокого навчання. Ця робота досліджує задачу класифікації веб-сторінок у 23 різні категорії за допомогою RNN – рекурентної нейронної мережі (з'єднання між вузлами утворюють граф, орієнтований у часі). Пристрої, за допомогою яких проводились дослідження, були розташовані у хмарі. Таким чином, дослідники використовували CPU Intel Xeon Gold 6126 із дванадцятьма ядрами, проте можливості хмари дозволили не обмежуватись одним процесором, а задіювати потрібну кількість ядер з кількох пристроїв Intel Xeon, а можлива частота роботи складала 1300 та 2600 МГц. Також хмарний сервіс надав доступ до GPU NVIDIA Tesla K80, тому дослідники мали у змогу виконати тести на графічному процесорі із 2496 ядрами, що працюють на частоті 562 МГц. Порівняльну характеристику можливостей пристроїв у роботі було наведено в таблиці, що на рис. 2.1.

В першу чергу було відмічено очікуваний приріст у швидкодії під час класифікації веб-сторінок за допомогою як CPU-ядер із вищою частотою, так і просто завдяки збільшенню кількості цих самих ядер. Результати, наведені на рис.2.2 були отримані в ході тренування цієї нейронної мережі у 3 епохи (повних обробок датасету) за різних розмірів батчів, тобто кількості екземплярів для тренування, що обробляються за одну ітерацію алгоритму. Загальна кількість екземплярів налічує 1 210 954 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. 2.1 – Таблиця характеристик пристроїв із [17]

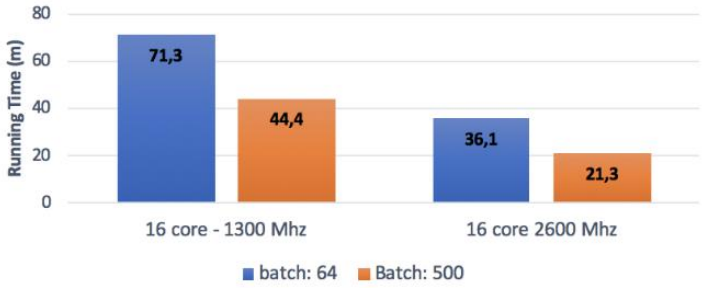

Fig. 3. CPU Core Frequency Comparison

Fig. 4. Core Count Comparison

Рис. 2.2 – Результати роботи із різною частотою та кількістю ядер CPU з [17]

Виконавши порівняння CPU і GPU у даній статті, автори навели результати, що представлені на рис. 2.3. Дослідження були проведені із використанням вищезгаданої графічної карти Tesla K80 та 16 ядер CPU із частотою у 2600 MHz. Так само було задіяне тренування у 3 епохи та розміри порцій даних на одну ітерацію розміром 64, 500, 1000 та 5000 одиниць.

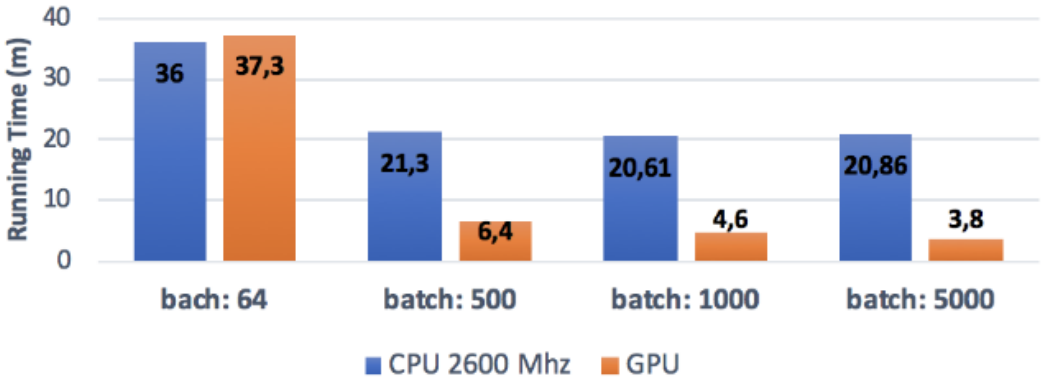

Fig. 5. The Effect of Batch Size

Рис. 2.3 – Порівняння ефективності CPU і GPU з [17]

Важливим спостереженням є те, що при малому розмірі батчу даних ефективність GPU є нижчою, аніж ефективність CPU, а зі збільшенням кількості одиниць даних для обробки за одну ітерацію ефективність роботи із графічною картою помітно зростає.

2.2 Огляд роботи з області еволюційних алгоритмів

Ще один розділ штучного інтелекту, що задіює графічні процесори для обрахунків це еволюційні алгоритми. Це питання також не раз досліджувалось у різноманітних роботах, то ж як приклад розглянемо працю [18], присвячену бенчмарку CPU та GPU на еволюційному алгоритмі. Еволюційні алгоритми імітують процеси природнього відбору, аби отримати найпридатніший з вибірки варіантів результат згідно із заданими критеріями. Сама ж множина вибірки із

кожною новою ітерацією формується за допомогою комбінації найоптимальніших екземплярів (особин) попередньої ітерації з існуючого набору (популяції) за заданими правилами і умовами. Для виконання бенчмарку були задіяні центральний процесор Intel i7-4790K із частотою 4 Гц та 4 ядрами та графічну карту NVIDIA GeForce 980 із 2048 ядрами, що працюють із частотою 1126 МГц.

Вибраний еволюційний алгоритм під назвою NSGA-II дослідники імплементували двома способами – на мові C# та на CUDA C. Реалізація псевдокоду алгоритму із лістингу 2.1 в обох варіантах наведена на рис. 2.4 та рис.2.5 [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
```

Лістинг 2.1 – Псевдокод 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. 2.4 – Імплементація 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. 2.5 – Імплементація NSGA-II двома способами [18]

На рис. 2.4 показано імплементацію функції OnePointSplitCrossover з рис. 2.5. Ця функція реалізовує схрещування двох особин з популяції шляхом розділення у випадковому місці векторів із набором характеристик цих особин (геномів) на 2 частини та обміну цими частинами між собою. Загалом, генетичний алгоритм працює наступним чином: на початку роботи задається початкова популяція з довільним набором значеннями характеристик (геномом), що входить до області допустимих розв'язків задачі, а потім на кожній ітерації відбуваються такі кроки:

1) випадковим чином обирається дві найкращі (за значенням цільової функції) особини з популяції, які є "батьками";

2) згідно функції схрещування (вищезгадана OnePointSplitCrossover), з них отримуються дві нові особини, "нащадки";

3) із деякою ймовірністю у геномі нових особин відбувається мутації – випадкові зміни значень деяких характеристик.

Отримані на кожному кроці особини додаються до нової популяції, доки її об'єм не буде таким же, як у початкової.

У даній роботі алгоритм відпрацьовував 100 поколінь із розміром геному 20 та розміром популяції 50.

Після запуску еволюційного алгоритму з вищезгаданими параметрами із кількістю потоків до 50 000 на кожному з пристроїв, був отриманий результат з рис. 2.6. Можна побачити, що зі збільшенням кількості потоків ефективність CPU падає, а GPU – зростає. На збільшеному графіку з рис. 2.7 видно, що зсув на користь GPU відбувається вже в районі 3000 потоків. Після проведення аналогічного досліду із розміром геному 2, а не 20 (таким чином, обробляється не 1000 характеристик у популяції, а лише 100), був отриманий результат з рис. 2.8.

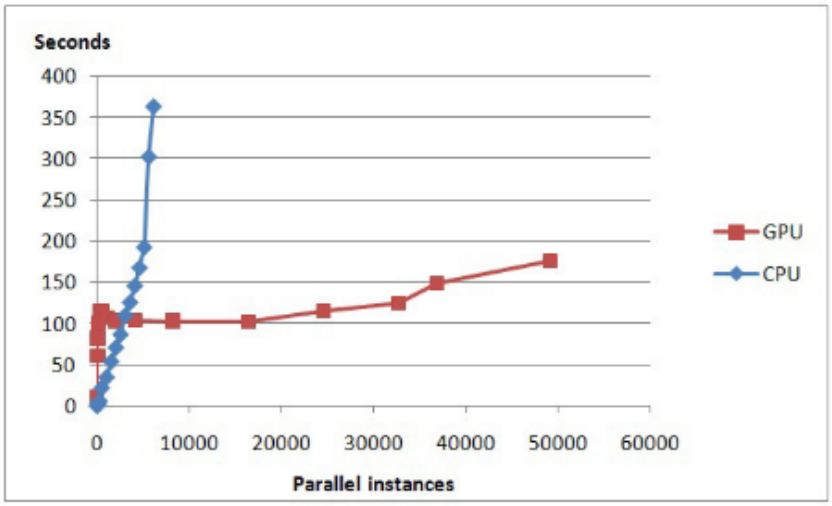

Рис. 2.6 – Результати роботи еволюційного алгоритму з використанням багатьох потоків [18]

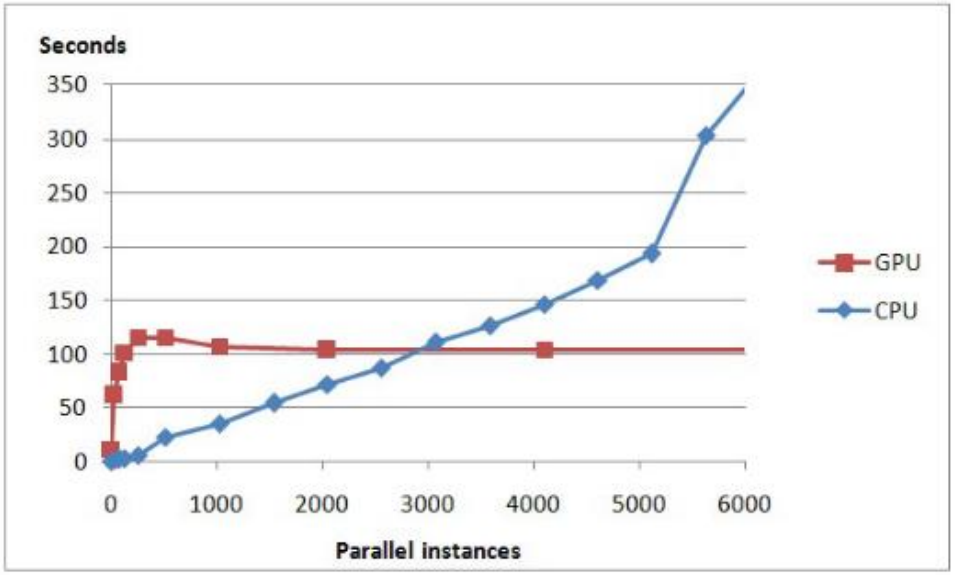

Рис. 2.7 – Збільшений вигляд [18]

На рис. 2.8 можна спостерігати, що переважання у бік GPU почалось раніше, в районі 1700 екземплярів відтворення генетичного алгоритму. Таким чином, автори публікації дійшли висновку, що ефективність GPU значно падає при простоюванні, а управління пам’яттю та копіювання даних з та до графічної карти вимагає ресурсів. Спостерігалось, що з невеликою кількістю даних та обмеженою

кількістю потоків центральні процесори показують більшу ефективність, проте ця ситуація змінюється зі збільшенням кількості даних та потоків, адже створення нових потоків з часом займає більше ресурсів процесора, аніж сама обробка даних.

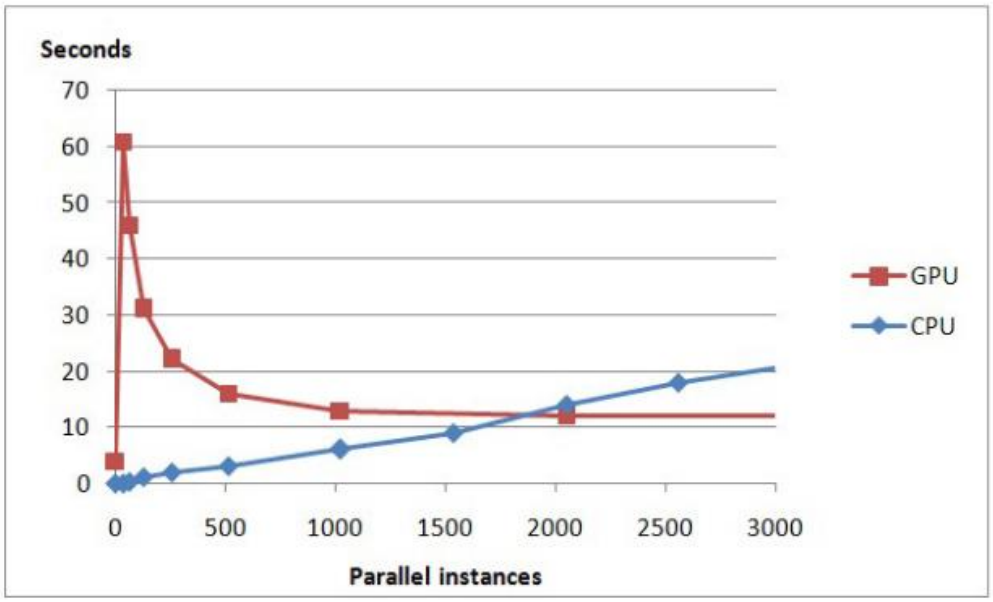

Рис. 2.8 – Експеримент із меншим розміром геному [18]

2.3 Огляд роботи з області медіаданих

Останньою роботою з теми порівняння ефективності центральних та графічних процесорів буде [19], що досліджує дане питання через тести із графічним форматом JPEG 2000. Використання відеокарт для обробки кодеків, особливо для відео, є цілою окремою галуззю у сфері роботи з GPU, а карти NVIDIA навіть містять окремий апаратний прискорювач для таких задач. Через це як приклад досліджень ми і розглядаємо роботу [19]. JPEG 2000 – спосіб стиснення даних, як зображено на рис. 2.9, вимагає значних розрахунків,

різноманітних трансформацій (CT, DWT), квантування, кодування (EBCOT), оптимізацій (PCRD) та багатьох інших операцій.

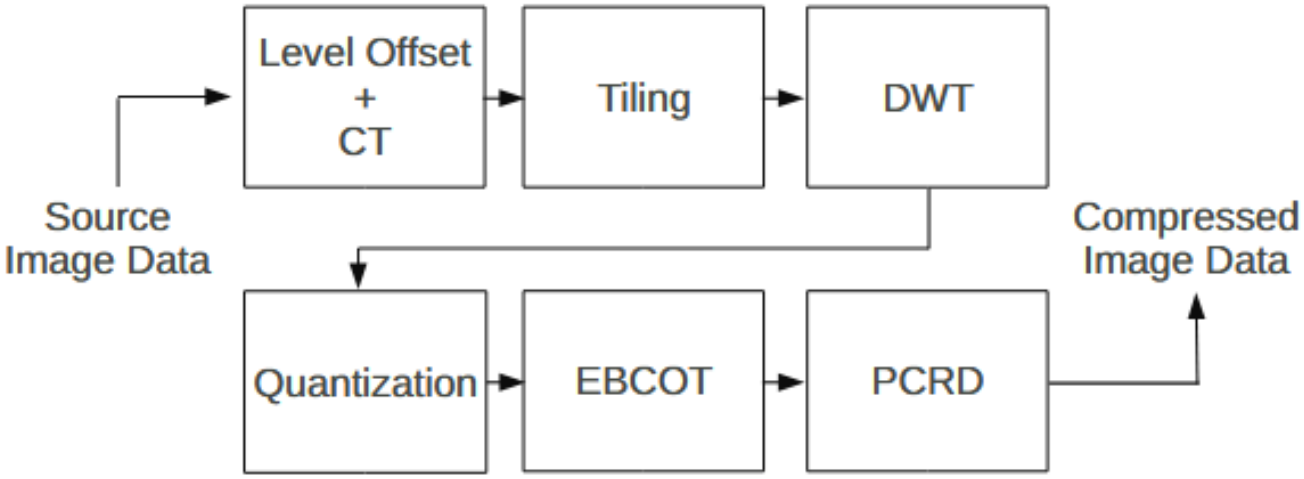

Рис. 2.9 – Операції під час кодування у формат JPEG 2000 [19]

Дослідники реалізували цей кодек на мові CUDA C, аби мати змогу запустити експеримент на графічному процесорі, а для тесту на центральному процесорі скористались готовою опен-сорс бібліотекою OpenJPEG 1.5.0., комерційною бібліотекою Kakadu 6.3.1 та бібліотекою Intel IPP. Під час замірів GPU не був врахований час передачі даних з RAM до графічної карти, проте враховувався час передачі в протилежному напрямі. Для експерименту були використані Intel Xeon X5670 (6 ядер, 2.93 ГГц), Intel Xeon E5-2660 (8 ядер, 2.2 ГГц), а також GPU NVIDIA GeForce GTX580 (512 ядер, 772 МГц), NVIDIA Tesla M2050 (448 ядер, 575 МГц), NVIDIA Tesla S1070 (960 ядер, 610 МГц). Результати порівняльних дослідів CPU і GPU наведені на рис. 2.10. Також результати запусків на одному ядрі з кожної моделі центарльного процесора за допомогою різних бібліотек наведені на рис. 2.11.

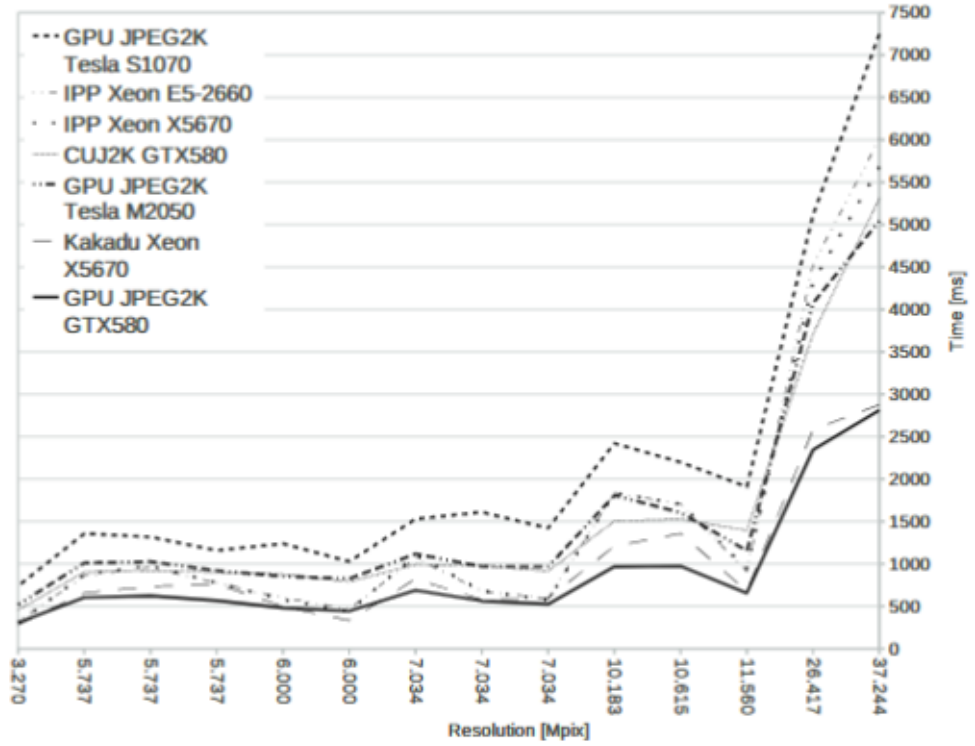

Рис. 2.10 – Досліди на алгоритмах кодеку JPEG 2000 [19]

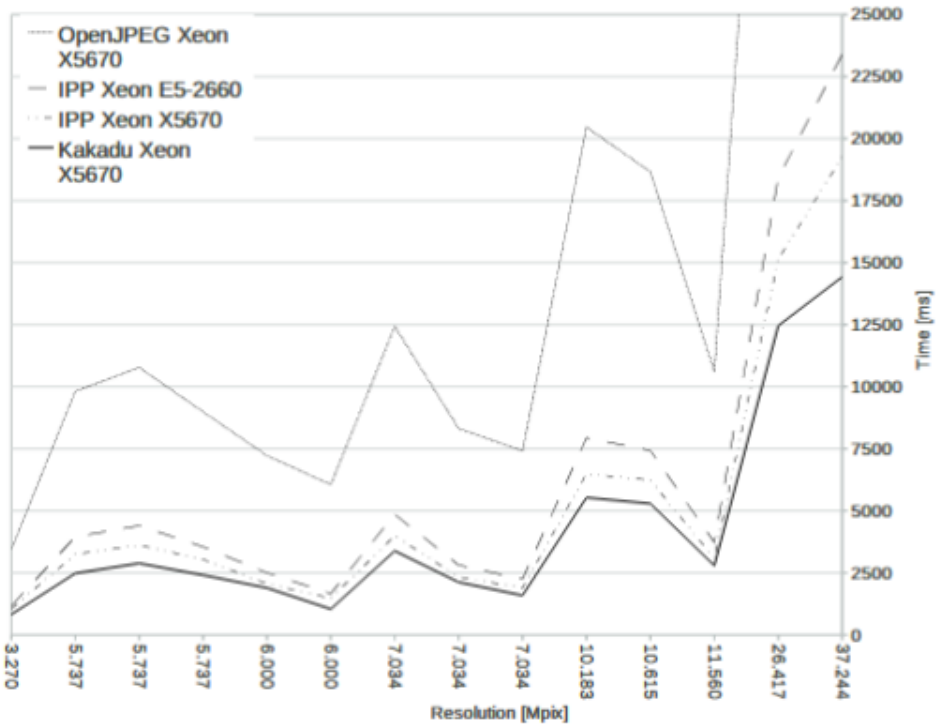

Рис. 2.11 – Досліди на одному ядрі CPU [19]

З цих результатів можна зробити заключення, що вирішальними для дослідження є характеристики та сучасність пристроїв, обраних для проведення тестів. Також важливо відмітити з графіку на рис. 2.11, що програмний код відіграє дуже важливу роль, і якщо мова йде про готові бібліотеки, необхідно враховувати закладені у цей код оптимізації та його можливість працювати із певною апаратною платформою чи пристроями, оптимально використовувати наявні ресурси.

Висновки, які можна зробити з усіх зазначених робіт, підсумовані у графіку з [20] (рис. 2.12).

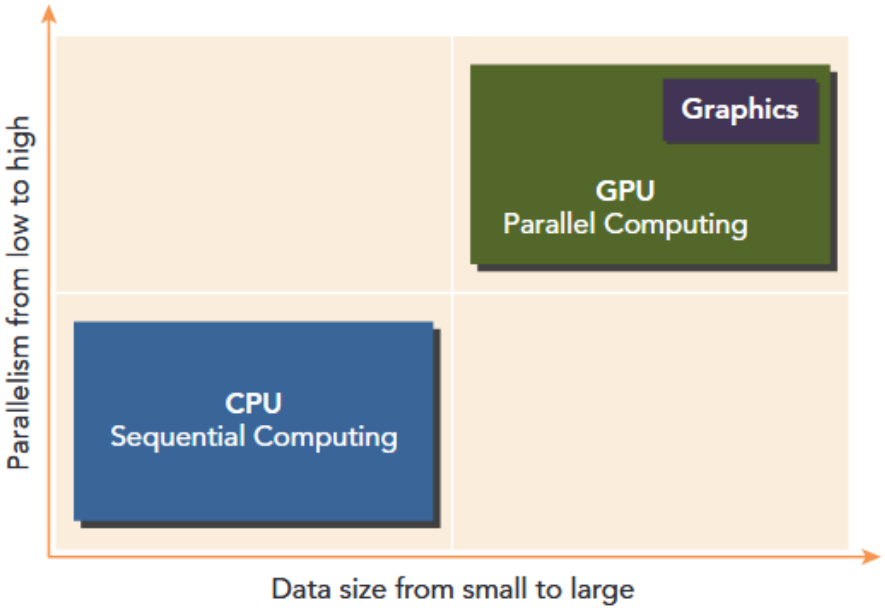

Рис. 2.12 – Використання GPU і CPU відповідно до задач [20]

Беручи до уваги усі заключення, винесені з розглянутих вище праць, дана бакалаврська робота має на меті віднайти оптимальний розміру вхідного набору даних для ефективного прискорення рішень на GPU для наступних задач:

1) скалярний добуток векторів;

2) множення матриць;

3) додавання матриць;

4) додавання матриць із коефіцієнтом;

5) ділення матриці на число

Ефективність буде наведена у порівнянні із ефективністю рішень цих же задач на CPU.

2.4. Висновки до розділу 2

Проаналізувавши описані вище роботи, можна стверджувати, що успішність використання GPU замість CPU для певних задач залежить від:

1) програмного коду самого виконуваного завдання, його коректності та правильного розбиття для паралельної обробки. Якщо ж для роботи використовуються готові бібліотеки, важливо перевірити яку підтримку паралелізму вони забезпечують;

2) у розглянутих роботах GPU показували вищу ефективність коли відбувалась обробка більшої кількості даних;

3) не останню роль грає сучасність пристроїв, на яких проводяться досліди.

Таким чином, для майбутніх дослідів на задачах з матрицями має місце припущення, що якість результатів роботи буде варіюватись між пристроями та залежати від їх новизни та апаратних можливостей; крім того, очікується, що при меншій кількості даних у ефективності переважатимуть CPU, проте зі збільшенням об’ємів для обробки, перевага з’явиться у GPU. Досліди у практичній частині перевірять, чи справджуються ці очікування для згаданих вище задач.

РОЗДІЛ 3 ВИБІР ПРОГРАМНИХ ІНСТРУМЕНТИ ДЛЯ ДОСЛІДЖЕННЯ

Даний розділ має на меті проаналізувати програмні інструменти для досліджень на центральних та графічних процесорах. Будуть розглянуті основні особливості тієї чи іншої платформи, які задовольняють головному критерію – написанню максимально ефективного коду, що є ключовою характеристикою та надалі буде використано у тестах.

3.1. Вибір мови для тестів на CPU

Для проведення тестів на центральному процесорі буде написана програма на мові C++. Вибір даної мови був зумовлений її швидкодією та можливостями наявних інструментів стандартної бібліотеки std. C++ є компільованою мовою. В цьому полягає перша перевага, адже код на компільованих мовах завжди швидше за інтерпретований код. Це зумовлено тим, що компілятор зразу створює цілий виконуваний файл із машинним кодом, на відміну від інтерпретатора, що виконує перетворення коду на машинні інструкції рядок за рядком.

Згідно із бенчмарками мов на різних задачах, C та C++ показують одні з найкращих результатів. Наприклад, на рис. 3.1 наведені результати тестів на задачі із створення bitmap-зображення множини Мандельброта із кількістю елементів 1600^2 на основі даних із проекту The Computer Language Benchmarks Game. Для цього графіку були обрані результати з найкращими показникам по кожній мові (включно з можливими у кожній мові оптимізаціями та паралелізмом).

Ще одним прикладом може слугувати дослідження [22] з теми біоінформатики. Для тесту було імплементовано алгоритм вирівнювання послідовностей – методу порівняння біологічних послідовностей шляхом знаходження схожих ділянок, – та запущено на двох послідовностях ДНК. Програми були запущені на платформах Linux та Windows. Результати наведені на рис. 3.2.

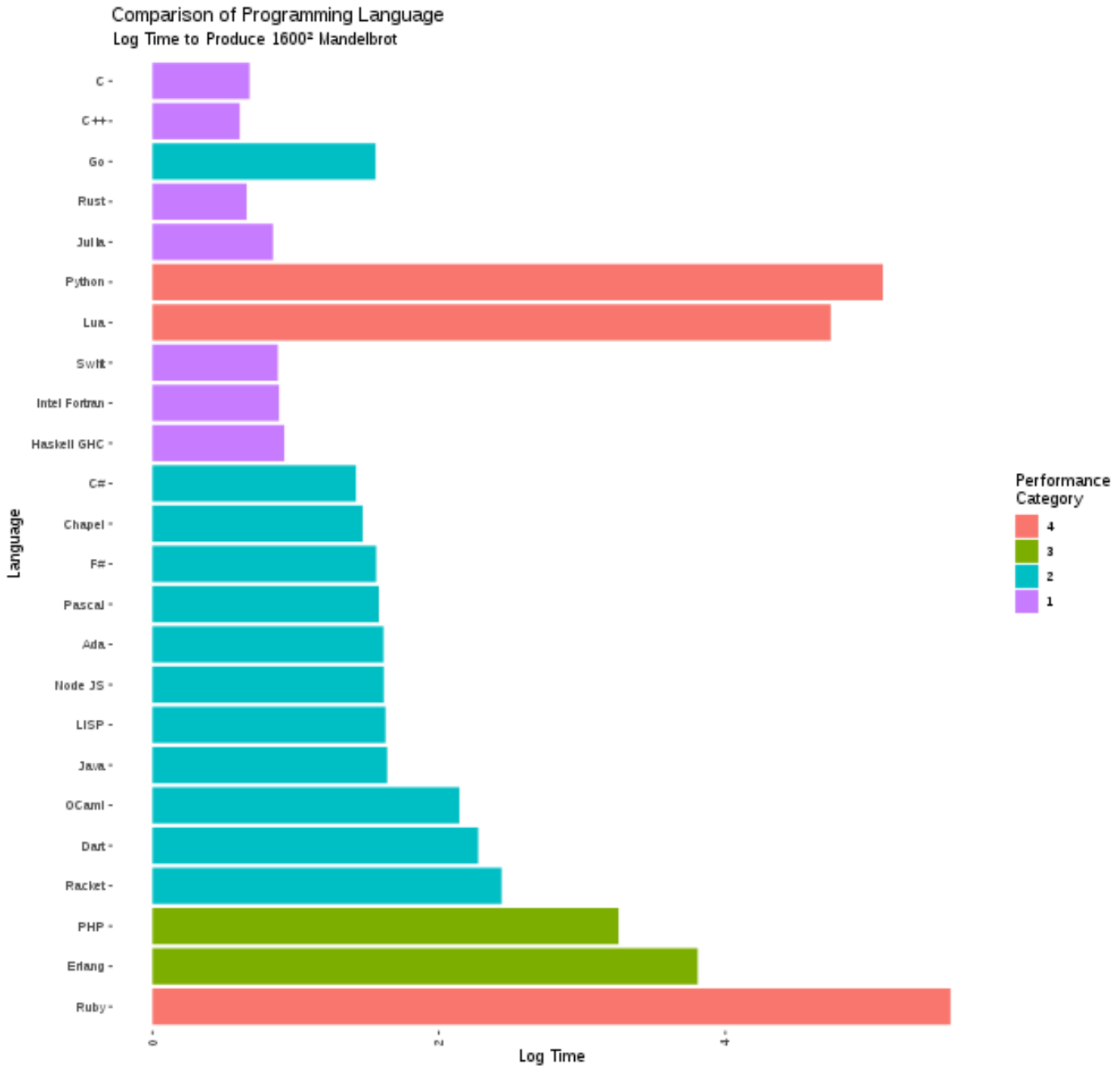

Рис. 3.1 – Результати згідно із The Computer Language Benchmarks Game [21]

Таким чином, подібного роду тести показують швидкодію мов C та C++. На рис. 3.1 окрім цих двох мов до групи високоефективних віднесені також інші мови, наприклад Rust чи Haskell. Проте вибір на користь C++ був зроблений з міркувань того, що ця мова дозволяє напряму працювати із пам’яттю машини і керувати нею на низькому рівні, що дає більше простору для вибору підходів та оптимізацій. В той самий час, ця мова має значно ширший набір інструментів у

своїй стандартній бібліотеці. Особливо це стосується функціоналу для паралелізму та синхронізації.

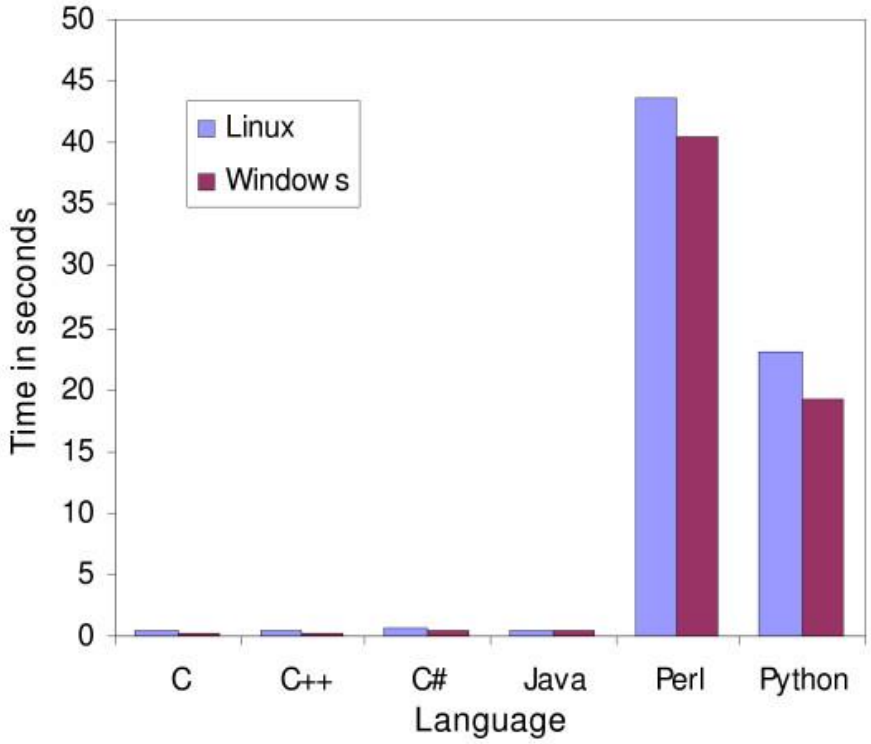

Рис. 3.2 – Результати згідно із [22]

Основним засобом стандартної бібліотеки C++ для імплементації паралелізму є std::thread. std::thread надає платформонезалежну обгортку над системними викликами тієї чи іншої ОС для зручної роботи із багатопоточністю. Потік в операційних системах є найменшою одиницею обробки; їх іноді називають «легковаговими» процесами. Потоки не можуть існувати за межами процесів. Кожен потік має власний програмний лічильник, стек і набір регістрів (рис. 3.3). Їхньою перевагою є швидке переключання контексту та відносно проста комунікація між потоками.

Кожна програма на C++ має принаймні один потік – main(). Далі програміст може створювати додаткові потоки. Потоки запускаються шляхом створення об’єкта std::thread, який визначає завдання для виконання в цьому потоці. Для запуску потоку треба вказати, яку функцію виконувати, та аргументи для неї.

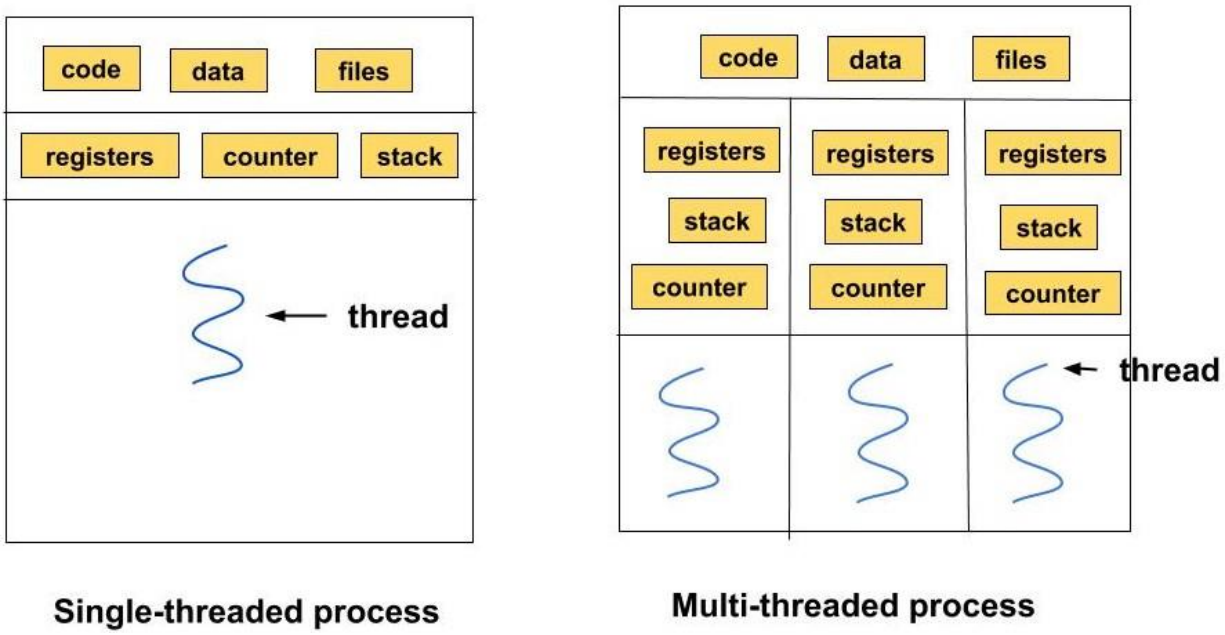

Рис. 3.3 – Схема роботи потоків [23]

Задати функцію для виконання можна різними способами, наприклад, через попереднє визначення цієї цільової функції:

```
void do_some_work(int a, int b);
std::thread my_thread(do_some_work, 1, 2);
```

Ще одним із варіантів є використання лямбда-функцій:

```
std::thread t([](){
        std::cout << "thread function\n";
});
```

Або ж через функціональний об’єкт:

```
class functor_task
{
public:
```

```
    void operator()() const
    {
      do_task_1();
      do_task_2 ();
    }
};
functor_task f;
std::thread my_thread(f);
```

Дочекатись виконання того чи іншого потоку можна через метод join(), його можна викликати для потоку тільки один раз. Виклик detach() для об’єкта std::thread залишає потік працювати у фоновому режимі, без прямих засобів зв’язку з ним. Більше не очікується завершення роботи цього потоку, неможливо отримати об’єкт std::thread, який посилається на нього, а отже і не можна викликати відносно нього join(). Право власності та контроль над такими потоками передаються бібліотеці C++ Runtime Library, яка гарантує, що ресурси, пов’язані з потоком, будуть правильно деалоковані, коли потік завершив роботу. Від’єднані потоки часто називають потоками-демонами, аналогічно до поняття процесів-демонів у UNIX, які працюють у фоновому режиму без комунікації із кінцевим користувачем.

Описані вище поняття є базовими концепціями, які пов'язані зі створенням потоків у C++ і які будуть використовуватись у практичній частині для реалізації та виконанні задач на CPU.

3.2 Вибір платформи для тестів на GPU

CUDA – це універсальна платформа та модель програмування, яка використовує механізм паралельних обчислень у графічних процесорах NVIDIA для вирішення складних обчислювальних задач у більш ефективний спосіб. CUDA дає можливість отримати доступ до графічного процесора для обчислень, як це традиційно робиться на центральному процесорі.

Модель програмування CUDA надає розробникам абстракцію для роботи із графічною картою. Головні поняття у моделі CUDA – хост та пристрій (device). Хост – це центральний процесор, доступний у системі. Системна пам'ять, пов'язана з центральним процесором, називається пам'яттю хоста. Пристрій – це графічний процесор, а пам’ять графічного процесора також називають пам’яттю пристрою. Це є втіленням неоднорідної (гетерогенної) архітектури, яка, на відміну від однорідної архітектури (гомогенної), використовує процесори різних архітектур для виконання програми, застосовуючи до них такі завдання, які для того чи іншого пристрою підходять найкраще, що в результаті забезпечує підвищення продуктивності. Схема такої архітектури наведена на рис. 3.4.

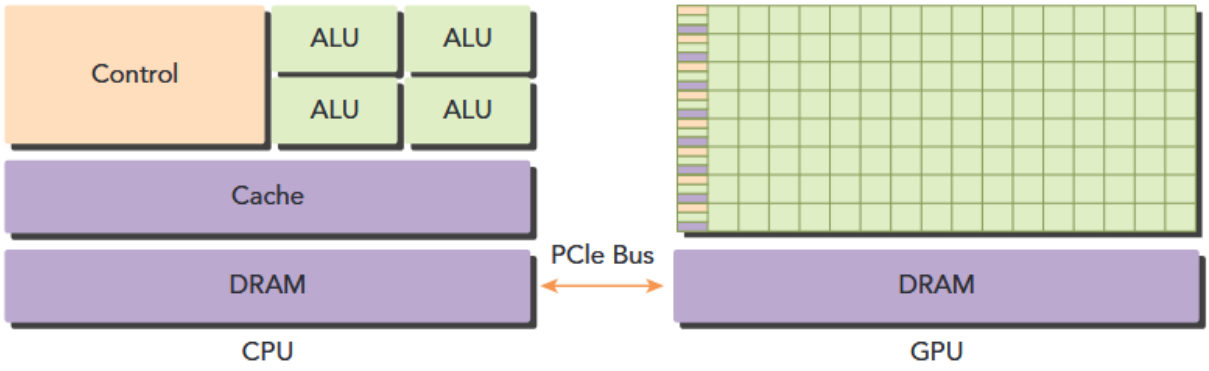

Рис. 3.4 – Неоднорідна (гетерогенна) архітектура [20]

Для виконання будь-якої програми на CUDA необхідно виконати три основні кроки:

1) скопіювати вхідні дані з пам’яті хоста в пам’ять пристрою;

2) завантажити програму GPU та виконати інструкції;

3) скопіювати результати з пам’яті пристрою в пам’ять хоста.

Ілюстрація даної моделі наведена на рис. 3.5.

Також важливим поняттям у CUDA є кернел (ядро). Це сама функція, яка буде виконуватись на графічному процесорі. Паралельна частина програми виконується N разів паралельно N різними потоками CUDA. Таким чином, графічний процесор реалізує потокову обчислювальну модель (stream computing model) – є потоки вхідних і вихідних даних, що складаються з однакових

елементів, які можуть бути оброблені незалежно один від одного. Самі ж функції і код загалом пишеться на розширенні мови C для графічних процесорів NVIDIA – CUDA C, а компіляція відбувається за допомогою компілятора nvcc.

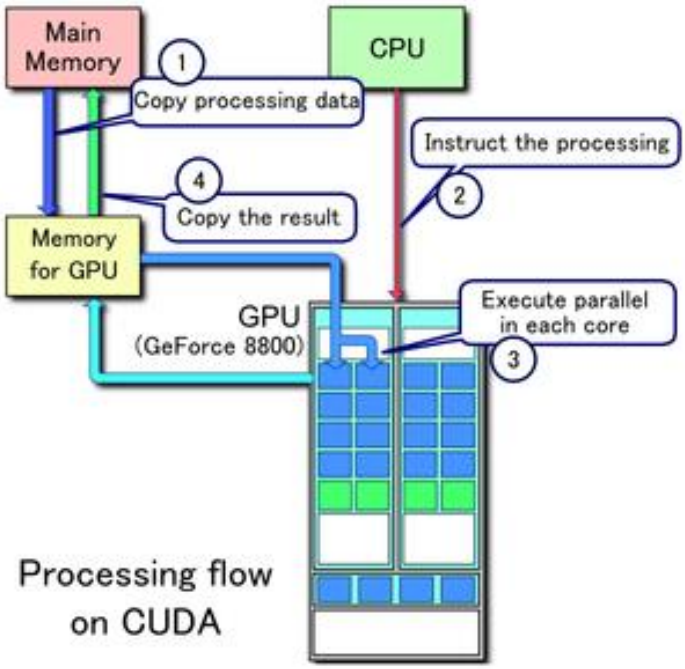

Рис. 3.5 – Алгоритм роботи із CUDA[24]

У моделі CUDA кожній одиниці виконуваного кернела співставлене апаратне ядро GPU, яке також називається потоковим процесором (streaming processor, SP). Групи потокових процесорів формують потокові мультипроцесори (streaming multiprocessor, SM), які, в свою чергу, формують кластери текстурних процесорів (Texture Processor Clusters, TPC). Відповідно до цих апаратних особливостей існують абстракції при написанні коду (рис. 3.6). Виконувані одиниці групуються у ієрархію потоків, блоків та сіток – thread, block, grid. Усі потоки, що породжуються одним запуском ядра, спільно називаються сіткою. Всі потоки в сітці спільно використовують глобальний простір пам'яті. Сітка складається з багатьох блоків потоків (рис. 3.7). Для потоків та блоків існують вбудовані тривимірні змінні в CUDA. Вбудована тривимірна змінна threadIdx використовується для індексації потоків. Тривимірне індексування полегшує програмування CUDA, забезпечуючи природний спосіб індексування елементів у

векторах та матрицях. Таким же чином блоки індексуються за допомогою вбудованої тривимірної змінної blockIdx.

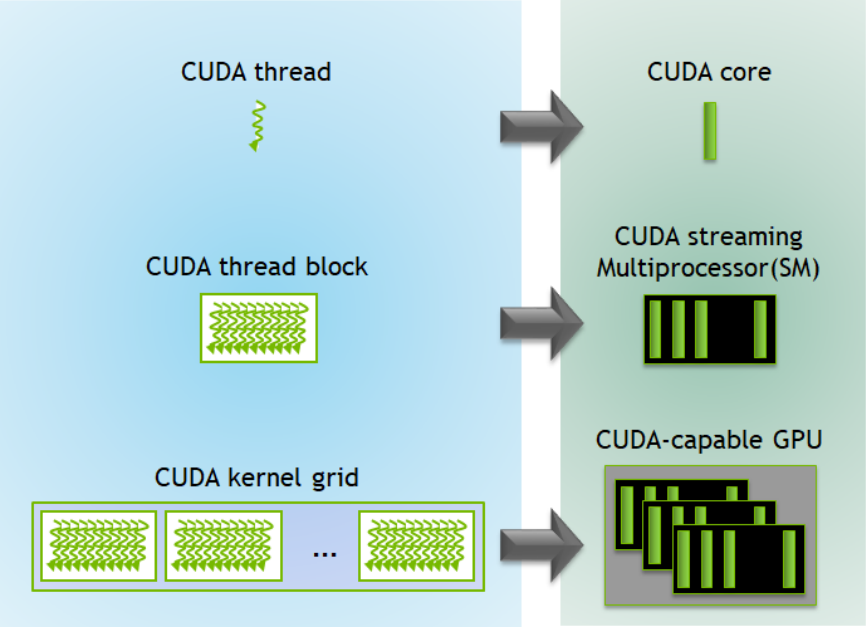

Рис. 3.6 – Абстракції CUDA [25]

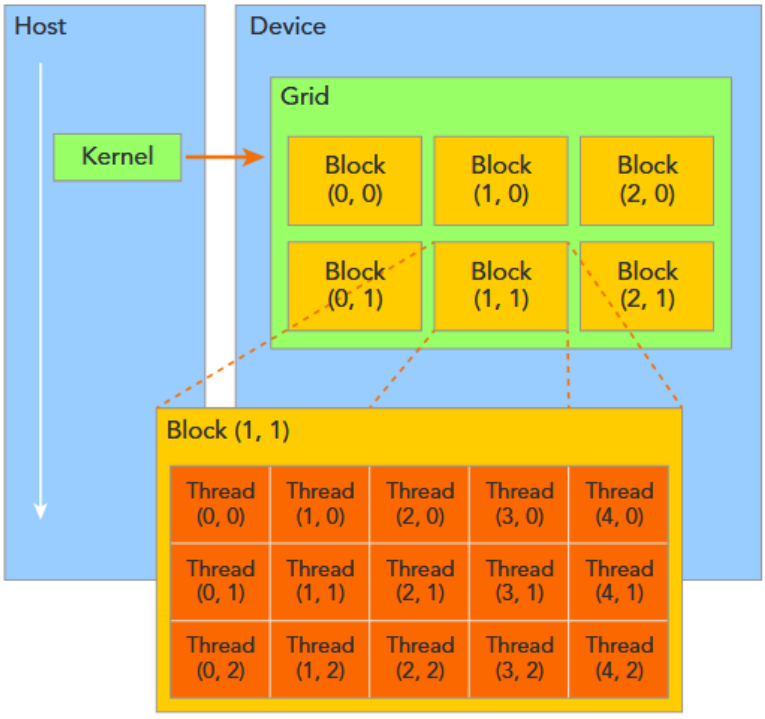

Рис. 3.7 – Структура потоків [20]

Завдяки цим координатам, можна потрібним чином розбити дані для обробки. Звертаючись до полів структури threadIdx.x, threadIdx.y, threadIdx.z та blockIdx.x, blockIdx.y та blockIdx.z можливо звертатись до потоків у тривимірній моделі. Архітектура CUDA обмежує кількість потоків на блок (ліміт 1024 потоків на блок).

Організація пам’яті у GPU із підтримкою CUDA є наступною:

1) Регістри – є приватними для кожного потоку, а це означає, що регістри, призначені потоку, не доступні для інших потоків, і компілятор самостійно приймає рішення про використання регістрів.

2) L1/спільна пам’ять (shared memory, SMEM) – кожен потоковий мультипроцесор має швидку вбудовану пам’ять, яку можна використовувати як кеш L1 і спільну пам’ять. Усі потоки в блоці CUDA можуть спільно використовувати цю пам’ять, а всі блоки CUDA, що працюють на даному потоковому мультипроцесорі, можуть спільно використовувати ресурс фізичної пам’яті, наданий цим SM.

3) Read-only пам’ять – кожний мультипроцесор має кеш інструкцій, постійну пам’ять, текстурну пам’ять та read-only кеш для кернел-коду.

4) L2-кеш – спільний для всіх мультипроцесорів, а отже кожен потік в усіх блоках може доступатись до цієї пам’яті.

5) Глобальна пам’ять – кадровий буфер (DRAM) GPU.

Дана ієрархія представлена на рис. 3.8.

Також для карт з підтримкою CUDA існує таке поняття як обчислювані можливості (compute capability). Це є кількісною характеристикою швидкості виконання певних операцій на GPU, тобто показує, наскільки швидко графічний процесор буде виконувати свою роботу. Цей номер версії може використовуватися програмами під час виконання, щоб визначити, які апаратні функції чи інструкції доступні на поточному графічному процесорі. Кожен графічний процесор має номер версії, позначений як X.Y, де X містить номер основної версії, а Y — номер другорядної версії. Другорядний номер версії відповідає поступовому вдосконаленню архітектури, наприклад, включенню нових функцій та

можливостей. Наприклад, відеокарта GeForce 360M 2010 року випуску має compute capability 1.0, GeForce GTX 750 2014 року випуску має compute capability 5.0, а карта Quadro RTX 2018 року — 7.5.

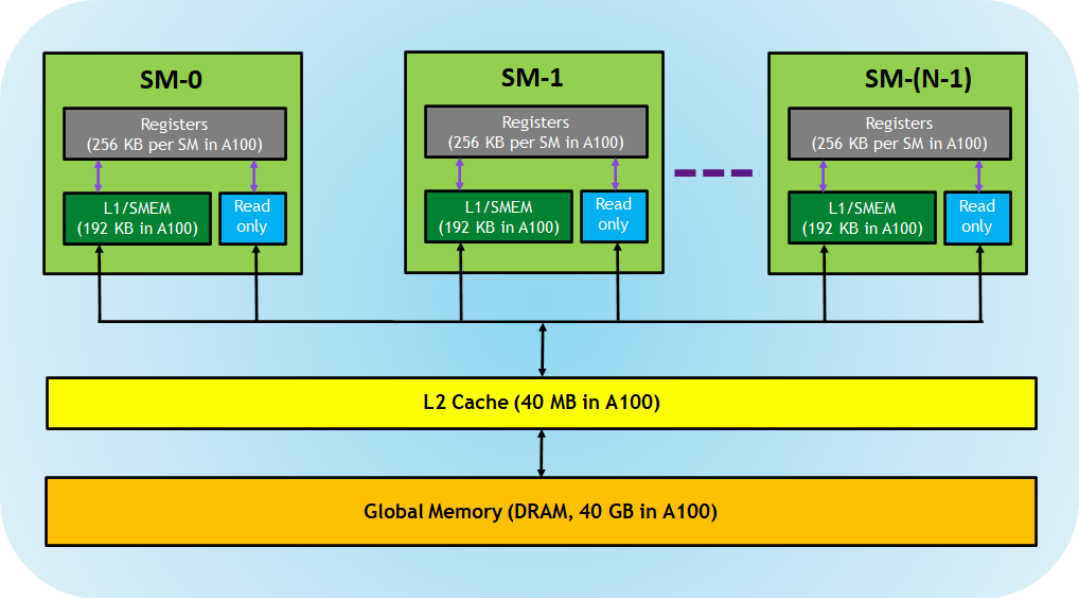

Рис. 3.8 – Структура пам’яті графічного процесору

Таким чином, платформа CUDA є зручним інструментом для неграфічних обчислень загального спрямування. Вона має свої особливості, свою мову, свій компілятор та правила роботи, які орієнтовані на максимальну оптимізацію роботи із графічною картою. Випуск CUDA дав великий поштовх області паралельних обчислень та зіграв значну роль у підвищенні ефективності роботи, та є на сьогоднішній день найпросунутішим інструментом для роботи із неграфічними обчисленнями на відеокатрах.

3.3. Висновки до розділу 3.

Мова C++ була обрана для даної роботи за свою швидкість та широкий набір інструментів. Пишучи програмний код на C++, програміст має змогу безпосередньо контролювати робочі ресурси, що є дуже важливим аспектом під час вимірювання ефективності. Підтверджують високу ефективність мови й

різноманітні тести, що були наведені у розділі. В той самий час обгортка std::thread дозволяє просто писати платформонезалежний багатопоточний код, не втрачаючи ефективності. Для роботи із GPU наявна платформа CUDA для графічних карт NVIDIA. Мова CUDA C є надмножиною мови C та містить функціонал для роботи зі специфічними концепціями для програмування GPU. Сама модель потоків CUDA орієнтована на максимальну оптимізацію та спрощення роботи під час рішення SIMD-орієнтованих задач, та надає широкий функціонал для розробника, багато бібліотек та можливостей. Поєднання C++ та CUDA дозволить отримувати одні з найкращих результатів, по максимуму використовуючи потенціал пристроїв, що дуже важливо при вимірюванні часу роботи та продуктивності.

РОЗДІЛ 4 ПРАКТИЧНА ЧАСТИНА. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПРИСТРОЇВ

4.1 Опис задачі та використаних пристроїв

Даний розділ має на меті створити порівняння ефективностей графічних та центральних процесорів на прикладі наступних задач:

1) скалярний добуток векторів;

2) множення матриць;

3) додавання матриць;

4) додавання матриць із коефіцієнтом (kA+B);

5) ділення матриці на число.

Як було встановлено із досліджень, описаних в попередніх розділах, кількість потоків та розмір даних є одними з вирішальних характеристик. Дана робота перевірить за яких розмірів даних оптимальніше працює той чи інший пристрій, будуть проведені заміри часу роботи для цих розмірів. Для векторів розмір буде від 256 до 81 920 000, для матриць – від 40^2 до 9064^2.

В першу чергу, вкажемо характеристики пристроїв, обраних для тестів. У якості CPU будуть використовуватись дві моделі: Intel i7-10850H (рис.4.1) та Intel і Intel i5-7200U (рис. 4.2). Обидва процесори представлені у персональних ноутбуках.

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. 4.1 – Характеристики Intel i7-10850H [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. 4.2 – Характеристики Intel i5-7200U [27]

У якості графічних карт були обрані NVIDIA GeForce 940MX, NVIDIA Tesla P100 та NVIDIA GeForce RTX 3080 Ti (у персональному ноутбуці, у клауді Google Colab та на віддаленій машині відповідно). Характеристики наведені на рис. 4.3, 4.4 та 4.5.

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. 4.3 – Характеристики GeForce 940MX [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. 4.4 – Характеристики Tesla P100 [29]

Graphic Engine	NVIDIA® GeForce RTX™ 3080 Ti
Bus Standard	PCI Express 4.0
OpenGL	OpenGL®4.6
Video Memory	12GB GDDR6X
Engine Clock	OC mode : 1785 MHz (Boost Clock) Gaming mode : 1755 MHz (Boost Clock)
CUDA Core	10240

Рис. 4.5 – Характеристики RTX 3080 [30]

Розглянемо детальніше представлені задачі. Матриці на CPU та GPU будуть представлені як одновимірні масиви потрібного типу, а коректна індексація відбуватиметься завдяки зсуву на розмірність матриці (які у цьому випадку є квадратними). Завдяки цьому алокація та адресація відбуватиметься швидше, адже робота буде проводитись із однією неперервною ділянкою пам'яті. Для обробки матриць за умовами даної роботи немає необхідності залучати додаткові інструменти синхронізації даних у потоках, адже існуватиме повний паралелізм даних по відношенню до кожного елемента. На рис. 4.6 представлений схематичний обхід матриці. Головний принцип – обробка одним потоком певного діапазону рядків, а всередині потоку відбувається обхід по індексах стовпчиків. На CUDA ж процес простіший: як описано в попередньому розділі, для індексації елементів будуть використовуватись змінні blockIdx і blockIdy, а розмірність сітки потоків буде визначена так, аби вона емулювала двовимірний блок (матрицю).

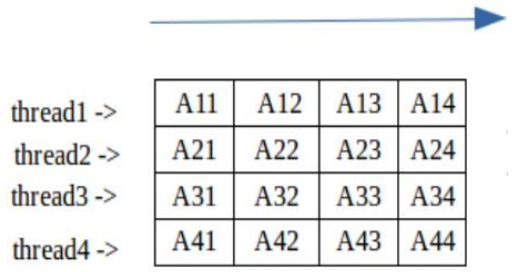

Рис. 4.6 – Схематичний обхід матриці [31]

4.2 Визначення найоптимальнішої кількості потоків на CPU

В першу чергу, визначимо оптимальний режим роботи CPU. Проведемо бенчмарк часу виконання задач із різною кількістю потоків на обидвох центральних процесорах. На графіках нижче (рис.4.7-4.14) будуть представлені середні значення після 10 запусків із перебором по різних конфігураціях. Також і в усіх графіках надалі усі значення рахуються як середні з 10 запусків. Час в мілісекундах, size – розмір квадратної матриці за одним виміром (відповідно, загальна кількість елементів складає $size^2$

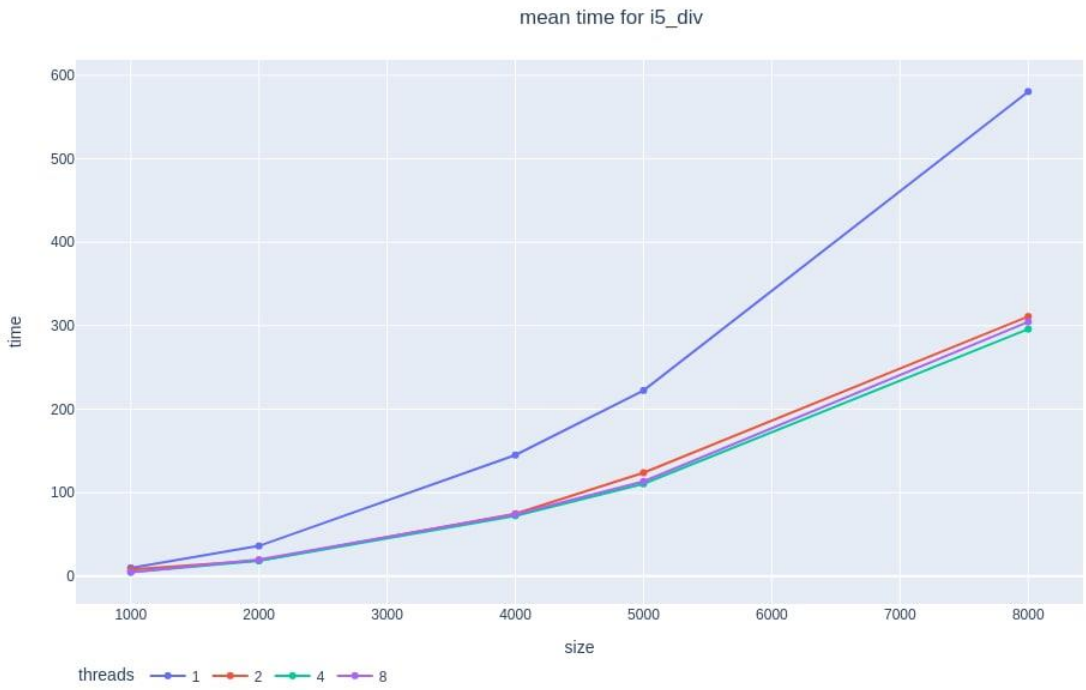

Рис. 4.7 – Середній час для ділення матриці на число на Intel i5

З рис. 4.7 видно, що в цьому випадку найоптимальніше програма відпрацювала у 4 потоки. У той самий час, ефективність у випадку двох та восьми потоків також була близька до найкращого результату.

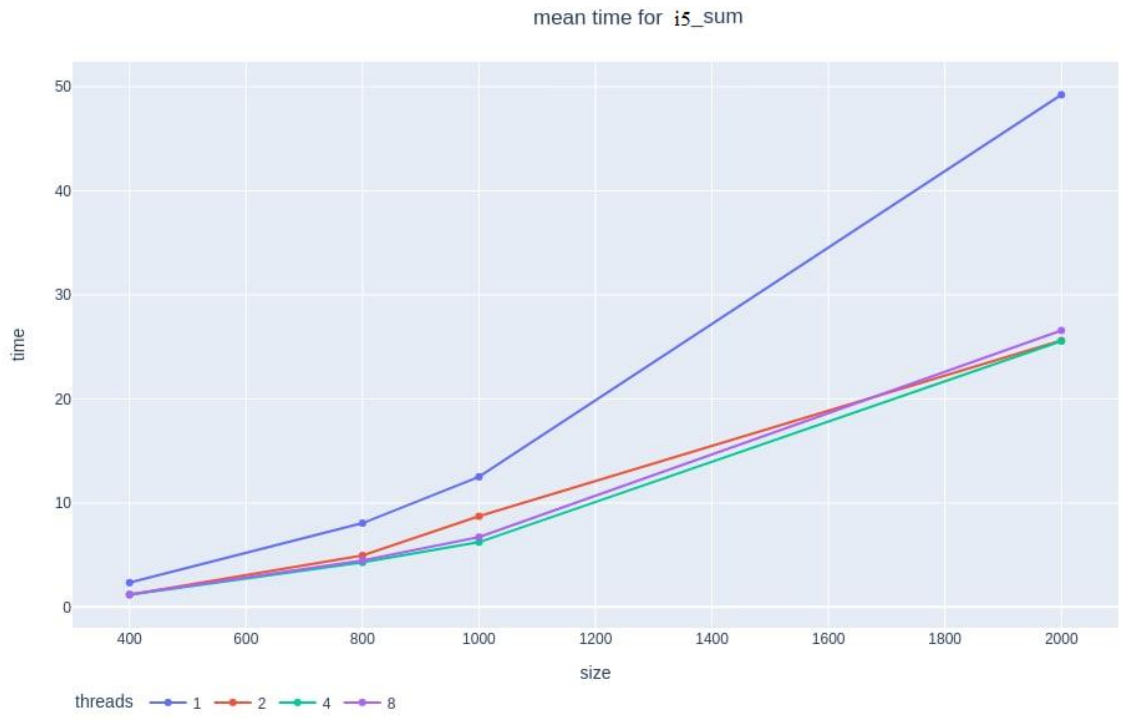

Рис. 4.8 – Середній час для додавання матриць на Intel i5

На рис. 4.8 ситуація аналогічна до рис. 4.7.

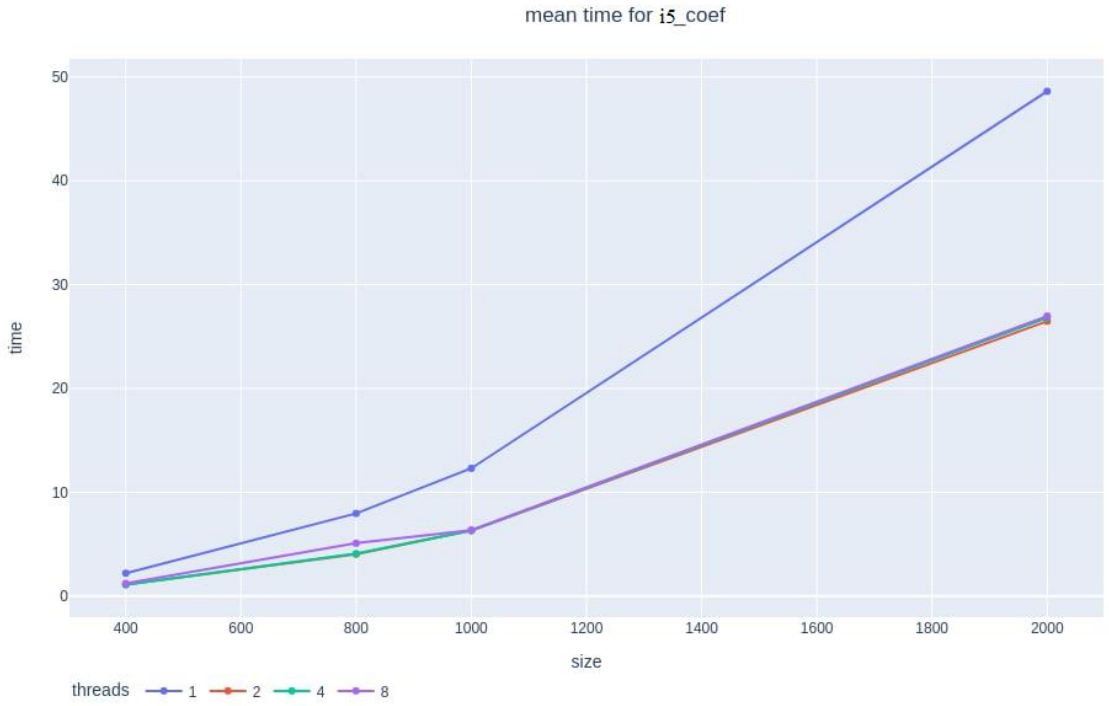

Рис. 4.9 – Середній час для додавання матриць з коефіцієнтом на Intel i5

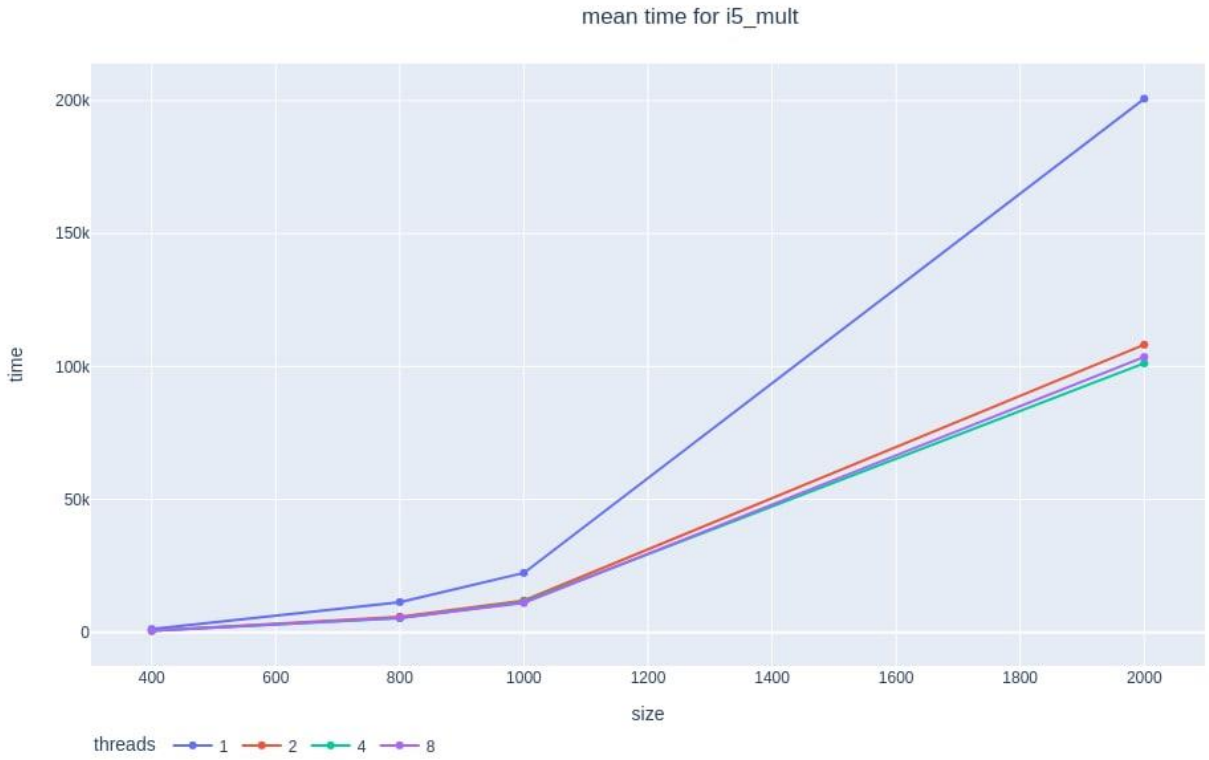

Рис. 4.10 – Середній час для множення матриць на Intel i5

На графіку з рис. 4.10 спостерігається найкращий час при чотирьох потоках.

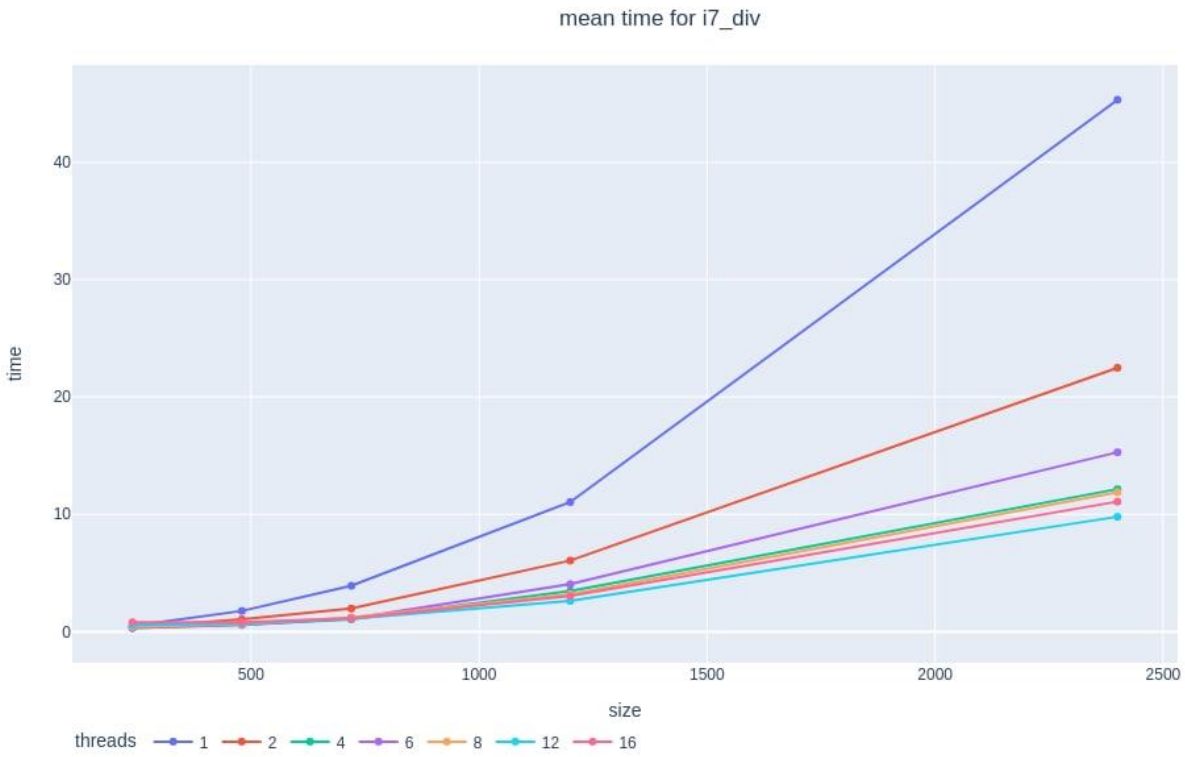

Рис. 4.11 – Середній час для ділення матриці на число на Intel i7

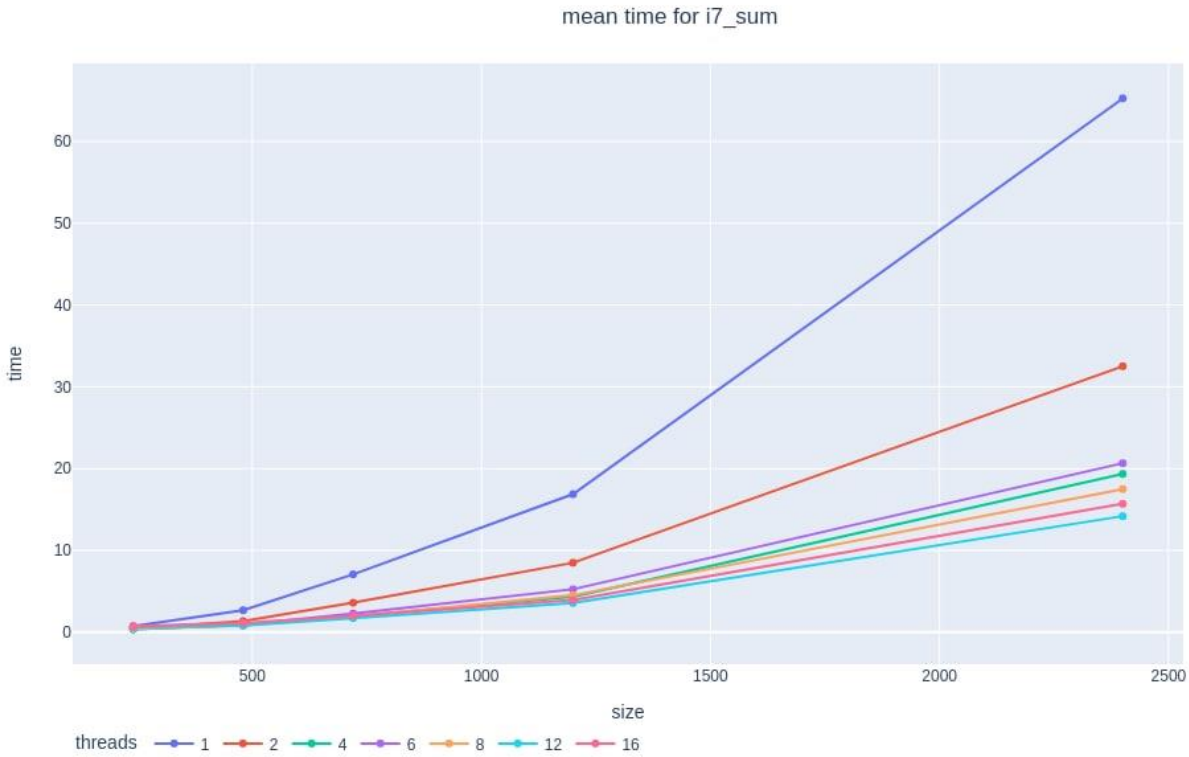

Рис. 4.12 – Середній час для додавання матриць на Intel i7

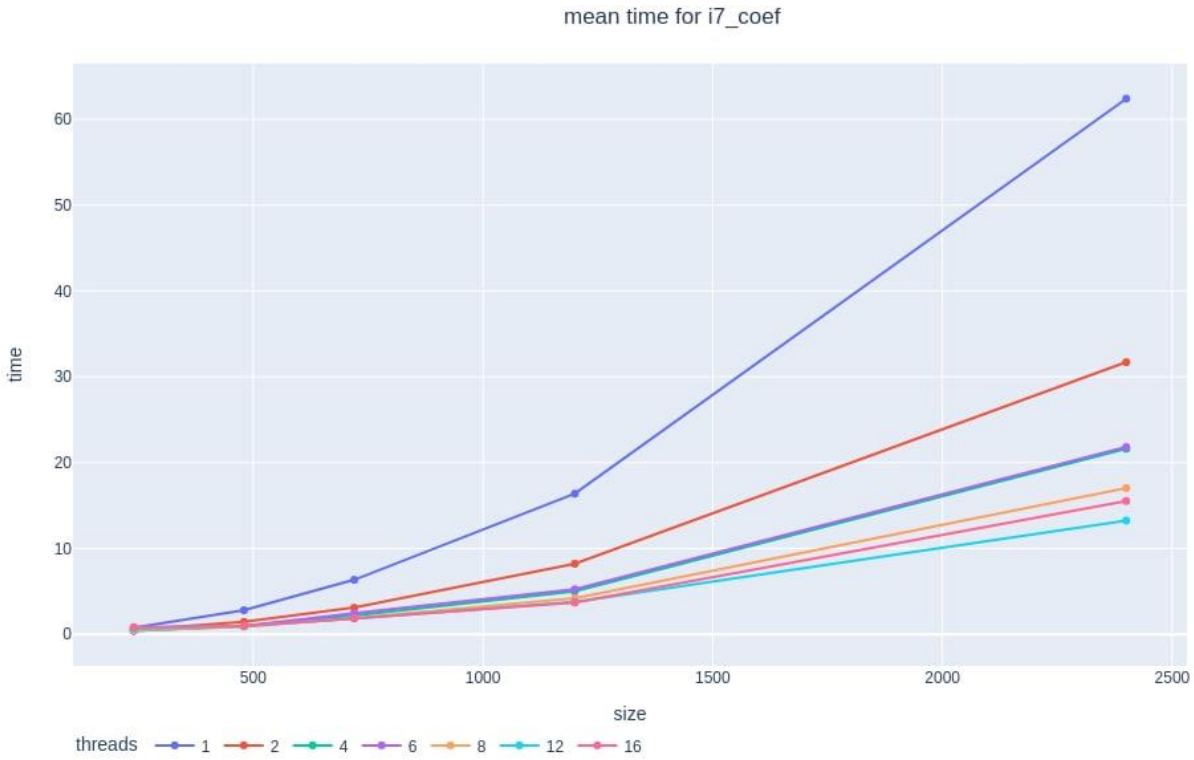

Рис. 4.13 – Середній час для додавання матриць з коефіцієнтом на Intel i7

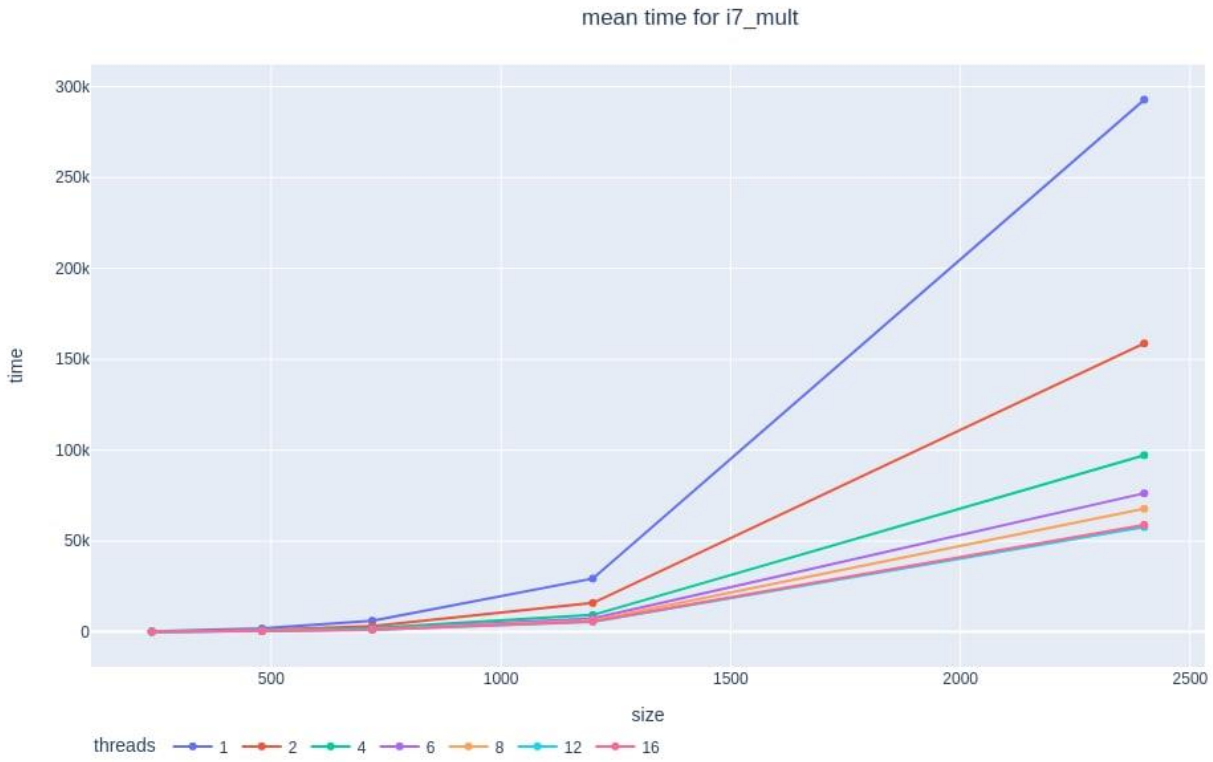

Рис. 4.14 – Середній час для множення матриць на Intel i7

З рис. 4.11 – 4.14 із тестами на процесорі i7 всі результати стабільно показують, що оптимумом є використання 12 потоків, а робота із більшою кількістю потоків, ніж передбачена архітектурою, створює зайві витрати на створення цих додаткових потоків, на які пристрій не розрахований. Із тестами на i5 найкращий час зазвичай теж на 4 потоках, проте тести із 2 та 8 не показують помітно гірші результати.

Окремо розглянемо задачу скалярного добутку векторів. Розрахунок відбувається за формулою $\vec{a} * \vec{b} = a_1 * b_1 + a_2 * b_2 + \cdots + a_n * b_n$. Отже, результатом роботи усіх потоків є одне число. У такому випадку не обійтись без інструментів синхронізації. В даному випадку було обрано такий інструмент як std::atomic. Використовуючи шаблонний клас std::atomic<> (в даному випадку std::atomic<float>), та метод fetch_add, що дозволяє робити атомарне додавання, було створене потокобезпечне збереження результатів кожної операції. Fetch_add для атомарних змінних типу float був визначений у стандарті C++20, який і використовувався для компіляції у даній роботі. Між std::mutex та std::atomic вибір був зроблений на користь останнього, тому що блокування м’ютексом створює більші затримки, аніж процедура неперервного додавання. Використання м’ютексів має сенс тоді, коли необхідно синхронізувати, наприклад, блок операцій, що йдуть підряд і працюють зі спільними даними. Відповідно, імплементація потоку для скалярного добуту представлена 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. 4.15 – Імплементація скалярного добутку

Результати виконання цієї задачі на кожному з процесорів наведені на рис. 4.16 і рис.4.17. Час вказаний в мілісекундах, а розмірність позначає загальний розмір досліджуваного вектора.

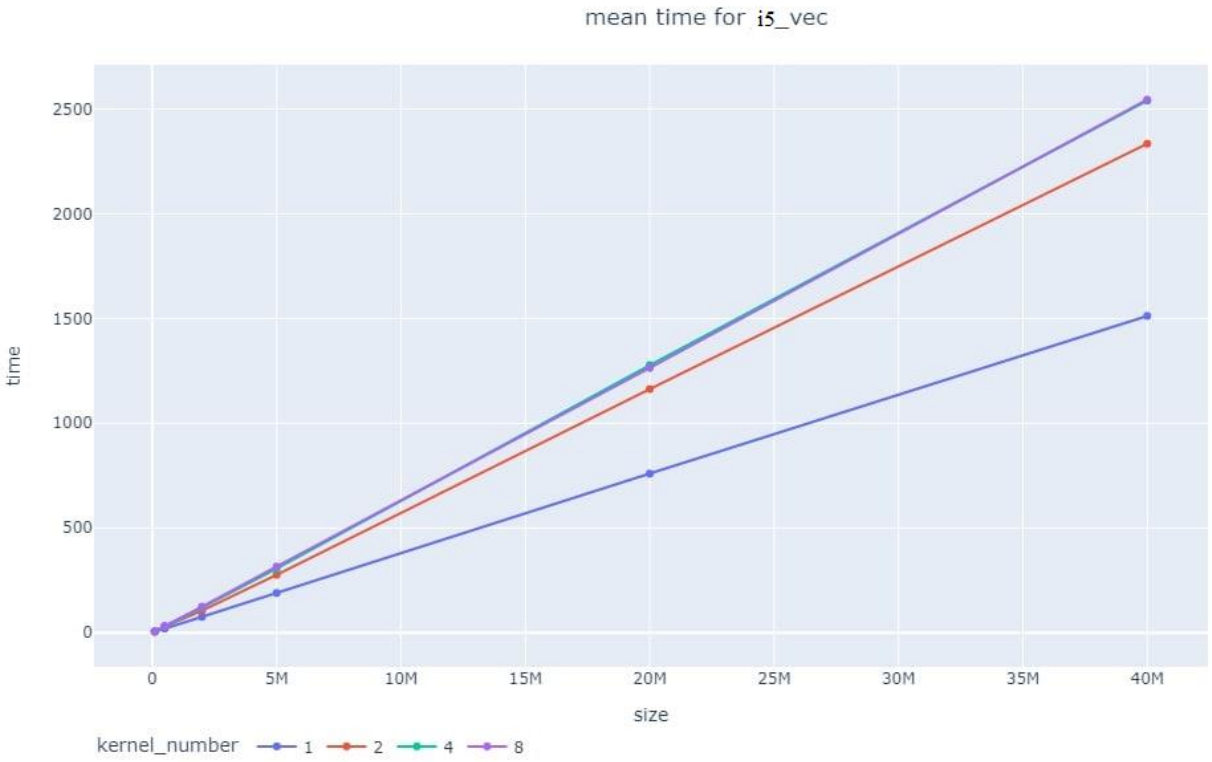

Рис. 4.16 – Результати виконання скалярного добутку векторів на Intel i5

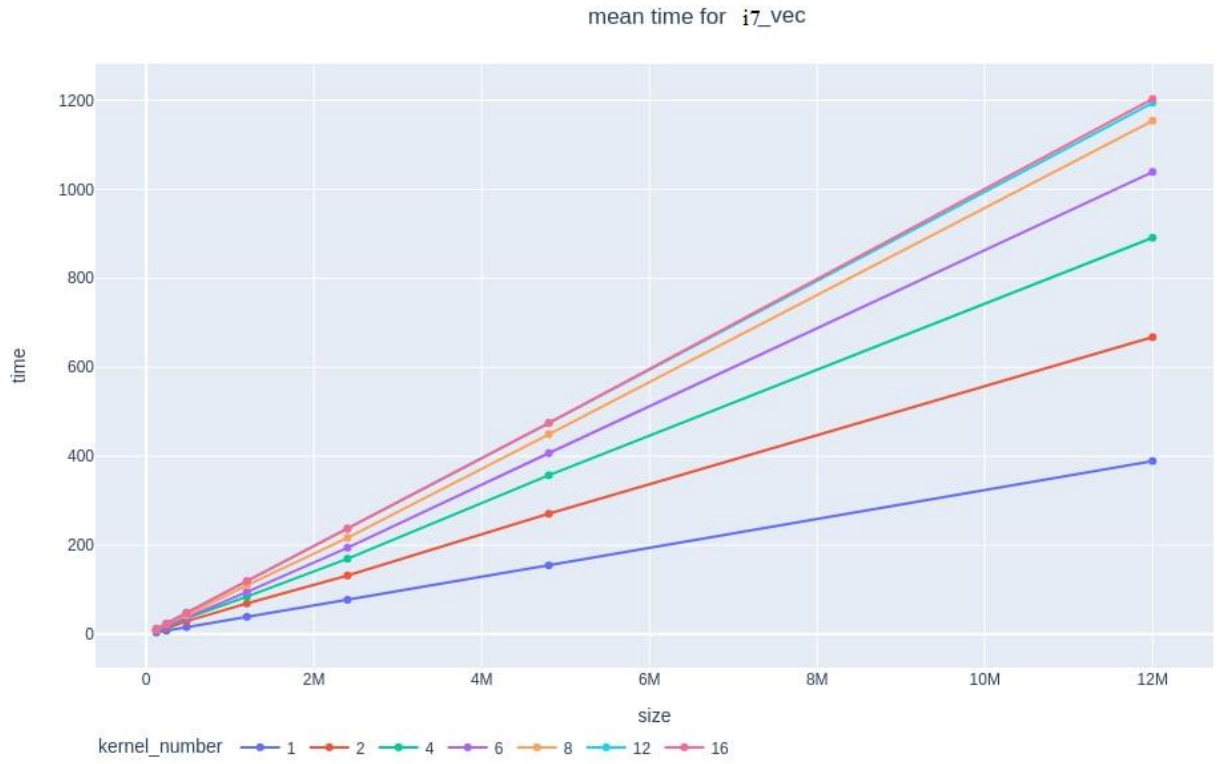

Рис. 4.17 – Результати виконання скалярного добутку векторів на Intel i7

З цих результатів можна побачити, що найкращий час виконання в обох випадках був при використанні лише одного потоку. Це свідчить про те, що навіть

такий інструмент синхронізації як атомарні операції не у всіх випадках користування дає прискорення, а в даному випадку навіть сповільнює роботу. Це пояснюєтсья тим, що після обробки кожної пари елементів векторів відбувається оновлення кінцевого результату і, відповідно, виконання атомарної операції. Часте виконання цієї операції створює затримку у роботі потоків. Для вирішення цієї проблеми була переписана функція, що виконується всередині потоків. Тепер вводиться проміжна змінна на кожен потік, яка акумулює в собі результати додавання у кожному потоці, адже обхід елементів всередині самої функції відбувається вжк послідовно. У кінці роботи потоку цей проміжний результат додається до цільової змінної, що є спільною для всіх. Така функція наведена на рис. 4.18.

Тепер, маючи покращений варіант методу багатопоточного розрахунку скалярного добутку векторів, повторимо попередні заміри із кількома значеннями потоків на обидвох процесорах. Результати наведені на рис. 4.19 та рис. 4.20.

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

Рис. 4.18 – Покращена функція для розрахунку скалярного добутку

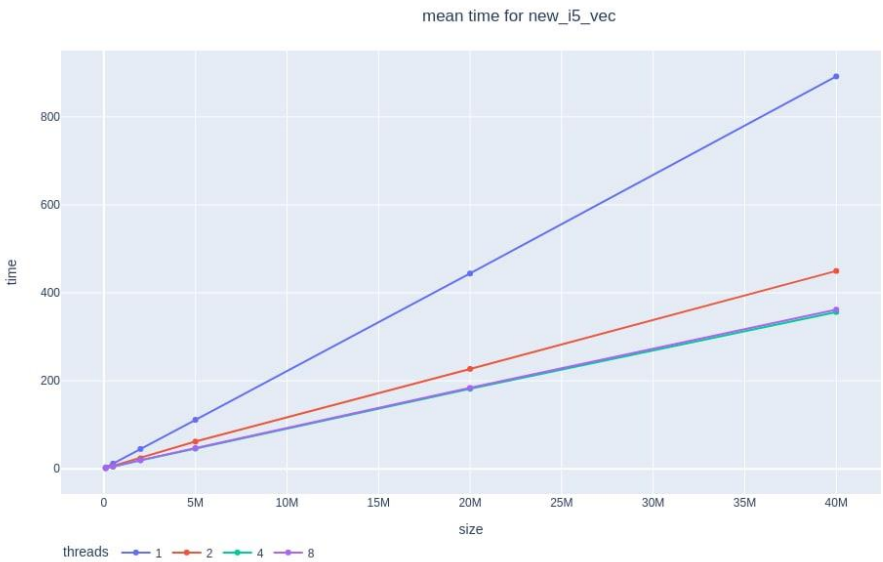

Рис. 4.19 – Результати виконання покращеного скалярного добутку на Intel i5

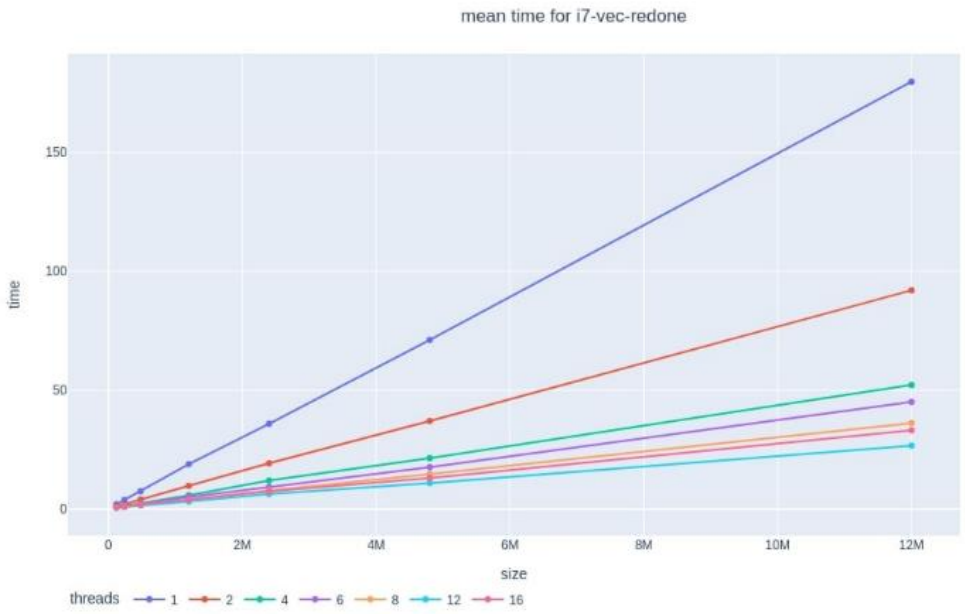

Рис. 4.20 – Результати виконання покращеного скалярного добутку на Intel i7

Тепер ці результати повністю співпадають із заключеннями, отриманими із операцій над матрицями. З цих міркувань, для порівняння роботи CPU та GPU тести на i7 будуть проводитись у 12 потоків, а на i5 – у 4 та 8.

4.3 Проведення бенчмарків

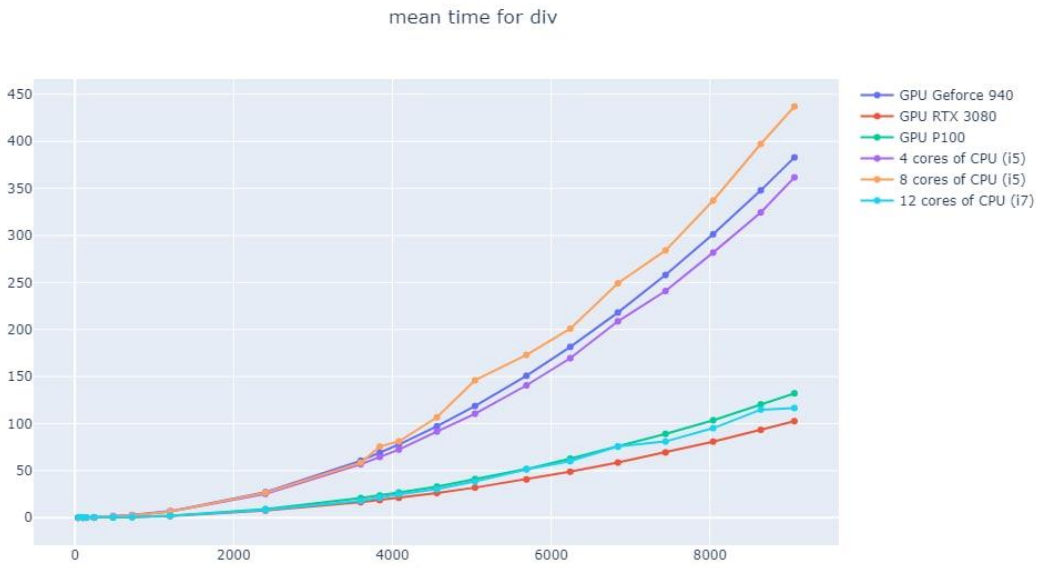

Рис. 4.21 – Результати виконання ділення матриці на різних пристроях

З цього рисунку можна бачити, що найкращі результати показує RTX 3080, найшірше працює Intel i5 на 8 потоках. 4 потоки на i5 працюють краще за GeForce 940, а Intel i7 працює краще за Intel i5, GeForce 940 та Tesla P100.

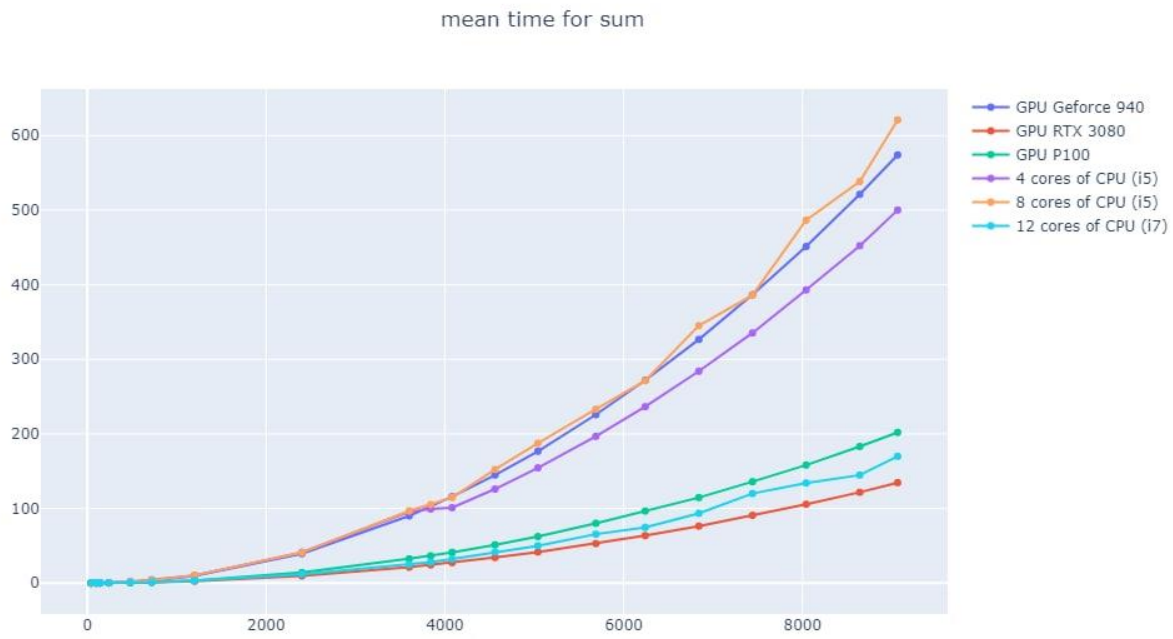

Рис. 4.22 – Результати виконання додавання матриць на різних пристроях

Дані результати повністю аналогічні до результатів на рис. 4.21.

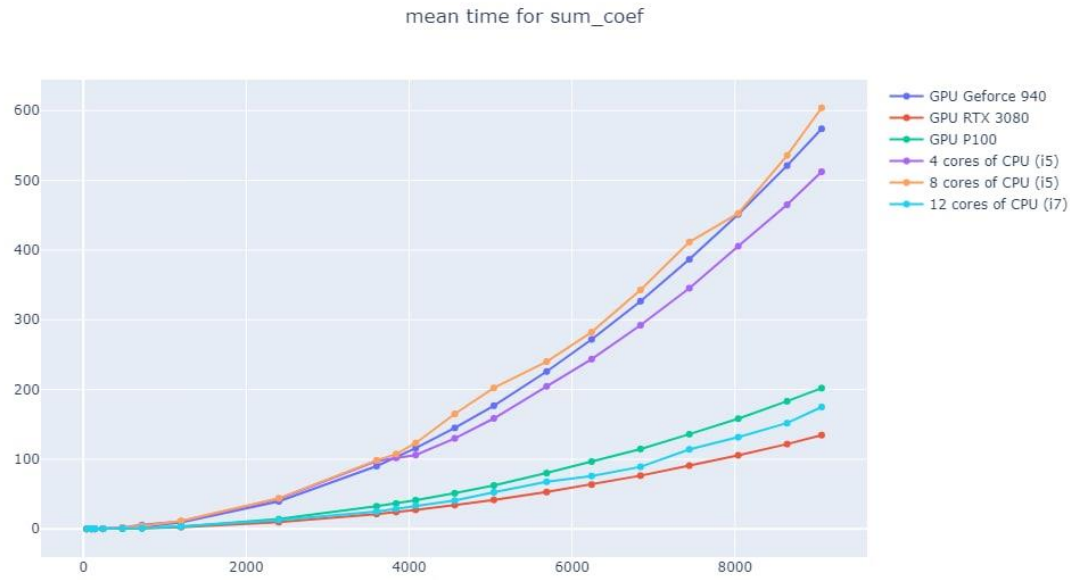

Рис. 4.23 – Результати виконання додавання матриць з коефіцієнтом на різних пристроях

На рис. 4.23 патерни ефективності пристроїв такі ж самі. Розглянемо детальніше результати додавання матриць, з коефіцієнтом та без, аби визначити, чи ця одна операція додає різницю у часі. Порівняймо середні значення для додавання без коефіцієнта та з коефецієнтом по пристроях (див. Додаток Б). З даних у додатку можна побачити, що помітна різниця у часі не спостерігається.

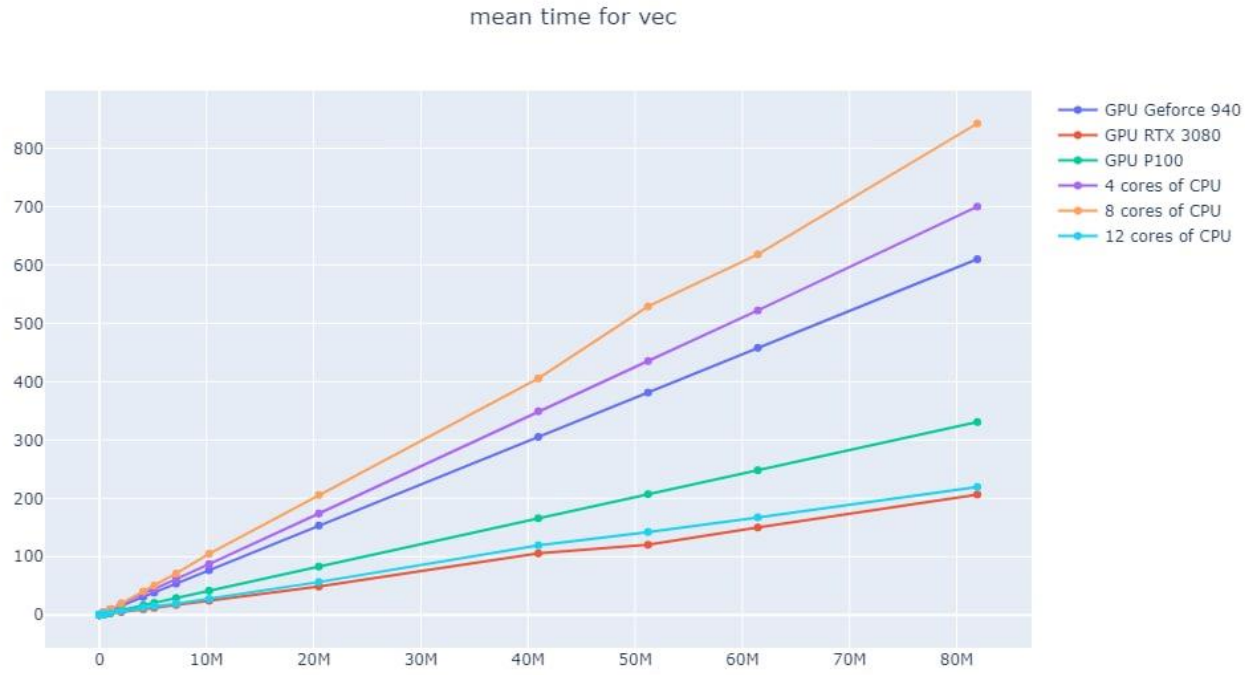

Рис. 4.24 – Результати виконання скалярного добутку двох векторів

На рис. 4.24 можна спостерігати, що тепер GeForce 940 показує кращі результати за Intel i5. Для решти пристроїв результати не змінились. Варто зазначити, що всередині CUDA-потоків також використовувалось атомарне додавання, аналогічне першій імплементації на CPU. Попри це, помітні проблеми зі сповільненням роботи не спостерігаються.

Розглянемо рис 4.25, на якому зображено графік виконання множення двох матриць та її зближена версія. Ситуація на цьому графіку помітно відрізняється від інших, адже на ньому чітко видно, що усі центральні процесори показують поганий результат, а усі графічні процесори працюють оптимально. Це пояснюється тим, що складність коду у потоках в C++ програмі складає O(n^3), а у

потоках CUDA – O(n). Так стається через те, що у потоці C++ необхідно проходити циклом по обраних рядках, всередині цього цикла – по усіх колонках, а всередині цього – ще раз пройтись циклом по розмірності матриці. У CUDA два перші цикли відсутні завдяки 2D-індексації. Через це множення матриць стандартним способом завжди краще робити на CUDA.

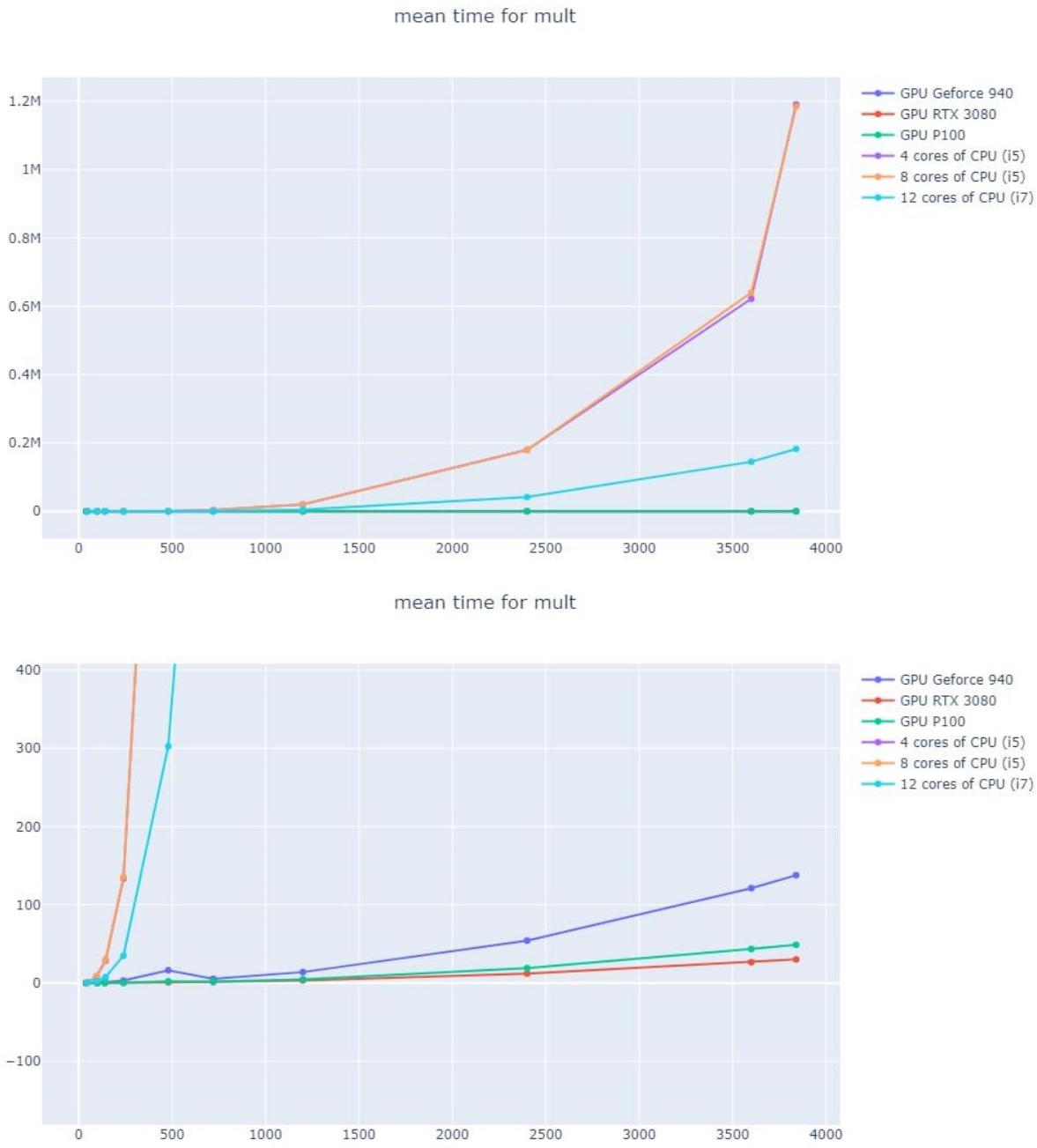

Рис. 4.25 – Результати виконання множення двох матриць

Переглянувши результати з інших графіків, можна побачити, що не завжди CPU програє GPU. Крім того, GeForce 940 показала гірші результати, аніж усі CPU у трьох тестах з п’яти. Варто також додати, що виконання на 8 потоках у Intel i5 на ще більшому пулі даних показало, що оптимальна кількість потоків для даного процесора складає все ж 4, як і вказано виробником у специфікації. У чому принципова різниця між графічним процесором із найкращим результатом, та графічним процесором із найгіршим? Чому GPU може програвати CPU? Розглянемо швидкості кожної операції (передача даних в обидві сторони та чистий час обрахунків) окремо на прикладі операцій в задачі скалярного добутку (рис. 4.26).

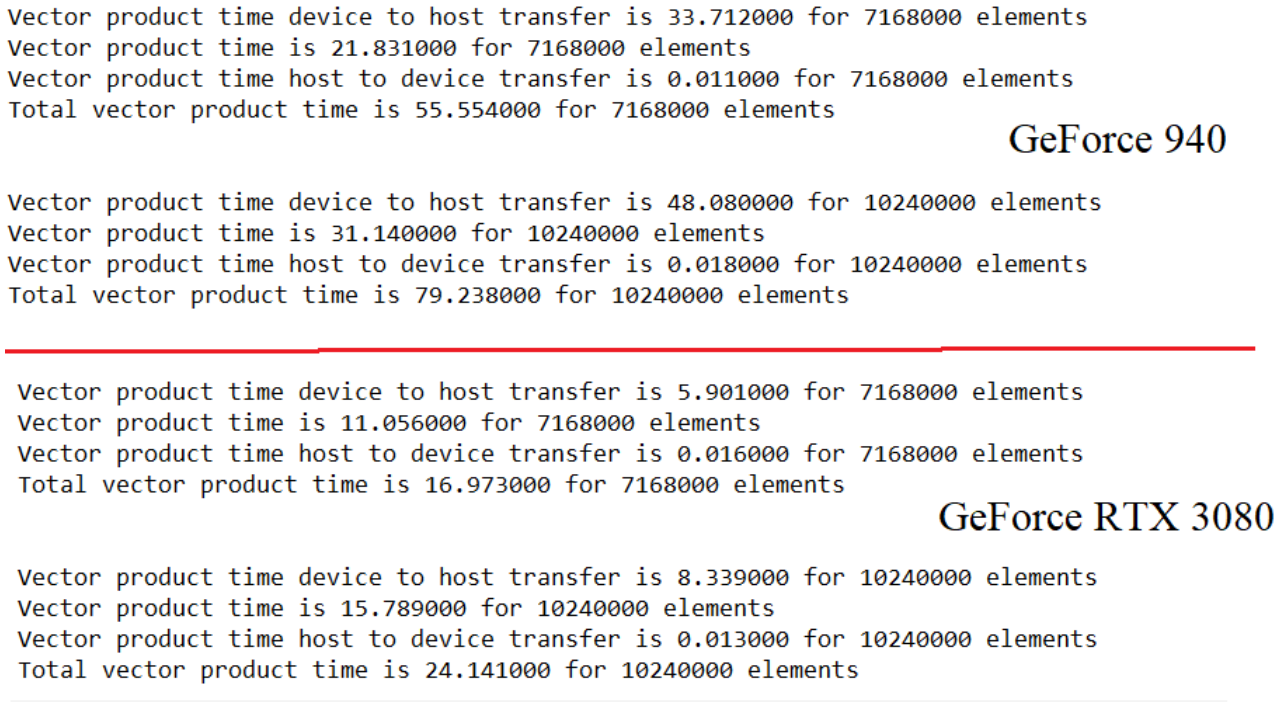

Рис. 4.26 – Порівняння швидкостей кожної окремої операції

З рисунку можна побачити, що більш вдосконалена RTX 3080 виконує усі операції швидше. Також важливо відмітити, що для випадку GeForce 940 час трансферу складає більшу частину часу заміру. Це ще раз підтверджує те, що питання швидкості передачі даних між пристроями у гетерогенній системі грає ключову роль в оптимізації обчислень. Іншими словами, обираючи GPU замість CPU, важливо для своїх задач всередині потоків мати достатню складність обчислень, щоб компенсувати трансфери даних. Спробуємо це продемонструвати

на задачах, для яких GeForce показував найгірші результати. Розглянемо Intel i5 на 4 потоках, Intel i7 на 2 потоках, GeForce 940 та Tesla P100. Замість виконання однієї операції в потоці (додавання, ділення), повторимо її 2, 5, 10 разів. Оскільки, як вже було відзначено, різниця в часі виконання для двох версій додавання матриць була незначною (додаток Б), для наступних тестів різниця в графіках цих двох випадків не очікується (рис. 4.27 – 4.35).

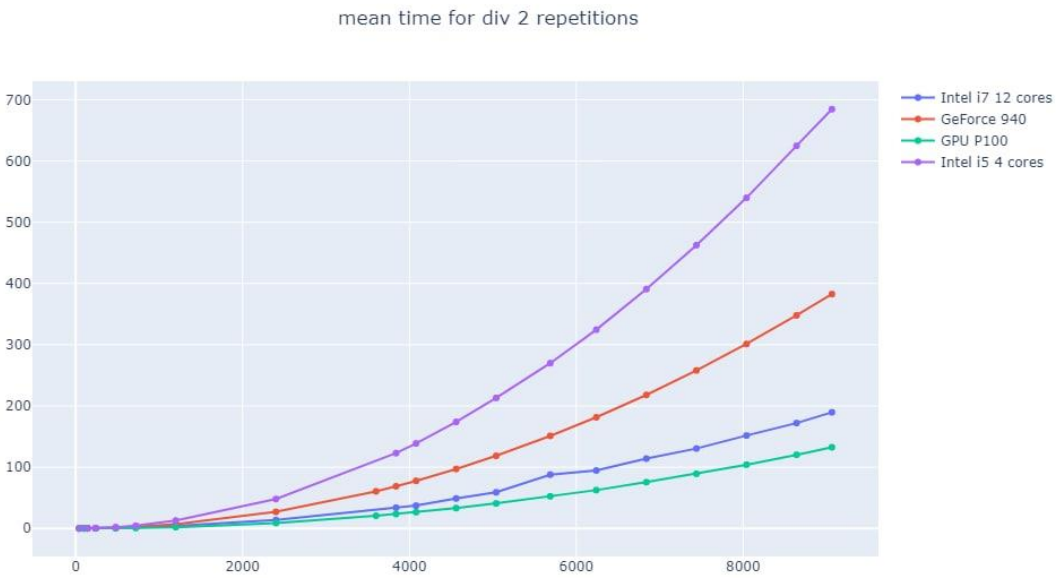

Рис. 4.27 – Ділення матриць із повторенням операції 2 рази всередині потоку

Рис. 4.28 – Додавання матриць із повторенням операції 2 рази всередині потоку

Рис. 4.29 – Додавання матриць із коефецієнтом із повторенням операції 2 рази всередині

Тепер можна побачити з рис. 4.27 – 4.29, що GeForce працює швидше за Intel i5. Ситуація з P100 та Intel i7 теж змінилась.

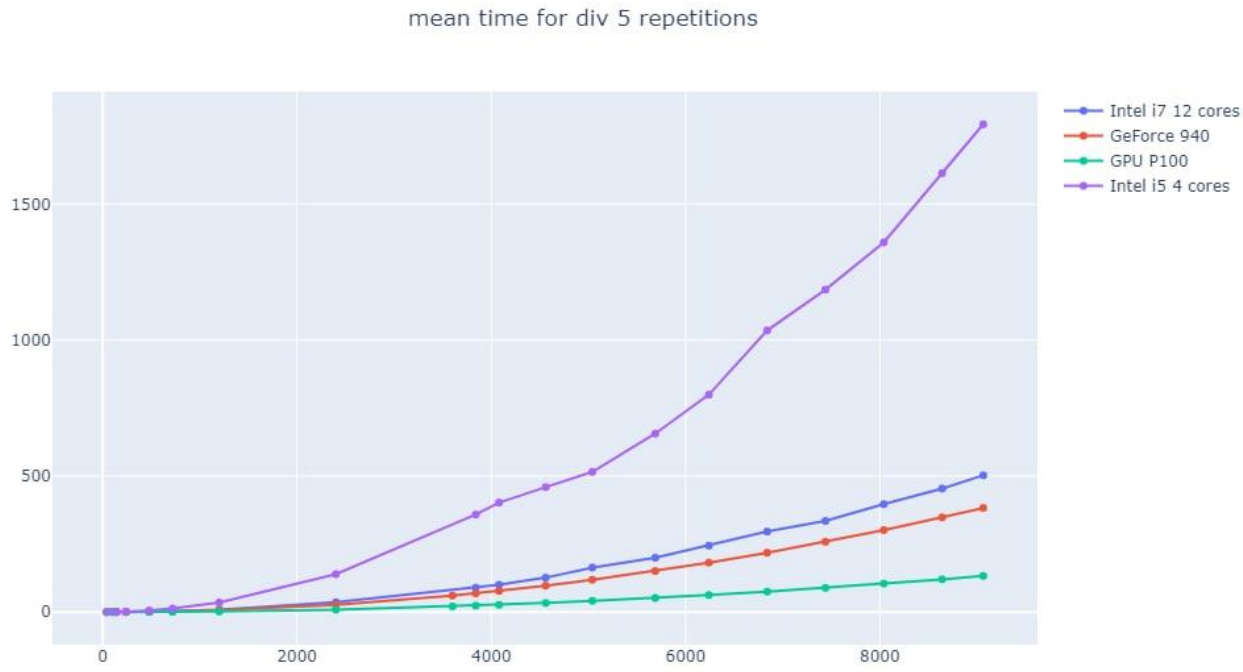

Рис. 4.30 – Ділення матриць із повторенням операції 5 разів всередині потоку

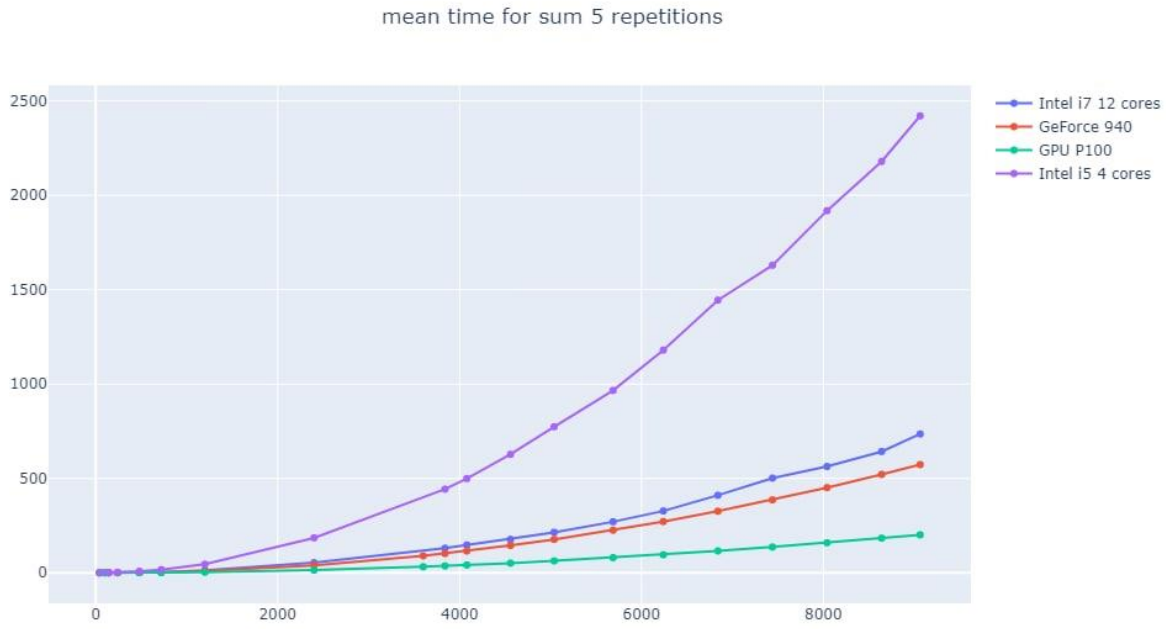

Рис. 4.31 – Додавання матриць із повторенням операції 5 разів всередині потоку

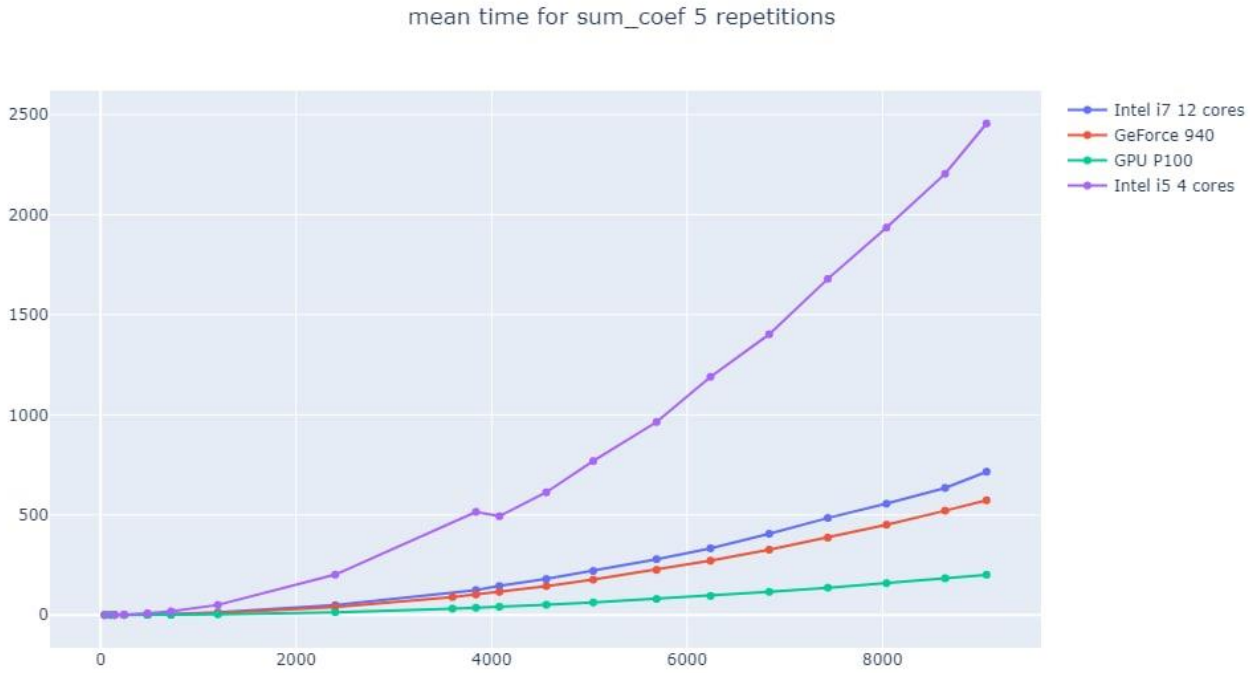

Рис. 4.32 – Додавання матриць із коефецієнтом із повторенням операції 5 разів всередині потоку

Як можна побачити з графіків 4.30 – 4.32, ситуація по відношенню до випадку з 2 потоками змінилась, всі GPU швидше за CPU.

Рис. 4.33 – Ділення матриць із повторенням операції 10 разів всередині потоку

Рис. 4.34 – Додавання матриць із повторенням операції 10 разів всередині потоку

Рис. 4.35 – Додавання матриць із коефецієнтом із повторенням операції 10 разів всередині потоку

З рис. 4.33 – 4.35 теж можна бачити, що в такому випадку CPU працюють гірше за всі GPU.

Таким чином, приходимо до висновку, що краща робота GPU в порівнянні з CPU в першу чергу залежить від балансу між ресурсомісткістю трансферу та складністю виконкуваних в потоках задач. При достатній кількості операцій всередині потоків дійсно спостерігається перевага GPU над CPU. Отже, питання оптимальних розмірів даних, при яких варто використовувати GPU, є другорядним та випливає з таких особливостей, як специфіка конкретної задачі та апаратні можливості швидкої передачі даних між хост-пристроєм та відеокартою.

Повертаючись до питання про оптимальні розміри даних, переглянемо наближені графіки з результатами виконання початкових задач та прослідкуємо, чи відбувались суттєві зміни у продуктивності пристроїв при менших розмірностях. Розглянемо рис. 4.36 – 4.40.

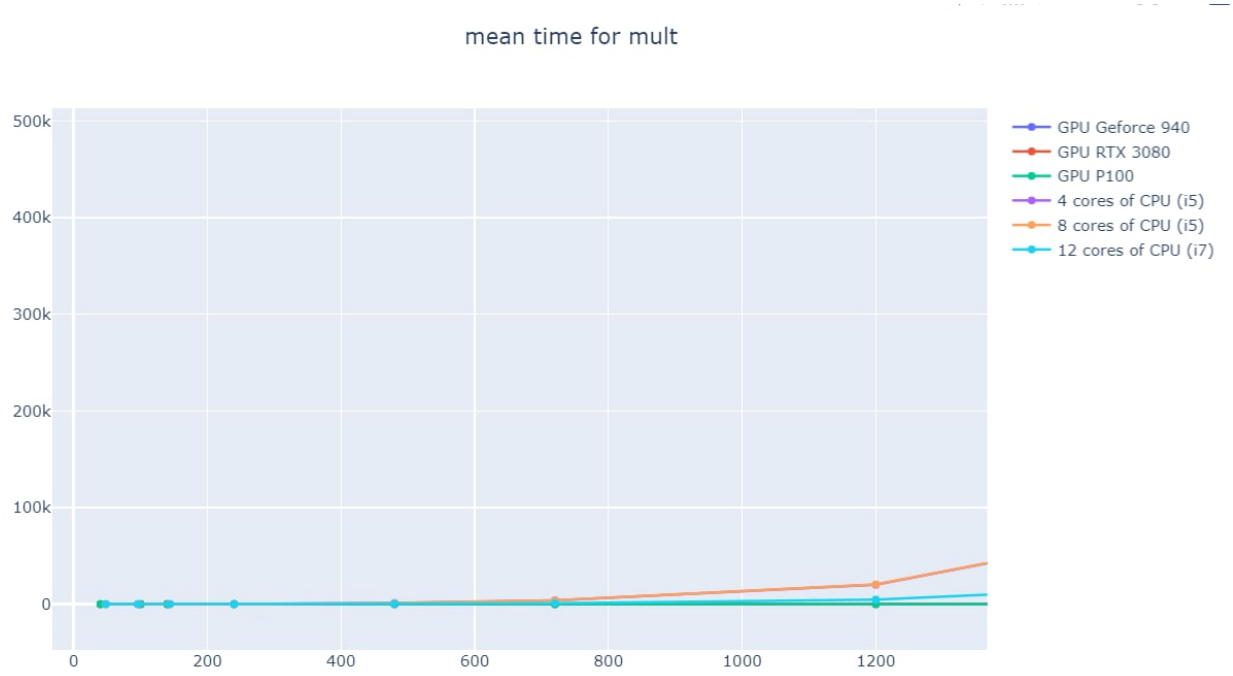

Рис. 4.36 – Результати множення матриць у ближчому масштабі

На графіку 4.36 ніяких суттєвих змін чи різких переходів у ефективності пристроїв в залежності від розмірності не спостерігається, тобто загальна картина ефективностей стала – розв'язок на CPU починає працювати помітно гірше, починаючи з матриць розмірністю близько 200*200, а до 1000*1000 GPU показують схожі результати.

Рис. 4.37 – Результати ділення матриці на число у ближчому масштабі

На графіку 4.37 так само немає особливої різниці між результатами на більшому і меншому масштабах. Розділення на 2 явні групи по ефективності відбувається на розмірності близько 700*700.

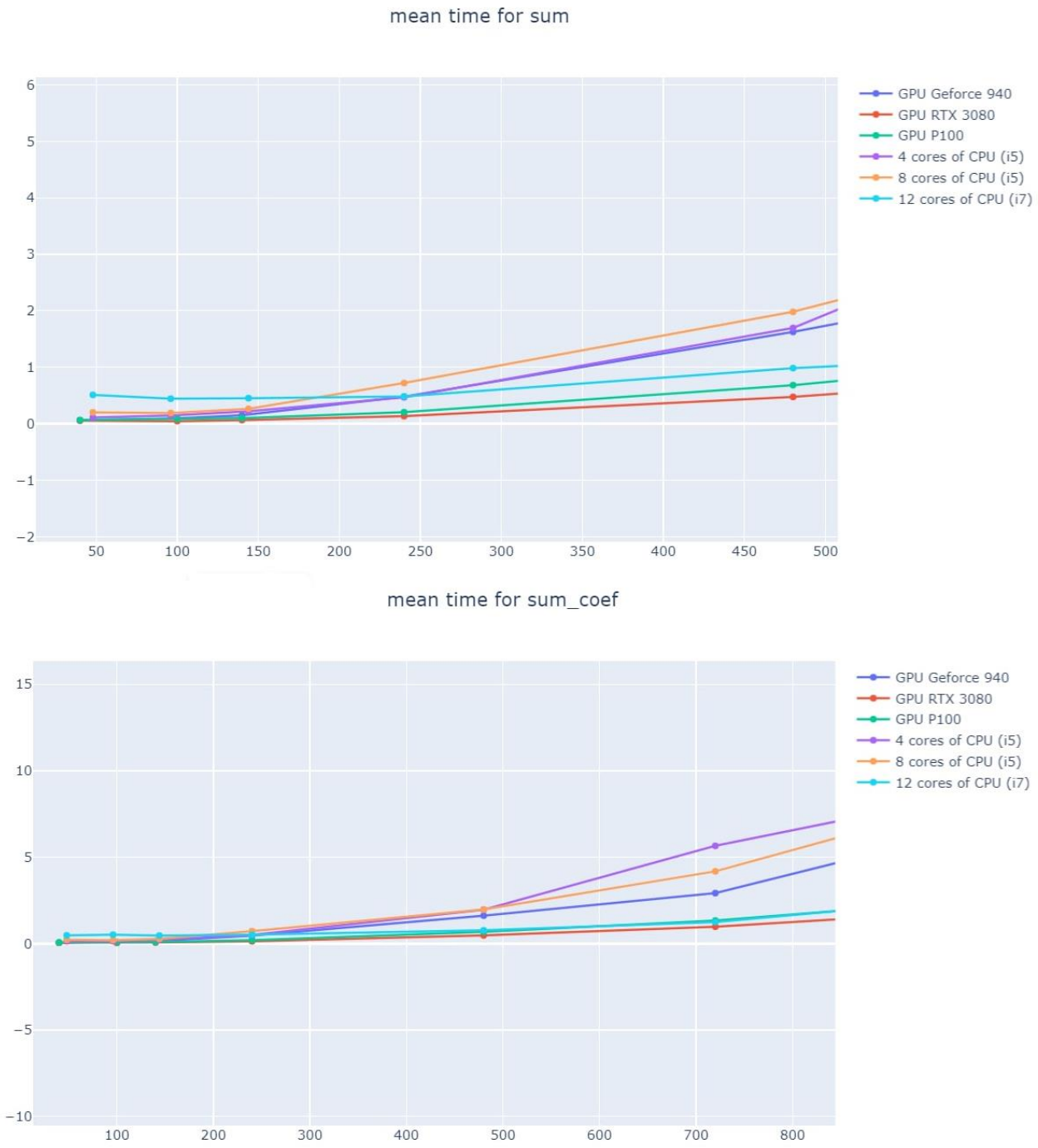

Рис. 4.38 – Результати додавань

На графіку на рис. 4.38 представлені 2 практично тотожні задачі. Можна побачити, що спочатку 12-потоковий центральний процесор на малих розмірностях працює довше за інші пристрої, та в районі 200 елементів починає

перевершувати інші пристрої. Далі картина на графіку в більшому масштабі загалом стала.

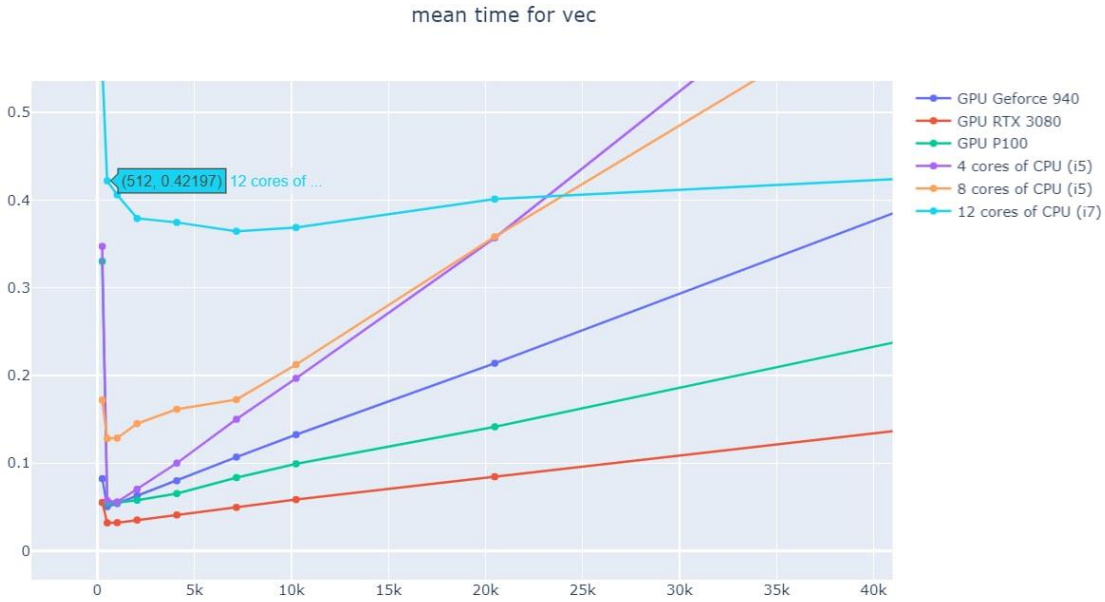

Рис. 4.39 – Результати скалярного добутку векторів у ближчому масштабі

Для скалярного добутку характерно, що на старті на малому розмірі (256 елементів) спостерігається скачок у часі виконання, особливо на центральних процесорах. Починаючи з 512 елементів час роботи починає стабілізуватись. Починаючи з 50 000 елементів Intel i7 починає переважати слабший GPU, а з приблизно 130 000 елементів – переважати Tesla P100.

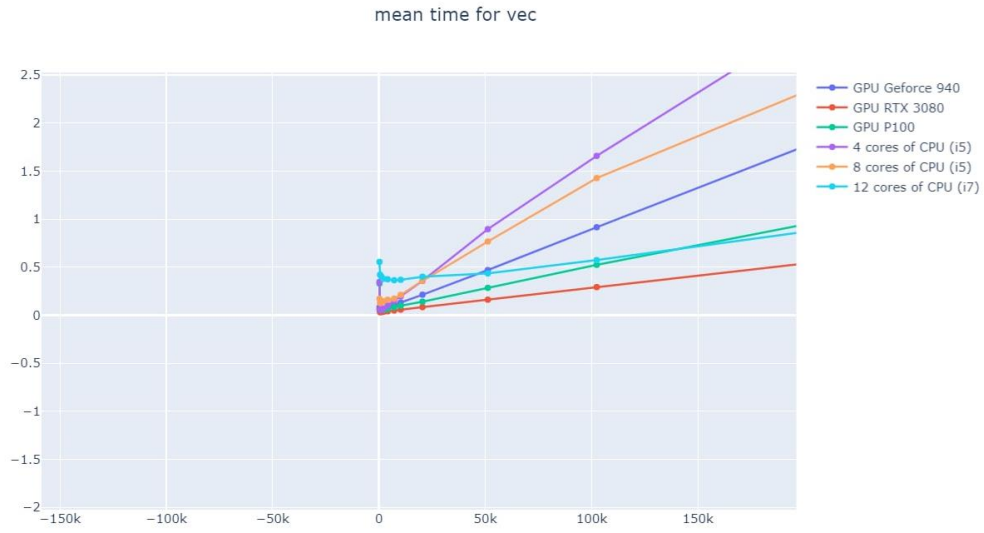

Рис. 4.40 – Результати скалярного добутку векторів у ближчому масштабі (2)

4.4 Висновки до розділу 4

У даному розділі були проведені тести із різними центральними та графічними процесорами. У ході роботи було встановлено, що в залежності від задачі та характеристик пристрою використання GPU не завжди є ефективніше для усіх SIMD-задач. Визначну роль грають такі фактори як швидкість передачі даних між графічним процесором та хост-машиною, а також складність (ресурсозатратність) самих досліджуваних алгоритмів. Швидкість передачі даних значною мірою залежить від сучасності робочого обладнання, саме тому RTX 3080 стабільно показувала найкращі результати. Для випадків із старішим пристроєм (GeForce 940) вдалось отримати більшу ефективність у випадку зі збільшенням операцій у потоках, що компенсувало трансфери.

РОЗДІЛ 5. ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

В даному розділі проведена оцінка основних характеристик програмного коду, розробленого для порівняння ефективностей CPU і GPU на прикладі різних задач.

Програмний продукт призначено для використання на будь-яких системах, що мають графічний процесор NVIDIA з підтримкою CUDA, та підтримуютьC++.

Нижче наведено аналіз різних варіантів реалізації модулю з метою вибору оптимальної стратегії створення програмного продукту. При цьому будуть враховуватись як економічні фактори, так і характеристики продукту, що впливають на продуктивність роботи і на його сумісність з апаратним забезпеченням. Для цього було використано апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) – це технологія для оцінки реальної вартості продукту або послуги. Ця оцінка не залежить від організаційної структури компанії. Прямі та побічні витрати розподіляються по продуктам та послугам у залежності від потрібних обсягів ресурсів на кожному етапі виробництва. Виконані на цих етапах дії у контексті метода ФВА називаються функціями.

Мета ФВА полягає у забезпеченні найбільш оптимального розподілу ресурсів, які виділені на виробництво продукції або надання послуг, на прямі та непрямі витрати. У даному випадку проводиться аналіз функцій програмного продукту й виявлення усіх витрат на реалізацію цих функцій.

Алгоритм ФВА є наступним:

1) визначення послідовності функцій, необхідних для виробництва продукту. Спочатку йдуть всі можливі функції, які розподіляються по двом групам: ті, що впливають на вартість продукту і ті, що не впливають. Також на цьому етапі оптимізується сама послідовність скороченням кроків, що не впливають на витрати;

2) для кожної функції визначається повний обсяг річних витрат та кількість робочих годин.
3) на основі оцінок з п.2 визначається кількісна характеристика джерел витрат.
4) після визначення джерел витрат проводиться кінцевий розрахунок витрат на виробництво продукту.

5.1. Постановка задачі

Метод ФВА був застосований для проведення техніко-економічний аналізу розробки системи для порівняння ефективностей центральних і графічних процесорів. . Оскільки основні проектні рішення стосуються всієї системи, кожна окрема підсистема має їм задовольняти. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, призначеного обробки та фільтрування даних.

До продукту були визначені наступні технічні вимоги:

1) можливість виконання на персональних комп’ютерах із графічними картами;
2) висока точність отримуваних результатів;
3) зручність та простота взаємодії;
4) можливість зручного налаштування, масштабування;
5) мінімальні витрати на впровадження програмного продукту.

5.1.1 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка програмного продукту, який вимірює ефективності роботи грфічних та центральних процесорів. Виходячи з конкретної мети, можна виділити наступні основні функції програмного продукту:

F_1 – вибір мови програмування;

F_2 – вибір інструменту для розрахунків загального призначення на відеокарті;

F_3 – вибір середовища розробки.

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1:

1) C
2) C++
3) C#

Функція F_2:

1) CUDA
2) OpenGL

Функція F_3:

1) Visual Studio
2) Текстовий редактор та окремий консольний компілятор

5.1.2 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті системи на рисунку 5.1. Морфологічна карта відображує всі можливі комбінації варіантів реалізації функцій, які складають повну множину варіантів ПП.

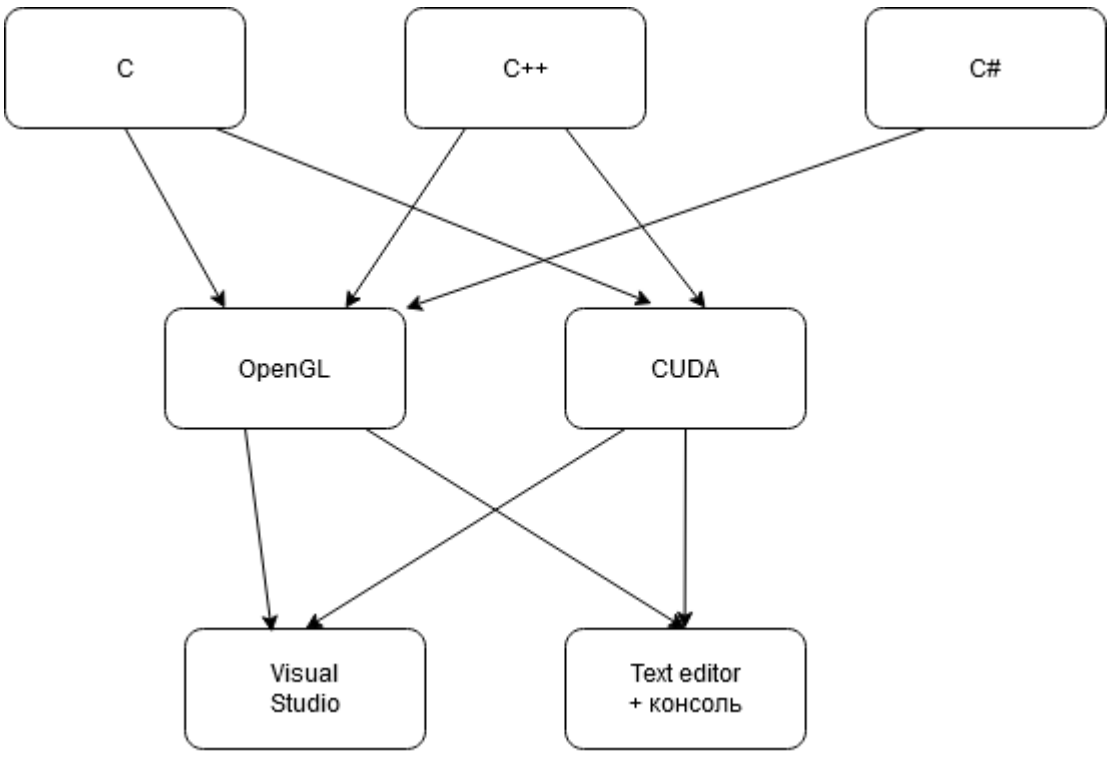

Рисунок 5.1 – Морфологічна карта варіантів реалізації функцій

Така карта дозволила побудувати позитивно-негативну матрицю варіантів основних функцій (табл. 5.1).

Основні функції	**Варіанти реалізації**	**Переваги**	**Недоліки**
F1	*А*	Висока швидкість роботи	Немає платформонезалежних інструментів для роботи з потоками
	Б	Є платформонезалежні інструменти та висока швидкість роботи	Відносно вища складність мови в порівнянні з іншими варіантами
	В	Є платформонезалежні інструменти	Нижча швидкість роботи
F2	А	Більша свобода з обрахунками для розробника	Підтримується лише відеокартами NVIDIA
	Б	Розроблена виключно для графічних розрахунків	Не залежна від виробника відеокарт
F3	*А*	Широкий функціонал	Вимагає багато ресурсів
	Б	Не вимагає багатьох ресурсів	Необхідність встановлювати і працювати з компілятором окремо

Таблиця 5.1 – Позитивно-негативна матриця

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути,

тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція *F1:*

Оскільки важливою характеристикою є швидкодія коду, варіант в) має бути відкинутий. Між варіантами а) і б) перевага на боці того, який дозволить зручніше працювати з потоками, тому варіант а) має бути відкинутий.

Функція *F2:*

Оскільки обхід обмежень OpenGL щодо виключно графічних розрахунків вимагає багато витрат часу від розробника, а відеокарти NVIDIA представлені в хмарних сервісах, то варіант б) має бути відкинутий.

Функція *F3:*

Використання тої чи іншої IDE не грає вирішальної ролі для кінцевого продукту, тому вважаємо варіанти а) та б) гідними розгляду.

Таким чином, будемо розглядати такі варіанти реалізації ПП:

1. F1б – F2а – F3а

2. F1б – F2а – F3б

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

5.2 Обґрунтування системи параметрів програмного продукту

5.2.1 Опис параметрів

На підставі даних про основні функції, що повинен реалізувати програмний продукт, вимог до нього, визначаються основні параметри виробу, що будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

– *X1* – швидкодія мови програмування;

– *X2* – об’єм пам’яті для збереження даних;

– *X3* – потенційний об’єм програмного коду

X1: Відображає швидкодію операцій залежно від обраної мови програмування.

X2: Відображає об’єм пам’яті в оперативній пам’яті персонального комп’ютера та графічної карти, необхідний для збереження та обробки даних під час виконання програми.

X3: Відображає час, який витрачається на виконання операцій.

5.2.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію програмного продукту як показано у таблиці 5.2.

Назва Параметра	**Умовні позначення**	**Одиниці виміру**	**Значення параметра**		
			гірші	**середні**	**кращі**
Швидкодія мови програмування	X1	Оп/мс	600	750	950
Об’єм пам’яті для збереження даних	X2	Мб	200	1024	2048
Потенційний об’єм програмного коду	X3	кількість рядків коду	3000	2000	1000

Таблиця 5.2 – Основні параметри ПП

За даними таблиці 5.2 будуються графічні характеристики параметрів (див. рис. 5.2–5.4).

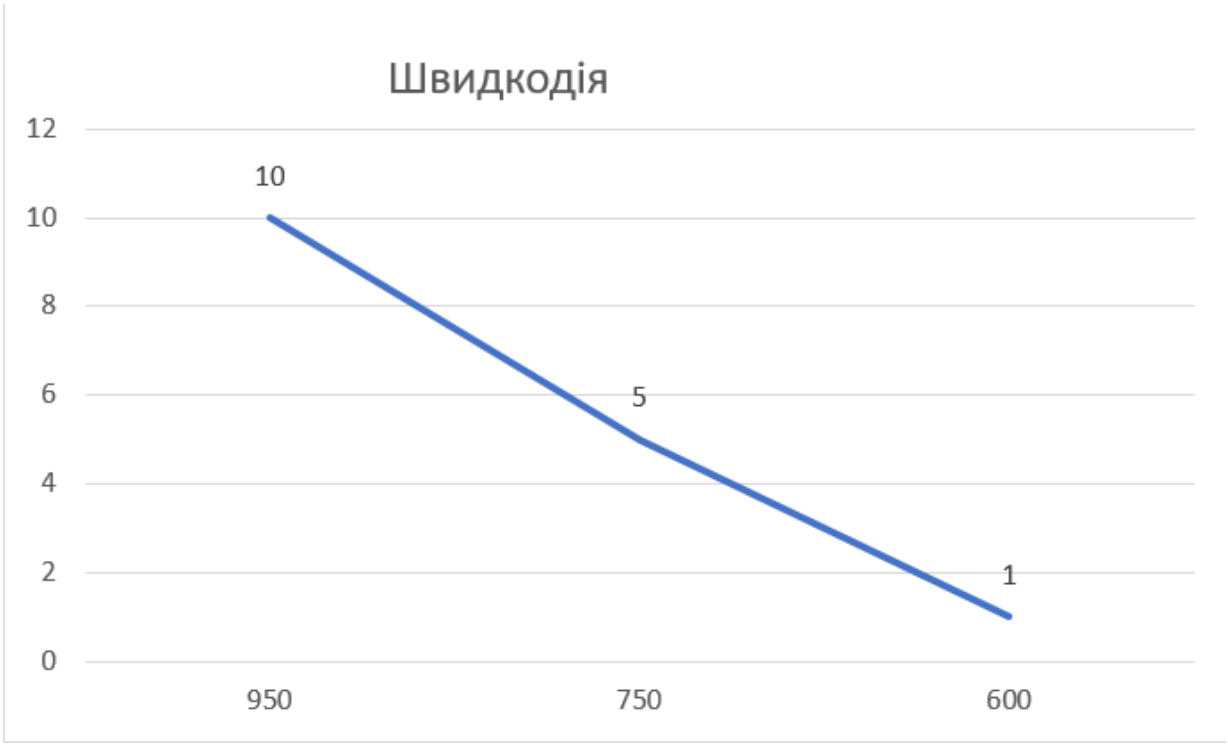

Рисунок 5.2 – Швидкодія мови програмування

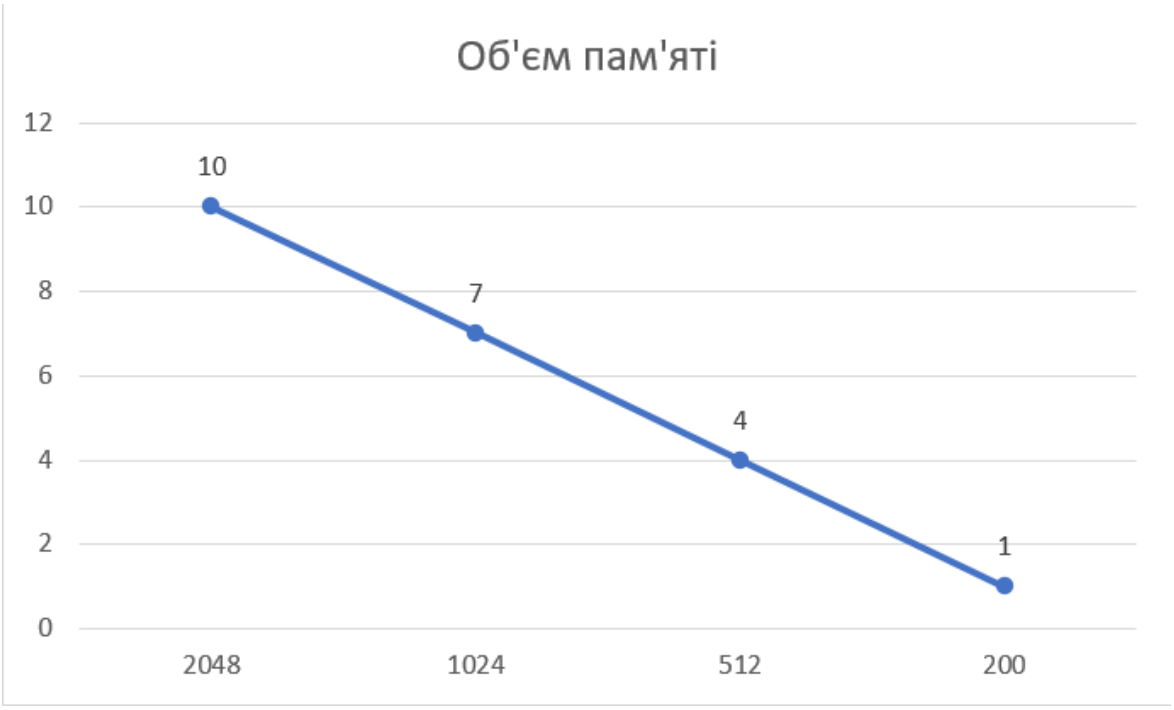

Рисунок 5.3 – Об’єм пам’яті для збереження даних

Рисунок 5.4 – Потенційний об’єм програмного коду

5.2.3 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який має найбільш зручний інтерфейс та зрозумілу взаємодію з користувачем

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

1) визначення рівня значимості параметра шляхом присвоєння різних рангів;
2) перевірку придатності експертних оцінок для подальшого використання;
3) визначення оцінки попарного пріоритету параметрів;
4) обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 5.3

ID	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія мови програмування	Оп/мс	1	2	1	1	1	1	1	8	-6	36
X2	Об'єм пам'яті для збереження даних	Мб	2	1	3	2	2	2	2	14	0	0
X3	Потенційний об'єм програмного коду	К-сть коду	3	3	2	3	3	3	3	20	6	36
	Разом		6	6	6	6	6	6	6	42	0	72

Таблиця 5.3 – Результати ранжування параметрів

Задля перевірки ступеню достовірності експертних оцінок, визначимо наступні параметри, наведені нижче:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} * r_{ij} = 42,$$

де r_{ij} – ранг *i*-го параметра, визначений *j*-м експертом; *N* – число експертів.

б) середня сума рангів *T*:

$$T = \frac{1}{n} R_i = 14.$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T.$$

г) загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{n} * \Delta_i^2 = 72.$$

д) коефіцієнт узгодженості (конкордації):

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 72}{7^2(3^3 - 3)} = 0{,}735 > W_k = 0{,}67.$$

Ранжування вважаємо правдивим, адже знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скористаємось результатами ранжирування та проведемо попарне порівняння всіх параметрів. Числове значення для визначення ступеня переваги *i*-го параметра над *j*-тим, a_{ij} визначається за формулою:

$$a_{ij} = \{1{,}5 \;\; x_i > x_j;\; 1{,}0 \;\; x_i = x_j;\; 0{,}5 \;\; x_i < x_j.$$

Результати занесемо у таблицю 5.4.

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	>	<	>	>	>	>	>	>	1,5
X1 і X3	>	>	>	>	>	>	>	>	1,5
X2 і X3	>	>	<	>	>	>	>	>	1,5

Таблиця 5.4 – Попарне порівняння параметрів

З отриманих числових оцінок переваги складемо матрицю A=$\| \; a_{ij} \; \|$.Для кожного параметра зробимо розрахунок вагомості $K_{ві}$ за наступними формулами: $K_{\text{ві}} = \frac{b_i}{\sum_{i=1}^{n} b_i}$, де $b_i = \sum_{i=1}^{N} a_{ij}$, а a_{ij} –коефіцієнт переваги *i*-го на *j*-тим параметром.

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступною формулою:

$$K_{B_i} = \frac{b_i'}{\sum_{i=1}^{n} * \, b_i'},$$

де $b_i' = \sum_{j=1}^{N} * \, a_{ij} b_j.$

Як видно з таблиці 5.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Параметриx_i	Параметриx_j			Перша ітер.		Друга ітер.	
	X1	X2	X3	b_i	$K_{\text{вi}}$	b_i^1	$K_{\text{вi}}^1$
X1	1,0	1,5	1,5	4	0.444	11.5	0.46
X2	0,5	1,0	1,5	3	0.333	8	0.32
X3	0,5	0,5	1,0	2	0.222	5.5	0.22
Всього:				9	1	25	1

Таблиця 5.5 – Розрахунок вагомості параметрів

5.3 Аналіз рівня якості варіантів реалізації функцій

Рівень якості кожного варіанту виконання основних функцій визначається окремо.

Абсолютне значення параметру X1 (швидкодія мови програмування) відповідають технічним вимогам умов функціонування даного програмного продукту

Абсолютне значення параметру X2 (об’єм пам’яті для збереження даних) було обрано не найгіршим, тобто б) 1024 Мб або в) 2048 Мб.

Абсолютне значення параметра X3 (кількість рядків коду) обрано не найгіршим, тобто це значення відповідає варіанту б) 2000 або в) 1000 рядків коду.

Коефіцієнт технічного рівня якості для кожного варіанта реалізації ПП розраховується за формулою:

$$K_{TP} = \sum_{i=1}^{n} * K_{B_i} B_i,$$

де n – кількість параметрів, K_{B_i} – коефіцієнт вагомості і–го параметра, B_i – оцінка i–го параметра в балах.

Розрахунок показників рівня якості представлено відповідно в таблиці 5.6.

Основні функції	Варіант реалізації функції	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1(X1)	А	950	10	0.46	4.6
F2(X2)	А	1024	5	0.32	1.6
	Б	2048	10	0.32	3.2
F3(X3)	А	2000	10	0.22	2.2
	Б	1000	5	0.22	1.1

Таблиця 5.6 – Розрахунок показників рівня якості варіантів реалізації

За даними з таблиці 5.6 за формулою.

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}],$$

визначаємо рівень якості кожного з варіантів:

$$\boldsymbol{K_{K1}} = 4.6 + 1.6 + 2.2 = 8.4$$

$$\boldsymbol{K_{K2}} = 4.6 + 1.6 + 1.1 = 7.3$$

$$\boldsymbol{K_{K3}} = 4.6 + 3.2 + 2.2 = 10$$

$$\boldsymbol{K_{K4}} = 4.6 + 3.2 + 1.1 = 8.9$$

Як видно з розрахунків, кращим є варіант 3. Для цього варіанту коефіцієнт технічного рівня має найбільше значення.

5.4 Економічний аналіз варіантів розробки програмного продукту

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти містять два окремих завдання:

1. написання коду тестів для CPU;
2. написання коду тестів для GPU;

Завдання 1 та 2 за ступенем новизни належати до групи В.

За складністю алгоритми, які використовуються в завданні 1 та 2 належать до групи 3.

Для реалізації завдання 1 та 2 використовуються згенеровані випадковим чином дані.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Загальна трудомісткість обчислюється за формулою.

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_М \cdot K_{СТ} \cdot K_{СТ.М}$$

де T_P – трудомісткість розробки програмного продукту;

$K_П$ – поправочний коефіцієнт;

$K_{СК}$ – коефіцієнт на складність вхідної інформації;

$K_М$ – коефіцієнт рівня мови програмування;

$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення.

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни В та групи складності алгоритму 3, трудомісткість дорівнює: $T_P = 12$ людино-днів. Поправочний коефіцієнт, який враховує вид вхідної інформації для першого завдання: $K_П = 1$ (інформація є ПІ, а складність алгоритму 3). Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації рівний $K_{СК} = 1$. Оскільки при розробці першого завдання не використовуються стандартні модулі, то $K_{СТ} = 1$. Також робимо поправку на мови низького рівня $K_М = 1.15$. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 12 \cdot 1 \cdot 1.15 = 13.8 \text{ людино-днів}$$

Проведемо аналогічні розрахунки для другого завдання, в якому усі отримані коефіцієнти та табличні людино-дні однакові:

$$T_2 = 12 \cdot 1 \cdot 1.15 = 13.8 \text{ людино-днів}$$

Об'єднаємо результати в одну групу, адже загальна трудомісткість усіх варіантів реалізації збігаються. Таким чином, загальна трудомісткість складає:

$$\mathrm{T}_0 = (13.8 + 13.8) \cdot 8 = 220{,}8 \text{ людино-годин;}$$

В розробці приймає участь програміст з окладом 30000 грн. Визначимо середню зарплату за годину за формулою:

$$\mathrm{C}_\mathrm{Ч} = \frac{M}{T_m \cdot t} \text{ грн.,}$$

де M – місячний оклад працівників;

T_m – кількість робочих днів на місяць;

t – кількість робочих годин в день.

$$\mathrm{C}_\mathrm{Ч} = \frac{30000}{20 \cdot 8} = 187.5 \text{ грн.}$$

Таким чином, заробітна плата отримується за формулою

$$\mathrm{C}_\mathrm{ЗП} = \mathrm{C}_\mathrm{Ч} \cdot T_i \cdot \mathrm{К}_\mathrm{Д},$$

де $\mathrm{C}_\mathrm{Ч}$ – величина погодинної оплати праці програміста;

T_i – трудомісткість відповідного завдання;

$\mathrm{К}_\mathrm{Д}$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробника становить:

$$\mathrm{C}_\mathrm{ЗП} = 187.5 \cdot 220.8 \cdot 1{,}2 = 49\,680 \text{ грн}$$

Відрахування на соціальний внесок становить 22%:

$$\mathrm{C}_\mathrm{СВ} = \mathrm{C}_\mathrm{ЗП} \cdot 0{,}22 = 49.680 \cdot 0{,}22 = 10929.6 \text{ грн.}$$

Тепер визначимо витрати на оплату однієї машино-години. Оскільки одна ЕОМ обслуговується одним інженером апаратного забезпечення з окладом 17000 грн. та коефіцієнтом зайнятості $\mathrm{К}_З = 0{,}2$ то для однієї машини отримаємо:

$$\mathrm{C}_\mathrm{Г} = 12 \cdot \mathrm{М} \cdot \mathrm{К}_З = 12 \cdot 17\,000 \cdot 0{,}2 = 40\,800 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$\mathrm{C}_\mathrm{ЗП} = \mathrm{C}_\mathrm{Г} \cdot (1 + \mathrm{К}_З) = 40\,800 \cdot (1 + 0{,}2) = 48960 \text{ грн.}$$

Відрахування на соціальний внесок становить 22%:

$$С_{СВ} = С_{ЗП} \cdot 0{,}22 = 48960 \cdot 0{,}22 = 10\,771.2 \text{ грн.}$$

Амортизаційні відрахування розраховуємо за формулою при амортизації 25% та вартості ЕОМ – 40 000 грн.:

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1{,}15 \cdot 0{,}25 \cdot 40000 = 11\,500 \text{ грн.}$$

де $К_{ТМ}$ – коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

$К_А$ – річна норма амортизації;

$Ц_{ПР}$ – договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо за формулою:

$$С_Р = К_{ТМ} \cdot К_Р \cdot Ц_{ПР} = 1{,}15 \cdot 0{,}05 \cdot 40000 = 2300 \text{ грн.}$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К - Д_В - Д_С - Д_Р) \cdot t \cdot К_В, Т_{ЕФ} = (365 - 104 - 12 - 16) \cdot 8 \cdot 0{,}9 = 1\,667{,}6 \text{ год.}$$

де $Д_К$ – календарна кількість днів у році;

$Д_В, Д_С$ – відповідно кількість вихідних та святкових днів;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t – кількість робочих годин в день;

$К_В$ – коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_C \cdot К_З \cdot Ц_{ЕЛ} = 1\,677{,}6 \cdot 0{,}75 \cdot 0{,}2 \cdot 3{,}6352 = 4573{,}81 \text{ грн.}$$

де N_C – середньо-споживча потужність приладу;

$К_З$ – коефіцієнтом зайнятості приладу;

$Ц_{ЕЛ}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0{,}67 = 40\,000 \cdot 0{,}67 = 26800 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть складати:

$$С_{ЕК} = С_{ЗП} + С_{СВ} + С_{А} + С_{Р} + С_{ЕЛ} + С_{Н}$$
$$= 49\,680 + 10\,771.2 + 11\,500 + 2300 + 4573{,}81 + 26800$$
$$= 105\,625.01 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{МГ} = \frac{С_{ЕК}}{Т_{ЕФ}} = \frac{105\,625.01}{1\,677{,}6} = 62.96 \text{ грн/год.}$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу складають:

$$С_{М} = С_{МГ} \cdot Т = 62.96 \cdot 220.8 = 13901.568 \text{ грн.}$$

Накладні витрати складають 67% від заробітної плати:

$$С_{Н} = С_{ЗП} \cdot 0{,}67 = 49\,680 \cdot 0{,}67 = 33285.6 \text{ грн.}$$

Отже, вартість розробки програмного продукту за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{СВ} + С_{М} + С_{Н} = 49\,680 + 10\,771.2 + 13\,901.568 + 33\,285.6$$
$$= 107\,638.368 \text{ грн.}$$

Розрахуємо коефіцієнти техніко-економічного рівня за формулою:

$$К_{ТЕР1} = \frac{\boldsymbol{K_{K_1}}}{С_{ПП}} = \frac{8.4}{107\,638.368\text{грн}} = 7.8 \cdot 10^{-5}$$

$$К_{ТЕР2} = \frac{\boldsymbol{K_{K_2}}}{С_{ПП}} = \frac{7.3}{107\,638.368\text{грн}} = 6.78 \cdot 10^{-5}$$

$$К_{ТЕР3} = \frac{\boldsymbol{K_{K_3}}}{С_{ПП}} = \frac{10}{107\,638.368\text{грн}} = 9.3 \cdot 10^{-5}$$

$$К_{ТЕР4} = \frac{\boldsymbol{K_{K_4}}}{С_{ПП}} = \frac{8.9}{107\,638.368\text{грн}} = 8.64 \cdot 10^{-5}$$

Як можна спостерігати, найбільш ефективним є третій варіант реалізації програми. Його коефіцієнт техніко-економічного рівня складає $К_{ТЕР3} = 9.3 \cdot 10^{-5}$ [32].

5.5 Висновки

У ході виконання економічного розділу були систематизовані і закріплені теоретичні знання в галузі економіки та організації виробництва використанням їх

для техніко-економічного обґрунтування розробки методом функціонально-вартісного аналізу. ФАВ був застосований до програмного продукту, який реалізувався в рамках даної бакалаврської роботи. Мета даного ПП полягала у замірі ефективностей двох обчислювальних пристроїв

Аналізуючи дані про зміст основних функцій, які повинен реалізувати програмний продукт, та мету, з якою вони використовуються, були визначені чотири найбільш оптимальні та перспективні варіанти створення продукту. Як показали обрахунки, найбільш ефективним виявився третій варіант реалізації функцій даного програмного продукту, тобто був віднайдений коефіцієнт техніко-економічного рівня, який має максимальну величину. Вартість витрат для нього становить $C_{ПП} = 107\,638.368$ грн.

Цей варіант передбачає:

- використання мови програмування C++;
- використання платформи CUDA C на відеокартах NVIDIA;
- бенчмарки часу роботи кожного пристрою на різних задачах.

ВИСНОВКИ

Метою даної дипломної роботи було провести дослідження, аби визначити, за яких розмірів даних оптимально використовувати GPU для визначених SIMD-задач (скалярний добуток векторів, множення матриць, сума матриць з коефіцієнтом та без, ділення матриці на число). Для досліду використовувались процесори Intel i5 та Intel i7, а тести на платформі CUDA запускались на GPU NVIDIA GeForce 940, Tesla P100 та RTX 3080. Результати показали, що стабільно краще за всіх працювала відеокарта RTX 3080. З іншого боку, було помічено, що не для всіх задач GPU працювали краще за CPU.

Для задачі множення матриць всі GPU відпрацьовували краще за всі CPU. Це пояснювалось тим, що вбудована 2D-індексація потоків на платформі CUDA дозволяє зменшити складність алгоритму множення матриць всередині потоку з $O(n^3)$ до $O(n)$.

Для решти задач були випадки, коли GPU відпрацьовували гірше за CPU, наприклад, 12-потоковий Intel i7 був стабільно другим за показниками після RTX 3080, обходячи Tesla P100 та GeForce 940. У 3 випадках з 5 GeForce 940 працював гірше за 4-потоковий Intel i5, показуючи найдовший час роботи. Розглянувши детально час трансефрів даних між хостом та пристроєм, а також чистий час роботи над даними, було встановлено, що велику частку часу для відеокарти з повільнішим результатом складає саме переміщення пам'яті. Вартість переміщення пам'яті не була компенсована самими задачами, які виконувало GPU.

Для того, аби протестувати цю тезу, до задач додавання і ділення матриці на число була внесена зміна, а саме базова операція додавання або ділення виконувалась всередині функції потоку не один раз, а два, п'ять та десять. Після цього тесту були помітні зміни у ефективності роботи пристроїв: на випадку з 2 і 5 потоками P100 показав кращий результат за Intel i7, а GeForce 940 впорався краще за Intel i5. При цьому i7 все одно виконав задачу за менше часу, аніж Intel i5. У випадку з 10 потоками ситуація змінилась таким чином, що всі GPU стали швидшими за всі CPU.

Таким чином, питання оптимальної розмірності даних для роботи із GPU можна назвати другорядним. Ключовими аспектами ефективної роботи графічного процесора є швидкість передачі даних з та до відеокарти (що напряму залежить від новизни та вдосконалень пристроїв) та складності виконуваної задачі, яка має бути достатньо, аби компенсувати трансфери пам’яті. Чистий час обробки на відеокарті на порядки менший за час обробки на CPU, тому збільшення складності операцій не настільки критично впливає на час роботи GPU, на відміну від центрального процесора. Підсумовуючи, новіші GPU завдяки покращеним апаратним можливостям передачі даних з великою ймовірністю будуть обходити за результативністю багато центральних процесорів. Для випадків, коли графічний процесор показує гірші результати через трансефри пам’яті, починає грати визначну роль аспект так званого “tradeoff”, який залежить від складності виконуваної задачі. Таким чином, питання оптимального розміру даних для GPU є похідним від цих двох ключових моментів, і для кожної задачі і комбінації порівнюваних пристроїв відповідь варіюється.

ДЖЕРЕЛА

1. The Story of the Intel® 4004 [Електронний ресурс] // Intel. – Режим доступу: https://www.intel.co.uk/content/www/uk/en/history/museum-story-of-intel-4004.html (дата звернення: 06.06.2022). – Назва з екрана.

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. [Електронний ресурс] / 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 (дата звернення: 06.06.2022). – Назва з екрана.

4. Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions [Електронний ресурс] / 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 (дата звернення: 06.06.2022). – Назва з екрана.

5. Contributors to Wikimedia projects. Transistor count - Wikipedia [Електронний ресурс] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. – Режим доступу: https://en.wikipedia.org/wiki/Transistor_count#Microprocessors (дата звернення: 06.06.2022). – Назва з екрана.

6. CPU DB - Looking At 40 Years of Processor Improvements | A complete database of processors for researchers and hobbyists alike. [Електронний ресурс] // cpudb.stanford.edu. – Режим доступу: http://cpudb.stanford.edu/visualize/clock_frequency (дата звернення: 06.06.2022). – Назва з екрана.

7. Bohr M. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper [Електронний ресурс] / 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 (дата звернення: 06.06.2022). – Назва з екрана.

8. Contributors to Wikimedia projects. Multi-core processor - Wikipedia [Електронний ресурс] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. – Режим доступу: https://en.wikipedia.org/wiki/Multi-core_processor (дата звернення: 06.06.2022). – Назва з екрана.

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 (дата звернення: 06.06.2022). – Назва з екрана.

12. Contributors to Wikimedia projects. Non-uniform memory access - Wikipedia [Електронний ресурс] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. – Режим доступу: https://en.wikipedia.org/wiki/Non-uniform_memory_access (дата звернення: 06.06.2022). – Назва з екрана.

13. Neelap A. K. Master's thesis in Electrical Engineering. Performance analysis of GPGPU and CPU On AES Encryption [Електронний ресурс] / Akash Kiran Neelap. – Режим доступу: https://www.diva-portal.org/smash/get/diva2:831353/FULLTEXT01.pdf (дата звернення: 06.06.2022). – Назва з екрана.

14. A communication-aware solution framework for mapping AUTOSAR runnables on multi-core systems [Електронний ресурс] / 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. – Назва з екрана.

15. Singer G. History of the Modern Graphics Processor, Part 2 [Електронний ресурс] / Graham Singer // TechSpot. – Режим доступу: https://www.techspot.com/article/653-history-of-the-gpu-part-2 (дата звернення: 06.06.2022). – Назва з екрана.

16. Samel B. GPU Computing and Its Applications [Електронний ресурс] / 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. – Назва з екрана.

18. Syberfeldt A. A Comparative Evaluation of the GPU vs The CPU for Parallelization of Evolutionary Algorithms Through Multiple Independent Runs [Електронний ресурс] / 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_Multiple_Independent_Runs. – Назва з екрана.

19. Benchmarking Data and Compute Intensive Applications on Modern CPU and GPU Architectures [Електронний ресурс] / 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 (дата звернення: 06.06.2022). – Назва з екрана.

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

21. File:Barplot language speeds (Benchmarks Game Mandelbrot).svg - Wikimedia Commons [Електронний ресурс] // Wikimedia Commons. – Режим доступу: https://commons.wikimedia.org/wiki/File:Barplot_language_speeds_(Benchmarks_Game_Mandelbrot).svg (дата звернення: 06.06.2022). – Назва з екрана.

22. Fourment M. A comparison of common programming languages used in bioinformatics [Електронний ресурс] / Mathieu Fourment, Michael R. Gillings // BMC Bioinformatics. – 2008. – Т. 9, № 1. – Режим доступу: https://doi.org/10.1186/1471-2105-9-82 (дата звернення: 06.06.2022). – Назва з екрана.

23. What is a Thread in OS and what are the differences between a Process and a Thread? [Електронний ресурс] // 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 (дата звернення: 06.06.2022). – Назва з екрана.

24. Parallel Computing for Data Science (EN 600.420/620) [Електронний ресурс] // http://parallel.cs.jhu.edu – Режим доступу: http://parallel.cs.jhu.edu/assets/slides/lec17.1.gpu.pdf (дата звернення: 06.06.2022). – Назва з екрана.

25. CUDA Refresher: The CUDA Programming Model. [Електронний ресурс] // Edge AI and Vision Alliance – Режим доступу: https://www.edge-ai-vision.com/2020/08/cuda-refresher-the-cuda-programming-model/ (дата звернення: 06.06.2022). – Назва з екрана.

26. Product Specifications [Електронний ресурс] // Intel® Product Specifications. – Режим доступу: https://ark.intel.com/content/www/us/en/ark/products/201897/intel-core-i710850h-processor-12m-cache-up-to-5-10-ghz.html (дата звернення: 06.06.2022). – Назва з екрана.

27. Product Specifications [Електронний ресурс] // Intel® Product Specifications. – Режим доступу: https://ark.intel.com/content/www/us/en/ark/products/95443/intel-core-i57200u-processor-3m-cache-up-to-3-10-ghz.html (дата звернення: 06.06.2022). – Назва з екрана.

28. NVIDIA GeForce 940MX (GDDR5) specs - GPUZoo [Електронний ресурс] // https://www.gpuzoo.com/. – Режим доступу: https://www.gpuzoo.com/GPU-NVIDIA/GeForce_940MX_GDDR5.html. – Назва з екрана.

29. NVIDIA Tesla P100 16 GB specs - GPUZoo [Електронний ресурс] // https://www.gpuzoo.com. – Режим доступу: https://www.gpuzoo.com/GPU-NVIDIA/Tesla_P100_16_GB.html. – Назва з екрана.

30. ASUS TUF Gaming GeForce RTX 3080 Ti OC Edition 12GB GDDR6X | Graphics Card - Tech Specs [Електронний ресурс] // ASUS Deutschland. – Режим доступу: https://www.asus.com/us/Motherboards-Components/Graphics-Cards/TUF-

Gaming/TUF-RTX3080TI-O12G-GAMING/techspec/ (дата звернення: 06.06.2022). – Назва з екрана.

31. Addition and Subtraction of Matrix using pthreads - GeeksforGeeks. [Електронний ресурс] // GeeksforGeeks – Режим доступу: https://www.geeksforgeeks.org/addition-subtraction-matrix-using-pthreads/ (дата звернення: 06.06.2022). – Назва з екрана.

32. Пашін В. П. Методичні вказівки до виконання економіко-організаційного розділу дипломних проектів (робіт) бакалаврів і спеціалістів для студентів інституту прикладного системного аналізу / В. П. Пашін, В. В. Романов, Н. В. Єгорова. // НТУУ “КПІ”. – 2011. – С. 118.

ДОДАТОК А. ПРОГРАМНИЙ КОД ПРОДУКТУ

1. Бенчмарк на CPU

```
#include <iostream>
#include <thread>
#include <chrono>
#include <vector>
#include <atomic>
#include <random>
#include <ctime>
#include <fstream>
#define LOW -1
#define HIGH 1

using std::chrono::duration;
using std::chrono::duration_cast;
using std::chrono::high_resolution_clock;
using std::chrono::milliseconds;

template <class T>
class Matrix
{
public:
   Matrix(int n, bool fill) : dim(n)
   {
      m_matrix = new T[dim * dim];
      for (int k = 0; k < dim * dim; ++k)
         if (fill)
            m_matrix[k] = (T)(rand() % 10);
         else
            m_matrix[k] = 0;
   }
   ~Matrix()
   {
      delete[] m_matrix;
   }
   void print()
   {
      for (int i = 0; i < dim; ++i)
      {
         for (int k = 0; k < dim; ++k)
         {
            std::cout << m_matrix[i * dim + k] << "\t";
         }
         std::cout << "\n";
      }
      std::cout << "\n";
   }

   void set(int i, int k, T value)
   {
      m_matrix[i * dim + k] = value;
   }

   T get(int i, int k)
   {
      if (i < dim && k < dim)
         return m_matrix[i * dim + k];
      else
         return 0;
   }
```

```
    void clear()
    {
        for (int k = 0; k < dim * dim; ++k)
            m_matrix[k] = 0;
    }

private:
    Matrix();
    int dim;
    T *m_matrix;
};

void divide_matrix(int start, int end, int dim, Matrix<float> &m1, int d)
{
    for (int i = start; i < end; ++i)
    {
        for (int k = 0; k < dim; ++k)
        {
            m1.set(i, k, m1.get(i, k) / d);
        }
    }
}

void sum_matrix(int start, int end, int dim, Matrix<int> &m1, Matrix<int> &m2)
{
    for (int i = start; i < end; ++i)
    {
        for (int k = 0; k < dim; ++k)
        {
            m2.set(i, k, m1.get(i, k) + m2.get(i, k));
        }
    }
}

void sum_matrix_coef(int start, int end, int dim, Matrix<int> &m1, Matrix<int> &m2, int coef)
{
    for (int i = start; i < end; ++i)
    {
        for (int k = 0; k < dim; ++k)
        {
            m2.set(i, k, coef * m1.get(i, k) + m2.get(i, k));
        }
    }
}

void multiply_matrix(int start, int end, int dim, Matrix<int> &m1, Matrix<int> &m2, Matrix<int> &m3)
{
    for (int i = start; i < end; ++i)
    {
        for (int k = 0; k < dim; ++k)
        {
            for (int j = 0; j < dim; ++j)
            {
                int interm = m3.get(i, k) + m1.get(i, j) * m2.get(j, k);
                m3.set(i, k, interm);
            }
        }
    }
}

void vec_sum(std::vector<float> *vec1, std::vector<float> *vec2, std::atomic<float> &res, int start, int end)
{
    float interm;
```

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

void vec_product()
{
    std::vector<std::thread> thread_list;
    constexpr int dims[] = {100000, 500000, 2000000, 5000000, 20000000, 40000000};
    constexpr int num_threads[] = {1, 2, 4, 8};
    srand(time(NULL));

    std::ofstream vec_file;
    vec_file.open("output2/new_i5_vec.txt", std::ios::app);
    for (int i = 0; i < sizeof(dims) / sizeof(dims[0]); ++i)
    {
        std::atomic<float> res(0);
        std::vector<float> *vec1 = new std::vector<float>(dims[i]);
        std::vector<float> *vec2 = new std::vector<float>(dims[i]);

        for (int j = 0; j < dims[i]; ++j)
        {
            vec1->at(j) = LOW + static_cast<float>(rand()) * static_cast<float>(HIGH - LOW) / RAND_MAX;
            vec2->at(j) = LOW + static_cast<float>(rand()) * static_cast<float>(HIGH - LOW) / RAND_MAX;
        }

        for (int j = 0; j < sizeof(num_threads) / sizeof(num_threads[0]); ++j)
        {
            auto t1 = high_resolution_clock::now();
            for (int k = 0; k < num_threads[j]; ++k)
            {
                int start = (dims[i] / num_threads[j]) * k;
                int end = (dims[i] / num_threads[j]) * (k + 1);
                thread_list.push_back(std::thread(vec_sum, vec1, vec2, std::ref(res), start, end));
            }
            for (auto &th_i : thread_list)
            {
                th_i.join();
            }
            auto t2 = high_resolution_clock::now();
            duration<double, std::milli> ms_double = t2 - t1;
            std::cout << "Vector product for " << dims[i] << " elements with " << num_threads[j] << " threads is " <<
ms_double.count() << " ms\n\n";
            vec_file << dims[i] <<  " " << ms_double.count() << "  " << num_threads[j] <<"\n";
            thread_list.clear();
        }

        delete vec1;
        delete vec2;
    }
    vec_file.close();
}

template <typename... Types>
void int_matrix_operations(void (*fun_ptr)(int, int, int, Matrix<int> &, Matrix<int> &, Types...), const std::string
&comment)
{
    //constexpr int dims[] = {1000, 2000, 4000, 5000, 8000};
    constexpr int dims[] = {400, 800, 1000, 2000};
    constexpr int num_threads[] = {1, 2, 4, 8};
    std::vector<std::thread> thread_list;
```

```
    std::ofstream target_file;
    if ((void *)fun_ptr == (void *)&sum_matrix)
    {
        target_file.open("output2/cpu_sum.txt", std::ios::app);
    }
    else if ((void *)fun_ptr == (void *)&sum_matrix_coef)
    {
        target_file.open("output2/cpu_coef.txt", std::ios::app);
    }
    else
    {
        target_file.open("output2/cpu_mult.txt", std::ios::app);
    }

    for (int i = 0; i < sizeof(dims) / sizeof(dims[0]); ++i)
    {
        Matrix<int> matrix1(dims[i], true);
        Matrix<int> matrix2(dims[i], true);
        Matrix<int> matrix3(dims[i], false);
        int coef = rand() % 100 + 1;
        for (int j = 0; j < sizeof(num_threads) / sizeof(num_threads[0]); ++j)
        {
            auto t1 = high_resolution_clock::now();
            for (int k = 0; k < num_threads[j]; ++k)
            {
                int start = (dims[i] / num_threads[j]) * k;
                int end = (dims[i] / num_threads[j]) * (k + 1);
                if ((void *)fun_ptr == (void *)&sum_matrix)
                {
                    thread_list.push_back(std::thread(&sum_matrix, start, end, dims[i], std::ref(matrix1), std::ref(matrix2)));
                }

                else if ((void *)fun_ptr == (void *)&sum_matrix_coef)
                {
                    thread_list.push_back(std::thread(&sum_matrix_coef, start, end, dims[i], std::ref(matrix1), std::ref(matrix2),
coef));
                }
                else if ((void *)fun_ptr == (void *)&multiply_matrix)
                {
                    thread_list.push_back(std::thread(&multiply_matrix, start, end, dims[i], std::ref(matrix1), std::ref(matrix2),
std::ref(matrix3)));
                }
            }
            for (auto &th_i : thread_list)
            {
                th_i.join();
            }
            auto t2 = high_resolution_clock::now();
            duration<double, std::milli> ms_double = t2 - t1;
            std::cout << comment << " for " << dims[i] << " elements with " << num_threads[j] << " threads is " <<
ms_double.count() << " ms\n\n";
            target_file << dims[i] <<  " " << ms_double.count() << "  " << num_threads[j] <<"\n";
            matrix3.clear();
            thread_list.clear();
        }
    }
    target_file.close();
}

void float_matrix_operations()
{
    constexpr int dims[] = {1000, 2000, 4000, 5000, 8000};
    constexpr int num_threads[] = {1, 2, 4, 8};
```

```
    std::vector<std::thread> thread_list;
    std::ofstream div_file;
    div_file.open("output/cpu_div.txt", std::ios::app);

    for (int i = 0; i < sizeof(dims) / sizeof(dims[0]); ++i)
    {
        Matrix<float> matrix1(dims[i], true);
        int coef = rand() % 100 + 1;
        for (int j = 0; j < sizeof(num_threads) / sizeof(num_threads[0]); ++j)
        {
            auto t1 = high_resolution_clock::now();

            for (int k = 0; k < num_threads[j]; ++k)
            {
                int start = (dims[i] / num_threads[j]) * k;
                int end = (dims[i] / num_threads[j]) * (k + 1);
                thread_list.push_back(std::thread(&divide_matrix, start, end, dims[i], std::ref(matrix1), coef));
            }

            for (auto &th_i : thread_list)
            {
                th_i.join();
            }
            auto t2 = high_resolution_clock::now();
            duration<double, std::milli> ms_double = t2 - t1;
            std::cout << "Matrix division for " << dims[i] << " elements with " << num_threads[j] << " threads is " << ms_double.count() << " ms\n\n";
            div_file << dims[i] <<  " " << ms_double.count() << "  " << num_threads[j] <<"\n";
            matrix1.clear();
            thread_list.clear();
        }
    }
    div_file.close();
}

int main()
{
    vec_product();
    std::cout << "\n\n";
    int_matrix_operations(&sum_matrix, "Matrix sum");
    std::cout << "\n\n";
    int_matrix_operations(&sum_matrix_coef, "Matrix sum with coef");
    std::cout << "\n\n";
    int_matrix_operations(&multiply_matrix, "Matrix multiplication");
    std::cout << "\n\n";
    float_matrix_operations();

    return 0;
}
```

2. Бенчмарк на GPU

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/time.h>
#define LOW -1
#define HIGH 1
#define MAX_THREADS 1024
#define THREADXY 20

__global__ void vec_product_GPU(float *a, float *b, float *res)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
```

```
    atomicAdd(res, a[index] * b[index]);
}

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

__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;

    a[row * N + col] /= d;
}

__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;

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

__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;

    a[row * N + col] += b[row * N + col];
}

__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;

    a[row * N + col] = a[row * N + col] * coef + b[row * N + col];
}

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");
    }
}

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

   for (int j = 0; j < sizeof(list) / sizeof(list[0]); ++j)
   {

      h_a = (float *)malloc(sizeof(float) * list[j]);
      h_b = (float *)malloc(sizeof(float) * list[j]);
      h_product = (float *)malloc(sizeof(float));

      for (int i = 0; i < list[j]; ++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) * list[j]);
      cudaMalloc((void **)&d_b, sizeof(float) * list[j]);
      cudaMalloc((void **)&d_product, sizeof(float));

      gettimeofday(&t1_htod, 0);
      cudaMemcpy(d_a, h_a, sizeof(float) * list[j], cudaMemcpyHostToDevice);
      cudaMemcpy(d_b, h_b, sizeof(float) * list[j], cudaMemcpyHostToDevice);
      cudaMemcpy(d_product, h_product, sizeof(float), cudaMemcpyHostToDevice);
      gettimeofday(&t2_htod, 0);

      int grids;
      int num_threads_per_block;
      if (list[j] < MAX_THREADS)
      {
         grids = 1;
         num_threads_per_block = list[j];
      }
      else
      {
         grids = list[j] % MAX_THREADS ? list[j] / MAX_THREADS + 1 : list[j] / MAX_THREADS;
         num_threads_per_block = MAX_THREADS;
      }

      gettimeofday(&t1, 0);
      vec_product_GPU<<<grids, num_threads_per_block>>>(d_a, d_b, d_product);
      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;
      printf("Vector product time device to host transfer is %lf for %d elements\n", htod_time, list[j]);
      double time = (1000000.0 * (t2.tv_sec - t1.tv_sec) + t2.tv_usec - t1.tv_usec) / 1000.0;
      printf("Vector product time is %lf for %d elements\n", time, list[j]);
      double dtoh_time = (1000000.0 * (t2_dtoh.tv_sec - t1_dtoh.tv_sec) + t2_dtoh.tv_usec - t1_dtoh.tv_usec) / 1000.0;
      printf("Vector product time host to device transfer is %lf for %d elements\n", dtoh_time, list[j]);
```

```
        double total = htod_time + time + dtoh_time;
        printf("Total vector product time is %lf for %d elements\n\n\n", total, list[j]);

        fprintf(vec_ptr, "%d %lf\n", list[j], total);

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

void matrix_operations(void)
{
    int *h_a, *h_b, *d_a, *d_b, *h_product, *d_product;
    float *h_c, *d_c;
    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, *coef_ptr, *div_ptr, *mult_ptr;

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

    for (int j = 0; j < sizeof(list) / sizeof(list[0]); ++j)
    {
        long m_size = list[j] * list[j];
        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);
        gettimeofday(&htod_2, 0);
        cudaMemcpy(d_product, h_product, sizeof(int) * m_size, cudaMemcpyHostToDevice);
        gettimeofday(&htod_3, 0);

        int thread_dim, block_dim;
        if (list[j] < THREADXY)
        {
            thread_dim = list[j];
            block_dim = 1;
        }
        else
        {
            thread_dim = THREADXY;
            block_dim = list[j] / THREADXY;
        }
```

```
    dim3 threads(thread_dim, thread_dim);
    dim3 blocks(block_dim, block_dim);

    gettimeofday(&t1, 0);
    matrix_mult_GPU<<<threads, blocks>>>(d_a, d_b, d_product, list[j]);
    cudaDeviceSynchronize();
    gettimeofday(&t2, 0);
    matrix_add_GPU<<<threads, blocks>>>(d_a, d_b, list[j]);
    cudaDeviceSynchronize();
    gettimeofday(&t3, 0);
    matrix_add_coef_GPU<<<threads, blocks>>>(d_a, d_b, coef, list[j]);
    cudaDeviceSynchronize();
    gettimeofday(&t4, 0);

    gettimeofday(&dtoh_1, 0);
    cudaMemcpy(h_a, d_a, sizeof(int) * m_size, cudaMemcpyDeviceToHost);
    gettimeofday(&dtoh_2, 0);
    cudaMemcpy(h_product, d_product, sizeof(int) * m_size, cudaMemcpyDeviceToHost);
    gettimeofday(&dtoh_3, 0);

    printf("\n\n");

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

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

    h_c = (float *)malloc(sizeof(float) * m_size);

    for (int i = 0; i < m_size; ++i)
    {
       h_c[i] = (float)(rand() % 10000);
    }

    cudaMalloc(&d_c, m_size * sizeof(float));
    gettimeofday(&htod_4, 0);
    cudaMemcpy(d_c, h_c, sizeof(float) * m_size, cudaMemcpyHostToDevice);
    gettimeofday(&htod_5, 0);
    matrix_divide_GPU<<<threads, blocks>>>(d_c, coef, list[j]);
    cudaDeviceSynchronize();
    gettimeofday(&t5, 0);
    cudaMemcpy(h_c, d_c, sizeof(float) * m_size, cudaMemcpyDeviceToHost);
    gettimeofday(&dtoh_4, 0);
    cudaFree(d_c);
    free(h_c);

    double htod_time_sum = (1000000.0 * (htod_2.tv_sec - htod_1.tv_sec) + htod_2.tv_usec - htod_1.tv_usec) / 1000.0;
    double htod_time_product_delta = (1000000.0 * (htod_3.tv_sec - htod_2.tv_sec) + htod_3.tv_usec - htod_2.tv_usec) /
1000.0;
    double product_time = (1000000.0 * (t2.tv_sec - t1.tv_sec) + t2.tv_usec - t1.tv_usec) / 1000.0;
    double sum_time = (1000000.0 * (t3.tv_sec - t2.tv_sec) + t3.tv_usec - t2.tv_usec) / 1000.0;
    double dtoh_time_sum = (1000000.0 * (dtoh_2.tv_sec - dtoh_1.tv_sec) + dtoh_2.tv_usec - dtoh_1.tv_usec) / 1000.0;
    double dtoh_time_product = (1000000.0 * (dtoh_3.tv_sec - dtoh_2.tv_sec) + dtoh_3.tv_usec - dtoh_2.tv_usec) /
1000.0;
    double sum_coef_time = (1000000.0 * (t4.tv_sec - t3.tv_sec) + t4.tv_usec - t3.tv_usec) / 1000.0;

    double htod_time_divide = (1000000.0 * (htod_5.tv_sec - htod_4.tv_sec) + htod_5.tv_usec - htod_4.tv_usec) / 1000.0;
    double division_time = (1000000.0 * (t5.tv_sec - htod_5.tv_sec) + t5.tv_usec - htod_5.tv_usec) / 1000.0;
    double dtoh_time_divide = (1000000.0 * (dtoh_4.tv_sec - t5.tv_sec) + dtoh_4.tv_usec - t5.tv_usec) / 1000.0;
```

```
        printf("Matrix product time device to host transfer is %lf for %d elements\n", htod_time_sum + htod_time_product_delta, list[j]);
        printf("Matrix product time is %lf for %d elements\n", product_time, list[j]);
        printf("Matrix product time host to device transfer is %lf for %d elements\n", dtoh_time_product, list[j]);
        printf("Total matrix product time is %lf for %d elements\n\n\n", htod_time_sum + htod_time_product_delta + product_time + dtoh_time_product, list[j]);

        printf("Matrix sum time device to host transfer is %lf for %d elements\n", htod_time_sum, list[j]);
        printf("Matrix sum time is %lf for %d elements\n", sum_time, list[j]);
        printf("Matrix sum time host to device transfer is %lf for %d elements\n", dtoh_time_sum, list[j]);
        printf("Total matrix sum time is %lf for %d elements\n\n\n", htod_time_sum + sum_time + dtoh_time_sum, list[j]);

        printf("Matrix sum with coef time device to host transfer is %lf for %d elements\n", htod_time_sum, list[j]);
        printf("Matrix sum with coef time is %lf for %d elements\n", sum_coef_time, list[j]);
        printf("Matrix sum with coef time host to device transfer is %lf for %d elements\n", dtoh_time_sum, list[j]);
        printf("Total matrix sum with coef time is %lf for %d elements\n\n\n", htod_time_sum + sum_coef_time + dtoh_time_sum, list[j]);

        printf("Matrix division time device to host transfer is %lf for %d elements\n", htod_time_divide, list[j]);
        printf("Matrix division time is %lf for %d elements\n", division_time, list[j]);
        printf("Matrix division time host to device transfer is %lf for %d elements\n", dtoh_time_divide, list[j]);
        printf("Total matrix division time is %lf for %d elements\n\n\n", htod_time_divide + division_time + dtoh_time_divide, list[j]);

        fprintf(sum_ptr, "%d %lf\n", list[j], htod_time_sum + sum_time + dtoh_time_sum);
        fprintf(coef_ptr, "%d %lf\n", list[j], htod_time_sum + sum_coef_time + dtoh_time_sum);
        fprintf(div_ptr, "%d %lf\n", list[j], htod_time_divide + division_time + dtoh_time_divide);
        fprintf(mult_ptr, "%d %lf\n", list[j], htod_time_sum + htod_time_product_delta + product_time + dtoh_time_product);
    }

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

int main(void)
{
    srand(time(NULL));
    scalar_product();
    printf("-------------\n");
    matrix_operations();
    printf("-------------\n");

    return 0;
}
```

ДОДАТОК Б.СЕРЕДНІЙ ЧАС РОБОТИ ПО РОЗМІРНОСТЯХ І ПРИСТРОЯХ ДЛЯ ОПЕРАЦІЙ З ДОДАВАННЯМ

sum_coef GPU Geforce 940

	size	time
0	40	0.0595
1	100	0.0969
2	140	0.1499
3	240	0.4677
4	480	1.6225
5	720	2.9228
6	1200	9.5975
7	2400	39.5460
8	3600	90.1109
9	3840	102.6598
10	4080	115.8756
11	4560	144.9391
12	5040	176.8189
13	5688	225.8630
14	6240	271.9195
15	6840	326.6359
16	7440	386.6828
17	8040	451.2775
18	8640	521.1888
19	9064	574.0188

sum GPU Geforce 940

	size	time
0	40	0.0606
1	100	0.0975
2	140	0.1510
3	240	0.4810
4	480	1.6258
5	720	2.9247
6	1200	9.5985
7	2400	39.5466
8	3600	90.1111
9	3840	102.6606
10	4080	115.8761
11	4560	144.9398
12	5040	176.8192
13	5688	225.8630
14	6240	271.9202
15	6840	326.6362
16	7440	386.6835
17	8040	451.2778
18	8640	521.1895
19	9064	574.0193

sum_coef GPU RTX 3080

	size	time
0	40	0.0553
1	100	0.0461
2	140	0.0654
3	240	0.1351
4	480	0.4761
5	720	0.9809
6	1200	2.6088
7	2400	9.6945
8	3600	21.4387
9	3840	24.3510
10	4080	27.4304
11	4560	34.3544
12	5040	41.7064
13	5688	52.9935
14	6240	63.6549
15	6840	76.4560
16	7440	90.8896
17	8040	105.6720
18	8640	121.7376
19	9064	134.6185

sum GPU RTX 3080

	size	time
0	40	0.0563
1	100	0.0463
2	140	0.0659
3	240	0.1350
4	480	0.4758
5	720	0.9810
6	1200	2.6091
7	2400	9.6948
8	3600	21.4386
9	3840	24.3510
10	4080	27.4301
11	4560	34.3544
12	5040	41.7067
13	5688	52.9936
14	6240	63.6551
15	6840	76.4560
16	7440	90.8895
17	8040	105.6722
18	8640	121.7377
19	9064	134.6181

sum_coef GPU P100

	size	time
0	40	0.0683
1	100	0.0871

2	140	0.1004
3	240	0.2040
4	480	0.6829
5	720	1.3388
6	1200	3.4511
7	2400	14.1330
8	3600	32.5528
9	3840	36.4959
10	4080	41.2699
11	4560	51.2695
12	5040	62.4135
13	5688	80.2111
14	6240	96.6345
15	6840	114.6651
16	7440	135.9381
17	8040	158.2460
18	8640	183.1915
19	9064	202.0482

sum GPU P100

	size	time
0	40	0.0676
1	100	0.0882
2	140	0.1009
3	240	0.2045
4	480	0.6842
5	720	1.3391
6	1200	3.4522
7	2400	14.1338
8	3600	32.5533
9	3840	36.4964
10	4080	41.2702
11	4560	51.2701
12	5040	62.4145
13	5688	80.2114
14	6240	96.6348
15	6840	114.6654
16	7440	135.9389
17	8040	158.2463
18	8640	183.1923
19	9064	202.0489

sum 4 cores of CPU (i5)

	size	time
0	48	0.112954
3	96	0.145768
6	144	0.219729
9	240	0.468280
12	480	1.696044
15	720	4.524182
18	1200	10.177246
21	2400	41.308610
24	3600	95.506600
27	3840	99.340265
30	4080	101.095700
33	4560	126.102100
36	5040	154.555000
39	5688	196.713700
42	6240	236.494100
45	6840	284.073700
48	7440	335.341600
51	8040	393.049800
54	8640	452.299600
57	9064	500.120400

sum_coef 4 cores of CPU (i5)

	size	kernel_number	time
0	48		0.141317
3	96		0.145782
6	144		0.226388
9	240		0.499721
12	480		1.967285
15	720		5.656602
18	1200		11.056571
21	2400		43.369530
24	3600		97.406240
27	3840		102.087060
30	4080		106.062500
33	4560		129.997800
36	5040		158.581900
39	5688		204.517900
42	6240		243.642600
45	6840		292.194700
48	7440		345.358700
51	8040		405.561800
54	8640		465.116200
57	9064		512.438400

sum_coef 8 cores of CPU (i5)

	size	time
1	48	0.219769
4	96	0.202161

7	144	0.277091
10	240	0.723031
13	480	1.972658
16	720	4.182278
19	1200	11.556835
22	2400	43.953490
25	3600	98.440560
28	3840	107.404810
31	4080	123.207200
34	4560	165.097600
37	5040	202.339400
40	5688	240.023200
43	6240	282.392600
46	6840	342.717600
49	7440	411.648900
52	8040	452.648500
55	8640	535.789000
58	9064	604.117800

sum 8 cores of CPU (i5)

size	kernel_number	time
1	48	0.202894
4	96	0.191034
7	144	0.265181
10	240	0.720773
13	480	1.981199
16	720	3.762450
19	1200	10.312927
22	2400	41.003580
25	3600	96.783700
28	3840	105.304220
31	4080	114.648200
34	4560	152.335500
37	5040	187.438800
40	5688	233.089900
43	6240	271.299000
46	6840	345.093500
49	7440	386.082800
52	8040	486.566600
55	8640	538.306900
58	9064	620.976600

sum_coef 12 cores of CPU (i7)

size	time
2 48	0.477990
5 96	0.517630
8 144	0.468510
11 240	0.517290
14 480	0.769250
17 720	1.249770
20 1200	3.683820
23 2400	12.770400
26 3600	24.935510
29 3840	28.933693
32 4080	33.731460
35 4560	40.674590
38 5040	52.638390
41 5688	67.728320
44 6240	75.964950
47 6840	89.097750
50 7440	114.183910
53 8040	131.626500
56 8640	151.785400
59 9064	174.949100

sum 12 cores of CPU (i7)

size	time
2 48	0.51093
5 96	0.44401
8 144	0.45281
11 240	0.48362
14 480	0.98504
17 720	1.31458
20 1200	3.58449
23 2400	11.87020
26 3600	24.92016
29 3840	28.09802
32 4080	32.38910
35 4560	41.60727
38 5040	50.05700
41 5688	65.61312
44 6240	74.65165
47 6840	93.54246
50 7440	120.06224
53 8040	134.20430
56 8640	144.81560
59 9064	169.94780