ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Високопродуктивний графічний процесор»

на здобуття освітнього ступеня бакалавра
зі спеціальності 123 – Комп'ютерна інженерія
освітньо-професійної програми «Комп'ютерна інженерія»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Дмитро ОХОТНІКОВ
(підпис)

Виконав: здобувач вищої освіти групи КІД-42
Дмитро ОХОТНІКОВ

Керівник: Олександр ЗУБ

Рецензент: _____
Ім'я, ПРІЗВИЩЕ

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерна інженерія

Ступінь вищої освіти Бакалавра

Спеціальність 123 – Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерної інженерії
_____Наталія ЛАЩЕВСЬКА
«_____» _____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Охотніков Дмитро Григорович

1. Тема кваліфікаційної роботи: «Високопродуктивний графічний процесор»

керівник кваліфікаційної роботи Олександр ЗУБ,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.
2. Строк подання кваліфікаційної роботи «05» червня 2025 р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, стандарти розподілу навантаження в енергомережах, моделі керування енергетичними системами, методики оптимізації та аналізу алгоритмів.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд літератури та аналіз сучасного стану розвитку графічних процесорів у високопродуктивних обчисленнях.
2. Аналіз архітектурних особливостей GPU та моделей паралелізму.
3. Огляд програмних технологій прискорення обчислень на GPU.
4. Узагальнення результатів і розробка рекомендацій щодо доцільності

застосування GPU у задачах високої складності.

5. Перелік ілюстративного матеріалу: *презентація*

Проблематика обмеженої продуктивності CPU та потреба у високопродуктивних обчисленнях.

Огляд архітектури графічних процесорів та їхніх переваг у паралельних обчисленнях.

Вплив обсягу даних та кількості потоків на ефективність GPU.

Результати бенчмарків та візуалізація обчислювального прискорення.

Висновки щодо доцільності використання GPU у високопродуктивних обчисленнях.

6. Дата видачі завдання «24» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз науково-технічної літератури з питань високопродуктивних обчислень та застосування GPU	16.02–23.02.2025	Виконано
2	Вивчення архітектурних особливостей сучасних графічних процесорів	23.02–02.03.2025	Виконано
3	Аналіз програмних технологій для GPU-прискорення (CUDA, OpenCL, GPGPU-бібліотеки)	02.03–09.03.2025	Виконано
4	Розробка експериментальних задач для тестування CPU та GPU	09.03–19.03.2025	Виконано
5	Проведення експериментів, збір результатів, побудова графіків продуктивності	19.03–29.03.2025	Виконано
6	Оформлення роботи: висновки, рекомендації, реферат	29.03–30.03.2025	Виконано
7	Підготовка демонстраційних матеріалів та презентації	30.03–31.03.2025	Виконано
8	Попередній захист роботи	27.05–29.05.2025	Виконано
9	Здача в деканат	02.05–05.06.2025	Виконано

Здобувач вищої освіти _____ Дмитро ОХОТНІКОВ
(підпис)

Керівник
кваліфікаційної роботи _____ Олександр ЗУБ
(підпис)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 82 сторінок, 56 рисунки, 51 джерел.

Об'єкт дослідження – процеси високопродуктивних обчислень із застосуванням графічних процесорів.

Предмет дослідження – архітектура та принципи роботи сучасних GPU, технології паралельної обробки даних, а також ефективність використання графічного процесора порівняно з центральним процесором у задачах різної складності.

Мета роботи – дослідити архітектурні та програмні особливості високопродуктивних графічних процесорів, провести порівняльний аналіз продуктивності CPU та GPU в реальних задачах обробки даних, і визначити доцільність використання GPU для підвищення ефективності обчислювальних процесів.

GPU, CUDA, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ПРОДУКТИВНІСТЬ, БАГАТОЯДЕРНА АРХІТЕКТУРА, GPGPU, БЕНЧМАРКИ, МАТРИЧНІ ОБЧИСЛЕННЯ, SIMD, ВИСОКОПРОДУКТИВНІ СИСТЕМИ

ЗМІСТ

ВСТУП 8

1 ТЕОРЕТИЧНІ ОСНОВИ ВИСОКОПРОДУКТИВНИХ ГРАФІЧНИХ ПРОЦЕСОРІВ 9

1.1 Еволюція та роль GPU в сучасних обчислювальних системах 9

1.2 Архітектура центрального та графічного процесора: порівняльний аналіз 12

1.3 Моделі паралелізму в GPU (SIMD, MIMD, GPGPU) 15

1.4 Архітектурні особливості сучасних графічних процесорів (CUDA, OpenCL, Tensor Cores) 16

2 ПРОГРАМНІ ЗАСОБИ ТА ТЕХНОЛОГІЇ GPU-ПРИСКОРЕННЯ 18

2.1 Загальні принципи обчислень загального призначення на GPU 18

2.2 Огляд технологій GPGPU: CUDA, OpenCL, DirectCompute, AMD ROCm 22

2.3 Бібліотеки для GPU-прискорення: cuDNN, cuBLAS, Numba, Thrust, Dask, PyTorch 24

2.4 Проблеми передачі даних між CPU та GPU 26

3 ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ТА ЗАСТОСУВАННЯ ВИСОКОПРОДУКТИВНОГО GPU 29

3.1 Постановка експерименту: методика, обґрунтування 29

3.2 Вибір пристроїв та програмного забезпечення для тестування 30

3.3 Бенчмарки та аналіз ефективності GPU в порівнянні з CPU 43

3.4 Вплив обсягу даних, кількості потоків і типу задач на продуктивність ... 75

3.5 Узагальнення результатів та рекомендації щодо використання GPU 81

ВИСНОВКИ 83

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 85

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 90

ВСТУП

У сучасних умовах стрімкого зростання обсягів даних та складності обчислювальних задач зростає потреба в ефективних і високопродуктивних методах обробки інформації. Традиційні центральні процесори (CPU) часто не забезпечують достатньої продуктивності при виконанні задач, що вимагають масивних паралельних обчислень, таких як машинне навчання, обробка зображень, фізичне моделювання, рендеринг тощо.

Графічні процесори (GPU), завдяки своїй багатоядерній архітектурі та підтримці паралельного виконання великої кількості потоків, відкривають нові можливості для значного прискорення обчислень. Розвиток технологій CUDA, OpenCL та спеціалізованих бібліотек дає змогу переносити ресурсоємні обчислення з CPU на GPU, що дозволяє суттєво підвищити продуктивність без суттєвого збільшення витрат на апаратне забезпечення.

Актуальність дослідження зумовлена необхідністю глибокого аналізу архітектурних та програмних можливостей GPU, а також розробки методик ефективного використання графічних процесорів у задачах, де важлива швидкість, масштабованість і енергоефективність. У цьому контексті важливо оцінити переваги та обмеження GPU у порівнянні з CPU, зокрема в умовах реальних сценаріїв обчислень.

Методологія дослідження включає аналіз та узагальнення науково-технічної літератури з тематики високопродуктивних обчислень, архітектури сучасних графічних процесорів і принципів паралельної обробки даних. Значна увага приділяється вивченню технологій GPU-прискорення, таких як CUDA, OpenCL, а також бібліотек і фреймворків, що використовуються у задачах наукових обчислень, штучного інтелекту та обробки великих обсягів даних.

Дослідження передбачає реалізацію типових обчислювальних задач із використанням як CPU, так і GPU, побудову бенчмарків, а також експериментальне вимірювання продуктивності залежно від розміру даних, кількості паралельних

потоків і типу алгоритму. У ході роботи застосовуються методи системного аналізу, комп’ютерного експерименту, емпіричного моделювання та порівняльного аналізу результатів.

Практична частина включає реалізацію експериментальних задач, що виконуються як на CPU, так і на GPU, з подальшим вимірюванням часу виконання, навантаження на ресурси та масштабованості. Проводиться серія тестів, які дозволяють порівняти ефективність обчислень при різних параметрах – від кількості потоків до обсягу вхідних даних.

1 ТЕОРЕТИЧНІ ОСНОВИ ВИСОКОПРОДУКТИВНИХ ГРАФІЧНИХ ПРОЦЕСОРІВ

Розвиток графічних процесорів (GPU) розпочався на початку 1980-х років із появою простих 2D-графічних прискорювачів, які виконували базові операції рендерингу, розвантажуючи центральний процесор (CPU). Основним завданням таких пристроїв було відображення зображень на екрані, зокрема – операції з растровою графікою, бітовим копіюванням та базовим затіненням.

У 1990-х роках розвиток 3D-графіки пришвидшився завдяки впровадженню апаратного прискорення для роботи з полігонами, текстурами та ефектами перспективи. Знаковими прикладами того часу стали 3dfx Voodoo Graphics (1996) та серія NVIDIA RIVA, які започаткували перехід до програмованих графічних рішень.

У 1999 році компанія NVIDIA представила GeForce 256, який вперше було офіційно названо графічним процесором (GPU). Цей пристрій поєднував функції трансформації та освітлення (T&L) в апаратному забезпеченні, що стало революційним кроком до створення програмованої графіки в реальному часі.
У 2000-х роках GPU почали швидко розвиватися не лише як графічні прискорювачі, але й як інструменти для загальних обчислень. Випуск технології CUDA (Compute Unified Device Architecture) компанією NVIDIA у 2006 році відкрив можливість використання GPU для загального призначення (GPGPU). Відтоді графічні процесори почали широко використовуватись у наукових симуляціях, обробці великих даних, штучному інтелекті та високопродуктивних обчисленнях.

1.1 Еволюція та роль GPU в сучасних обчислювальних системах

Сучасні GPU – це високопродуктивні паралельні обчислювальні пристрої, що містять тисячі ядер і здатні одночасно обробляти великі масиви даних. Вони

відіграють ключову роль у таких галузях, як машинне навчання, моделювання фізичних процесів, блокчейн-технології, рендеринг і наукова візуалізація.

Первинне призначення графічних процесорів (GPU) полягало у виконанні виключно графічних завдань – обробці зображень, рендерингу 2D/3D сцени, текстуруванні та виведенні графіки на екран. Архітектура таких GPU була жорстко спеціалізованою, орієнтованою на фіксовані графічні пайплайни та виконання однотипних інструкцій у високій кількості.

Однак із розвитком обчислювальних потреб і ускладненням завдань, зокрема в науковій сфері, стало очевидним, що високий рівень паралелізму GPU може бути ефективно використаний не лише для візуалізації, а й для вирішення задач загального призначення.

З середини 2000-х років почався активний перехід до обчислень загального призначення на графічних процесорах (GPGPU). Це стало можливим завдяки появі програмних платформ, які надали розробникам інструменти для прямого доступу до обчислювальних ресурсів GPU без прив’язки до графічних API. Найважливішим етапом у цьому переході стало створення технології CUDA (2006) компанією NVIDIA, яка дозволила програмувати GPU за допомогою розширення мови C та запускати обчислювальні "кернели" паралельно на тисячах ядер.

Завдяки GPGPU графічні процесори почали активно застосовуватись у таких сферах, як:

- чисельне моделювання;
- машинне навчання та нейронні мережі;
- криптографія та блокчейн;
- обробка великих даних (Big Data);
- біоінформатика;
- фінансове моделювання.

GPGPU став основою сучасного тренду до гетерогенних обчислень, коли CPU виконує логічну та послідовну обробку, а GPU – масово-паралельні задачі. Такий розподіл дозволяє досягати суттєвого приросту продуктивності та ефективності використання апаратних ресурсів.

Перехід від суто графічних завдань до GPGPU означає еволюцію графічного процесора у повноцінний інструмент для загальних високопродуктивних обчислень, який суттєво розширив межі його застосування у сучасних інформаційних технологіях.

У сучасних обчислювальних системах графічні процесори (GPU) стали незамінними інструментами для виконання наукомістких задач, які вимагають високої обчислювальної потужності, паралельності та ефективної обробки великих обсягів даних.

1. Машинне навчання та штучний інтелект (ML/AI):

GPU активно використовуються в задачах навчання нейронних мереж, де обчислення включають мільйони операцій з матрицями та векторами. Завдяки паралельній архітектурі, GPU здатні значно пришвидшити такі операції, як:

– зворотне поширення помилки (backpropagation);
– обчислення градієнтів;
– множення великих матриць у глибоких нейронних мережах.

Фреймворки на кшталт TensorFlow, PyTorch та Keras мають оптимізовану підтримку GPU через CUDA та cuDNN, що дозволяє скоротити час навчання моделей у десятки разів порівняно з CPU.

2. Високопродуктивні обчислення (HPC):

У галузях фізики, хімії, біоінформатики, метеорології GPU використовуються для моделювання складних процесів:

– моделювання молекулярної динаміки;
– кліматичні симуляції;
– вирішення диференціальних рівнянь;
– моделювання ядерних реакцій.

Використання GPU-кластерів дозволяє реалізовувати масштабовані обчислення на тисячах вузлів, що значно перевищує потужність традиційних суперкомп’ютерів, побудованих лише на CPU.

3. Обробка великих даних (Big Data):

У сфері Big Data GPU застосовуються для:

– обробки потокових даних у реальному часі;

– аналізу структурованих і неструктурованих даних;

– прискорення алгоритмів кластеризації, фільтрації, сортування.

GPU-платформи, такі як RAPIDS, дозволяють реалізовувати повноцінні аналітичні пайплайни безпосередньо на відеокарті, що значно скорочує час між збором даних та отриманням результатів.

1.2 Архітектура центрального та графічного процесора: порівняльний аналіз

Центральний процесор (CPU) і графічний процесор (GPU) мають різну архітектурну побудову, оскільки орієнтовані на виконання принципово різних типів задач. CPU призначений для загального призначення – обробки логіки програм, послідовних операцій, взаємодії з пам’яттю та системними ресурсами. GPU, натомість, розроблений для масово-паралельних обчислень, зокрема – для обробки графіки, чисельного моделювання та задач високої інтенсивності даних.

Кількість ядер:

– CPU зазвичай містить від 2 до 32 високопродуктивних ядер, оптимізованих для складних послідовних інструкцій, що вимагають складної логіки гілкування, планування задач, переривань;

– GPU може містити від сотень до тисяч простіших ядер, організованих у блоки (Streaming Multiprocessors), що дозволяють виконувати одну інструкцію над багатьма даними одночасно (SIMD – Single Instruction, Multiple Data).

Кеш-пам’ять:

– CPU має багаторівневу кеш-пам’ять великого обсягу (L1, L2, L3), що забезпечує мінімальні затримки при роботі з пам’яттю та високий рівень передбачення даних;

– GPU використовує менший обсяг кешу, але має спеціалізовані рівні пам’яті: глобальну (Global Memory), спільну (Shared Memory), текстурну, постійну та регістри, що дозволяють оптимізувати обчислення залежно від задачі.

Логіка виконання:

– CPU обробляє складні команди з точним керуванням, підтримує віртуалізацію, багатозадачність і захищені режими.

– GPU спрощує керування для зменшення витрат і виконує однотипні обчислення у тисячах потоків без складного планування.

Узагальнюючи, можна сказати, що CPU – це потужний «універсальний обчислювальний інтелект», який ефективний у складних послідовних задачах, тоді як GPU – це «масово-паралельна обчислювальна машина», яка ідеально підходить для однотипних, добре масштабованих операцій на великих обсягах даних. Їх поєднання формує основу гетерогенних обчислювальних систем.

Графічні процесори (GPU) та центральні процесори (CPU) мають принципово різні архітектурні підходи до обробки інформації, що визначає їх ефективність у різних типах задач. CPU призначений для обробки послідовних обчислень. Його архітектура включає невелику кількість потужних ядер, які мають доступ до великого обсягу кеш-пам’яті та підтримують складні інструкції, умовні переходи, логічні розгалуження і управління ресурсами. Саме тому CPU ідеально підходить для задач, у яких важлива логіка, динамічне прийняття рішень, взаємодія з пам’яттю або операційною системою, а також обробка невеликої кількості, але складних операцій у строго визначеній послідовності.

GPU, натомість, спроєктований для максимально ефективного виконання великої кількості однотипних обчислень одночасно. Завдяки сотням або навіть тисячам простих ядер, GPU здатен працювати у масово-паралельному режимі, де одна інструкція виконується над великою кількістю незалежних даних. Саме така

особливість робить графічний процесор надзвичайно потужним у задачах, які легко розпаралелюються, – наприклад, у машинному навчанні, обробці зображень, відео, чисельному моделюванні, обробці великих даних тощо.

Архітектурна відмінність між CPU та GPU зумовлює різний підхід до виконання обчислювальних задач. Центральний процесор (CPU) розроблений для ефективної обробки послідовних інструкцій, що потребують складної логіки, гнучкого управління та тісної взаємодії з операційною системою. Графічний процесор (GPU), натомість, орієнтований на масово-паралельне виконання однотипних операцій над великими обсягами даних, що робить його незамінним у задачах, пов’язаних із машинним навчанням, обробкою зображень, моделюванням та іншими високонавантаженими сценаріями. Для наочності ключові відмінності між архітектурою CPU та GPU узагальнено в Таблиці 1.1.

Таблиця 1.1 – Порівняння архітектури CPU та GPU

Параметр	**CPU (Central Processing Unit)**	**GPU (Graphics Processing Unit)**
Кількість ядер	Невелика кількість потужних ядер (2–32)	Сотні або тисячі простих ядер
Архітектура виконання	Оптимізована для послідовного виконання інструкцій	Орієнтована на паралельне виконання однотипних операцій (SIMD)
Кеш-пам’ять	Велика багаторівнева кеш-пам’ять (L1, L2, L3)	Обмежена кеш-пам’ять; використовується глобальна, shared, текстурна пам’ять
Призначення	Загальні задачі, логіка, контроль, операції з ОС	Важкі обчислення, графіка, машинне навчання, GPGPU
Тип обчислень	Послідовні, умовні, логічні, контрольні	Масово-паралельні, одноманітні, числово-інтенсивні
Гнучкість	Висока (підтримка складної логіки, планування задач)	Обмежена (фокус на повторюваних операціях над даними)
Продуктивність на 1 ядро	Висока	Нижча, але компенсується кількістю потоків
Оптимальні задачі	Компіляція, обробка тексту, керування ресурсами	Матричні обчислення, рендеринг, моделювання, машинне навчання

1.3 Моделі паралелізму в GPU (SIMD, MIMD, GPGPU)

У сучасних обчислювальних системах широко використовуються різні моделі паралелізму, кожна з яких має свою архітектурну специфіку та сферу ефективного застосування. Найбільш поширеними є SIMD, MIMD та GPGPU – моделі, що реалізуються як у класичних процесорах, так і у графічних прискорювачах.

SIMD (Single Instruction, Multiple Data) – це модель, у якій один потік інструкцій одночасно виконується над багатьма елементами даних. Такий тип паралелізму дозволяє обробляти масиви чисел, зображень або векторів за допомогою однієї інструкції на множині обчислювальних ядер. SIMD реалізується, зокрема, у векторних процесорах та сучасних GPU, які виконують операції над пікселями, елементами матриць або векторами. Найефективніше SIMD використовується у задачах обробки сигналів, комп'ютерної графіки, фільтрації, шифрування, а також у глибокому навчанні.

MIMD (Multiple Instruction, Multiple Data) – це гнучкіша модель, у якій декілька незалежних потоків інструкцій працюють над різними наборами даних. Вона реалізується в багатоядерних CPU або у гібридних обчислювальних системах. Кожне ядро може виконувати свій власний код, що дозволяє ефективно розподіляти задачі з нерівномірною структурою або складною логікою. MIMD найбільш ефективна у сценаріях багатозадачності, керування потоками, емуляції, базах даних та серверних системах.

GPGPU (General-Purpose computing on Graphics Processing Units) – це підхід, за якого графічні процесори використовуються не лише для графіки, а й для обчислень загального призначення. Він поєднує переваги SIMD-моделі з архітектурою GPU та дозволяє виконувати масово-паралельні обчислення в задачах, які потребують обробки великих обсягів даних. GPGPU ефективно застосовується в науковому моделюванні, фінансовій аналітиці, біоінформатиці,

рендерингу, криптографії та машинному навчанні.

Вибір моделі паралелізму залежить від характеру задачі. SIMD забезпечує високу ефективність у випадках однакових обчислень над великими наборами даних, MIMD – у задачах з гнучкою логікою та розгалуженою структурою, а GPGPU – у високопродуктивних обчисленнях, де ключовим є масштаб і паралельність.

1.4 Архітектурні особливості сучасних графічних процесорів (CUDA, OpenCL, Tensor Cores)

Сучасні графічні процесори мають складну архітектуру, оптимізовану для масово-паралельних обчислень, що реалізуються за допомогою спеціалізованих апаратних та програмних рішень. Основу обчислювального потенціалу GPU становлять платформи CUDA та OpenCL, а також модулі, такі як Tensor Cores, Streaming Multiprocessors (SM), Texture Processing Clusters (TPC) та Warp-структури, які забезпечують високу масштабованість і продуктивність при виконанні паралельних задач.

CUDA (Compute Unified Device Architecture) – це пропрієтарна платформа, розроблена компанією NVIDIA, яка надає можливість програмування графічного процесора мовами, подібними до C/C++, без потреби використовувати графічні API. Архітектура CUDA передбачає поділ на хост (CPU) та пристрій (GPU), де CPU ініціалізує та керує виконанням, а GPU обробляє масиви даних за допомогою тисяч паралельних потоків. CUDA забезпечує гнучкий контроль над ресурсами, синхронізацією та управлінням пам’яттю, що робить її провідним стандартом у сфері GPGPU-обчислень.

OpenCL (Open Computing Language) – це відкритий стандарт, який підтримується як NVIDIA, так і AMD, Intel, Apple та іншими компаніями. На відміну від CUDA, OpenCL є кросплатформенним і може працювати з різними

типами пристроїв: GPU, CPU, FPGA. Архітектура OpenCL також базується на моделі "хост-пристрій", але має більш універсальний інтерфейс програмування, що дозволяє реалізовувати паралельні обчислення на широкому спектрі обладнання.

У сучасних GPU, особливо архітектур NVIDIA (Volta, Turing, Ampere), реалізовано низку апаратних блоків, які відповідають за паралельне виконання інструкцій:

– *Tensor Cores.* Спеціалізовані ядра для прискорення тензорних обчислень, зокрема у сфері машинного навчання та глибоких нейронних мереж. Вони дозволяють обчислювати матричні добутки з високою швидкістю, що особливо актуально для тренування та інференсу моделей;

– *SM (Streaming Multiprocessor).* Базовий обчислювальний блок GPU, який містить десятки ядер, що виконують паралельні потоки. Кожен SM має власний набір регістрів, кеш-пам’яті та ресурсів для обробки кернелів;

– *TPC (Texture Processing Cluster).* Група SM, яка об’єднується для ефективної обробки текстур, геометрії та загальних обчислень. TPC є важливою одиницею масштабування всередині GPU;

– *Warp.* Група з 32 потоків, які одночасно виконують одну інструкцію в рамках SIMD-моделі. Управління Warp’ами дозволяє GPU ефективно реалізовувати паралелізм без втрат продуктивності через розгалуження коду.

Ієрархія пам’яті. Архітектура пам’яті сучасного GPU побудована за багаторівневою схемою, що дозволяє ефективно керувати доступом до даних:

– *Регістри*. Найшвидший рівень пам’яті, призначений для зберігання приватних змінних кожного потоку;

– *Shared Memory*. Швидка пам’ять, спільна для всіх потоків в межах одного блока. Вона дозволяє обмінюватися даними без затримок глобального доступу.

Сукупність цих апаратних і програмних елементів дозволяє сучасним GPU забезпечувати надзвичайно високу продуктивність у паралельних обчисленнях і залишатися ключовим компонентом у сфері наукомістких задач, таких як штучний інтелект, чисельне моделювання, рендеринг та аналітика великих даних.

2 ПРОГРАМНІ ЗАСОБИ ТА ТЕХНОЛОГІЇ GPU-ПРИСКОРЕННЯ

Обчислення загального призначення на графічних процесорах (англ. *General-Purpose computing on Graphics Processing Units*, GPGPU) є сучасною парадигмою, яка полягає у використанні GPU не тільки для графічної візуалізації, а й для розв'язання широкого кола обчислювальних задач, не пов'язаних із рендерингом. Цей підхід базується на ідеї, що архітектура графічного процесора – яка включає велику кількість паралельних ядер – може бути ефективно використана для обробки великих масивів даних у різних галузях, від наукових розрахунків до штучного інтелекту.

2.1 Загальні принципи обчислень загального призначення на GPU

Програмування GPGPU передбачає побудову спеціального обчислювального коду (кернелу), який виконується одночасно на багатьох потоках графічного процесора. На відміну від CPU, де програміст зазвичай пише логіку послідовного виконання з можливим розгалуженням, GPGPU орієнтований на масове однотипне обчислення. Це означає, що один і той самий фрагмент коду виконується паралельно тисячами потоків над різними ділянками даних.

Основними мовами та технологіями, що використовуються для програмування GPU, є:

– *CUDA.* Власна технологія NVIDIA, яка дозволяє створювати кернели мовою C/C++ з використанням спеціальних розширень. Вона надає програмісту повний контроль над пам’яттю, синхронізацією, розміщенням блоків і потоків;

– *OpenCL.* Відкритий стандарт, який підтримується багатьма виробниками апаратного забезпечення (AMD, Intel, Apple). Дає змогу створювати кросплатформені рішення, однак має вищу складність у налаштуванні;

– *Широкий спектр бібліотек*. cuBLAS, cuDNN, Thrust, TensorRT, PyCUDA, забезпечують зручний доступ до високорівневих операцій та глибокої оптимізації під GPU.

Процес програмування обчислень на GPU включає кілька ключових етапів:

– *Підготовка даних*. Введення, ініціалізація масивів, визначення розміру задачі;

– *Копіювання даних на GPU*. Передача інформації з оперативної пам’яті CPU до глобальної пам’яті GPU;

– *Виклик кернелу*. Запуск паралельної функції з вказаною кількістю блоків і потоків;

– *Виконання обчислень*. GPU одночасно обробляє елементи даних, розподілених між потоками;

– *Повернення результатів*. Копіювання результатів назад у пам’ять CPU для подальшого використання або аналізу.

Система координат CUDA (threadIdx, blockIdx, blockDim, gridDim) дозволяє легко ідентифікувати, який саме потік за яку частину задачі відповідає, що є важливою складовою правильної організації паралельної логіки.

Переваги обчислень на GPU

– Масовий паралелізм, GPU дозволяє запускати тисячі потоків одночасно. Це дає значний приріст продуктивності у задачах, де одна й та сама операція повторюється багато разів (наприклад, множення матриць, конволюції, пошук максимуму тощо).

– Висока обчислювальна щільність, кількість FLOPS (floating point operations per second) на одиницю вартості або енерговитрат у GPU значно вища, ніж у CPU.

– Продуктивність у наукомістких задачах, GPU стали основою сучасного прогресу у глибокому навчанні, симуляціях фізичних процесів, рендерингу, фінансовому аналізі тощо.

– Гнучка архітектура, сучасні GPU мають спеціалізовані обчислювальні модулі (наприклад, Tensor Cores), що дозволяє виконувати високоточні або низькорівневі обчислення ще швидше.

– Багата екосистема, велика кількість бібліотек та фреймворків (TensorFlow, PyTorch, cuDNN, cuBLAS) дозволяють інтегрувати GPGPU в наукові та комерційні продукти без необхідності писати низькорівневий код.

Недоліки обчислень на GPU

– Складність у розробці, порівняно з CPU-програмуванням, створення GPGPU-програм вимагає розуміння паралельної логіки, ієрархії пам’яті, синхронізації потоків і оптимізації ресурсів.

– Витрати на копіювання даних, передача інформації з CPU до GPU і назад – це значний часовий ресурс. У задачах з невеликим обсягом обчислень витрати на копіювання можуть переважити виграш у швидкодії.

– Обмеження апаратної архітектури, наприклад, обмежена кількість потоків на блок (до 1024 у CUDA), обсяг shared memory, кількість SM – усе це впливає на масштабованість задачі.

– Неуспішність у задачах із сильною гілкою логіки, GPU неефективний, якщо потоки виконують різні інструкції або часто переходять між умовами – це знижує рівень паралелізму і породжує явище *warp divergence*.

– Залежність від постачальника, CUDA працює тільки на GPU від NVIDIA, що може бути обмеженням у гетерогенних середовищах або при роботі з AMD/Intel.

Обчислення загального призначення на GPU відкривають значні можливості для прискорення обчислень у багатьох сферах, але вимагають глибокого розуміння паралельного програмування, оптимізації пам’яті та специфіки архітектури. Вдале використання цієї технології дозволяє досягти якісно нового рівня продуктивності у задачах, які ще донедавна були недоступні для реального часу або стандартного апаратного забезпечення.

Перед використанням обчислень загального призначення на GPU важливо враховувати не лише переваги, а й потенційні обмеження цієї технології. У Таблиці

2.1 представлено узагальнене порівняння основних сильних та слабких сторін GPGPU, що дозволяє краще зрозуміти доцільність її застосування залежно від типу обчислювального завдання.

Таблиця 2.1 – Переваги та недоліки обчислень загального призначення на GPU

Аспект	**Переваги**	**Недоліки**
Продуктивність	Висока швидкість завдяки одночасному виконанню тисяч потоків	Виграш обмежений, якщо задача не піддається розпаралелюванню
Масштабованість	Легко масштабуються задачі з великою кількістю даних	Обмежена кількість потоків на блок і ресурсів на кожен SM
Енергетична ефективність	Більше FLOPS на ват енергії порівняно з CPU	Високе тепловиділення при повному навантаженні
Архітектурна гнучкість	Підтримка Tensor Cores, FP16/FP32/FP64, різних рівнів пам’яті	Не всі архітектури однаково ефективні для всіх типів задач
Бібліотечна підтримка	Велика кількість бібліотек: cuDNN, cuBLAS, PyTorch, TensorFlow	Більшість бібліотек оптимізовані лише під CUDA (тобто лише під NVIDIA)
Програмування	Можливість глибокої оптимізації через CUDA або OpenCL	Висока складність написання та налагодження паралельного коду
Передача даних	Можливість обробляти великі масиви даних без участі CPU	Затримки при копіюванні даних між CPU ↔ GPU через PCIe
Тип задач	Ідеальні для SIMD-задач: матричні операції, ML, рендеринг, симуляції	Неефективні при умовних переходах, ітераціях з розгалуженням (warp divergence)
Платформна підтримка	CUDA, OpenCL, Vulkan Compute, DirectCompute	CUDA працює лише на NVIDIA GPU, OpenCL менш ефективний

2.2 Огляд технологій GPGPU: CUDA, OpenCL, DirectCompute, AMD ROCm

У сфері обчислень загального призначення на GPU (GPGPU) найбільш поширеними платформами для програмування є CUDA, OpenCL та DirectCompute. Кожна з них має свої особливості, переваги та недоліки залежно від потреб розробника та обчислювального середовища.

CUDA – це пропрієтарна платформа компанії NVIDIA, яка дозволяє програмувати GPU мовами C/C++ та Python (через PyCUDA, Numba). Вона забезпечує повний контроль над потоками, пам’яттю та обчислювальними ресурсами.

Переваги:

– оптимізована під GPU NVIDIA, забезпечує максимальну продуктивність;

– має велику екосистему бібліотек: cuBLAS, cuDNN, TensorRT, Thrust;

– активно підтримується спільнотою та виробником.

Недоліки:

– працює лише на NVIDIA GPU;

– прив'язує розробника до однієї апаратної платформи.

OpenCL – це відкритий стандарт, розроблений Khronos Group. Він підтримується широким колом виробників (AMD, Intel, Apple, NVIDIA) і дозволяє запускати паралельний код на CPU, GPU, FPGA, DSP та інших пристроях.

Переваги:

– кросплатформеність – працює на багатьох апаратних архітектурах;

– можливість гібридних рішень із CPU та GPU в одній програмі;

– відкритість і незалежність від одного постачальника.

Недоліки:

– складніший у програмуванні порівняно з CUDA;

– менша кількість високорівневих бібліотек;

– не завжди стабільна підтримка драйверів і компіляторів.

DirectCompute – це компонент платформи Microsoft DirectX, який дозволяє виконувати обчислення на GPU у середовищі Windows. Його основне призначення – мультимедійні додатки, обробка графіки та відео в іграх.

Переваги:

– інтеграція з іншими компонентами DirectX, особливо з Direct3D;

– зручний для розробки графічних ефектів у Windows-додатках.

Недоліки:

– працює тільки в середовищі Windows;

– не підходить для наукових та інженерних задач;

– не підтримується на Linux або macOS.

Платформа ROCm (Radeon Open Compute) – це відкритий програмний стек, створений компанією AMD для реалізації обчислень загального призначення (GPGPU) на відеокартах Radeon. ROCm є відповіддю AMD на пропрієтарну платформу CUDA від NVIDIA та орієнтована на відкритість, масштабованість і ефективну інтеграцію з екосистемами високопродуктивних обчислень.

Основна ідея ROCm полягає у створенні повноцінного середовища для розробки та запуску паралельних обчислень на GPU, з підтримкою мов програмування C++, HIP (Heterogeneous-Compute Interface for Portability), а також OpenCL. HIP є проміжною платформою, яка дозволяє писати код, що може бути скомпільований як для CUDA (на NVIDIA), так і для ROCm (на AMD) – це забезпечує кращу переносимість застосунків між різними виробниками GPU.

Переваги ROCm:

– відкритість і кросплатформеність – повністю open-source екосистема;

– підтримка HIP – дозволяє адаптувати існуючі CUDA-програми для AMD GPU;

– інтеграція з HPC – активне використання в наукових і промислових суперкомп’ютерах, таких як Frontier (США), де використано GPU AMD;

– сумісність із фреймворками – підтримка TensorFlow, PyTorch, ONNX Runtime, а також бібліотек для лінійної алгебри (rocBLAS, MIOpen, rocFFT).

Недоліки:

– обмежена підтримка моделей GPU – не всі відеокарти AMD сумісні з ROCm, платформа найкраще працює з професійними картами (MI Instinct, серії Radeon PRO);

– менша зрілість екосистеми – у порівнянні з CUDA, ROCm ще не має такої кількості інструментів і бібліотек;

– вища складність налаштування – потребує більше часу для конфігурації середовища, особливо під Linux.

ROCm є перспективним інструментом для розробників, які бажають працювати з відеокартами AMD у задачах високопродуктивних обчислень, машинного навчання та симуляцій. Хоча платформа ще не досягла рівня зрілості CUDA, вона активно розвивається, особливо в академічному середовищі, де цінується відкритість і доступність.

2.3 Бібліотеки для GPU-прискорення: cuDNN, cuBLAS, Numba, Thrust, Dask, PyTorch

Наявність спеціалізованих бібліотек забезпечують ефективне використання графічних процесорів для загального програмування. Ці бібліотеки значно спрощують розробку обчислювальних додатків, знижують поріг входу в GPGPU-програмування та забезпечують доступ до високопродуктивних апаратних функцій GPU без потреби писати низькорівневий код.

cuDNN (CUDA Deep Neural Network library) – це високопродуктивна GPU-оптимізована бібліотека, розроблена NVIDIA для прискорення навчання глибоких нейронних мереж. Вона включає реалізації основних операцій, таких як згортки, нормалізація, активації, пулінг та зворотне поширення помилки. Використання cuDNN дозволяє значно зменшити час тренування моделей у фреймворках, таких

як TensorFlow або PyTorch, забезпечуючи гнучкість і продуктивність, наближену до максимальної пропускної здатності GPU.

cuBLAS – це GPU-версія відомої бібліотеки BLAS (Basic Linear Algebra Subprograms), яка реалізує основні лінійні алгебраїчні операції: векторно-матричні множення, транспонування, LU-розклади та інші. Завдяки оптимізації під CUDA, cuBLAS дозволяє досягати надзвичайної швидкості виконання математичних операцій, що є критично важливим у машинному навчанні, симуляціях, комп'ютерному зорі та інших технічних застосуваннях.

Numba – це компілятор Python-коду, який дозволяє перетворювати високорівневі функції в машинний код за допомогою JIT-компіляції (Just-in-Time). У поєднанні з CUDA, Numba надає розробникам можливість писати кернели на чистому Python, які потім виконуються на GPU. Це дуже зручно для швидкого прототипування наукових алгоритмів.

Thrust – це C++ STL-подібна бібліотека з високорівневими абстракціями для паралельного програмування. Вона підтримує як CPU, так і GPU-виконання, що дозволяє легко переносити код між архітектурами. Thrust має вбудовані алгоритми сортування, редукції, сканування, фільтрації, які значно прискорюють обробку даних на GPU.

Dask – це бібліотека для паралельного обчислення великих масивів даних у Python, яка підтримує інтеграцію з RAPIDS – набором бібліотек від NVIDIA для GPU-прискореної обробки даних. Завдяки цьому Dask дозволяє ефективно розподіляти обчислення між CPU та GPU, масштабуючи аналітичні задачі навіть у хмарному середовищі.

PyTorch – це один із найпопулярніших фреймворків глибокого навчання, який має нативну підтримку GPU. Використовуючи CUDA і cuDNN, PyTorch дозволяє легко переносити тензори та моделі на GPU, забезпечуючи прискорення в десятки разів порівняно з CPU.

Ключові переваги PyTorch:

– автоматичне диференціювання (autograd);
– інтеграція з бібліотеками TorchScript, TorchVision, HuggingFace;

– динамічна побудова графів, що зручно для досліджень і адаптивних моделей.

GPU-прискорення у PyTorch не потребує низькорівневого втручання – достатньо перенести тензори на пристрій командою .to("cuda"), і обчислення автоматично відбуватимуться на GPU.

Використання спеціалізованих бібліотек дає змогу розробникам зосередитися на логіці алгоритмів, не витрачаючи час на оптимізацію обчислень вручну. Вони стали основою ефективного GPGPU-програмування у сучасному середовищі – від наукових досліджень до комерційних продуктів у сфері штучного інтелекту, обробки даних та автоматизації.

2.4 Проблеми передачі даних між CPU та GPU

Незважаючи на високий обчислювальний потенціал графічних процесорів, одним із основних обмежуючих чинників при їх використанні є передача даних між CPU та GPU. У парадигмі GPGPU архітектура зазвичай передбачає розподіл ролей: центральний процесор (CPU) відповідає за загальне управління програмою, а графічний процесор (GPU) – за масово-паралельні обчислення. Проте цей поділ призводить до необхідності постійного обміну інформацією між обома пристроями, що створює додаткові затрати часу та ресурсів.

Передача даних з оперативної пам’яті (host memory), яка контролюється CPU, до відеопам’яті GPU (device memory) виконується через шину PCI Express (PCIe). Хоча сучасні версії PCIe (наприклад, PCIe 4.0 або 5.0) забезпечують високу пропускну здатність, вона все ще значно нижча порівняно з пропускною здатністю локальної пам’яті GPU. Це означає, що великі обсяги копіювання можуть стати вузьким місцем у продуктивності, особливо якщо дані часто змінюються або обчислення виконуються на невеликих масивах, де час обчислення менший за час передачі.

В окремих випадках витрати на копіювання можуть повністю нівелювати переваги GPU. Наприклад, якщо операція над невеликою матрицею виконується за мілісекунди, але передача цієї матриці відбувається довше – використання GPU стає нераціональним.

Для зменшення цих витрат у сучасних системах застосовуються такі підходи:

– використання асинхронних копій із перекриттям обчислень та передавання;

– застосування уніфікованої пам’яті (Unified Memory), яка дозволяє автоматично управляти даними між CPU і GPU;

– оптимізація обсягу переданих даних – робота з підмножинами або потоковими структурами.

GPU має складну ієрархічну систему пам’яті, яка безпосередньо впливає на продуктивність обчислень. Неефективне використання цієї структури може призвести до затримок доступу до даних навіть у межах самого GPU.

Основні рівні пам’яті:

– *Global memory*. Основна відеопам’ять пристрою (DRAM), доступна всім потокам, але має найвищу затримку. Часто використовується для зберігання вхідних і вихідних даних, проте доступ до неї повинен бути структурованим (coalesced), інакше значно падає пропускна здатність;

– *Shared memory*. Швидка пам’ять, спільна для потоків одного блока. Завдяки низькій затримці і прямому доступу вона ідеально підходить для обміну даними між потоками та для тимчасового кешування під час обчислень. Неправильне використання цієї пам’яті (наприклад, конфлікти банків) може знижувати ефективність;

– *Registers*. Найшвидший тип пам’яті, доступний для кожного окремого потоку. Їх використання забезпечує максимальну швидкість, проте їх кількість обмежена;

– *Constant memory*. Використовується для зберігання незмінних даних, доступних усім потокам. Оптимізована для одночасного читання багатьма потоками;

– *Texture/Surface memory*. Спеціалізована пам’ять, яка має кешування та інтерполяцію, корисна для доступу до зображень або багатовимірних масивів.

Ефективна робота з пам’яттю GPU передбачає:

– зведення до мінімуму доступу до глобальної пам’яті;

– використання shared memory для обміну між потоками;

– уникнення банківських конфліктів у спільній пам’яті;

– розподіл регістрів з урахуванням обмежень кожного ядра.

Передача даних між CPU і GPU, а також правильне використання внутрішньої пам’яті GPU – критичні фактори, які визначають загальну продуктивність обчислень. Навіть найефективніші обчислювальні алгоритми можуть втратити свою перевагу, якщо витрати на обмін інформацією перевищують виграш від паралелізму. Тому оптимізація доступу до пам’яті та мінімізація копіювань є ключовими елементами при розробці високопродуктивних GPGPU-застосунків.

3 ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ТА ЗАСТОСУВАННЯ ВИСОКОПРОДУКТИВНОГО GPU

Метою даного експериментального дослідження є виявлення оптимального розміру вхідного набору даних, при якому доцільно використовувати графічний процесор (GPU) замість центрального процесора (CPU) для виконання базових матричних операцій. Такий підхід дозволяє визначити поріг ефективності переходу до паралельних обчислень на GPU залежно від обсягу задачі.

3.1 Постановка експерименту: методика, обґрунтування

У процесі дослідження обрана низка базових матричних операцій, які найчастіше використовуються в задачах високопродуктивних обчислень:

- додавання матриць;
- множення матриць;
- скалярне множення вектора;
- ділення матриці на число;
- додавання матриць із коефіцієнтом.

Для реалізації тестів на CPU обрано мову програмування C++, зокрема використання стандартної бібліотеки std::thread для побудови багатопотокових рішень. Такий вибір обґрунтовується високою продуктивністю компільованого коду та можливістю детального управління пам’яттю й ресурсами.

Тести на GPU реалізовані з використанням платформи CUDA від NVIDIA. CUDA дозволяє безпосередньо програмувати обчислення на графічних процесорах, обминаючи графічні API, що значно підвищує продуктивність та адаптованість коду під обчислювальні задачі. CUDA C, як розширення мови C,

забезпечує паралельне виконання кернелів на великій кількості потоків, що є ідеальним для SIMD-задач з великим обсягом даних.

Методика проведення експерименту включає наступні етапи:

– реалізація однакових алгоритмів матричних операцій для CPU (на C++) та GPU (на CUDA C);

– варіювання розміру вхідного набору даних: від малих матриць (напр., 32×32) до великих (до декількох тисяч елементів);

– проведення серії бенчмарків із фіксацією часу виконання та ресурсів, що використовуються;

– порівняння результатів CPU і GPU в залежності від обсягу обчислень;

– визначення критичної межі, при якій GPU починає перевершувати CPU за продуктивністю.

Для досягнення достовірних результатів, експерименти проводились на декількох моделях CPU та GPU, доступних на серверному обладнанні кафедри системного проєктування. Зокрема, були використані два CPU і три GPU, що дозволило охопити різні покоління та технічні характеристики пристроїв, а також дослідити вплив апаратного забезпечення на швидкодію.

Результати експерименту мають практичне значення для розробників високопродуктивних обчислювальних систем, а також для оптимізації розподілу ресурсів у задачах, орієнтованих на інтенсивні обчислення з великими обсягами даних.

3.2 Вибір пристроїв та програмного забезпечення для тестування

У цьому підрозділі розглянуто програмно-апаратні засоби, що були використані для проведення експериментального дослідження продуктивності при виконанні базових матричних операцій. Основна мета аналізу полягає у визначенні найбільш доцільних технологічних платформ для забезпечення максимально

ефективного виконання обчислень як на центральному (CPU), так і на графічному процесорі (GPU).

Під час вибору програмного забезпечення враховувались такі критерії, як: здатність до ефективного паралельного виконання задач, підтримка сучасних архітектур обчислювальних пристроїв, а також зручність інтеграції з мовами програмування низького рівня. Усі інструменти, що увійшли до експериментального середовища, дозволяють створювати високопродуктивні рішення з акцентом на обчислювальну інтенсивність.

Застосування відповідного набору апаратних та програмних засобів є критично важливим для забезпечення об'єктивності результатів. Подальший опис охоплює обґрунтування обраних рішень для CPU- та GPU-реалізацій та пояснює доцільність їх використання у рамках поставленої задачі.

У межах цього дослідження для реалізації тестів на центральному процесорі було обрано мову програмування C++, що вирізняється високою ефективністю виконання програмного коду, особливо у задачах з інтенсивними обчисленнями. Основною перевагою C++ є її компільована природа: на відміну від інтерпретованих мов, компілятор одразу створює виконуваний машинний код, що значно знижує витрати на перетворення інструкцій під час виконання.

У численних незалежних бенчмарках C++ стабільно демонструє високі показники продуктивності. Наприклад, згідно з результатами тестування задачі побудови bitmap-зображення множини Мандельброта, мова C++ увійшла до числа лідерів серед найпродуктивніших мов, що підтверджується даними проєкту The Computer Language Benchmarks Game. Це підтверджує здатність мови ефективно працювати з великими обсягами даних та реалізовувати паралелізм на рівні потоків (рис. 3.1).

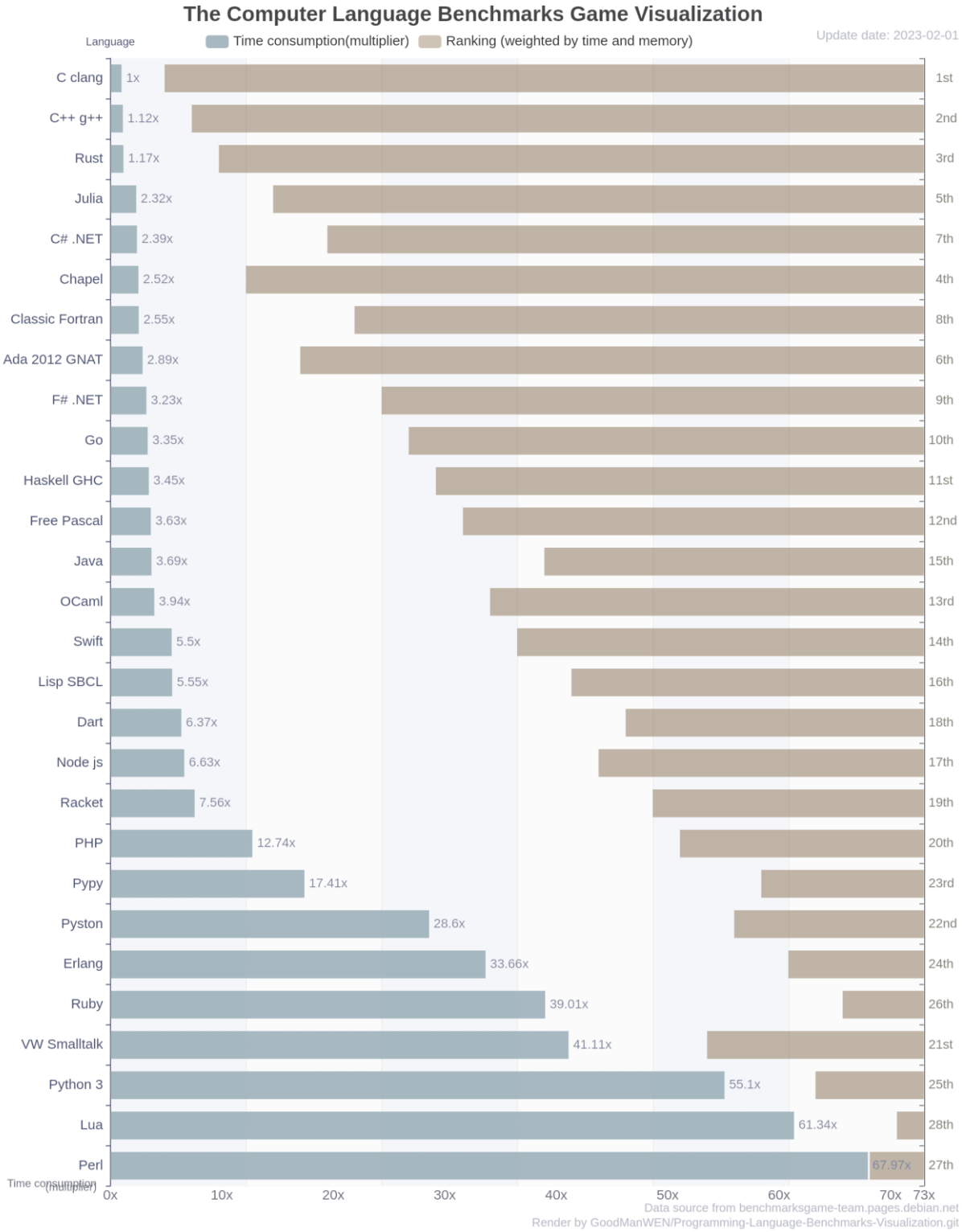


Рисунок 3.1 – Порівняння продуктивності мов програмування у задачі генерації множини Мандельброта

Ще одним прикладом, що ілюструє доцільність використання C++ у наукових обчисленнях, є реалізація алгоритму вирівнювання ДНК-послідовностей у задачах біоінформатики. У відповідному дослідженні була використана мова C++

для запуску коду на різних операційних системах (Windows та Linux), що дозволило оцінити стабільність результатів та універсальність підходу. За результатами тестів, що наведені на рисунку 3.2, мова C++ показала високу швидкодію на обох платформах.

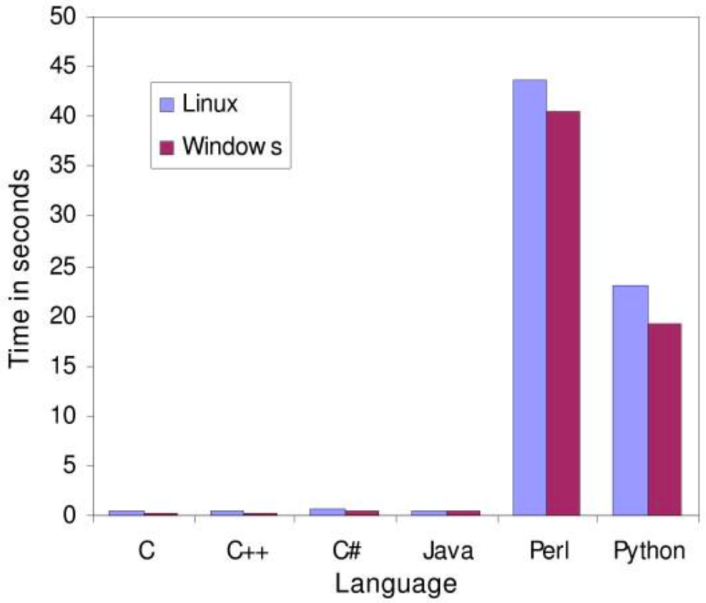


Рисунок 3.2 – Результати виконання алгоритму вирівнювання ДНК-послідовностей на різних ОС

Результати згаданих тестів підтверджують високу продуктивність мов програмування C та C++ у задачах, що вимагають інтенсивних обчислень. Як показано на рисунку 3.1, окрім цих двох мов, до категорії високопродуктивних також увійшли мови Rust та Haskell, які демонструють хорошу оптимізацію при належному використанні. Втім, вибір саме C++ був зумовлений низкою практичних переваг, які роблять її надзвичайно зручною для реалізації ефективних обчислювальних рішень у контексті даного дослідження.

Зокрема, мова C++ надає розробнику можливість безпосередньо працювати з оперативною пам’яттю, що відкриває доступ до низькорівневого керування ресурсами – важливої властивості при реалізації оптимізованих рішень для задач із великим навантаженням на апаратну частину. Крім того, C++ має розвинену стандартну бібліотеку, яка включає засоби для реалізації багатопотокових

обчислень, синхронізації потоків і роботи з паралельними структурами, що є критичним у задачах, орієнтованих на тестування продуктивності CPU.

Одним із ключових інструментів стандартної бібліотеки C++ для реалізації паралелізму є клас std::thread. Цей механізм забезпечує платформонезалежний інтерфейс для створення та керування потоками, виступаючи як обгортка над низькорівневими системними викликами конкретної операційної системи. З точки зору архітектури, потік вважається найменшою одиницею планування виконання програми, і нерідко його характеризують як «легковаговий процес». Потоки завжди існують у межах певного процесу і не можуть бути функціонально незалежними від нього.

Кожен потік має окремий програмний лічильник, власний стек і набір регістрів, що дозволяє виконувати задачі паралельно з іншими потоками в межах того самого процесу. Основною перевагою багатопоточності є мінімізація накладних витрат на переключення контексту та ефективна організація взаємодії між паралельними блоками обчислень (рис. 3.3).

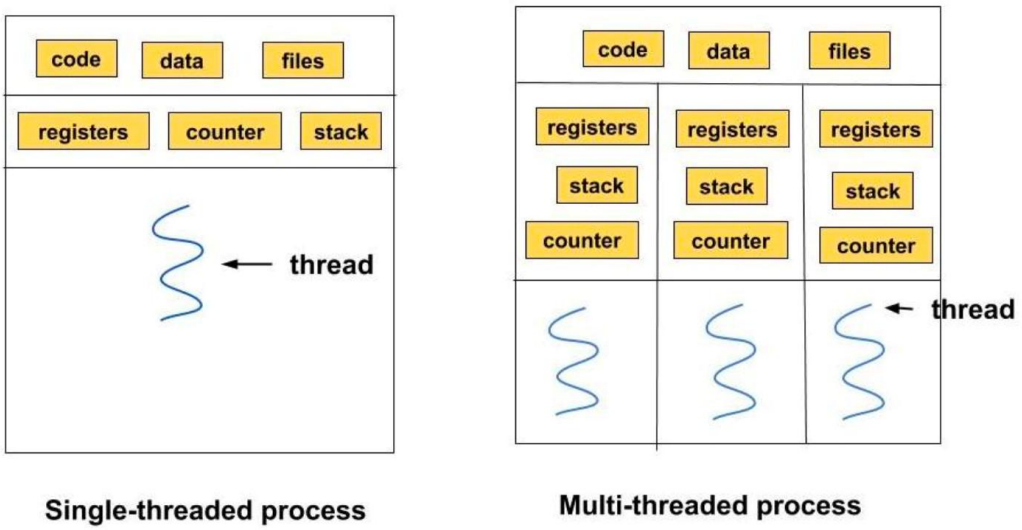


Рисунок 3.3 – Структурна модель організації потоків в операційній системі

У мові програмування C++ існує кілька способів ініціалізації потоків за допомогою класу std::thread, що дозволяє гнучко організовувати багатопоточне виконання в залежності від потреб програми.

1. Виклик попередньо визначеної функції Найбільш прямолінійний спосіб запуску потоку – це передача посилання на існуючу функцію з відповідними аргументами:

```
void do_some_work(int a, int b) {
    std::cout << "Сума: " << a + b << std::endl;
}

std::thread my_thread(do_some_work, 1, 2);
my_thread.join(); // Обов'язково очікуємо завершення потоку
```

Рисунок 3.4 – Прямолінійний спосіб запуску потоку в бібліотеці std::endl

Такий підхід зручний у випадках, коли логіка обчислення вже реалізована у вигляді функції.

2. Використання лямбда-функції Іншим поширеним способом є створення лямбда-виразу, який дозволяє швидко визначити логіку виконання "на місці":

```
std::thread t([](){
    std::cout << "Потік виконує лямбда-функцію" << std::endl;
});
t.join();
```

Рисунок 3.5 – Потік виконує лямбда-функцію в бібліотеці std::endl

Цей варіант особливо корисний у випадках, коли функція є короткою або коли необхідно передати локальні змінні в область дії потоку (через захоплення значення або посилання).

3. Запуск функціонального об’єкта (функтору) Ще один ефективний підхід – створення окремого класу, який реалізує функціональний об’єкт (overload оператора ()):

```
class functor_task {
public:
    void operator()() const {
        do_task_1();
        do_task_2();
    }

    void do_task_1() const {
        std::cout << "Задача 1 у потоці" << std::endl;
    }

    void do_task_2() const {
        std::cout << "Задача 2 у потоці" << std::endl;
    }
};

functor_task f;
std::thread my_thread(f);
my_thread.join();
```

Рисунок 3.6 – Реалізація функціонального обєкта в бібліотеці std::endl

Такий підхід зручний у разі складної логіки, коли необхідно інкапсулювати функціональність у класі, зберігаючи при цьому читабельність коду.

У C++ важливим аспектом керування життєвим циклом потоку є методи join() та detach(), які визначають подальшу поведінку програми щодо запущених потоків.

Метод join() використовується для синхронного очікування завершення потоку. Викликавши join() для об’єкта std::thread, головний потік блокується доти, доки не завершиться виконання відповідного потоку. Варто зауважити, що метод join() може бути викликаний лише один раз, і повторне звернення до нього спричинить помилку виконання. Цей підхід є типовим для ситуацій, коли результати роботи потоку необхідні для подальших обчислень у головному потоці програми.

Альтернативний варіант – використання методу detach(), який від’єднує потік від об’єкта std::thread, дозволяючи йому продовжувати виконання у фоновому режимі без зв’язку з основною програмною логікою. У такому випадку неможливо дочекатися завершення потоку або отримати до нього доступ через join(). Вся відповідальність за завершення потоку та очищення відповідних ресурсів переходить до бібліотеки виконання C++ Runtime, яка забезпечує правильну деалокацію після завершення фонової задачі.

Від’єднані потоки в деяких випадках називають демон-потоками, аналогічно до концепції daemon-процесів в UNIX-системах, які функціонують у фоновому режимі без безпосередньої взаємодії з користувачем або головним процесом.

Описані механізми є базовими концептами управління багатопоточністю у C++, які мають принципове значення у реалізації експериментального програмного коду в рамках цієї роботи. Зокрема, у практичній частині буде реалізовано приклади, що передбачають як синхронну координацію потоків через join(), так і асинхронне виконання фонових завдань за допомогою detach().

Аналіз вибору платформи для реалізації обчислень на GPU

Для реалізації тестування на графічному процесорі в рамках даного дослідження була обрана платформа CUDA (Compute Unified Device Architecture), яка є інструментом компанії NVIDIA, спеціально створеним для виконання паралельних обчислень. Основна перевага використання CUDA полягає в можливості реалізації обчислень загального призначення (GPGPU) на GPU без використання традиційних графічних API, що значно розширює діапазон задач, які можуть бути ефективно виконані на відеокарті.

Модель програмування CUDA пропонує розробнику зручну абстракцію для керування архітектурою пристрою, базовану на розподілі між "хостом" (host) – центральним процесором системи, та "пристроєм" (device) – графічним процесором. Відповідно, вся оперативна пам’ять поділяється на пам’ять хоста, до якої має доступ CPU, та пам’ять пристрою, що використовується GPU для зберігання даних під час обчислень.

CUDA реалізує неоднорідну (гетерогенну) модель обчислень, яка передбачає розподіл завдань між різними типами процесорів – кожен з яких відповідає за ті задачі, які виконує найефективніше. Таким чином, CPU може виконувати послідовні або менш паралелізовані операції, тоді як GPU бере на себе значні обсяги однотипних обчислень, які легко масштабуються за рахунок великої кількості ядер. Це дозволяє досягти суттєвого приросту продуктивності у задачах з високим рівнем паралелізму, як-от обробка матриць, векторних масивів чи елементних трансформацій даних.

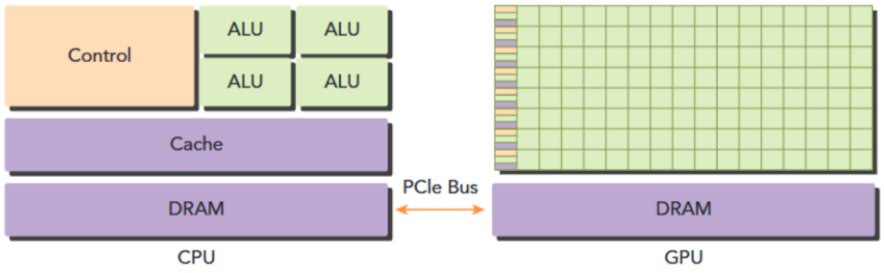


Рисунок 3.7 – Архітектурна схема взаємодії хоста та пристрою у моделі CUDA (джерело: [умовно – CUDA Developer Guide або авторська ілюстрація])

Для реалізації обчислень на графічному процесорі за допомогою платформи CUDA необхідно дотримуватись певної послідовності дій, яка є фундаментальною для запуску будь-якої CUDA-програми. Архітектура взаємодії між хостом (CPU) та пристроєм (GPU) базується на трьох ключових етапах:

1. Передача вхідних даних з оперативної пам’яті хоста (CPU) у пам’ять пристрою (GPU).
2. Виконання інструкцій на графічному процесорі шляхом запуску спеціальної функції, що реалізує обчислення.
3. Зворотне копіювання результатів із пам’яті GPU назад у пам’ять CPU для подальшого аналізу або обробки.

Ця схема забезпечує повний цикл взаємодії між центральним та графічним процесорами, що ілюструється на рисунку 3.8.

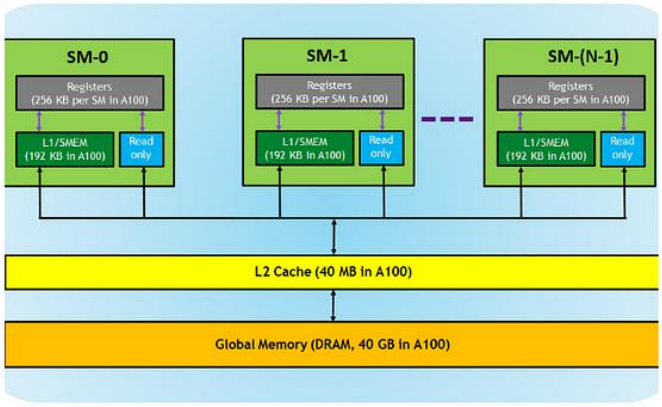


Рисунок 3.8 – Узагальнена схема етапів виконання обчислювального процесу в середовищі CUDA

Центральним елементом обчислень у CUDA є так зване "кернел-функція" (kernel) – це функція, що виконується безпосередньо на графічному процесорі. Особливістю кернелу є те, що він запускається паралельно у великій кількості потоків, де кожен потік обробляє окремий фрагмент даних. Така модель обчислень відома як потокова (stream) модель обробки даних, у межах якої всі елементи вхідного масиву можуть бути оброблені незалежно один від одного.

Код програм пишеться за допомогою розширення мови C, відомого як CUDA C, що дозволяє оголошувати спеціальні кернел-функції, вказувати кількість потоків для виконання і керувати пам’яттю пристрою. Компіляція відбувається за допомогою спеціального компілятора nvcc, який дозволяє транслювати код у вигляд, сумісний із архітектурою NVIDIA GPU.

Таким чином, описана модель реалізує повноцінну підтримку масивно-паралельних обчислень, що є основною перевагою використання CUDA у задачах високої обчислювальної складності.

У контексті архітектури CUDA кожна одиниця виконання кернел-функції відповідає апаратному ядру графічного процесора, яке має назву потоковий процесор (Streaming Processor, SP). Ці елементарні обчислювальні блоки об’єднуються у більші функціональні одиниці – потокові мультипроцесори (Streaming Multiprocessor, SM). У свою чергу, кілька мультипроцесорів формують кластери текстурних процесорів (Texture Processor Clusters, TPC), що є характерною рисою апаратної структури GPU-пристроїв від NVIDIA.

Для узгодження програмного коду з цією апаратною структурою, у моделі CUDA передбачено абстрактну ієрархію виконуваних елементів, яка включає: потік (thread), блок потоків (block) та сітку блоків (grid). Усі потоки, які створюються в результаті одного запуску кернела, формують єдину сітку, яка, в свою чергу, складається з багатьох блоків. Усі потоки в межах однієї сітки мають доступ до глобальної пам’яті пристрою (рис. 3.9).

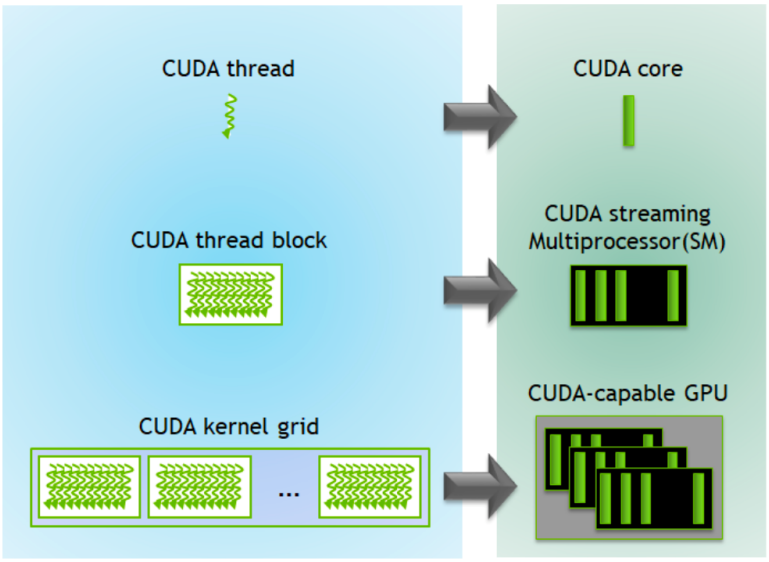


Рисунок 3.9 – Ієрархічна модель виконання CUDA-програми: сітка, блоки та потоки

Кожен блок складається з множини потоків, які можуть взаємодіяти між собою через спільну локальну пам’ять блока. Така структура дозволяє гнучко масштабувати обчислення, розподіляючи їх між великою кількістю паралельних потоків. Сітка може бути одновимірною, двовимірною або тривимірною, що робить модель надзвичайно придатною для роботи з векторами, матрицями або тривимірними масивами (рис. 3.10).

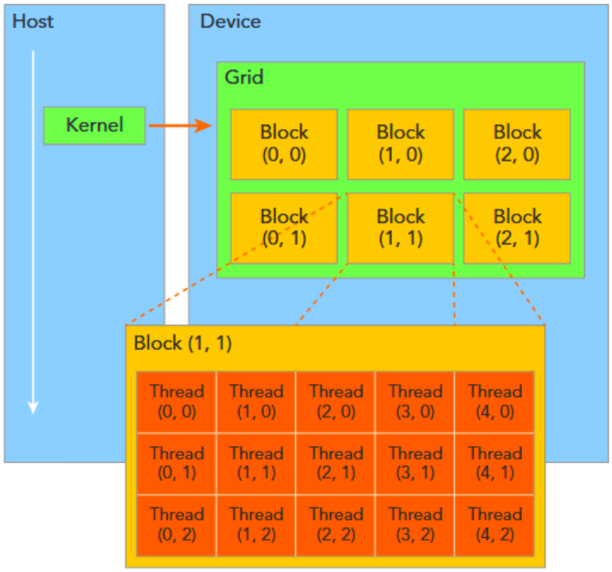


Рисунок 3.10 – Візуалізація сітки блоків і потоків у CUDA та їх ідентифікація

Для звертання до конкретного потоку або блока у CUDA передбачені вбудовані тривимірні змінні:

– threadIdx. Використовується для індексації окремих потоків усередині блока;
– blockIdx. Відповідає за індексацію блока всередині сітки.

Ці змінні дозволяють автоматично визначати позицію потоку або блока у багатовимірному обчислювальному просторі, що значно спрощує реалізацію паралельної логіки при обробці структурованих даних, зокрема векторів і матриць.

Завдяки доступу до тривимірних координат threadIdx та blockIdx, у CUDA можливо гнучко організувати обробку великих обсягів даних, розбиваючи їх на окремі фрагменти, кожен з яких обробляється відповідним потоком. Зокрема, структура threadIdx.x, threadIdx.y, threadIdx.z дозволяє адресувати окремі потоки всередині блока, а blockIdx.x, blockIdx.y, blockIdx.z – ідентифікувати конкретні блоки у межах всієї сітки. Це дає змогу реалізовувати ефективну паралельну обробку векторів, матриць та об’ємних масивів. Однак, архітектура CUDA накладає апаратне обмеження – не більше ніж 1024 потоки на блок, що потрібно враховувати при проектуванні алгоритмів.

Важливим аспектом ефективного використання CUDA є організація ієрархії пам’яті у графічному процесорі. Вона представлена у вигляді п’яти основних рівнів, кожен з яких виконує певну роль (рис. 3.11):

1. Регістри – найшвидший тип пам’яті, що є приватним для кожного потоку. Компілятор самостійно визначає, які змінні слід розмістити в регістрах.

2. Shared Memory (SMEM) – швидка локальна пам’ять, спільна для всіх потоків у межах одного блока. Вона реалізується як частина кешу L1 і забезпечує обмін даними між потоками блока з мінімальними затримками.

3. Read-only пам’ять – включає кеш інструкцій, постійну пам’ять (constant memory), текстурну пам’ять та кеш тільки для читання. Призначена для оптимізованого доступу до незмінних даних.

4. Кеш L2 – глобальний кеш, який використовується всіма мультипроцесорами. Забезпечує доступ до пам’яті між блоками, працюючи як посередник між глобальною та локальною пам’яттю.

5. Глобальна пам’ять – основна пам’ять GPU (DRAM), яка слугує кадровим буфером для зберігання даних, що надходять із хоста або передаються до нього.

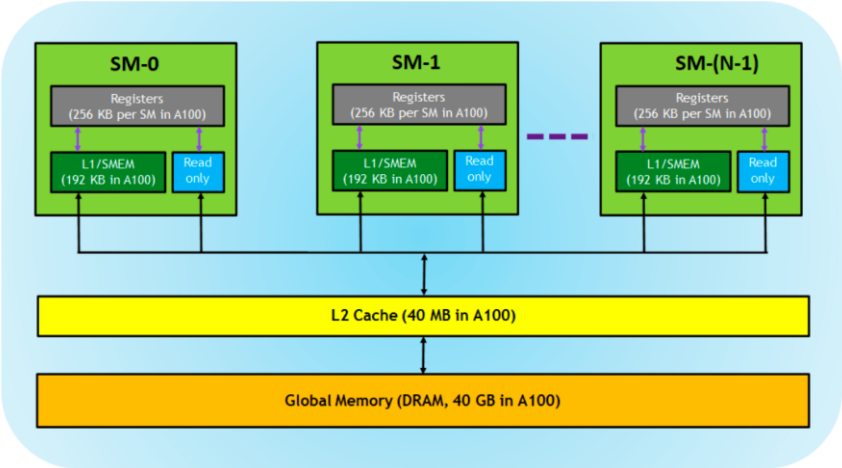


Рисунок 3.11 – Ієрархія організації пам’яті графічного процесора з підтримкою CUDA

Ще одним важливим параметром, який характеризує GPU у контексті обчислень, є обчислювальні можливості (Compute Capability). Це характеристика, що визначає функціональні можливості та підтримувані апаратні інструкції конкретної відеокарти. Версія Compute Capability подається у форматі X.Y, де X – основна версія архітектури, а Y – другорядне уточнення, що відповідає поступовому вдосконаленню мікроархітектури.

Наприклад:

- GPU GeForce 360M (2010 рік) має Compute Capability 1.0;
- GPU GeForce GTX 750 (2014 рік) – 5.0;
- GPU Quadro RTX (2018 рік) – 7.5.

Ця інформація використовується під час компіляції та виконання програм для визначення, які функції доступні на конкретному пристрої, і дозволяє коректно адаптувати код до наявної архітектури GPU.

Таким чином, платформа CUDA виступає потужним засобом для реалізації обчислень загального призначення (GPGPU), що виходять за межі традиційного

графічного рендерингу. Вона надає розробникам інтегрований набір інструментів – власну мову програмування (CUDA C), компілятор (nvcc) та чітко визначену модель виконання, орієнтовану на ефективну взаємодію з апаратною архітектурою GPU. Запровадження цієї платформи стало ключовим етапом у розвитку паралельних обчислень, зробивши можливим масштабування рішень у сфері інженерного моделювання, обробки великих даних, штучного інтелекту та інших ресурсоємних задач.

Завдяки своїм гнучким механізмам розподілу обчислень, підтримці багаторівневої пам’яті та можливості точного контролю над архітектурою виконання, CUDA на сьогоднішній день залишається одним із найефективніших інструментів для реалізації високопродуктивних неграфічних обчислень на базі відеокарт.

3.3 Бенчмарки та аналіз ефективності GPU в порівнянні з CPU

Постановка експерименту та характеристика обраних пристроїв

Метою даного етапу дослідження є проведення порівняльного аналізу ефективності виконання обчислень на графічному процесорі (GPU) та центральному процесорі (CPU). Аналіз здійснюється на прикладі базових математичних операцій, які широко застосовуються в технічних, наукових і прикладних обчисленнях:

- скалярний добуток векторів;
- множення матриць;
- додавання матриць;
- додавання матриць із коефіцієнтом (kA + B);
- ділення матриці на число.

Як було встановлено у попередніх розділах, розмір вхідних даних та кількість паралельних потоків виступають критичними факторами, що визначають

ефективність обчислень. Саме тому у межах цього дослідження буде здійснено серію тестів з поступовим масштабуванням обсягів даних:

– для векторних операцій – від 256 до 81 920 000 елементів;
– для матричних – від 40^2 до 300^2 елементів (тобто до понад 90 000 елементів у кожній матриці).

Кожне завдання буде реалізовано у двох варіантах: з використанням багатопотокової реалізації на C++ для CPU та паралельного кернелу на CUDA для GPU, що дозволить провести об’єктивне порівняння обчислювальної потужності пристроїв.

Для проведення тестування обрано два різних центральних процесори, представлених у ноутбуках:

– Intel Core i7-10850H. Сучасний високопродуктивний CPU з великою кількістю ядер та потоків, призначений для професійних задач і мобільних робочих станцій (рис. 3.12);
– Intel Core i5-7200U. Більш енергоефективний та бюджетний варіант, характерний для офісних або побутових систем (рис. 3.13).

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
Cache	12 MB Intel® Smart Cache
Bus Speed	8 GT/s
TDP	45 W
Configurable TDP-down	35 W

Рисунок 3.12 – Технічні характеристики процесора Intel Core i7-10850H

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
Cache	3 MB Intel® Smart Cache
Bus Speed	4 GT/s
TDP	15 W
Configurable TDP-up Base Frequency	2.70 GHz
Configurable TDP-up	25 W
Configurable TDP-down Base Frequency	800 MHz
Configurable TDP-down	7.5 W

Рисунок 3.13 – Технічні характеристики процесора Intel Core i5-7200U

Для проведення обчислень на графічних процесорах у рамках дослідження було обрано три моделі GPU різного класу, що дозволяє оцінити продуктивність паралельних обчислень як на споживчому, так і на професійному рівнях.

1. NVIDIA GeForce 940MX – представник мобільного сегмента, вбудований у персональний ноутбук, призначений для побутових завдань та обмежених обчислювальних сценаріїв.

2. NVIDIA Tesla P100 – серверна відеокарта з високою обчислювальною потужністю, яка надається у середовищі Google Colab, оптимізована для наукових досліджень та глибокого навчання.

3. NVIDIA GeForce RTX 3080 Ti – потужна відеокарта з підтримкою новітньої архітектури Ampere, що використовується на віддаленій машині для високопродуктивних розрахунків у реальному часі.

Codename	N16S-GTR-B/S
Architecture	Maxwell
Pipelines	384 - unified
TMUs	42
ROPs	16
Core Speed	1122 - 1242 (Boost) MHz
Memory Speed	4000 effective = 1253 MHz
Memory Bus Width	64 Bit
Memory Type	DDR3, GDDR5
Max. Amount of Memory	4 GB
Shared Memory	no
Memory Bandwidth	40.1 GB/s
API	DirectX 12 (FL 11_0), Shader 5.0, OpenGL 4.5
Power Consumption	15-25 Watt
Transistor Count	1.9 Billion
technology	28 nm
PCIe	3.0
Features	GPU Boost 2.0, Optimus, PhysX, CUDA, GeForce Experience, GameWorks
Notebook Size	medium sized
Date of Announcement	10.03.2016 = 3320 days old
Link to Manufacturer Page	www.geforce.com

Рисунок 3.14 – Технічні характеристики GPU NVIDIA GeForce 940MX

TECHNICAL SPECIFICATIONS	**Tesla P100**	
Peak double-precision floating point performance (board)	4.7 Tflops	
Peak single-precision floating point performance (board)	9.3 Tflops	
Number of GPUs	1x GP100	
Number of CUDA cores	3584	
Memory size per board	12GB HBM2	16GB HBM2
Memory Interface	3072-bit	4096-bit
Memory bandwidth for board (ECC off)2	540 Gbytes/sec	720 Gbytes/sec
Thermal Solution	Passive	
Max Power Consumption	250W	
Form Factor	4.376"H x 10.5" L	
System	NVIDIA Certified Servers	

Рисунок 3.15 – Технічні характеристики GPU NVIDIA Tesla P100

Model	TUF Gaming GeForce RTX" 3080 Ti
Graphic Engine	NVIDIA® GeForce RTX" 3080 Ti
Bus Standard	PCI Express 4.0
OpenGL	0penGL4.6
Video Memory	12GB GDDR6X (Boost Clock) Gaming mode: 1665 MHzost Clock)
CUDA Core	10240
Memory Speed	19 Gbps
Memory Interface	384-bit
Resolution	Digital Max Resoution 7680 x 4320
Interface	Yes x 2 (Native HDMI 2.1) Yes x 3 (Native DisplayPort 1.4a)

Рисунок 3.16 – Технічні характеристики GPU NVIDIA GeForce RTX 3080 Ti

У цьому підрозділі розглядаються особливості реалізації обробки матриць у рамках експерименту як на CPU, так і на GPU. У всіх реалізаціях матриці представляються у вигляді одновимірних масивів, що дозволяє спростити роботу з пам’яттю, уникнути зайвого накладного коду для доступу до елементів і забезпечити неперервне розміщення даних у пам’яті. У контексті квадратних матриць, які застосовуються у даній роботі, доступ до елементів здійснюється шляхом обчислення зсуву за допомогою індексу рядка та розмірності:

element[i][j] ⇨ array[i * N + j]

Така адресація забезпечує ефективну алокацію і доступ до елементів, зменшуючи ризики фрагментації пам’яті та підвищуючи швидкодію обчислень. Завдяки повному розділенню обробки кожного елемента, не виникає потреби у синхронізації потоків – усі обчислення можуть бути виконані незалежно один від одного, що є ідеальним випадком для паралельної реалізації. На центральному

процесорі обробка матриці здійснюється шляхом розподілу діапазонів рядків між окремими потоками, де кожен потік послідовно проходить усі стовпці у межах своєї області відповідальності. Такий підхід ілюструється на рисунку 3.17.

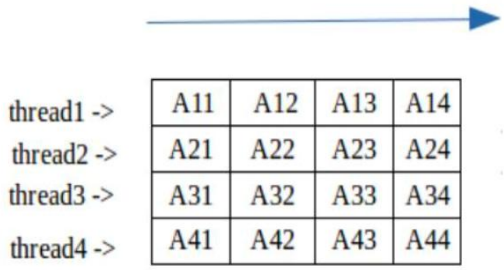


Рисунок 3.17 – Схема обходу матриці у багатопотоковій CPU-реалізації: розподіл рядків по потоках

У реалізації на CUDA індексація елементів матриці значно спрощується завдяки використанню вбудованих змінних blockIdx, threadIdx, blockDim, що дозволяють емулювати двовимірну структуру сітки потоків, яка відповідає структурі самої матриці. Таким чином, кожному потоку автоматично відповідає унікальний елемент матриці, що дає змогу досягти максимальної простоти та ефективності реалізації без додаткових розрахунків.

Визначення оптимальної кількості потоків при виконанні задач на CPU Перед початком безпосереднього порівняння CPU та GPU у виконанні обчислювальних задач, було проведено серію попередніх експериментів, метою яких було визначення оптимального режиму багатопотокової роботи центрального процесора. Для цього реалізовано тестування з різною кількістю активних потоків на кожному з двох CPU: Intel Core i7-10850H та Intel Core i5-7200U.

Бенчмарк проводився на основі серії тестів, у межах яких виконувалась одна з обраних задач (наприклад, додавання або множення матриць), а кількість потоків поступово змінювалася. Усі заміри часу виконання проводилися у мілісекундах, а вхідні дані формувались у вигляді квадратної матриці розміром size × size. Кожен експеримент повторювався 10 разів, після чого обчислювалося середнє значення часу виконання, що дозволяє нівелювати випадкові флуктуації та підвищити достовірність результатів.

Результати тестування наведено на рисунках 3.18–3.19. На кожному з графіків представлено залежність часу виконання від кількості потоків при різних розмірах вхідних матриць. Аналіз цих графіків дозволяє зробити висновки щодо того, за яких конфігурацій паралелізації досягається мінімальний час виконання для кожного процесора.

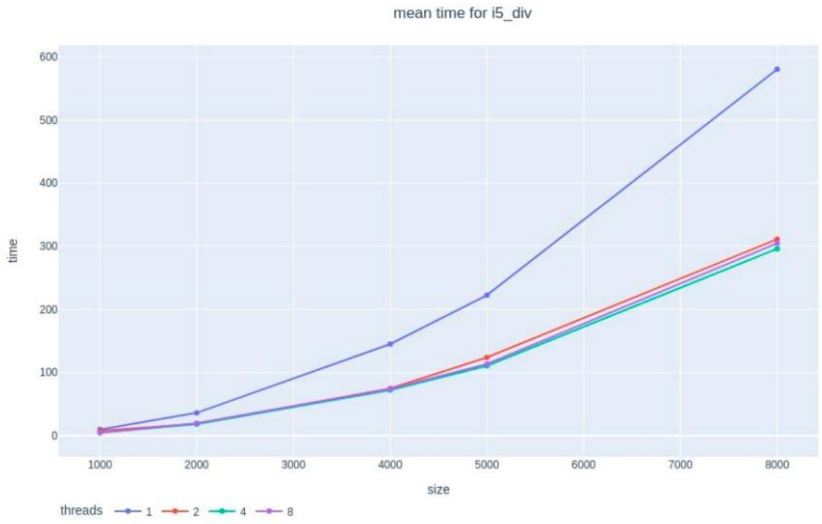


Рисунок 3.18 – Середній час виконання операції ділення матриці на скаляр на процесорі Intel Core i5-7200U (усереднено за 10 запусків)

Як видно з рисунка 3.18, найменший середній час виконання операції ділення матриці на скаляр на процесорі Intel Core i5-7200U було досягнуто при використанні чотирьох потоків. Така конфігурація виявилась оптимальною для даної задачі, забезпечивши найвищу продуктивність. Водночас, результати, отримані при використанні двох та восьми потоків, також виявилися достатньо близькими до мінімального значення, що свідчить про відносну стабільність ефективності у межах цього діапазону рівнів паралелізації.

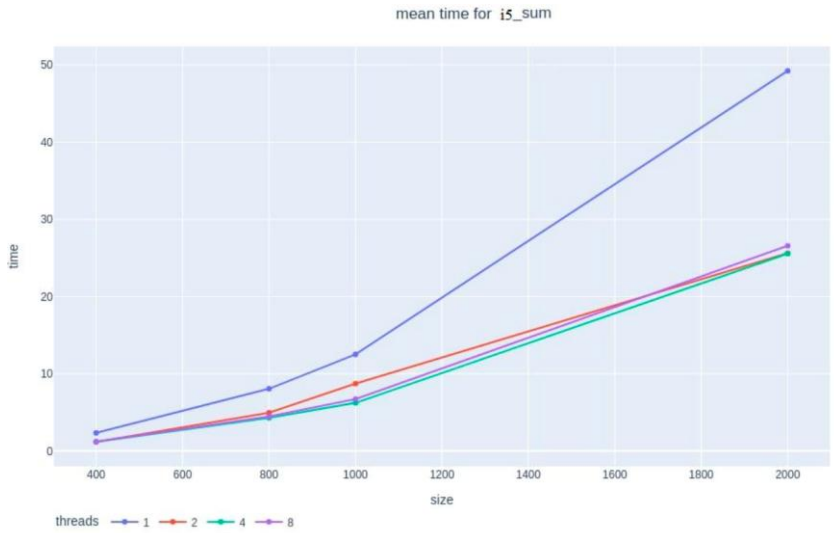


Рисунок 3.19 – Середній час виконання операції додавання матриць на процесорі Intel Core i5-7200U (усереднено за 10 запусків)

Як свідчать результати, наведені на рисунку 3.19, характер зміни часу виконання операції додавання матриць на Intel Core i5-7200U є подібним до спостереженого для операції ділення (див. рис. 3.18). Оптимальним варіантом знову виявилось використання чотирьох потоків, що забезпечило найнижчий середній час виконання. Водночас, двопотокова та восьмипотокова конфігурації також продемонстрували порівняно високий рівень ефективності, що підтверджує стійкість продуктивності цього процесора в межах помірної багатопоточності.

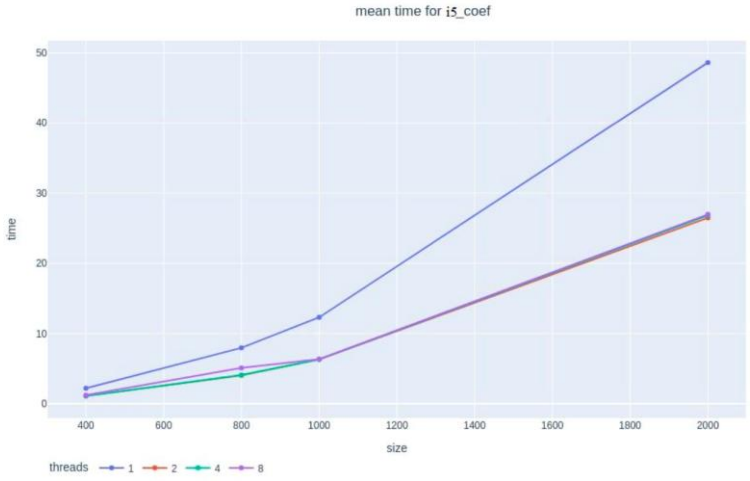


Рисунок 3.20 – Середній час виконання операції додавання матриць з коефіцієнтом (kA + B) на процесорі Intel Core i5-7200U (усереднено за 10 запусків)

Згідно з результатами, поданими на рисунку 3.20, виконання операції додавання матриць з коефіцієнтом (kA + B) демонструє аналогічну залежність продуктивності від кількості потоків, як і у попередніх випадках. Найменший час обчислень було зафіксовано при використанні чотирьох потоків, що підтверджує сталість цієї конфігурації як оптимальної для i5-7200U у задачах додавання. Як і раніше, конфігурації з двома та вісьмома потоками також забезпечують результати, наближені до оптимальних, що свідчить про ефективне масштабування задач такого типу на цьому процесорі.

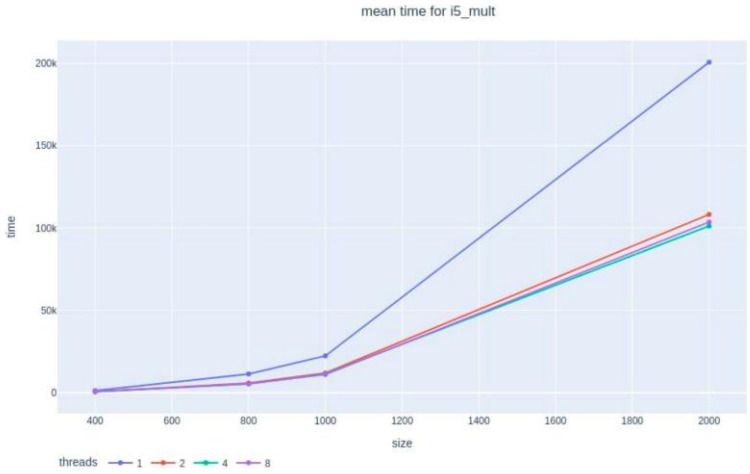


Рисунок 3.21 – Середній час виконання операції множення матриць на процесорі Intel Core i5-7200U (усереднено за 10 запусків)

Як видно з рисунка 3.21, у випадку множення матриць спостерігається дещо інша динаміка, порівняно з попередніми операціями. Найкращий результат з точки зору часу виконання знову був досягнутий при використанні чотирьох потоків, однак відмінності між конфігураціями стали більш помітними.

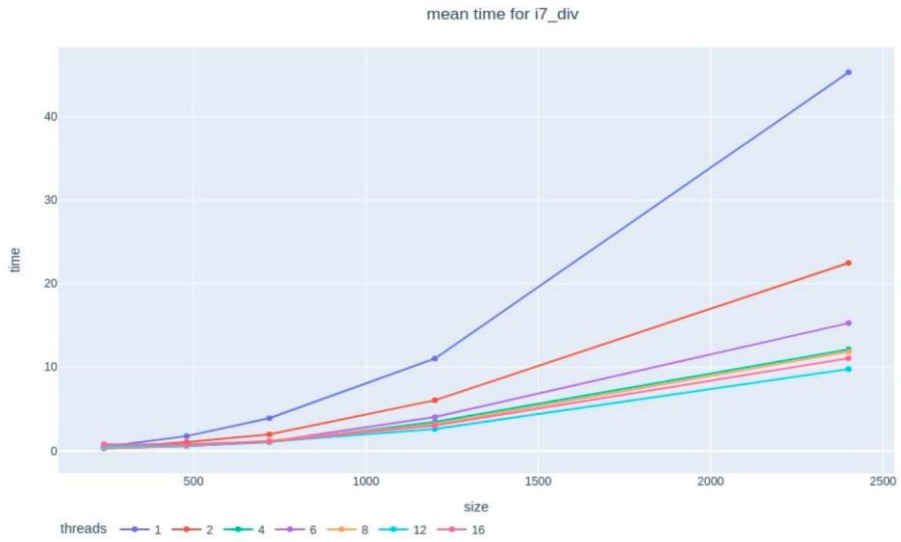


Рисунок 3.22 – Середній час виконання операції ділення матриці на скаляр на процесорі Intel Core i7-10850H (усереднено за 10 запусків)

Як демонструє рисунок 3.22, при виконанні операції ділення матриці на скаляр на процесорі Intel Core i7-10850H найнижчий середній час було досягнуто при використанні восьми потоків. Ця конфігурація виявилась найбільш ефективною серед протестованих, що узгоджується з архітектурними особливостями i7, зокрема – більшою кількістю фізичних та логічних ядер порівняно з i5. Слід також відзначити, що приріст продуктивності при переході від чотирьох до восьми потоків є суттєвим, що підтверджує добру масштабованість цього CPU в умовах багатопотокових обчислень.

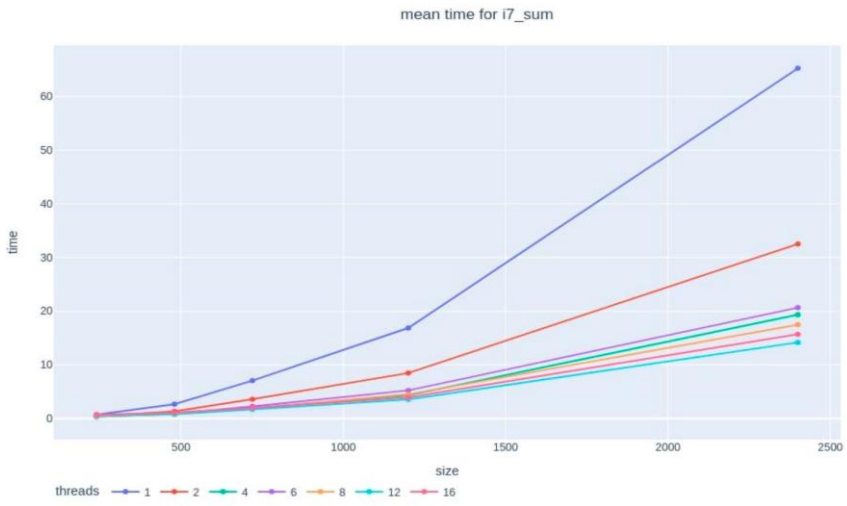


Рисунок 3.23 – Середній час виконання операції додавання матриць на процесорі Intel Core i7-10850H (усереднено за 10 запусків)

На рисунку 3.23 відображено результати тестування операції додавання матриць на Intel Core i7-10850H. Найвища продуктивність зафіксована при використанні восьми потоків, що свідчить про ефективну реалізацію паралелізму на цьому процесорі. Конфігурації з чотирма та шістнадцятьма потоками демонструють також високі результати, проте подальше збільшення кількості потоків не завжди призводить до значного покращення – це може бути пов’язано з накладними витратами на синхронізацію та розподіл задач при надмірній кількості паралельних гілок.

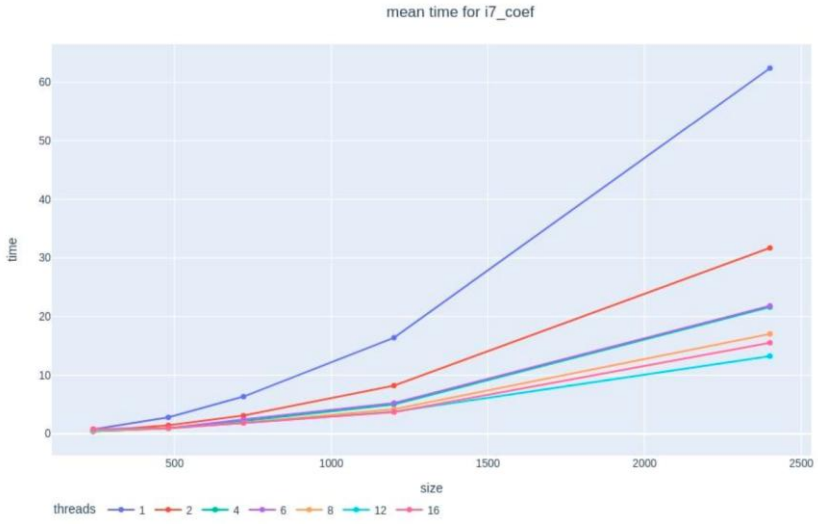


Рисунок 3.24 – Середній час виконання операції додавання матриць з коефіцієнтом (kA + B) на процесорі Intel Core i7-10850H

Як показано на рисунку 3.24, при виконанні операції додавання матриць з коефіцієнтом на Intel Core i7-10850H оптимальним знову виявився восьмипотоковий режим роботи. Це підтверджує загальну закономірність, встановлену у попередніх тестах: i7-10850H найбільш ефективно реалізує паралельні обчислення при середньому рівні навантаження, яке відповідає кількості фізичних або логічних ядер. Як і у випадку з базовим додаванням, зростання кількості потоків понад оптимальне значення не забезпечує лінійного покращення, а в деяких випадках навіть дещо знижує ефективність через перевитрати на міжпотокову координацію.

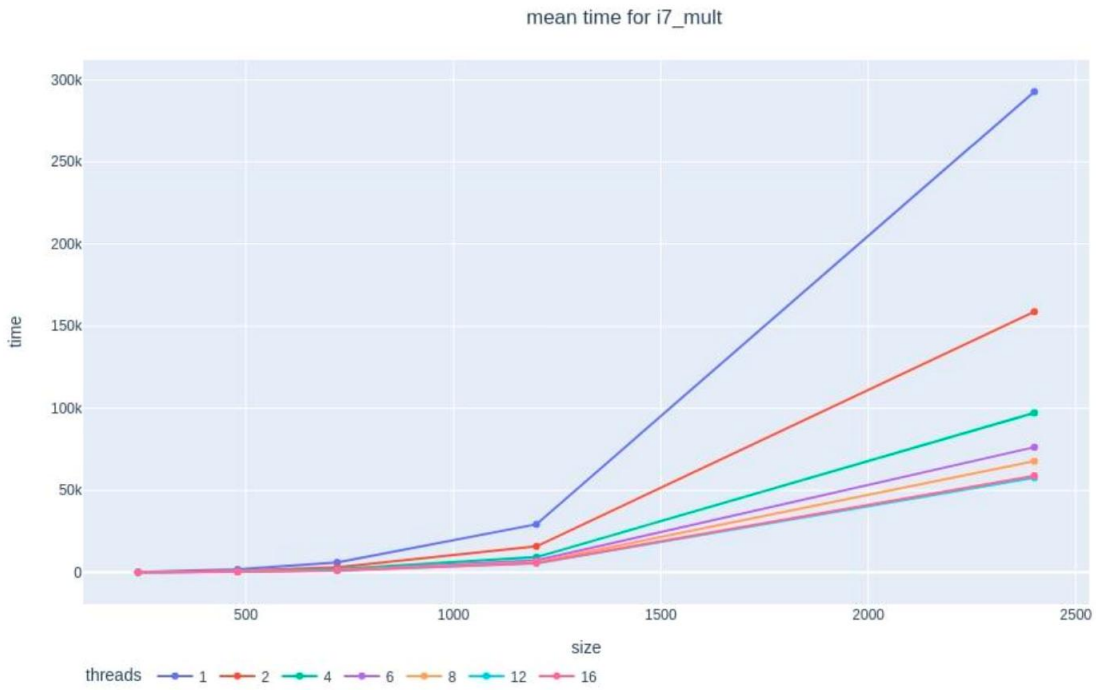


Рисунок 3.25 – Середній час виконання операції множення матриць на процесорі Intel Core i7-10850H (усереднено за 10 запусків)

Аналіз продуктивності багатопотокового виконання задач та реалізація скалярного добутку

На основі результатів, отриманих у ході експериментів, відображених на рисунках 3.22–3.25, можна зробити висновок, що для процесора Intel Core i7-10850H оптимальною конфігурацією є використання 12 потоків. Залучення більшої кількості потоків, ніж фактично передбачено архітектурою CPU, не лише не дає приросту продуктивності, але й створює додаткові накладні витрати на керування надлишковими потоками, що призводить до зниження загальної ефективності.

У випадку з процесором Intel Core i5-7200U, найкращий результат стабільно досягається при використанні чотирьох потоків, однак конфігурації з двома або вісьмома потоками демонструють дуже близькі до оптимальних показники, що свідчить про певну стійкість платформи до варіацій рівня паралелізму у межах зазначеного діапазону.

Окрему увагу в межах цього дослідження заслуговує реалізація операції скалярного добутку векторів, що виконується за формулою:

$$\vec{a} * \vec{b} = a_1 * b_1 + a_2 * b_2 + \cdots + a_3 * b_3 \quad (3.1)$$

Результатом такої операції є єдине скалярне значення, що формується шляхом поступового додавання добутків відповідних елементів векторів. У цьому випадку виникає необхідність синхронізації доступу до спільної змінної, у якій накопичується результат. Для вирішення цієї задачі було використано механізм атомарних змінних.

У рамках реалізації було застосовано шаблонний клас std::atomic<float> із методом fetch_add, який виконує атомарне додавання у конкурентному середовищі. Варто зазначити, що повноцінна підтримка атомарних операцій над типом float з'явилася починаючи зі стандарту C++20, який і був використаний для компіляції коду у даному дослідженні.

Між альтернативами std::mutex та std::atomic вибір було зроблено на користь останнього, оскільки блокування за допомогою м'ютекса створює додаткові затримки, особливо у випадку, коли потоки мають доступ до однієї змінної лише для виконання елементарної операції. М'ютекси доцільніше застосовувати у тих випадках, коли необхідно синхронізувати виконання більшого блоку інструкцій або декількох змінних.

```
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);
    }
}
```

Рисунок 3.26 – Фрагмент реалізації скалярного добутку векторів з використанням std::atomic у багатопотоковому середовищі

Результати виконання операції скалярного добутку векторів з використанням атомарних змінних наведено на рисунках 3.30 та 3.31. У цих графіках відображено середній час виконання задачі на процесорах Intel Core i5-7200U та Intel Core i7-10850H відповідно, залежно від розміру вхідного вектора.

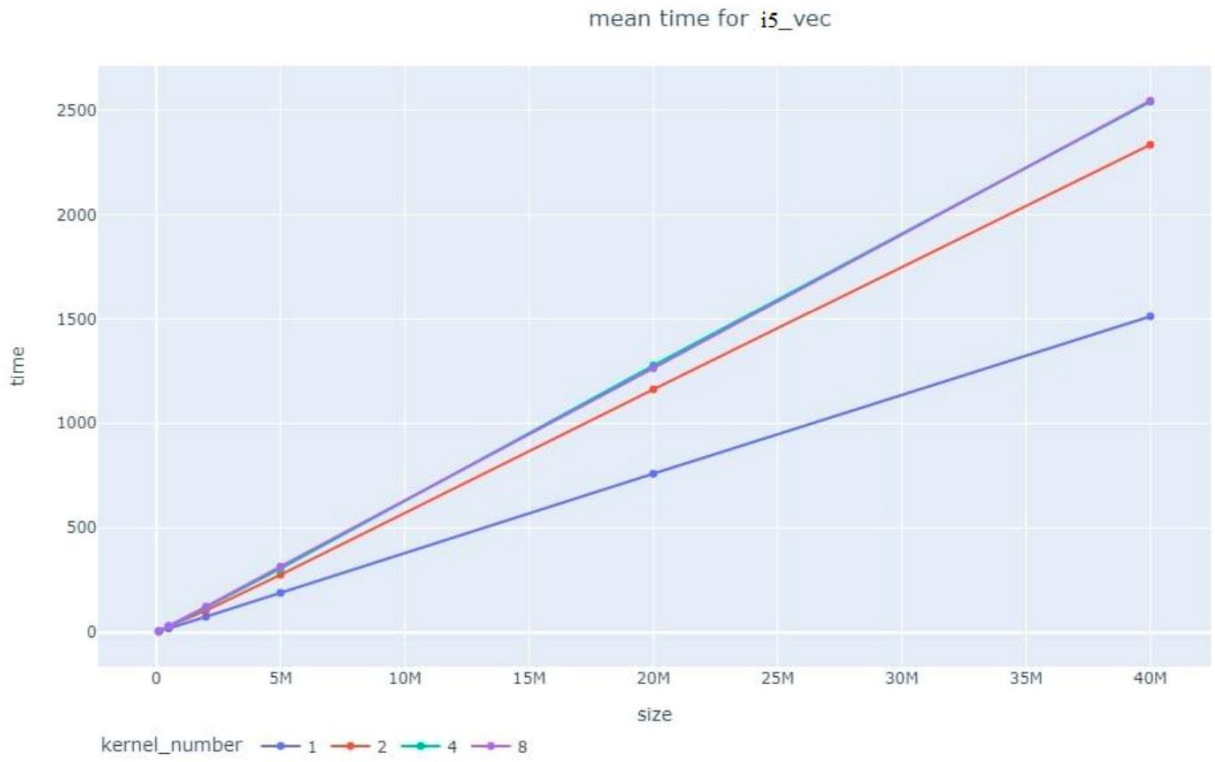


Рисунок 3.27 – Час виконання скалярного добутку на процесорі Intel Core i5-7200U (у мілісекундах, усереднено)

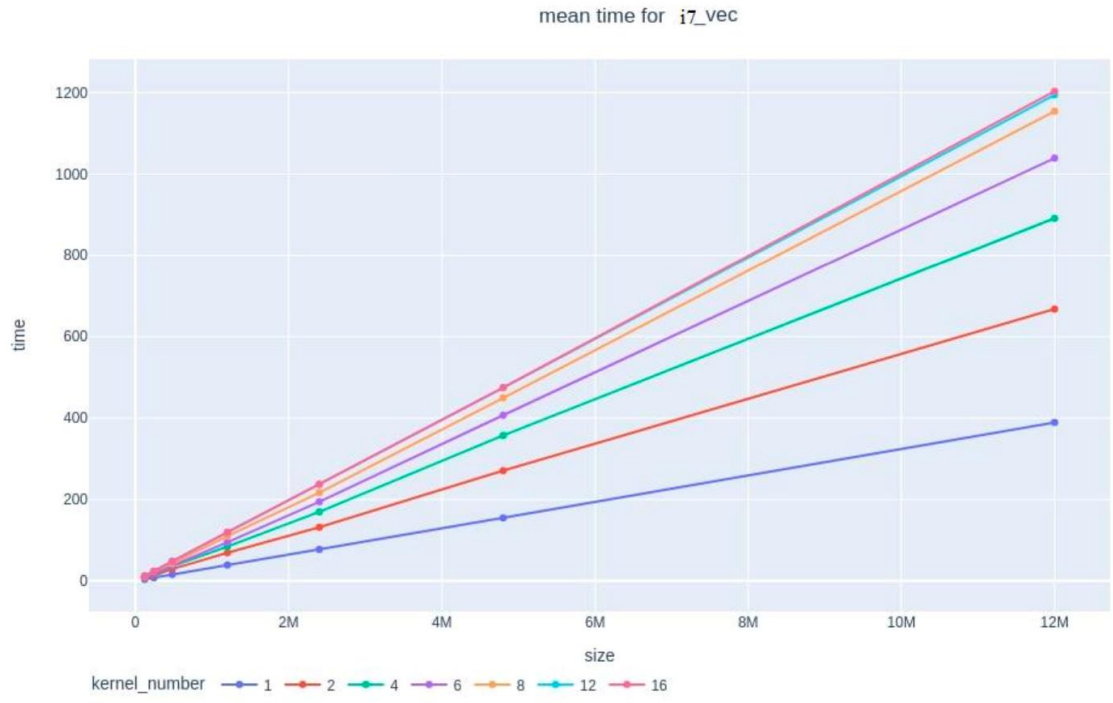


Рисунок 3.28 – Час виконання скалярного добутку на процесорі Intel Core i7-10850H (у мілісекундах, усереднено)

Аналіз результатів, представлених на рисунках 3.27 та 3.28, свідчить про те, що найменший час виконання операції скалярного добутку в обох випадках – як для процесора Intel Core i5-7200U, так і для Intel Core i7-10850H – було досягнуто при використанні лише одного потоку. Цей факт свідчить про те, що навіть сучасні засоби синхронізації, зокрема атомарні операції, не завжди забезпечують приріст продуктивності. У даному випадку їх використання призводило до зниження ефективності через надмірну кількість викликів атомарної операції fetch_add після кожного множення елементів векторів.

Причина зниження продуктивності полягає у тому, що кожен потік після обробки елементів здійснював атомарне оновлення спільної змінної, що створювало додаткове блокування та впливало на швидкодію через часту синхронізацію між потоками.

З метою вирішення цієї проблеми було реалізовано модифікований підхід до багатопоточного обчислення, у якому кожен потік використовує локальну (проміжну) змінну для накопичення результатів. Цей обхід елементів вектора всередині потоку виконується послідовно, без синхронізації. Лише після завершення обробки всього діапазону елементів потік виконує одне атомарне додавання до загального результату. Такий підхід суттєво знижує кількість синхронізаційних операцій і, відповідно, зменшує накладні витрати на доступ до спільного ресурсу.

```
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);
}
```

Рисунок 3.29 – Оптимізована реалізація багатопоточного скалярного добутку з використанням локального накопичення результатів

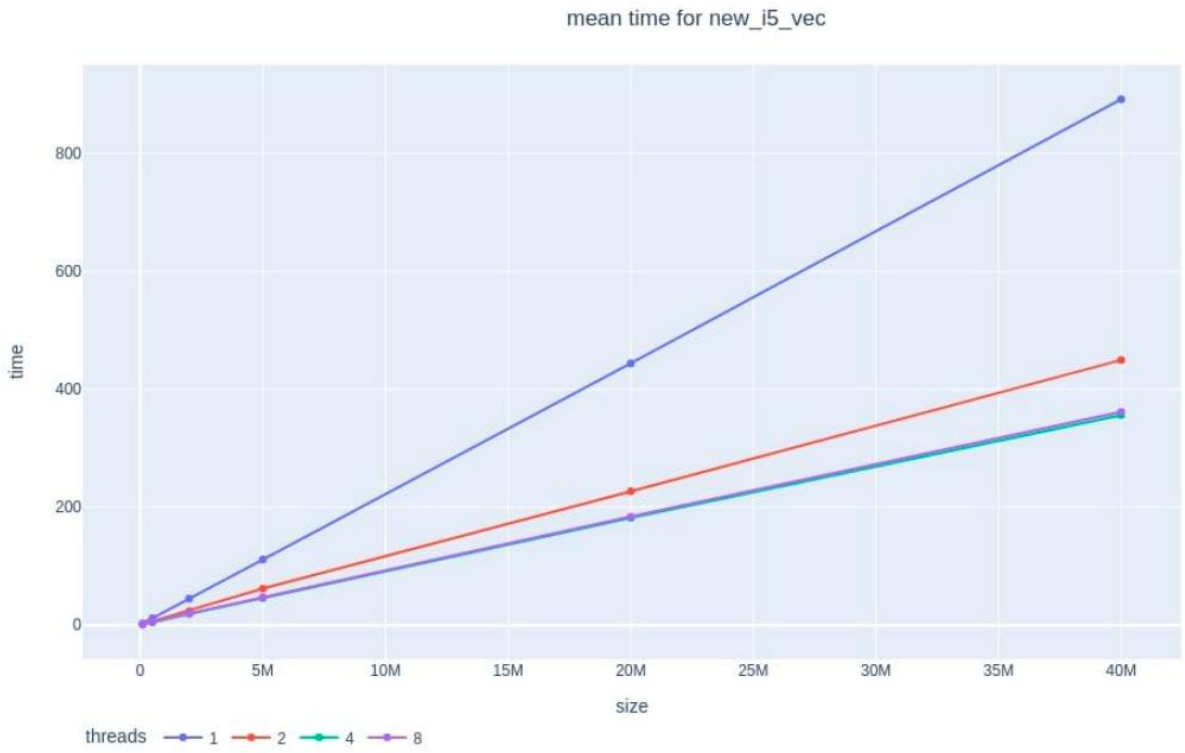


Рисунок 3.30 – Середній час виконання оптимізованої реалізації скалярного добутку на Intel Core i5-7200U

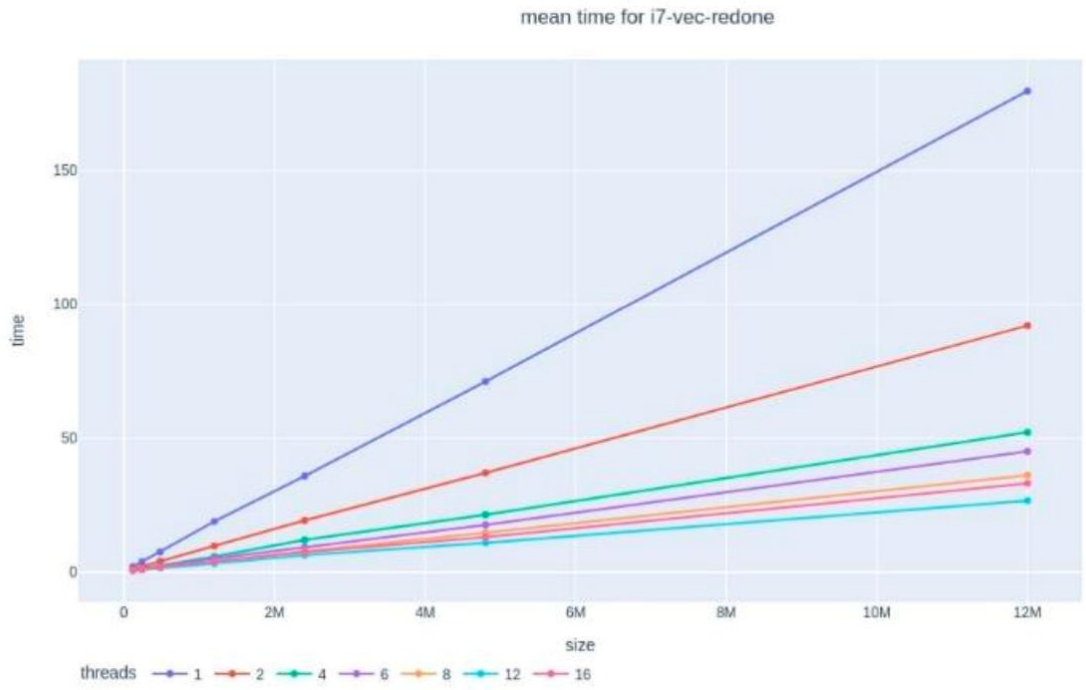


Рисунок 3.31 – Середній час виконання оптимізованої реалізації скалярного добутку на Intel Core i7-10850H

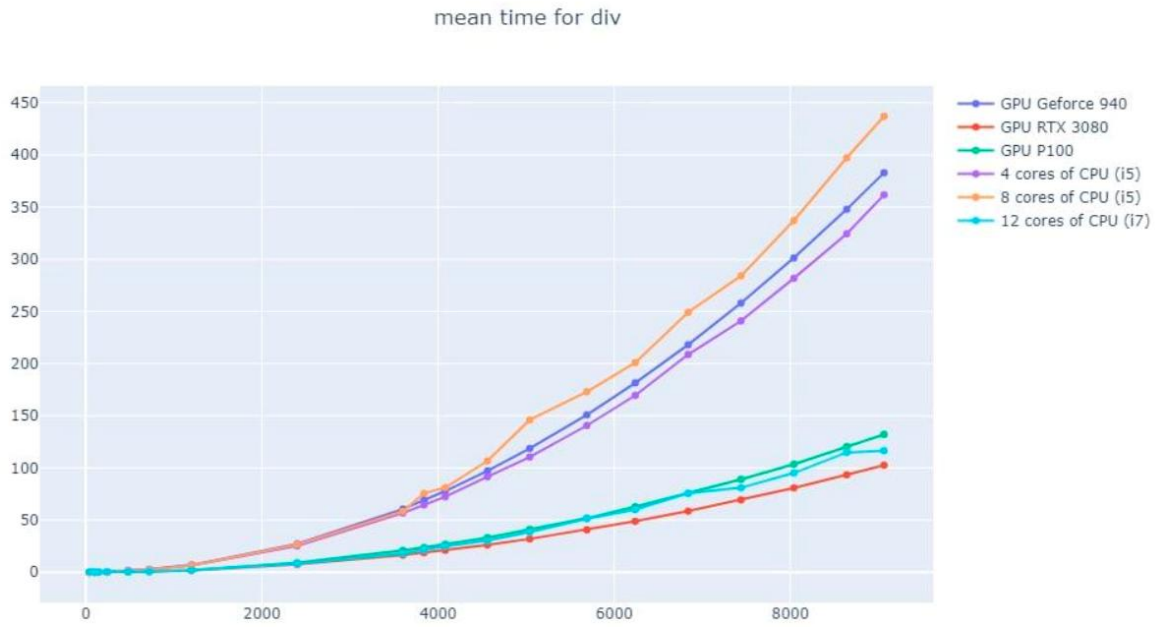


Рисунок 3.32 – Порівняльні результати виконання операції ділення матриці на скаляр на різних обчислювальних пристроях (усереднено за 10 запусків)

Як видно з рисунка 3.32, найкращий результат за часом виконання операції ділення матриці на скаляр було досягнуто на графічному процесорі NVIDIA GeForce RTX 3080, що відповідає її високому рівню обчислювальної потужності та сучасній архітектурі. Натомість найменшу ефективність продемонстрував процесор Intel Core i5-7200U при використанні восьми потоків, що, ймовірно, пов’язано з перевищенням оптимального рівня паралелізму для цієї моделі CPU та збільшенням накладних витрат на координацію потоків.

Цікаво, що конфігурація з чотирма потоками на Intel i5 виявилася ефективнішою за GPU GeForce 940MX, що підтверджує обмежену здатність цієї відеокарти до обчислень загального призначення. Процесор Intel Core i7-10850H стабільно демонструє кращі результати порівняно з Intel i5, GeForce 940MX та навіть серверною GPU Tesla P100, що може бути обумовлено низькою оптимізацією коду під архітектуру Tesla або специфікою задачі, де CPU здатен краще справлятися завдяки високій частоті та широкому кешу.

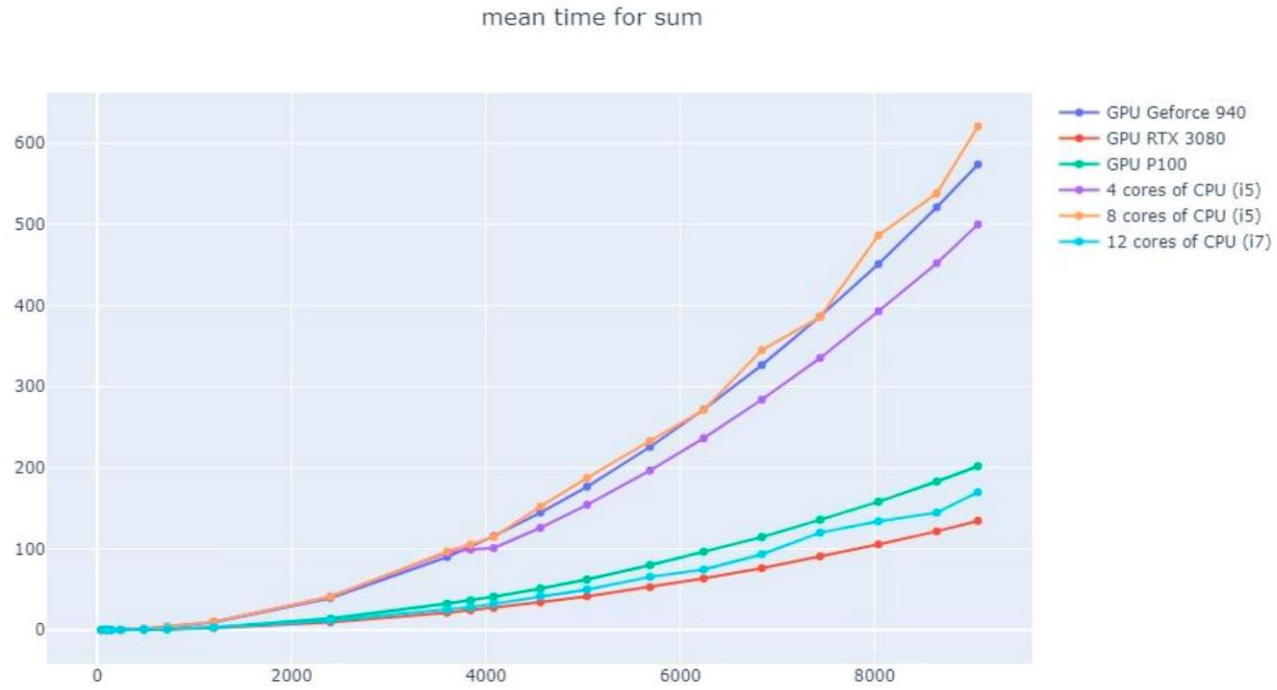


Рисунок 3.33 – Порівняльні результати виконання операції додавання матриць на різних обчислювальних пристроях (усереднено за 10 запусків)

Як і у випадку з операцією ділення матриці на число (рис. 3.32), результати, представлені на рисунку 3.33, демонструють аналогічну тенденцію у розподілі ефективності між пристроями. Графічний процесор RTX 3080 знову є беззаперечним лідером за швидкістю виконання, тоді як Intel Core i5-7200U при восьмипотоковій конфігурації стабільно демонструє найнижчі показники продуктивності.

Чотирипотоковий режим для Intel i5 знову перевершує GeForce 940MX, а Intel i7-10850H впевнено утримує середню позицію між споживчими та серверними пристроями, випереджаючи навіть Tesla P100. Така повторюваність результатів дозволяє підтвердити стійкість продуктивності пристроїв у задачах однотипної арифметики, незалежно від специфіки операції (додавання чи ділення).

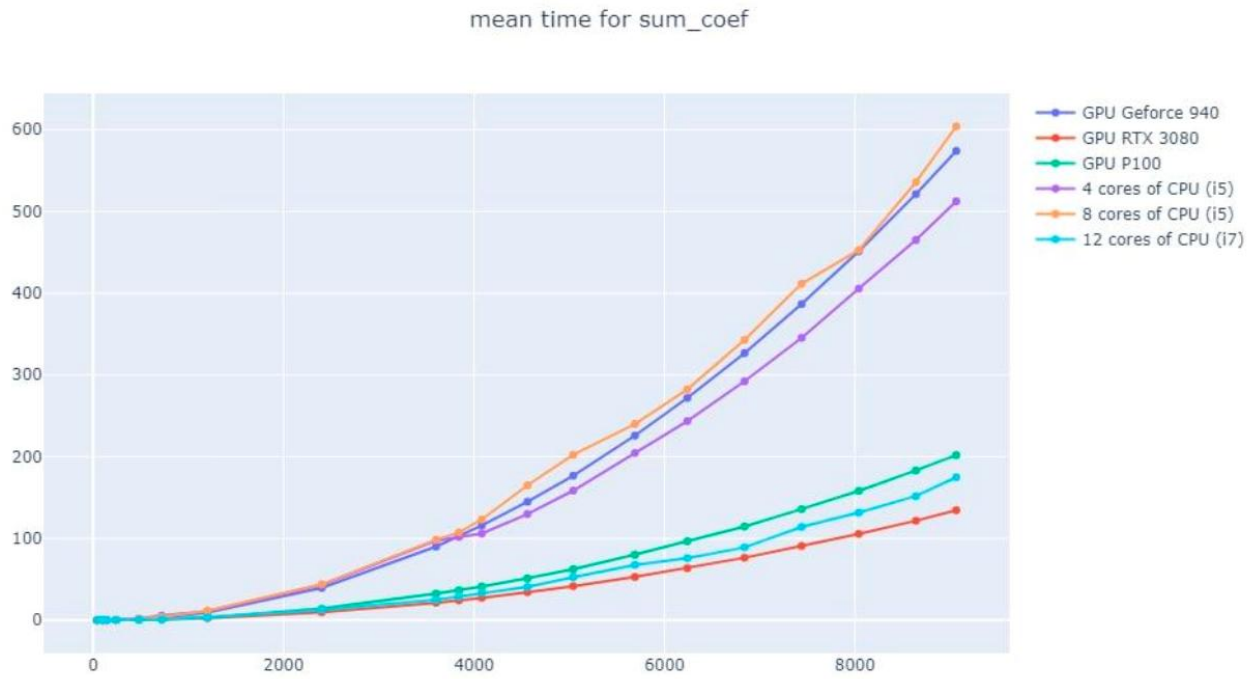


Рисунок 3.34 – Порівняльні результати виконання операції додавання матриць з коефіцієнтом (kA + B) на різних обчислювальних пристроях (усереднено за 10 запусків)

Результати, подані на рисунку 3.34, підтверджують тенденції, зафіксовані у попередніх випробуваннях: GPU RTX 3080 зберігає лідерство у швидкодії, продемонструвавши найнижчий середній час виконання задачі. Незважаючи на незначне ускладнення операції порівняно з базовим додаванням (через необхідність попереднього множення на коефіцієнт), розподіл ефективності між пристроями залишається практично незмінним.

Intel Core i7-10850H вкотре демонструє високу стабільність і масштабованість продуктивності, обганяючи Tesla P100, GeForce 940MX та всі конфігурації i5. Як і раніше, найнижчі результати показує Intel i5-7200U з вісьмома потоками, тоді як чотирипотокова конфігурація для цього ж процесора демонструє вищу ефективність, навіть перевершуючи результати GeForce 940MX.

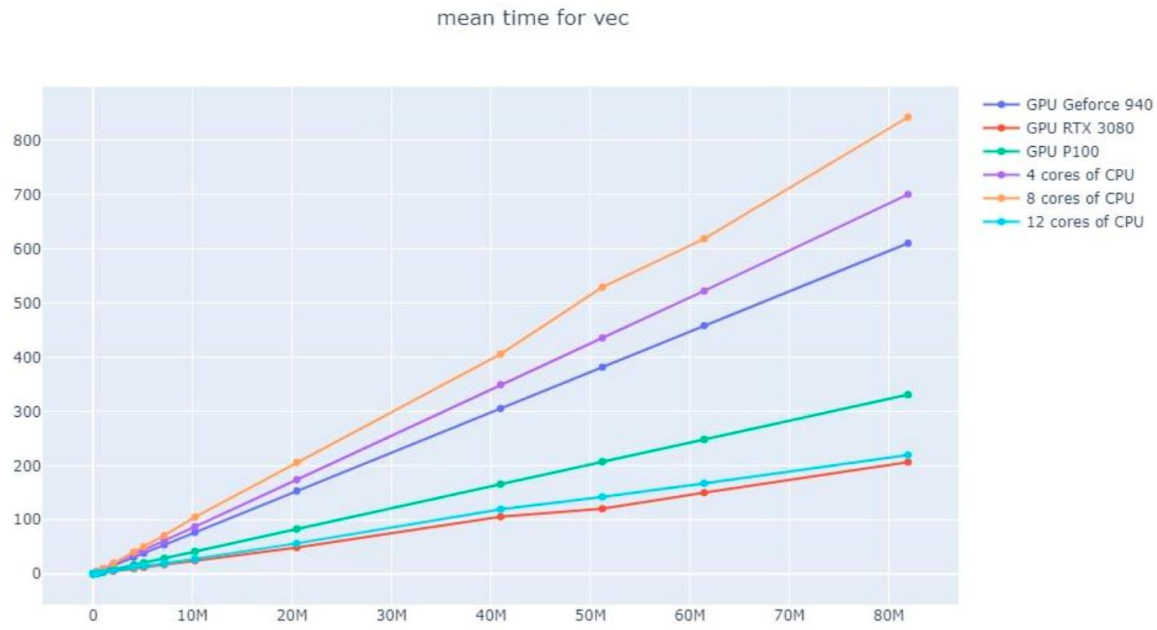


Рисунок 3.35 – Порівняльні результати виконання операції скалярного добутку векторів на різних обчислювальних пристроях (усереднено за 10 запусків)

Аналіз результатів, поданих на рисунку 3.35, демонструє незначну зміну розподілу ефективності між пристроями у порівнянні з попередніми задачами. Зокрема, відеокарта GeForce 940MX показала кращу продуктивність, ніж процесор Intel Core i5-7200U, що є нетиповим результатом у загальному контексті проведених тестів. Для решти пристроїв – RTX 3080, Tesla P100 та Intel Core i7 – продуктивність залишилася на сталому високому рівні, відповідно до їхніх апаратних можливостей.

Варто зауважити, що CUDA-реалізація скалярного добутку також використовувала атомарні операції для збереження результату, подібно до початкової імплементації на CPU. Однак, на відміну від CPU, затримок у виконанні або помітного зниження швидкодії виявлено не було, що свідчить про кращу оптимізацію роботи з атомарними змінними у середовищі CUDA.

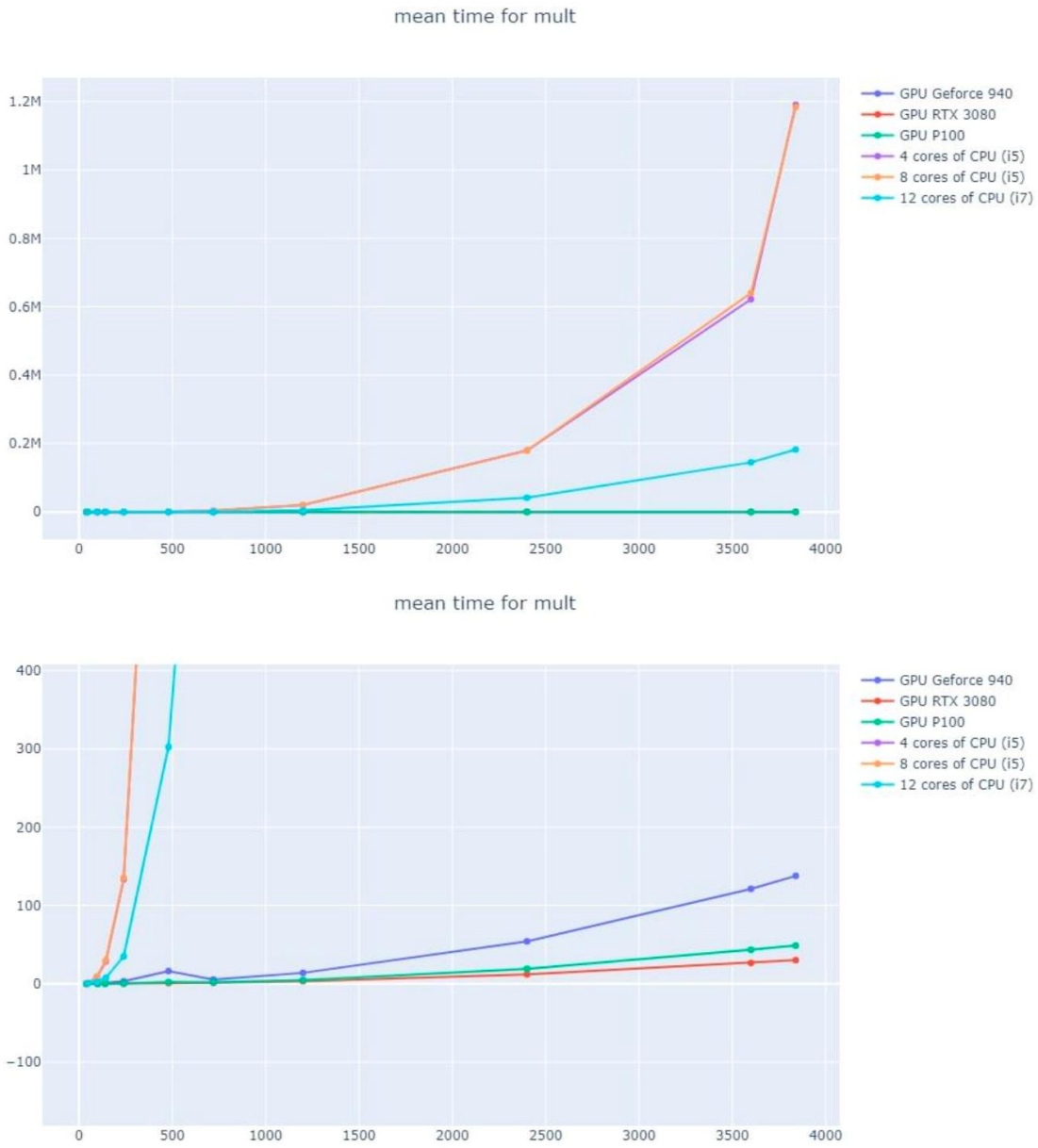


Рисунок 3.36 – Результати виконання операції множення матриць на різних пристроях та збільшений фрагмент графіка для центральних процесорів

На рисунку 3.36 спостерігається якісно інша картина: усі графічні процесори демонструють значно кращі результати, тоді як усі центральні процесори – як i5, так і i7 – показують істотно нижчу продуктивність. Така ситуація пояснюється різною алгоритмічною складністю реалізацій для різних платформ.

У реалізації на CPU множення матриць реалізоване як трикратний вкладений цикл, що має алгоритмічну складність $O(n^3)$: зовнішній цикл іде по рядках, середній – по стовпцях, і внутрішній – по розмірності матриці. У CUDA реалізація

побудована за принципом двовимірної індексації сітки та блоків, що дозволяє уникнути перших двох вкладених циклів. У результаті складність фактичної реалізації знижується до O(n) для кожного потоку, що й обумовлює значне зростання продуктивності на GPU.

Аналізуючи результати, представлені на рисунках 3.31–3.36, можна зробити висновок, що центральні процесори не завжди поступаються графічним за продуктивністю. Зокрема, у трьох із п’яти протестованих задач відеокарта NVIDIA GeForce 940MX продемонструвала нижчі результати, ніж навіть енергоефективний процесор Intel Core i5-7200U, що підкреслює невисоку придатність GPU цього класу до задач загального обчислювального призначення.

Також слід зауважити, що 8-потокова конфігурація i5 у низці задач показала гірший час виконання, ніж при використанні 4 потоків. Це підтверджує, що оптимальна кількість потоків для цієї моделі відповідає чотирьом логічним ядрам, як зазначено у специфікації виробника. Надмірна кількість створених потоків для такого процесора створює додаткові накладні витрати на розподіл задач та перемикання контексту, що негативно впливає на загальну ефективність.

Постає логічне запитання: чому GPU з високою теоретичною обчислювальною потужністю іноді програє CPU? Така ситуація пов’язана із комплексом факторів, серед яких:

- затрати на копіювання даних між хостом (CPU) і пристроєм (GPU);
- вузьке місце у пропускній здатності шини PCIe;
- наявність значних накладних витрат при запуску GPU-ядер для задач з малим обсягом обчислень;
- низька оптимізація коду для конкретної архітектури GPU.

Для більш детального аналізу було проведено роздільне вимірювання часу на передачу даних до GPU, обчислення та повернення результату назад на CPU – на прикладі операції скалярного добутку векторів. Отримані результати представлені на рисунку 3.37.

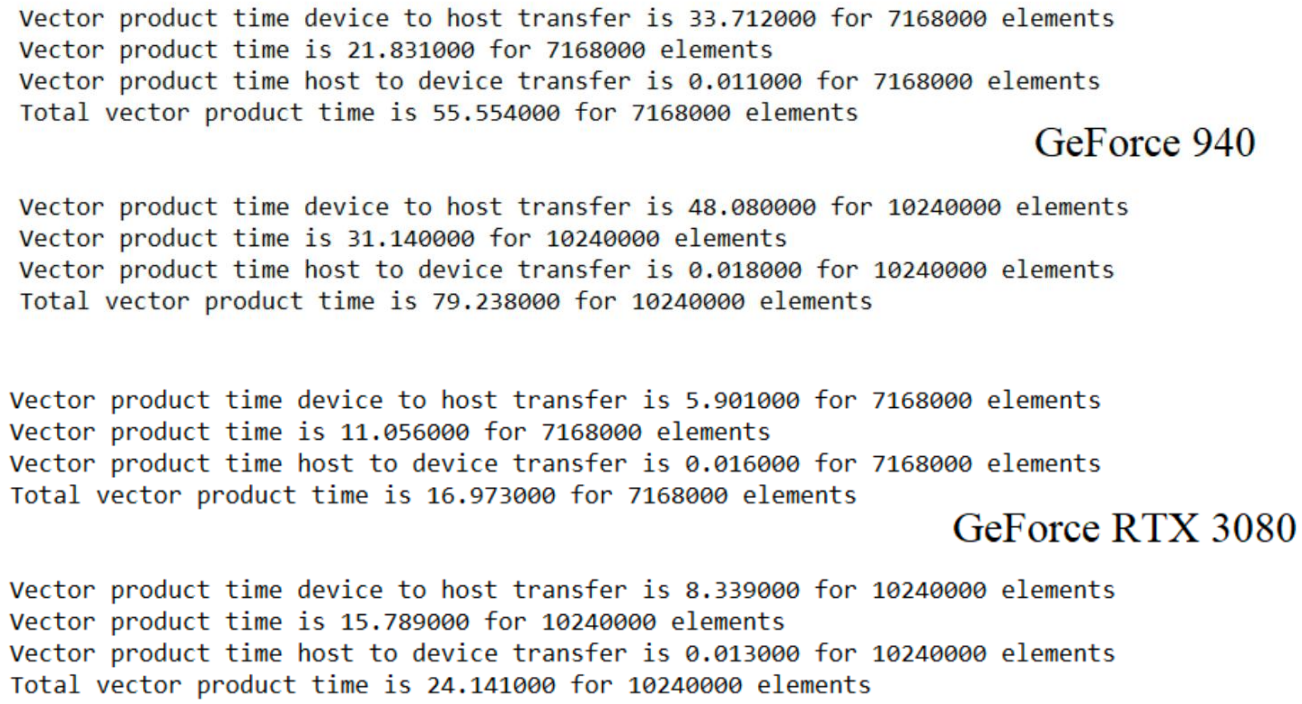


Рисунок 3.37 – Порівняння часу копіювання даних та обчислень при виконанні операції скалярного добутку на CPU та GPU

Як видно з рисунка 3.37, графічний процесор NVIDIA GeForce RTX 3080 демонструє найкращі результати як у частині обчислень, так і передачі даних між хостом та пристроєм, що свідчить про високу оптимізацію архітектури та ефективну взаємодію з пам’яттю. У той же час, для GeForce 940MX було зафіксовано, що переважна частина часу витрачається саме на трансфер даних, а не на обчислення, що суттєво впливає на загальний час виконання.

Цей факт підтверджує, що ефективність використання GPU в гетерогенній архітектурі значною мірою залежить не лише від потужності пристрою, а й від обсягу та складності обчислень, які необхідно виконати всередині потоку. Іншими словами, продуктивність GPU виправдовується лише тоді, коли тривалість обчислювального навантаження достатня для компенсації витрат на копіювання даних.

Щоб наочно продемонструвати цей принцип, було вирішено провести додаткову серію експериментів, у межах яких для задач, що передбачають прості обчислення (наприклад, додавання або ділення матриць), одна й та сама операція

повторюється кілька разів у межах одного потоку. Мета цього тесту – визначити, чи зменшується вплив трансферу даних при зростанні "обчислювальної ваги" потоку.

Для цього було обрано чотири пристрої:

– Intel Core i5-7200U у конфігурації з 4 потоками;
– Intel Core i7-10850H з 2 потоками (як помірний багатопотоковий режим);
– GeForce 940MX;
– Tesla P100.

У кожному випадку було змодельовано виконання однієї, двох, п’яти та десяти послідовних операцій додавання або ділення матриць у межах потоку. Оскільки, як було зазначено раніше, час виконання двох варіантів додавання матриць (звичайного та з коефіцієнтом) практично не відрізняється, для цих тестів результати будуть представлені узагальнено.



Рисунок 3.38 – Час виконання операції ділення матриці на скаляр із дворазовим повторенням обчислення всередині потоку на різних пристроях

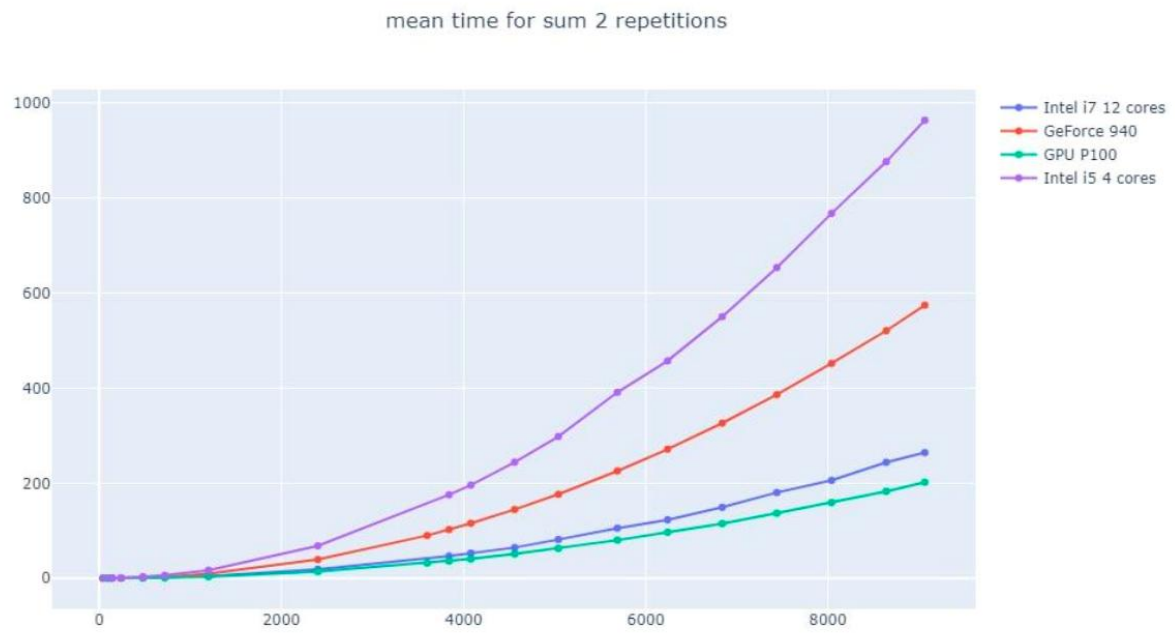


Рисунок 3.39 – Час виконання операції додавання матриць із дворазовим повторенням обчислення всередині потоку на різних обчислювальних пристроях



Рисунок 3.40 – Час виконання операції додавання матриць з коефіцієнтом (kA + B) із дворазовим повторенням обчислення всередині потоку на різних пристроях

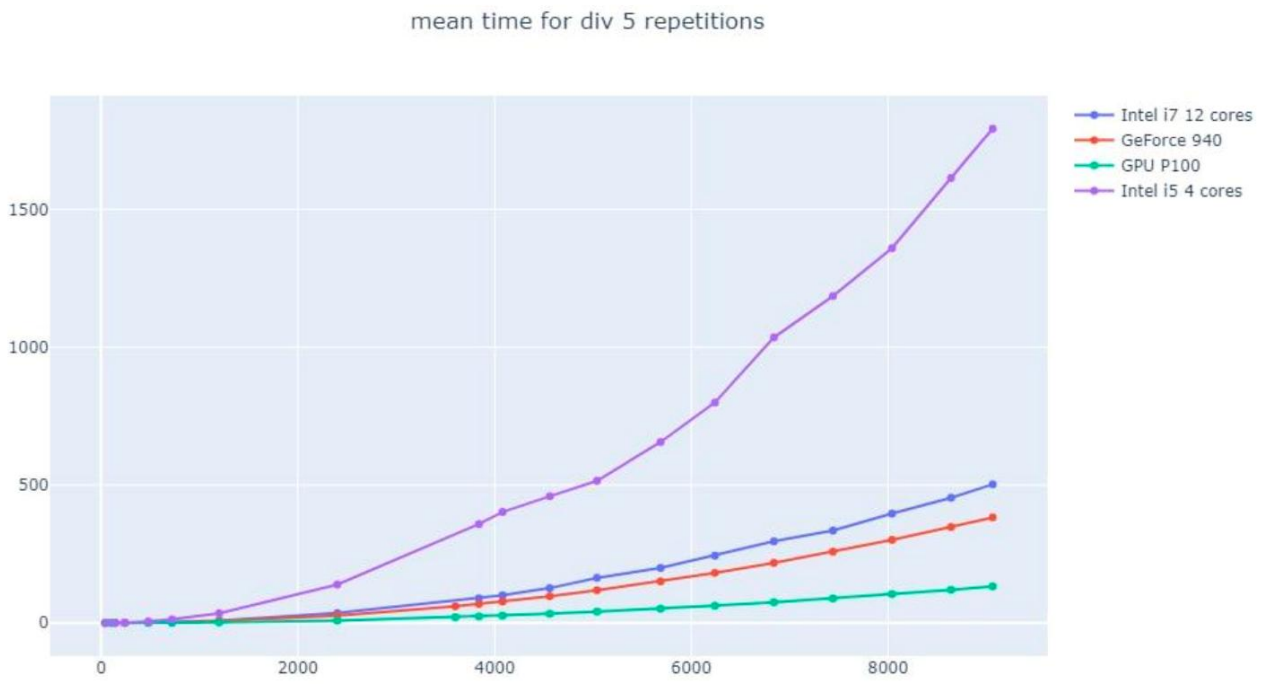


Рисунок 3.41 – Час виконання операції ділення матриці на скаляр із п’ятикратним повторенням обчислення всередині потоку на різних обчислювальних пристроях

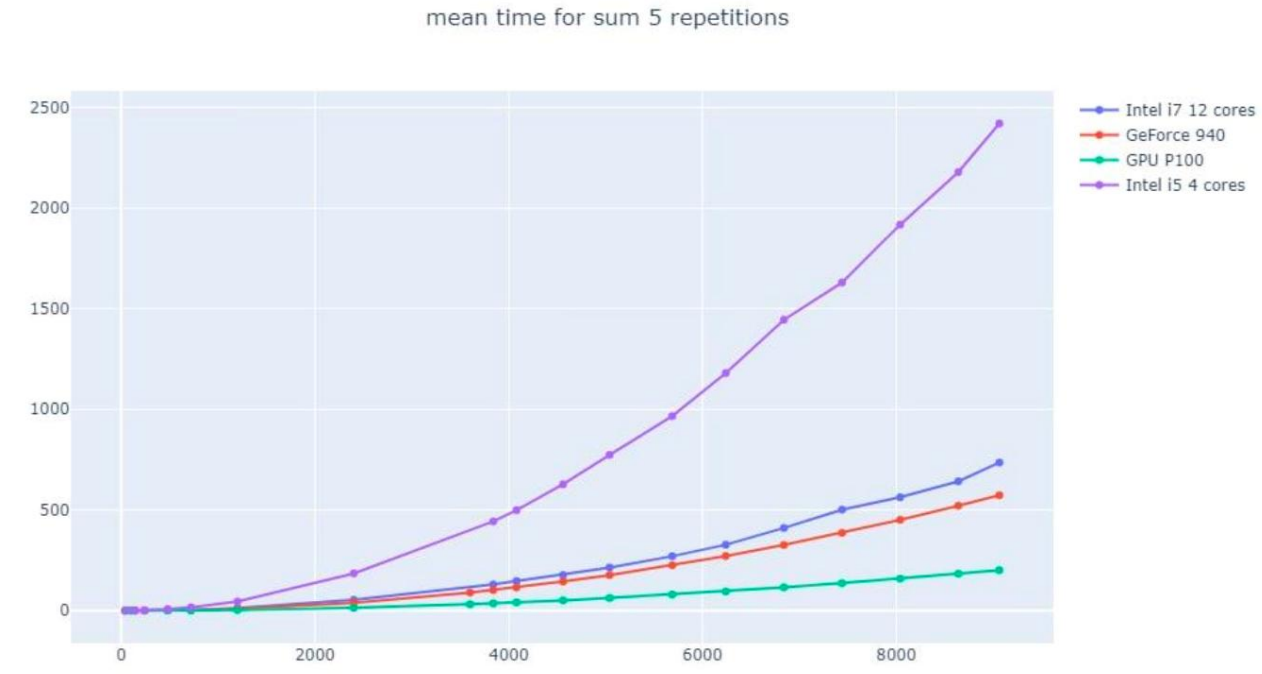


Рисунок 3.42 – Час виконання операції додавання матриць із п’ятикратним повторенням обчислення всередині потоку на різних обчислювальних пристроях

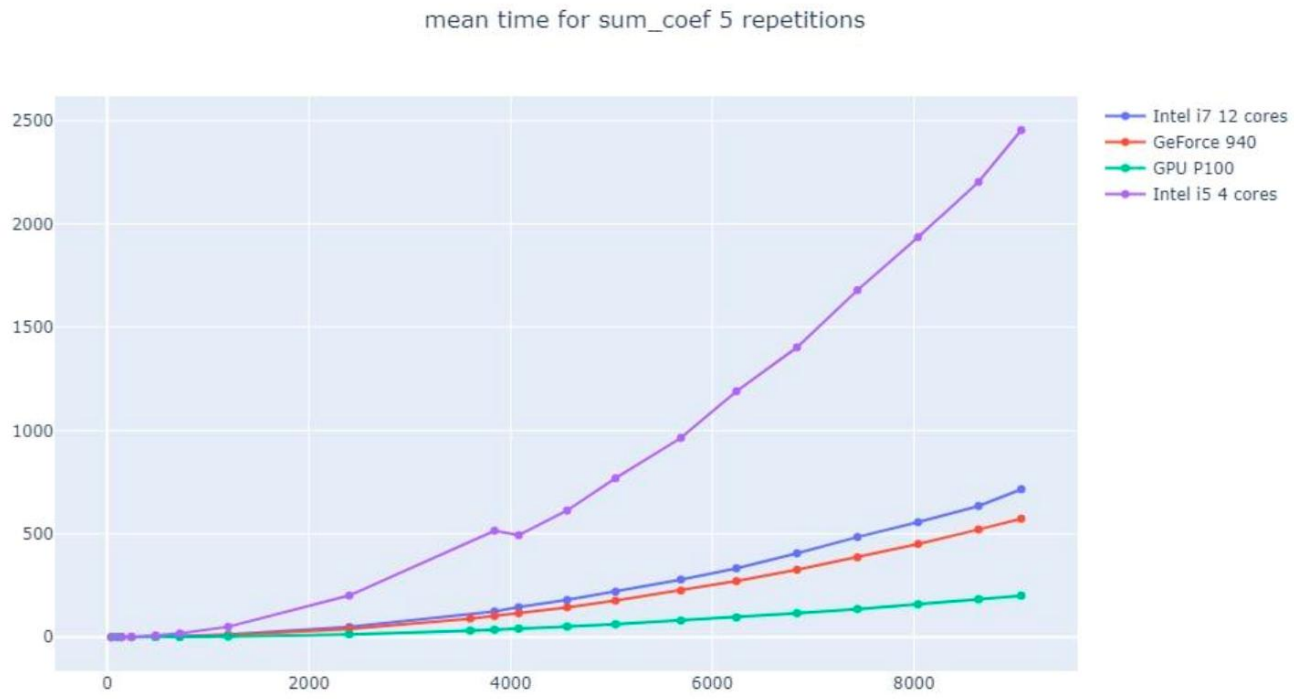


Рисунок 3.43 – Час виконання операції додавання матриць з коефіцієнтом (kA + B) із п’ятикратним повторенням обчислення всередині потоку на різних обчислювальних пристроях

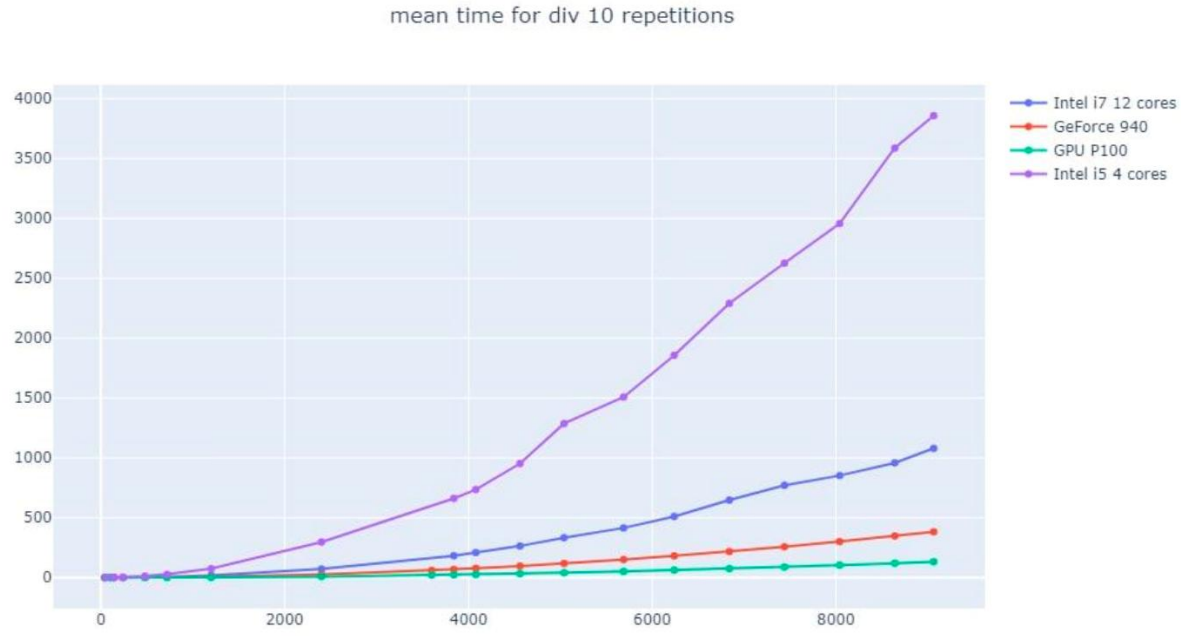


Рисунок 3.44 – Час виконання операції ділення матриці на скаляр із десятикратним повторенням обчислення всередині потоку на різних обчислювальних пристроях



Рисунок 3.45 – Час виконання операції додавання матриць із десятикратним повторенням обчислення всередині потоку на різних обчислювальних пристроях



Рисунок 3.46 – Час виконання операції додавання матриць з коефіцієнтом (kA + B) із десятикратним повторенням обчислення всередині потоку на різних обчислювальних пристроях

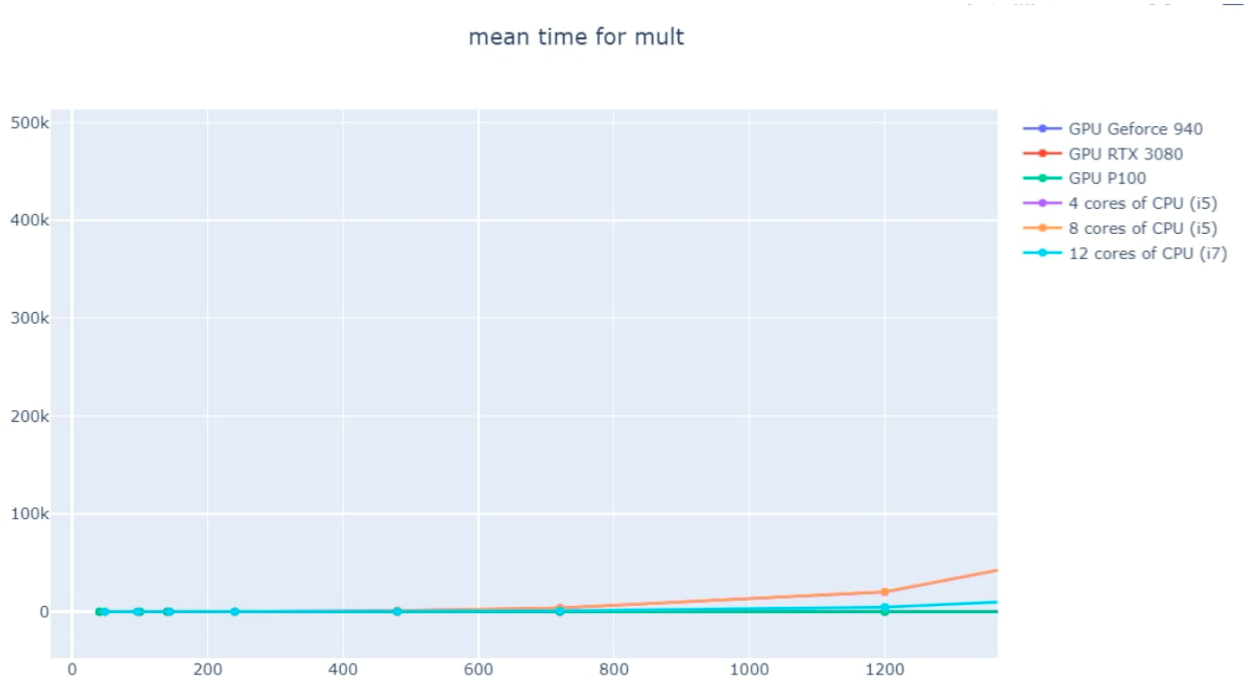


Рисунок 3.47 – Результати виконання операції множення матриць у збільшеному масштабі для центральних процесорів

Як свідчать дані, наведені на рисунку 3.47, суттєвих змін або стрибків у продуктивності пристроїв у залежності від розмірності матриці не спостерігається. Загальна тенденція лишається сталою: продуктивність CPU різко погіршується при збільшенні розмірності задачі, тоді як GPU демонструють більш передбачувану та стабільну ефективність.

Ключовим порогом, після якого обчислення на центральному процесорі стають значно менш ефективними, є розмір матриці близько 200×200. Починаючи з цього моменту, вплив алгоритмічної складності ($O(n^3)$) на CPU стає критичним, і навіть багато ядер не здатні компенсувати навантаження.

Натомість у діапазоні від 200×200 до 1000×1000 всі протестовані GPU – незалежно від покоління – демонструють подібні результати, що свідчить про добру масштабованість паралельної обробки та низький вплив зростання розмірності на загальний час виконання.

Таким чином, множення матриць є показовою задачею, у якій обчислювальна модель GPU значно перевершує CPU за ефективністю вже на середніх розмірах

задачі, що підтверджує доцільність використання графічних прискорювачів для подібних обчислень.

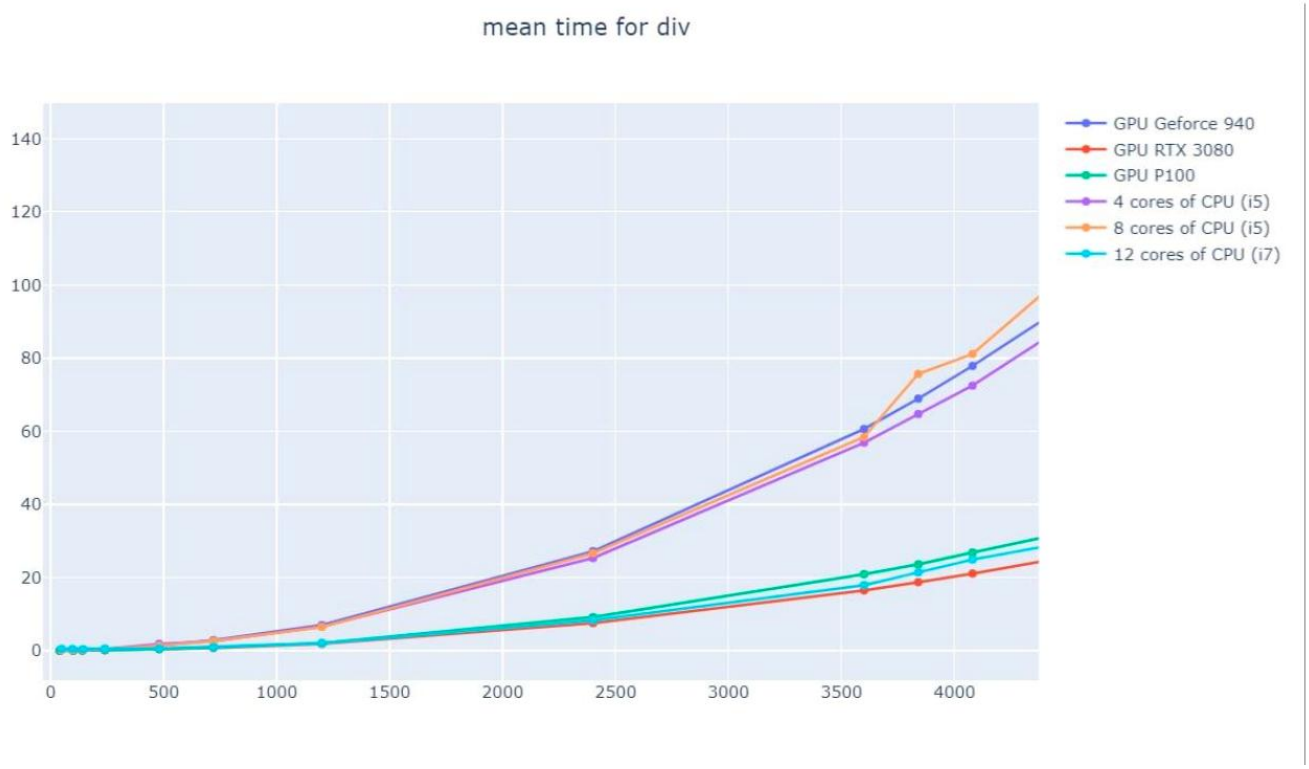


Рисунок 3.48 – Результати виконання операції ділення матриці на скаляр у збільшеному масштабі для центральних процесорів

Як видно з рисунка 3.48, збільшення масштабу графіка для операції ділення матриці на число дозволяє детальніше проаналізувати продуктивність CPU-пристроїв, які раніше були «згладжені» візуально високими показниками GPU. Графік демонструє, що порядок ефективності між CPU залишається незмінним: Intel Core i7-10850H стабільно перевершує Intel Core i5-7200U за всіма розмірностями матриці.

Разом із тим, на відміну від задачі множення матриць (див. рис. 3.44), у випадку ділення розрив у продуктивності між CPU та GPU є менш різким. Це пояснюється нижчою обчислювальною складністю самої операції та менш вираженим ефектом масштабування на CPU. Попри це, починаючи з розмірності приблизно 400×400, графічні процесори все ж виходять на суттєво кращі показники, демонструючи високу стабільність при зростанні обсягу обчислень.

Таким чином, у задачах із низькою внутрішньою складністю, таких як

ділення матриць на скаляр, центральні процесори залишаються конкурентоспроможними до певного порогу, однак надалі GPU поступово перехоплюють перевагу завдяки ширшому паралелізму та оптимізації доступу до пам’яті.

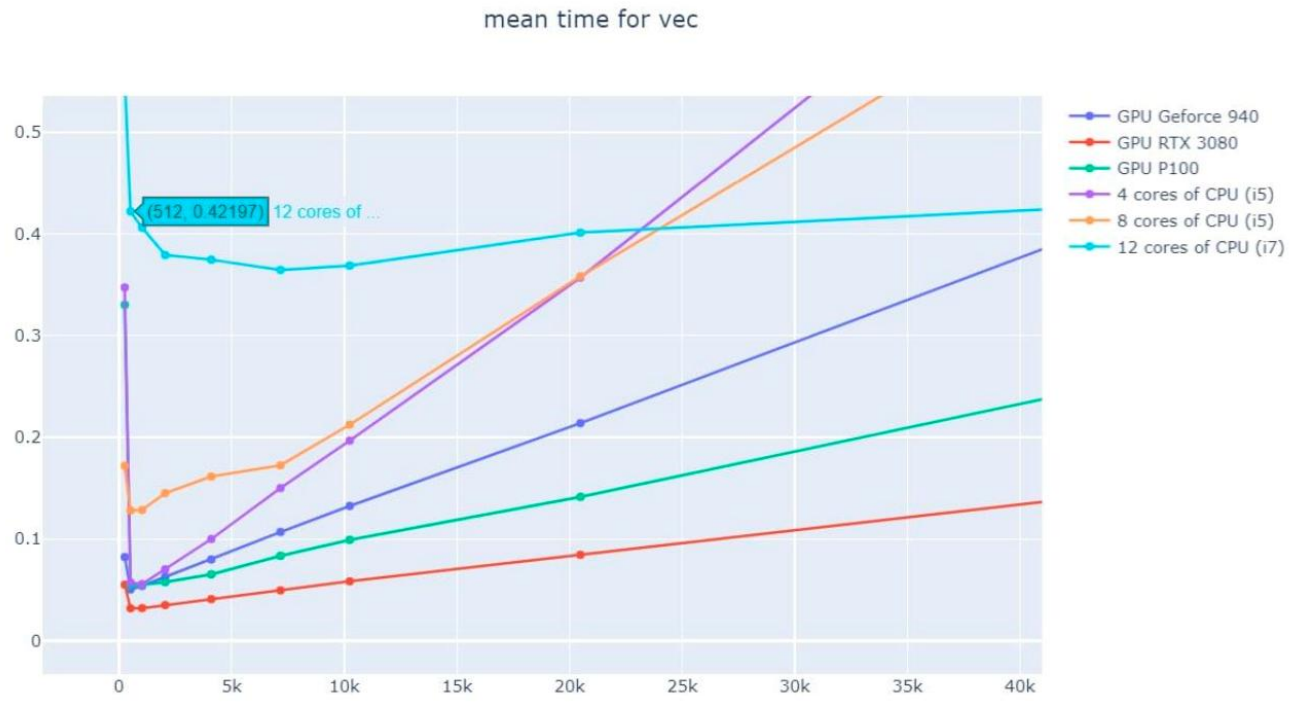


Рисунок 3.49 – Результати виконання операції скалярного добутку векторів у збільшеному масштабі для центральних процесорів

Як свідчить рисунок 3.49, у задачі скалярного добутку векторів спостерігається цікава динаміка продуктивності на центральних процесорах, особливо на початкових розмірностях векторів. Зокрема, при розмірі 256 елементів помітний різкий стрибок часу виконання, що характерно насамперед для CPU. Така поведінка зумовлена накладними витратами на ініціалізацію потоків, синхронізацію та атомарні операції, які на малих обсягах даних переважають над фактичними обчисленнями.

Починаючи з розмірності 512 елементів, спостерігається стабілізація часу виконання, що свідчить про перехід у режим, де обчислювальне навантаження починає переважати накладні витрати.

Особливої уваги заслуговує Intel Core i7-10850H, який при розмірності понад 50 000 елементів перевершує менш потужний графічний процесор GeForce 940MX,

а починаючи з приблизно 130 000 елементів – навіть Tesla P100. Така поведінка пояснюється двома ключовими факторами:

1. Оптимізація локального обчислення у CPU-реалізації, яка зменшує кількість атомарних операцій;

2. Відносна неефективність GPU при низькій складності задачі, що не дозволяє повністю завантажити паралельну архітектуру, особливо коли синхронізація не масштабована лінійно.

Таким чином, у випадку обчислювальних задач із синхронізованим накопиченням результатів, таких як скалярний добуток, центральні процесори можуть демонструвати конкурентну або навіть переважну продуктивність, особливо при великому обсязі даних і правильно підібраному підході до обробки.

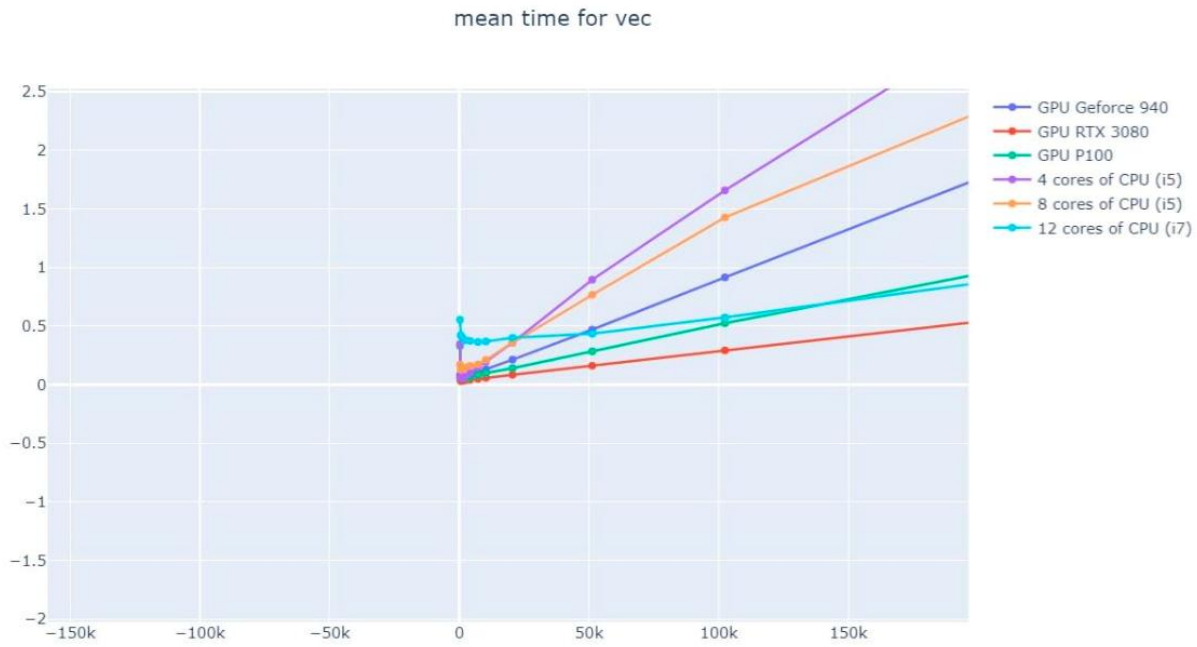


Рисунок 3.50 – Результати виконання операції скалярного добутку векторів у збільшеному масштабі (фрагмент 2) для центральних та графічних процесорів

3.4 Вплив обсягу даних, кількості потоків і типу задач на продуктивність

Для дослідження впливу апаратного забезпечення на продуктивність було обрано два графічні процесори: NVIDIA GeForce MX230, встановлений у локальному ноутбуці, та NVIDIA Tesla T4, доступна через безкоштовне віддалене середовище Google Colab. Вибір на користь середовища Colab був зумовлений суттєвою перевагою Tesla T4 за ключовими характеристиками: зокрема, обсяг відеопам’яті у Tesla T4 є у 8 разів більшим, ніж у MX230, а також загальна обчислювальна потужність і пропускна здатність пам’яті є значно вищими.

Згідно з порівняльним аналізом характеристик, наведеним у джерелі, Tesla T4 є значно кращим кандидатом для виконання обчислювально-інтенсивних задач, що потребують обробки великих масивів даних. Це дозволяє використовувати складніші задачі з більшим розміром вхідних даних без ризику перевищення ліміту пам’яті та гарантує вищу стабільність результатів при масштабуванні.

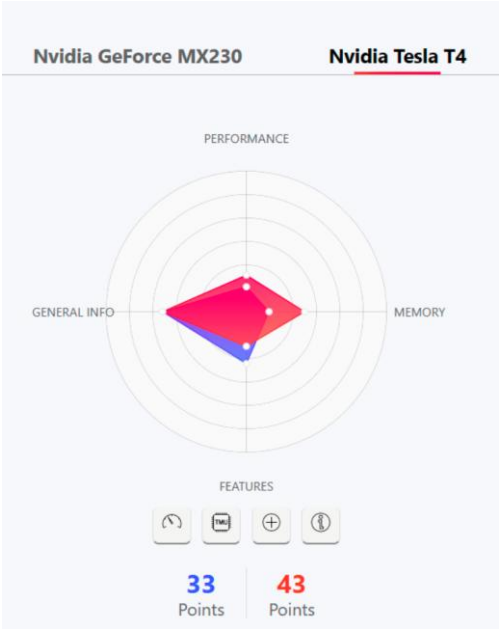


Рисунок 3.51 – Результати порівняння характеристик графічних процесорів Nvidia Tesla T4 та GeForce MX230

Згідно з джерелом у процесі виконання експериментів було зафіксовано, що обсяг використаної відеопам’яті GPU досягав 9 ГБ, що перевищує максимально доступний обсяг пам’яті на GeForce MX230. Це означає, що проведення аналогічного експерименту на MX230 було б технічно неможливим через обмеження апаратного рівня, зокрема – недостатню місткість відеопам’яті для зберігання великих масивів даних та проміжних обчислень.

Таким чином, вибір Tesla T4 у середовищі Google Colab був не лише обґрунтованим з точки зору продуктивності, а й необхідним для забезпечення можливості масштабного експериментального моделювання в рамках цього дослідження.

Порівняння продуктивності обробки даних на CPU та GPU з використанням CuDF

У сучасних обчислювальних проєктах часто порушується питання доцільності перенесення обробки даних з центрального процесора (CPU) на графічний процесор (GPU). Одним з популярних інструментів для обробки табличних структур на GPU є бібліотека CuDF, яка входить до складу екосистеми RAPIDS. У свою чергу, традиційною альтернативою для обробки структурованих даних на CPU залишається бібліотека Pandas.

У мережі опубліковано багато прикладів порівняння швидкодії Pandas і CuDF, у яких GPU у більшості випадків демонструє значну перевагу. Проте оцінка продуктивності не може бути обмежена лише часом виконання окремої операції, оскільки практичні задачі передбачають комплексну взаємодію між компонентами системи, включаючи перенесення даних між GPU та CPU, наприклад, при візуалізації або тренуванні моделі машинного навчання. У такому випадку виграш у швидкодії може бути знівельований витратами на синхронізацію та копіювання даних.

Ще одним важливим аспектом є функціональна різниця між Pandas та CuDF. У ряді випадків Pandas дозволяє використовувати гнучкі високорівневі конструкції, зокрема лямбда-функції для обробки рядкових та об’єктних типів, тоді як аналоги в CuDF можуть бути недоступними або потребувати нетривіальних рішень.

З метою оцінки реального впливу CuDF на продуктивність, було адаптовано фрагмент проєкту, присвяченого аналізу Telegram-датасету, де частину задач обробки було перенесено на GPU. Реальна потреба у пришвидшенні обчислень була обґрунтована ще під час попереднього досвіду: на етапі проєктування більшість студентів зіштовхувались із проблемою повільної обробки даних у Pandas, що значно уповільнювало весь процес аналізу.

У межах цього експерименту було протестовано декілька задач, і для їх вирішення частково змінено логіку обчислень, щоб досягнути оптимального балансу між швидкодією, легкістю реалізації та стабільністю виконання. Таке відношення можна умовно оцінити формулою:

$$\text{Оптимальність} = \frac{\text{Швидкодія}}{\text{Легкість написання коду}} \tag{3.1}$$

У середньому, на CPU завдання виконувались за ~30 секунд, тоді як на GPU – до 10 секунд, проте перенесення логіки потребувало додаткового часу на розробку, що в деяких випадках ставило під сумнів доцільність такого прискорення.

У рамках подальшого аналізу буде розглянуто п’ять прикладів, серед яких – найтиповіші для аналітики операції concat та merge, з метою кількісного порівняння продуктивності та аналізу доцільності перенесення конкретних задач на GPU.

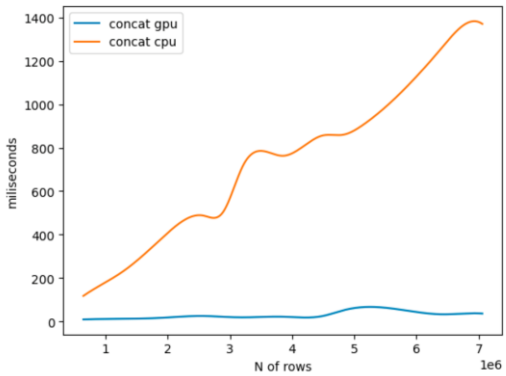


Рисунок 3.52 – Залежність часу виконання операції concat від об’єму даних при використанні бібліотек Pandas (CPU) та CuDF (GPU)

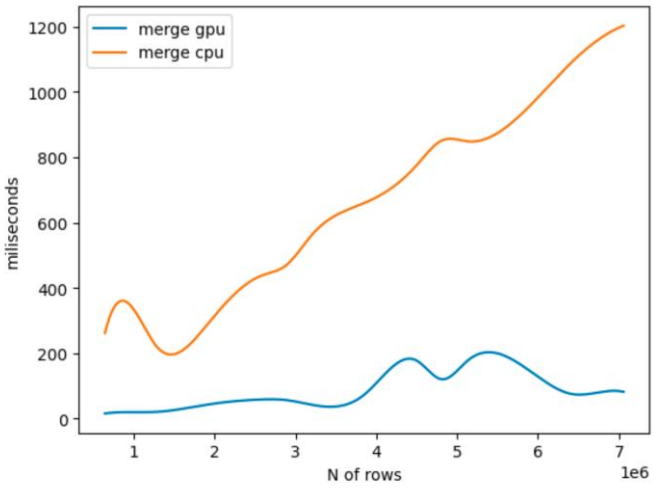


Рисунок 3.53 – Залежність часу виконання операції merge від об’єму даних при використанні бібліотек Pandas (CPU) та CuDF (GPU)

Аналіз інтенсивності листування з окремим контактом

Першою задачею в рамках експериментального дослідження було обрано аналіз інтенсивності листування з визначеним контактом. Основною метою є визначення динаміки спілкування у часі за допомогою агрегації повідомлень, що дозволяє виявити часові періоди підвищеної чи зниженої активності.

Для реалізації задачі на CPU використовувалась бібліотека Pandas, а для реалізації на GPU – CuDF. Послідовність дій включала:

– фільтрацію повідомлень за визначеним ID контакту (операція вибірки);

– групування за часовими мітками (наприклад, за днями або тижнями);

– обчислення кількості повідомлень у кожному часовому інтервалі;

– перенесення результатів на CPU для побудови графіка за допомогою бібліотеки Matplotlib, яка не має нативної підтримки GPU-даних.

Особливістю цієї задачі є наявність змішаної взаємодії між GPU та CPU, що дозволяє оцінити не лише швидкодію обчислень, а й ефективність передачі даних між пристроями. Незважаючи на присутність копіювання, виграш у швидкодії на етапі фільтрації та агрегації дозволив зменшити загальний час виконання, особливо на великих обсягах вхідних повідомлень.

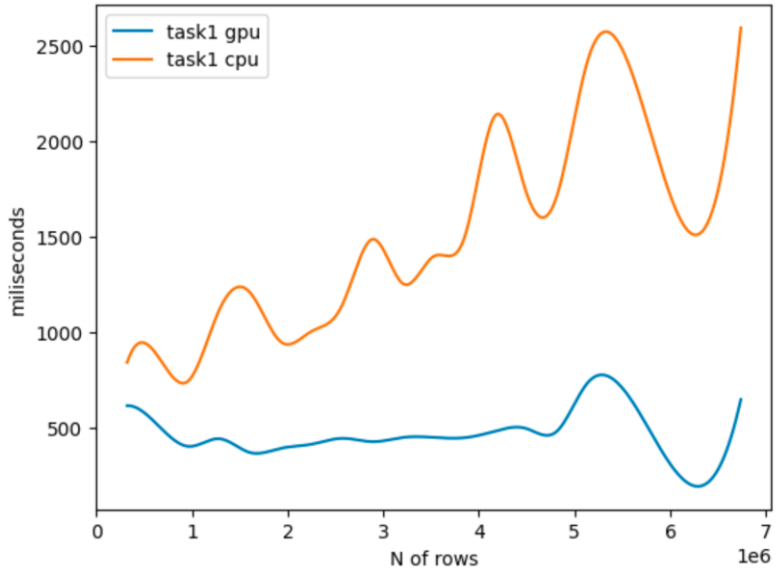


Рисунок 3.54 – Залежність часу обчислення задачі аналізу інтенсивності листування (Задача 1) від об’єму даних при використанні Pandas (CPU) та CuDF (GPU)

Задача 2. Обчислення середньої довжини повідомлень за місяць

Друга експериментальна задача полягала у визначенні середньої довжини повідомлень у розрізі кожного місяця. Вона є типовою для аналізу текстових повідомлень у месенджерах або соціальних мережах. З технічної точки зору, ця задача передбачає:

– розрахунок довжини кожного повідомлення;
– групування за місяцями на основі дати надсилання;
– усереднення довжин по кожній групі.

Для реалізації на CPU використовувалась бібліотека Pandas, зокрема метод .apply() у поєднанні з функцією len() для обчислення довжини рядків. У GPU-реалізації за допомогою CuDF використовувались оптимізовані методи для обробки рядків та списків у серіях Series, які дозволяють досягти подібного функціонального результату без необхідності в повноцінному .apply().

Причина такого підходу полягає в тому, що рядкові методи у Pandas працюють досить повільно, особливо при роботі з великими наборами даних, у той

час як CuDF забезпечує апаратне прискорення таких операцій, зменшуючи обчислювальне навантаження за рахунок векторизації та паралельного виконання.

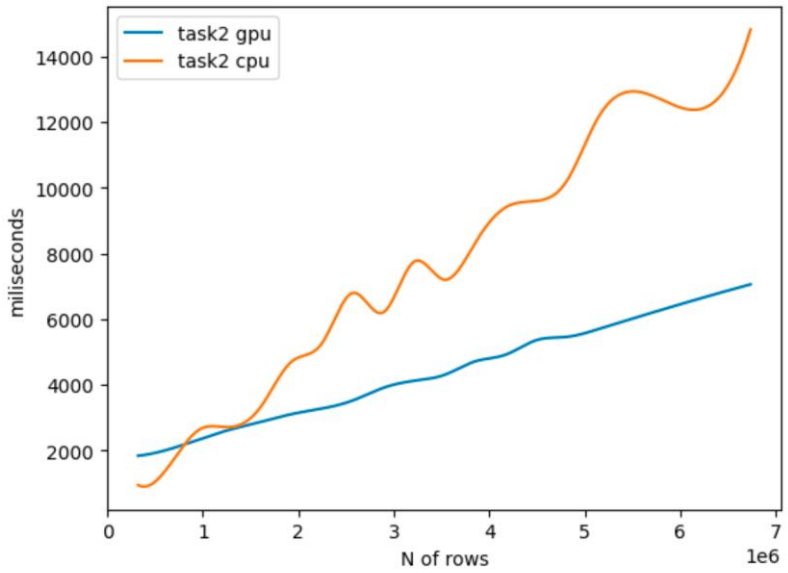


Рисунок 3.55 – Залежність часу обчислення задачі обчислення середньої довжини повідомлень за місяць (Задача 2) від об’єму даних при використанні Pandas (CPU) та CuDF (GPU)

Задача 3. Визначення тональності повідомлень

Третьою експериментальною задачею було створення алгоритму для визначення тональності текстових повідомлень. Такий аналіз є типовим для задач обробки природної мови (NLP), зокрема в контексті класифікації емоційного забарвлення в особистих повідомленнях чи відгуках. Основні кроки включали: попереднє очищення тексту; розділення датасету за ознакою «відправлені» / «отримані» повідомлення; використання рядкових методів для попередньої обробки (наприклад, приведення до нижнього регістру, пошук ключових слів тощо); передання результату в подальший блок аналізу або візуалізації. Принциповою особливістю цієї задачі є ідентичність логіки реалізації як у Pandas (CPU), так і у CuDF (GPU), за винятком одного етапу – розділення датафрейму, який реалізовано з використанням умовної фільтрації.

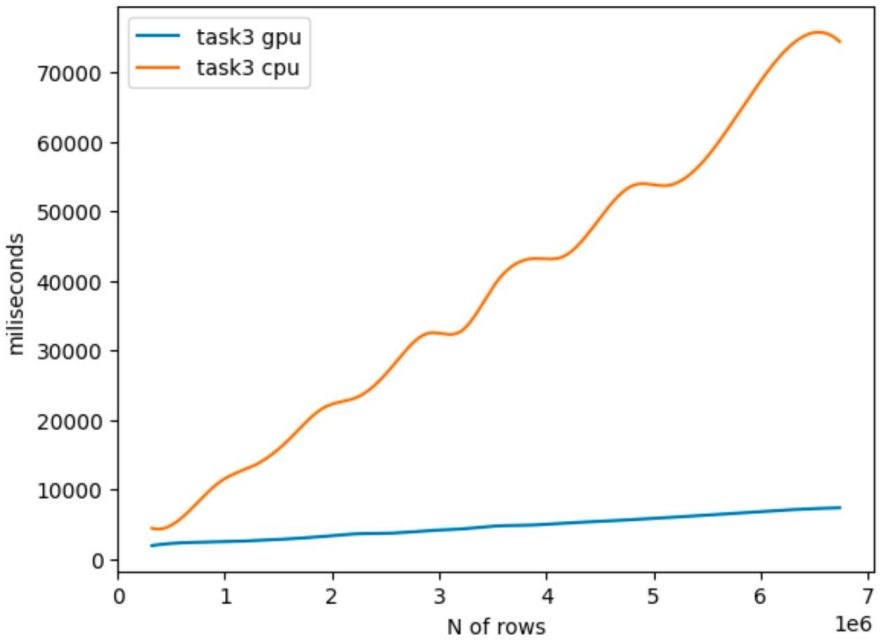


Рисунок 3.56 – Залежність часу обчислення задачі визначення тональності повідомлень (Задача 3) від об’єму даних при використанні Pandas (CPU) та CuDF (GPU)

3.5 Узагальнення результатів та рекомендації щодо використання GPU

На основі виконаних експериментів з трьома типовими задачами – аналіз інтенсивності листування, розрахунок середньої довжини повідомлень та визначення їхньої тональності – можна сформулювати низку узагальнень щодо продуктивності GPU-платформи на прикладі бібліотеки CuDF у порівнянні з Pandas на CPU.

Основні висновки:

1. Залежність ефективності GPU від обсягу даних. Для всіх задач спостерігається однакова закономірність: при невеликих наборах даних (до 500 тис. – 1 млн рядків) GPU не демонструє суттєвої переваги, а подекуди навіть поступається CPU через накладні витрати на передачу даних та ініціалізацію ядра.

Проте при зростанні обсягів даних – ефективність GPU зростає експоненційно, що забезпечує суттєве скорочення часу виконання на великих масивах.

2. Тип задачі має ключове значення. GPU є найефективнішим у задачах, що включають обробку рядкових даних (як у задачі 3), а також масові обчислювальні операції типу merge чи groupby на числових або часових даних (задачі 1–2). Для задач із високою частотою синхронізації, як-от скалярні обчислення, GPU може бути менш ефективним, якщо не враховувати архітектурну оптимізацію.

3. Вплив обчислювальної складності. Чим складніша структура обробки в кожному потоці (векторизація, агрегація, фільтрація), тим виразніше GPU реалізує свою перевагу. У випадках, де CPU використовує .apply() або обробку поелементно, CuDF з GPU здатен виконати ті самі задачі в десятки разів швидше.

4. Необхідність комплексної оцінки. Важливо враховувати не лише сам час обчислень, але і всі супутні операції – передача даних, синхронізація, побудова візуалізацій. У ряді випадків, особливо в середовищах з обмеженим доступом до GPU, як-от Google Colab, необхідна уважна оптимізація логіки передачі даних між CPU та GPU.

Рекомендації:

Використовувати GPU для:

- обробки великих датасетів (від 1 млн записів і більше);
- задач із фільтрацією, агрегацією та групуванням;
- обробки рядкових даних у високому обсязі;
- випадків, коли CPU виконує повільні .apply()-операції.

Залишити обробку на CPU для:

- невеликих обсягів даних (до 500 тис. рядків);
- задач з простою логікою або рідкісними викликами обчислень;
- ситуацій, коли важливіше мінімізувати час розробки, а не виконання.

ВИСНОВКИ

У межах даної кваліфікаційної роботи було проведено всебічне дослідження архітектури, програмних засобів та ефективності використання високопродуктивних графічних процесорів (GPU) у задачах загального призначення. Теоретична, аналітична та експериментальна частини дипломної роботи дозволили зробити низку важливих висновків і узагальнень.

У першому розділі розглянуто історію розвитку та місце GPU у сучасних обчислювальних системах. Встановлено, що з моменту появи перших графічних прискорювачів GPU зазнали еволюційного переходу від вузькоспеціалізованих графічних пристроїв до універсальних обчислювальних платформ. Здійснено порівняльний аналіз архітектур CPU та GPU, який показав суттєві відмінності в структурі ядер, принципах керування та організації пам’яті. Описано моделі паралелізму SIMD, MIMD та GPGPU, а також вивчено сучасні апаратні рішення на базі CUDA та OpenCL, включаючи Tensor Cores, Streaming Multiprocessors та структури Warp.

Другий розділ було присвячено програмним засобам реалізації обчислень на GPU. Розглянуто принципи програмування на CUDA, OpenCL, а також специфіку платформ DirectCompute та AMD ROCm. Значну увагу приділено бібліотекам GPU-прискорення, зокрема cuDNN, cuBLAS, Thrust, Numba, Dask і PyTorch. Описано їхнє призначення, переваги використання та сфери застосування. Також проаналізовано проблему передачі даних між CPU та GPU, яка є одним із критичних аспектів оптимізації продуктивності. Наголошено на важливості ефективного використання ієрархії пам’яті GPU, зокрема global, shared і register memory.

У третьому розділі було проведено експериментальне дослідження з метою визначення продуктивності GPU у порівнянні з CPU під час виконання обчислень різної складності. Обґрунтовано вибір апаратного та програмного забезпечення для тестування. Проведено серію бенчмарків, результати яких свідчать про суттєву

перевагу GPU при роботі з великими масивами даних та задачами, що добре піддаються розпаралелюванню. Встановлено, що продуктивність GPU зростає зі збільшенням обсягу даних і кількості потоків, тоді як у CPU швидко настає ефект насичення. Узагальнення результатів дозволило сформувати практичні рекомендації щодо ефективного використання GPU у залежності від типу задачі та її обчислювального профілю.

Проведене дослідження підтвердило доцільність використання GPU у високопродуктивних обчисленнях, машинному навчанні, симуляційних і візуалізаційних процесах. Отримані результати можуть бути використані як основа для побудови оптимізованих обчислювальних рішень у наукових, інженерних і прикладних галузях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Романюк О. Н. Довгалюк Р. Ю., Олійник С. В. Класифікація графічних відеоадаптерів.Наукові праці Донецького національного технічного університету. Сер. : Інформатика, кібернетика та обчислювальна техніка. - 2011. - Вип. 14. - С. 211-215.

2. Nickolls, J., Buck, I., Garland, M., & Skadron, K. Scalable Parallel Programming with CUDA. Queue, 6(2), 40–53. 2008. URL: https://doi.org/10.1145/1365490.1365500

3. Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. GPU Computing. Proceedings of the IEEE, 96(5), 879–899. 2008. URL: https://doi.org/10.1109/JPROC.2008.917757

4. Kirk, D. B., & Hwu, W.-M. W. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann. 2010. URL: https://www.elsevier.com/books/programming-massively-parallel-processors/kirk/978-0-12-381472-2

5. Baxter, S., & Luebke, D. The CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley. 2012. URL: https://developer.nvidia.com/cuda-handbook

6. Sanders, J., & Kandrot, E. CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley. 2010. URL: https://developer.nvidia.com/cuda-example

7. NVIDIA Corporation. CUDA C Programming Guide. 2021. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

8. Mittal, S., & Vetter, J. S. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. ACM Computing Surveys, 47(2), 1–23. 2015. URL: https://doi.org/10.1145/2656310

9. Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & Hwu, W.-M. W. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 73–82. 2008. URL: https://doi.org/10.1145/1345206.1345220

10. Fang, J., Varbanescu, A. L., & Sips, H. A Comprehensive Performance Comparison of CUDA and OpenCL. Proceedings of the 2011 International Conference on Parallel Processing, 216–225. 2011. URL: https://doi.org/10.1109/ICPP.2011.36

11. Stone, J. E., Gohara, D., & Shi, G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science & Engineering, 12(3), 66–73. 2010. URL: https://doi.org/10.1109/MCSE.2010.69

12. Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., ... & Sterling, T. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. Defense Advanced Research Projects Agency. 2008. URL: https://www.osti.gov/biblio/1032909

13. Cheng, H., & Wang, J. GPU Programming and Applications. CRC Press. 2014. URL: https://www.crcpress.com/GPU-Programming-and-Applications/Cheng-Wang/p/book/9781466598093

14. Zhang, Y., Owens, J. D., & Govindaraju, N. K. A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum, 26(1), 80–113. 2007. URL: https://doi.org/10.1111/j.1467-8659.2007.01010.x

15. Kirk, D. B., & Hwu, W.-M. W. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann. 2016. URL: https://www.elsevier.com/books/programming-massively-parallel-processors/kirk/978-0-12-811986-0

16. NVIDIA Corporation. CUDA Toolkit Documentation. 2021. URL: https://docs.nvidia.com/cuda/

17. AMD. ROCm - Open Software Platform for Accelerated Computing. 2021. URL: https://rocmdocs.amd.com/en/latest/

18. Intel. oneAPI Programming Guide. 2021. URL: [https://www.intel.com/content/www/us/en/developer/tools/

19. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., & Skadron, K. A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing, 68(10), 1370–1380. 2008. URL: https://doi.org/10.1016/j.jpdc.2008.05.014

20. Du, P., Weber, R., Luszczek, P., Tomov, S., & Peterson, G. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing, 38(8), 391–407. 2012. URL: https://doi.org/10.1016/j.parco.2011.10.002

21. AMD. GPU-Accelerated Applications with AMD Instinct™ Accelerators Enabled by AMD ROCm™ Software. 2024. URL: https://www.amd.com/content/dam/amd/en/documents/resources/gpu-accelerated-applications-catalog.pdf

22. NVIDIA. CUDA C Programming Guide. 2024. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

23. Khronos Group. The OpenCL Specification. 2024. URL: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html

24. NVIDIA. cuBLAS Library User Guide. 2024. URL: https://docs.nvidia.com/cublas/index.html

25. NVIDIA. cuDNN Developer Guide. 2024. URL: https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html

26. NVIDIA. Thrust: A Parallel Algorithms Library. 2024. URL: https://thrust.github.io/

27. Anaconda. Numba: A High Performance Python Compiler. 2024. URL: https://numba.pydata.org/

28. Dask Development Team. Dask: Parallel Computing with Task Scheduling. 2024. URL: https://docs.dask.org/en/stable/

29. PyTorch Team. PyTorch Documentation. 2024. URL: https://pytorch.org/docs/stable/index.html

30. CuPy Developers. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. 2024. URL: https://cupy.dev/

31. Google. JAX: Autograd and XLA. 2024. URL: https://github.com/google/jax

32. Apache. TVM: An Open Deep Learning Compiler Stack. 2024. URL: https://tvm.apache.org/

33. NVIDIA. RAPIDS: Open GPU Data Science. 2024. URL: https://rapids.ai/

34. Wilt, N. The CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley, 2013. URL: https://developer.nvidia.com/cuda-handbook.

35. Ansorge, R. Programming in Parallel with CUDA: A Practical Guide. CRC Press, 2022. URL: https://www.crcpress.com/Programming-in-Parallel-with-CUDA-A-Practical-Guide/Ansorge/p/book/9781032207110Reddit

36. Sanders, J., & Kandrot, E. CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2010. URL: https://developer.nvidia.com/cuda-example

37. Kirk, D. B., & Hwu, W.-M. W. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2016. URL: https://www.elsevier.com/books/programming-massively-parallel-processors/kirk/9780128119860

38. Klöckner, A., et al. GPU Scripting and Code Generation with PyCUDA. arXiv, 2013. URL: https://arxiv.org/abs/1304.5553arXiv

39. Dimitrov, M., & Esslinger, B. CUDA Tutorial: Cryptanalysis of Classical Ciphers Using Modern GPUs and CUDA. arXiv, 2021. URL: https://arxiv.org/abs/2103.13937arXiv

40. Ghorpade, J., et al. GPGPU Processing in CUDA Architecture. arXiv, 2012. URL: https://arxiv.org/abs/1202.4347arXiv

41. Holm, H. H., Brodtkorb, A. R., & Sætra, M. L. GPU Computing with Python: Performance, Energy Efficiency and Usability. arXiv, 2019. URL: https://arxiv.org/abs/1912.02607arXiv

42. Lutsiv, A. M., Lupenko, S. A., & Pasichnyk, V. V. Паралельні та розподілені обчислення: Навчальний посібник. Львів: Магнолія 2006, 2024. URL: https://magnolia.lviv.ua/wp-content/uploads/2024/03/Paralelni-ta-rozpodileni-obchyslennia_zmist.pdf

43. Семеренко, В. П. Технології паралельних обчислень: Навчальний посібник. Вінниця: ВНТУ, 2018. URL: https://www.researchgate.net/publication/334710599

44. Benotmane, M. *CUDA* Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming. Morgan Kaufmann, 2014.URL: https://www.elsevier.com/books/cuda-fortran-for-scientists-and-engineers/benotmane/978-0-12-416970-8

45. Cook, S. CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Morgan Kaufmann, 2013. URL: https://www.elsevier.com/books/cuda-programming/cook/978-0-12-415933-4

46. Hwu, W. W. GPU Computing Gems: Emerald Edition. Morgan Kaufmann, 2011. URL: https://www.elsevier.com/books/gpu-computing-gems-emerald-edition/hwu/978-0-12-384988-5

47. Sanders, J., Kandrot, E. CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2010. URL: https://developer.nvidia.com/cuda-example

48. Gregory, M. A., & Miller, T. Parallel Programming with OpenACC. Morgan Kaufmann, 2016. URL: https://www.elsevier.com/books/parallel-programming-with-openacc/gregory/978-0-12-410397-9

49. Baxter, S. Programming for Hybrid Multi/Manycore MPP Systems: Learn How to Design, Analyze, and Tune Multi-GPU Applications. Springer, 2020. URL: https://link.springer.com/book/10.1007/978-3-030-37816-8

50. Охотніков Д. Г., Зуб О. В. Теоретичні основи високопродуктивних графічних процесорів. Застосування програмного забезпечення в ІКТ : VI Науково-техн. конф., м. Київ, 24 трав. 2025 р. Київ, 2025. С. 603–604.URL: https://duikt.edu.ua/uploads/p_2779_56719466.pdf

51. Охотніков Д. Г., Зуб О. В. Програмні засоби та технології gpu-прискорення. Сучасні інтелектуальні інформаційні технології в науці та освіті : V Всеукр. Науково-практ. конф., м. Київ, 15 травня. 2025 р. Київ, 2025. – Подано на друкування.

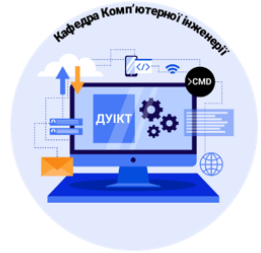
Додаток А

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій
Кафедра Комп'ютерної інженерії



КВАЛІФІКАЦІЙНА РОБОТА
НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

на тему: «Високопродуктивний графічний процесор»

Виконав студент групи КІД-42
Охотніков Дмитро Григорович
Керівник кваліфікаційної роботи:
Олександр ЗУБ

Київ 2025

Мета, об’єкт, предмет

Метою роботи є дослідити архітектурні та програмні особливості високопродуктивних графічних процесорів, провести порівняльний аналіз продуктивності CPU та GPU в реальних задачах обробки даних, і визначити доцільність використання GPU для підвищення ефективності обчислювальних процесів.

Об’єктом дослідження є процеси високопродуктивних обчислень із застосуванням графічних процесорів.

Предметом дослідження є архітектура та принципи роботи сучасних GPU, технології паралельної обробки даних, а також ефективність використання графічного процесора порівняно з центральним процесором у задачах різної складності..

Актуальність

Сучасні задачі в науці, штучному інтелекті, моделюванні та обробці великих даних вимагають потужних і швидких обчислень. Традиційні процесори не справляються з такими обсягами. Графічні процесори (GPU) завдяки паралельній архітектурі значно прискорюють обчислення, що робить дослідження їхньої ефективності надзвичайно актуальним у сучасних обчислювальних системах.

Порівняння архітектури CPU та GPU

Параметр	CPU (Central Processing Unit)	GPU (Graphics Processing Unit)
Кількість ядер	Невелика кількість потужних ядер (2–32)	Сотні або тисячі простих ядер
Архітектура виконання	Оптимізована для послідовного виконання інструкцій	Орієнтована на паралельне виконання однотипних операцій (SIMD)
Кеш-пам’ять	Велика багаторівнева кеш-пам’ять (L1, L2, L3)	Обмежена кеш-пам’ять; використовується глобальна, shared, текстурна пам’ять
Призначення	Загальні задачі, логіка, контроль, операції з ОС	Важкі обчислення, графіка, машинне навчання, GPGPU
Тип обчислень	Послідовні, умовні, логічні, контрольні	Масово-паралельні, одноманітні, числово-інтенсивні
Гнучкість	Висока (підтримка складної логіки, планування задач)	Обмежена (фокус на повторюваних операціях над даними)
Продуктивність на 1 ядро	Висока	Нижча, але компенсується кількістю потоків
Оптимальні задачі	Компіляція, обробка тексту, керування ресурсами	Матричні обчислення, рендеринг, моделювання, машинне навчання

Загальні принципи обчислень загального призначення на GPU

Аспект	Переваги	Недоліки
Продуктивність	Висока швидкість завдяки одночасному виконанню тисяч потоків	Виграш обмежений, якщо задача не піддається розпаралелюванню
Масштабованість	Легко масштабуються задачі з великою кількістю даних	Обмежена кількість потоків на блок і ресурсів на кожен SM
Енергетична ефективність	Більше FLOPS на ват енергії порівняно з CPU	Високе тепловиділення при повному навантаженні
Архітектурна гнучкість	Підтримка Tensor Cores, FP16/FP32/FP64, різних рівнів пам'яті	Не всі архітектури однаково ефективні для всіх типів задач
Бібліотечна підтримка	Велика кількість бібліотек: cuDNN, cuBLAS, PyTorch, TensorFlow	Більшість бібліотек оптимізовані лише під CUDA (тобто лише під NVIDIA)
Програмування	Можливість глибокої оптимізації через CUDA або OpenCL	Висока складність написання та налагодження паралельного коду
Передача даних	Можливість обробляти великі масиви даних без участі CPU	Затримки при копіюванні даних між CPU ↔ GPU через PCIe
Тип задач	Ідеальні для SIMD-задач: матричні операції, ML, рендеринг, симуляції	Неефективні при умовних переходах, ітераціях з розгалуженням (warp divergence)
Платформна підтримка	CUDA, OpenCL, Vulkan Compute, DirectCompute	CUDA працює лише на NVIDIA GPU, OpenCL менш ефективний

Постановка експерименту: мета, методика, обґрунтування

Мета дослідження:
Визначити розмір вхідного набору даних, при якому GPU перевершує CPU у базових матричних операціях.

Методика:

- Реалізація однакових алгоритмів на C++ (CPU) та CUDA C (GPU);
- Варіювання розмірів матриць від 32×32 до кількох тисяч елементів;
- Бенчмарки з фіксацією часу та ресурсів;
- Визначення критичної межі ефективності переходу на GPU.

Результат:
GPU демонструє перевагу при обробці великих обсягів даних, що дозволяє ефективно розподіляти ресурси в задачах HPC

Вибір пристроїв та програмного забезпечення для тестування

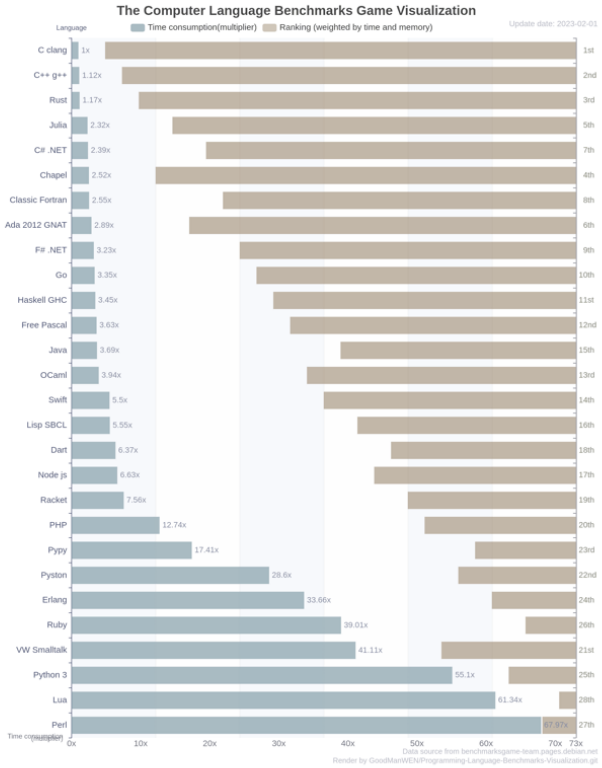


Рисунок 3.1 – Порівняння продуктивності мов програмування у задачі генерації множини Мандельброта

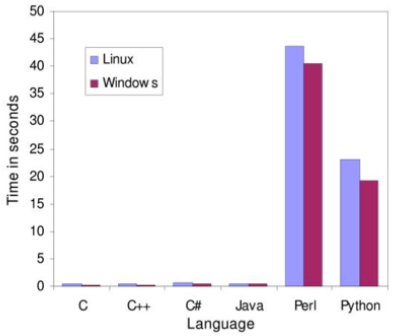


Рисунок 3.2 – Результати виконання алгоритму вирівнювання ДНК-послідовностей на різних ОС

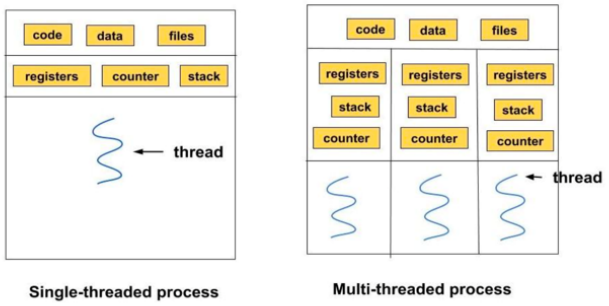


Рисунок 3.3 – Структурна модель організації потоків в операційній системі

Вибір пристроїв та програмного забезпечення для тестування

```cpp
void do_some_work(int a, int b) {
    std::cout << "Сума: " << a + b << std::endl;
}

std::thread my_thread(do_some_work, 1, 2);
my_thread.join(); // Обов'язково очікуємо завершення потоку
```

Рисунок 3.4 – Прямолінійний спосіб запуску потоку в бібліотеці std::endl

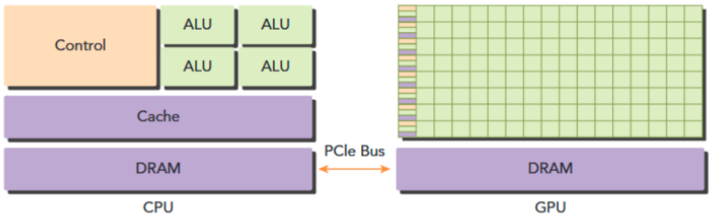


Рисунок 3.4 – Архітектурна схема взаємодії хоста та пристрою у моделі CUDA (джерело: [умовно – CUDA Developer Guide або авторська ілюстрація])

```cpp
std::thread t([](){
    std::cout << "Потік виконує лямбда-функцію" << std::endl;
});
t.join();
```

Рисунок 3.5 – Потік виконує лямбда-функцію в бібліотеці std::endl

```cpp
class functor_task {
public:
    void operator()() const {
        do_task_1();
        do_task_2();
    }

    void do_task_1() const {
        std::cout << "Задача 1 у потоці" << std::endl;
    }

    void do_task_2() const {
        std::cout << "Задача 2 у потоці" << std::endl;
    }
};

functor_task f;
std::thread my_thread(f);
my_thread.join();
```

Рисунок 3.6 – Реалізація функціонального обєкта в бібліотеці std::endl

Вибір пристроїв та програмного забезпечення для тестування

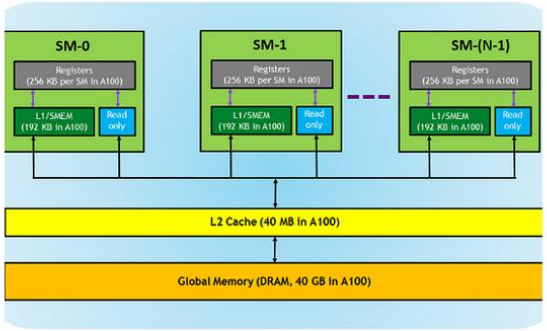


Рисунок 3.5 – Узагальнена схема етапів виконання обчислювального процесу в середовищі CUDA

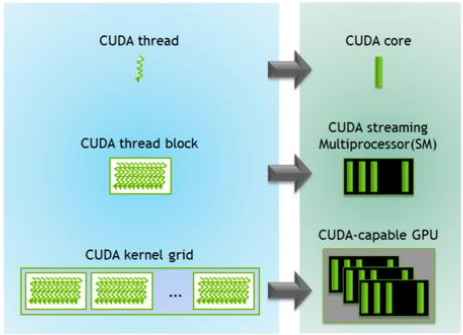


Рисунок 3.6 – Ієрархічна модель виконання CUDA-програми: сітка, блоки та потоки

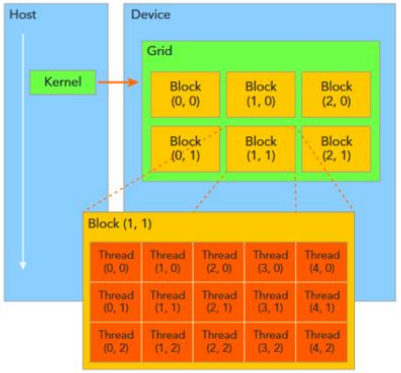


Рисунок 3.7 – Візуалізація сітки блоків і потоків у CUDA та їх ідентифікація

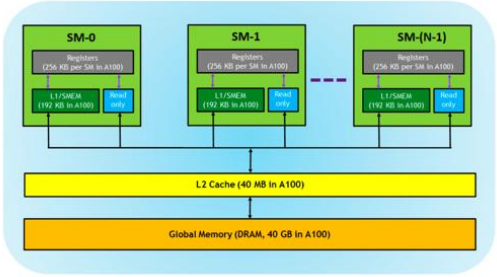


Рисунок 3.8 – Ієрархія організації пам'яті графічного процесора з підтримкою CUDA

Бенчмарки та аналіз ефективності GPU в порівнянні з CPU

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
Cache	12 MB Intel® Smart Cache
Bus Speed	8 GT/s
TDP	45 W
Configurable TDP-down	35 W

Рисунок 3.9 – Технічні характеристики процесора Intel Core i7-10850H

Codename	N16S-GTR-B/S
Architecture	Maxwell
Pipelines	384 - unified
TMUs	42
ROPs	16
Core Speed	1122 - 1242 (Boost) MHz
Memory Speed	4000 effective = 1253 MHz
Memory Bus Width	64 Bit
Memory Type	DDR3, GDDR5
Max. Amount of Memory	4 GB
Shared Memory	no
Memory Bandwidth	40.1 GB/s
API	DirectX 12 (FL 11_0), Shader 5.0, OpenGL 4.5
Power Consumption	15-25 Watt
Transistor Count	1.9 Billion
technology	28 nm
PCIe	3.0
Features	GPU Boost 2.0, Optimus, PhysX, CUDA, GeForce Experience, GameWorks
Notebook Size	medium sized
Date of Announcement	10.03.2016 = 3320 days old
Link to Manufacturer Page	www.geforce.com

Рисунок 3.11 – Технічні характеристики GPU NVIDIA GeForce 940MX

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
Cache	3 MB Intel® Smart Cache
Bus Speed	4 GT/s
TDP	15 W
Configurable TDP-up Base Frequency	2.70 GHz
Configurable TDP-up	25 W
Configurable TDP-down Base Frequency	800 MHz
Configurable TDP-down	7.5 W

Рисунок 3.10 – Технічні характеристики процесора Intel Core i5-7200U

TECHNICAL SPECIFICATIONS	Tesla P100	
Peak double-precision floating point performance (board)	4.7 Tflops	
Peak single-precision floating point performance (board)	9.3 Tflops	
Number of GPUs	1x GP100	
Number of CUDA cores	3584	
Memory size per board	12GB HBM2	16GB HBM2
Memory Interface	3072-bit	4096-bit
Memory bandwidth for board (ECC off)2	540 Gbytes/sec	720 Gbytes/sec
Thermal Solution	Passive	
Max Power Consumption	250W	
Form Factor	4.376"H x 10.5" L	
System	NVIDIA Certified Servers	

Рисунок 3.12 – Технічні характеристики GPU NVIDIA Tesla P100

Model	TUF Gaming GeForce RTX™ 3080 Ti
Graphic Engine	NVIDIA® GeForce RTX™ 3080 Ti
Bus Standard	PCI Express 4.0
OpenGL	0penGL4.6
Video Memory	12GB GDDR6X (Boost Clock) Gaming mode: 1665 MHzost Clock)
CUDA Core	10240
Memory Speed	19 Gbps
Memory Interface	384-bit
Resolution	Digital Max Resoution 7680 x 4320
Interface	Yes x 2 (Native HDMI 2.1) Yes x 3 (Native DisplayPort 1.4a)

Рисунок 3.13 – Технічні характеристики GPU NVIDIA GeForce RTX 3080 Ti

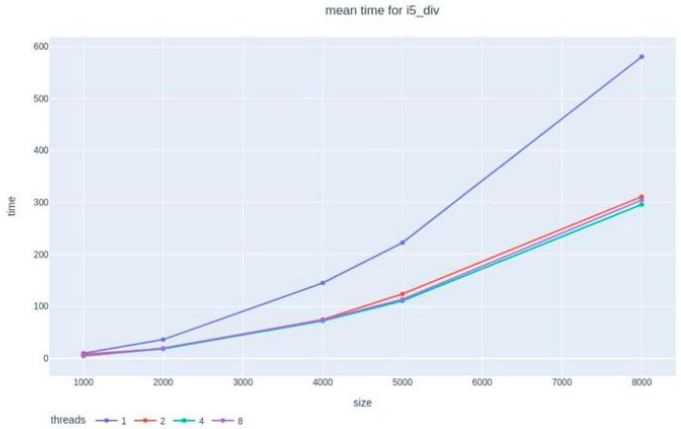


Рисунок 3.15 – Середній час виконання операції ділення матриці на скаляр на процесорі Intel Core i5-7200U (усереднено за 10 запусків)

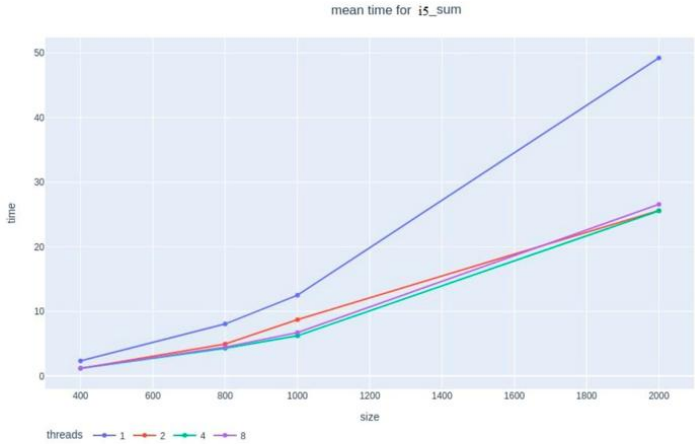


Рисунок 3.16 – Середній час виконання операції додавання матриць на процесорі Intel Core i5-7200U (усереднено за 10 запусків)

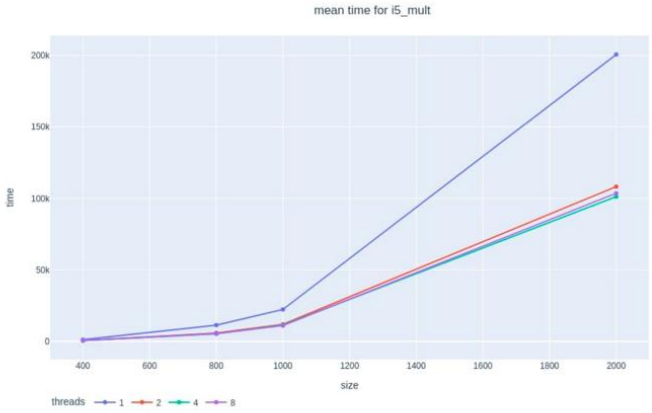


Рисунок 3.18 – Середній час виконання операції множення матриць на процесорі Intel Core i5-7200U (усереднено за 10 запусків)

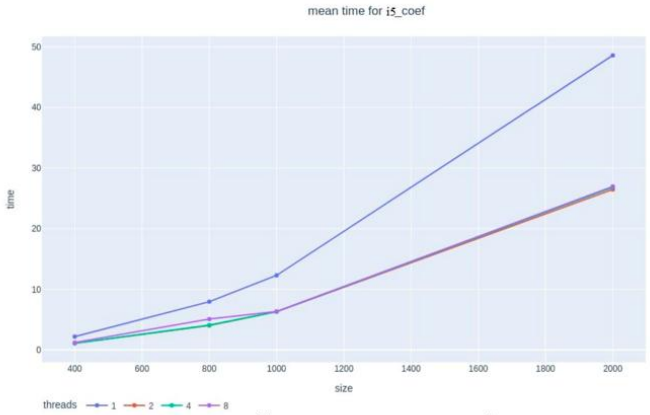


Рисунок 3.17 – Середній час виконання операції додавання матриць з коефіцієнтом (kA + B) на процесорі Intel Core i5-7200U (усереднено за 10 запусків)

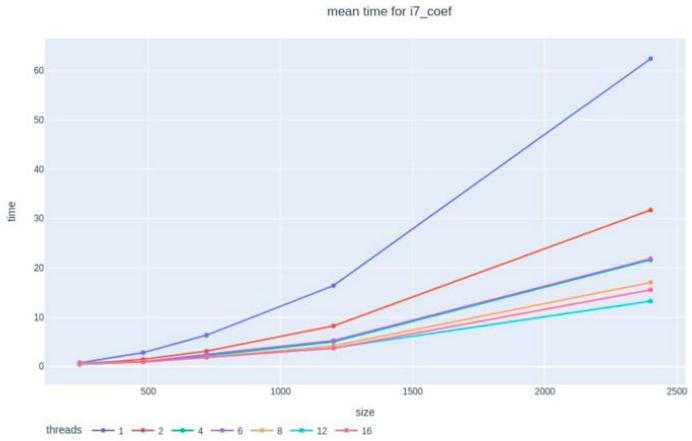


Рисунок 3.21 – Середній час виконання операції додавання матриць з коефіцієнтом (kA + B) на процесорі Intel Core i7-10850H

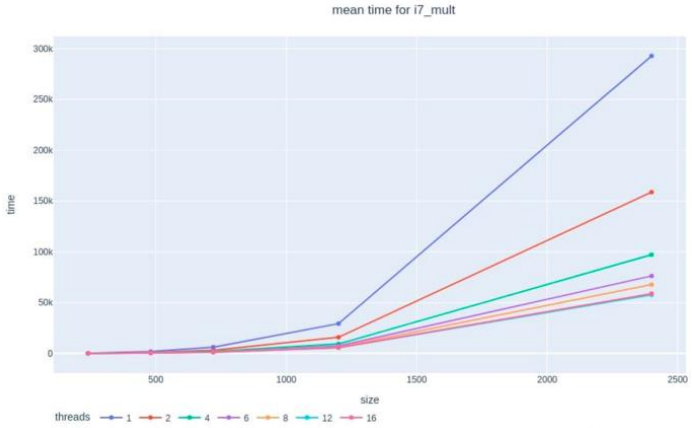


Рисунок 3.22 – Середній час виконання операції множення матриць на процесорі Intel Core i7-10850H (усереднено за 10 запусків)

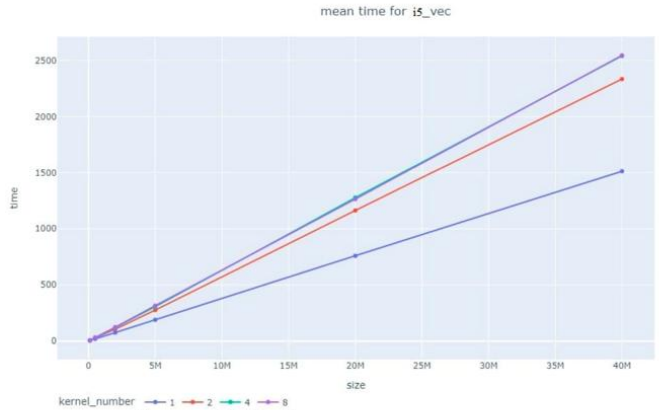


Рисунок 3.24 – Час виконання скалярного добутку на процесорі Intel Core i5-7200U (у мілісекундах, усереднено)

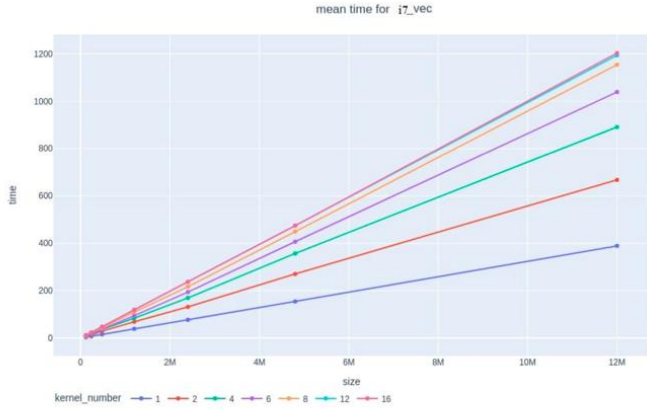


Рисунок 3.25 – Час виконання скалярного добутку на процесорі Intel Core i7-10850H (у мілісекундах, усереднено)

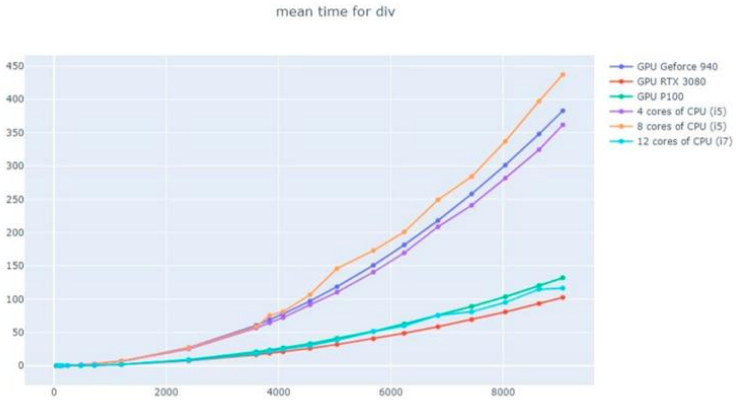


Рисунок 3.29 – Порівняльні результати виконання операції ділення матриці на скаляр на різних обчислювальних пристроях (усереднено за 10 запусків)

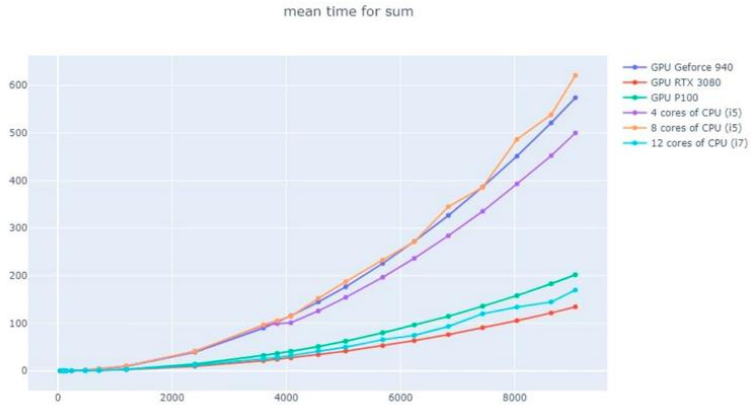


Рисунок 3.30 – Порівняльні результати виконання операції додавання матриць на різних обчислювальних пристроях (усереднено за 10 запусків)

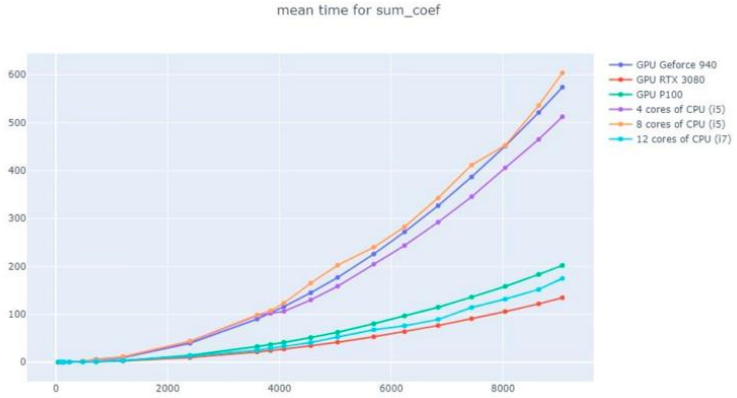


Рисунок 3.31 – Порівняльні результати виконання операції додавання матриць з коефіцієнтом (kA + B) на різних обчислювальних пристроях (усереднено за 10 запусків)



Рисунок 3.32 – Порівняльні результати виконання операції скалярного добутку векторів на різних обчислювальних пристроях (усереднено за 10 запусків)

Вплив обсягу даних, кількості потоків і типу задач на продуктивність

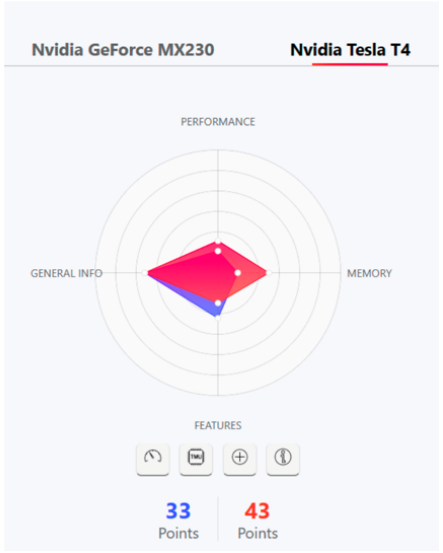
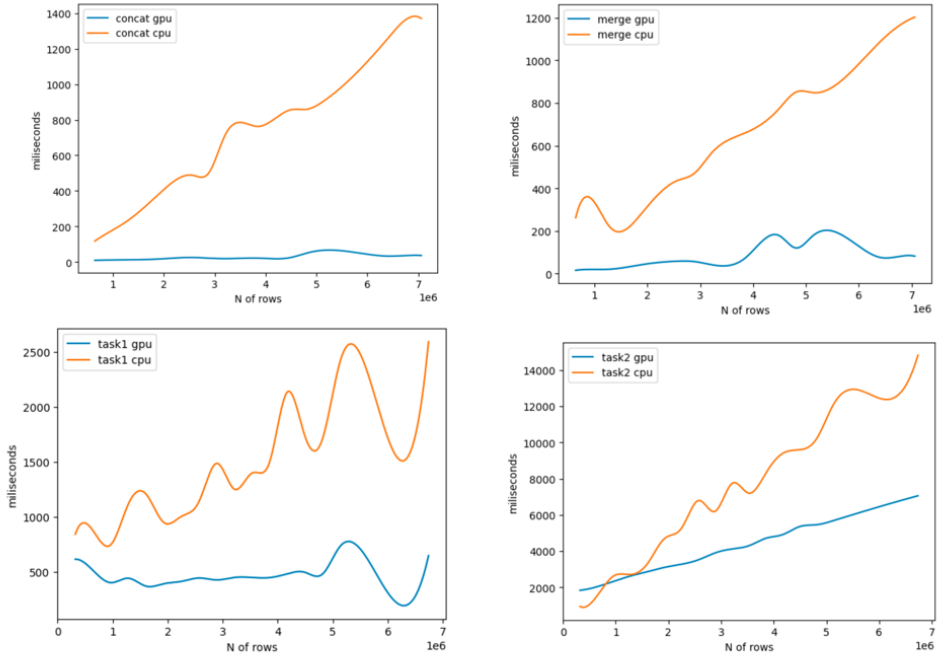


Рисунок 3.34 – Результати порівняння характеристик графічних процесорів Nvidia Tesla T4 та GeForce MX230 згідно з джерелом



Висновки

•У результаті виконання кваліфікаційної роботи було всебічно досліджено архітектуру, програмні засоби та продуктивність GPU у задачах загального призначення.

•Проаналізовано еволюцію GPU — від графічних прискорювачів до потужних універсальних обчислювальних платформ.

•Порівняльний аналіз CPU і GPU виявив ключові архітектурні та функціональні відмінності, що впливають на ефективність паралельних обчислень.

•Розглянуто сучасні платформи програмування GPU (CUDA, OpenCL) та бібліотеки прискорення, які значно спрощують реалізацію високопродуктивних обчислень.

•Експериментальні дослідження підтвердили значну перевагу GPU над CPU у задачах із високим рівнем паралелізму та великою кількістю обчислень.

•Запропоновано практичні рекомендації щодо доцільного використання GPU залежно від типу задачі та її обчислювального профілю.