МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

Спеціальність: 123 «Комп'ютерна інженерія»

Освітньо-професійна програма: «Комп'ютерна інженерія»

Група: 2БКС-29

КВАЛІФІКАЦІЙНА РОБОТА

здобувача освіти денної форми навчання

БКС.29.03.000.КРБ

БУХТЄЄВА
МАКСИМА ІГОРОВИЧА

м. Одеса
2025 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

Спеціальність: **123 «Комп'ютерна інженерія»**

Освітньо-професійна програма: **«Комп'ютерна інженерія»**

Група: **2БКС-29**

ПОЯСНЮВАЛЬНА ЗАПИСКА

До кваліфікаційної роботи бакалавра на тему: **Дослідження можливостей технологій GPGPU при виконанні SIMD-операцій**

Проектний матеріал складається з пояснювальної записки на 68 сторінках та графічного (презентаційного) матеріалу на 18 аркушах (слайдах)

Виконавець _____ (Бухтєєв М.І.)

Керівник проекту _____ (Кривченко А.А.)

Консультанти:

з розділу охорони праці та техніки безпеки _____ (Чорновол Н.І.)

з нормоконтролю _____ (Петрашова В.І.)

старший консультант _____ (Кривченко Ю.В.)

До захисту допущений

Завідувач кафедри _____ (Іванова Л.В.)

Завідувач відділення _____ (Краснокутська К.Г.)

Захист «28» 06 2025 р. Протокол ЕК № 3

Оцінка ЕК 5 (відмінно) / 90

Секретар ЕК _____

АНОТАЦІЯ

Випускна кваліфікаційна робота присвячена аналізу, розробці та апробації паралельних алгоритмів, що використовують обчислювальні можливості графічних процесорів для виконання SIMD-операцій. Метою роботи є створення ефективних методик розпаралелювання обчислювальних задач, зокрема множення матриць, обчислення скалярного добутку векторів та ділення матриці на число, що дозволяє досягнути високої продуктивності при обробці великих обсягів даних.

Процес виконання роботи розбитий на кілька основних етапів. На першому етапі було проведено аналіз літератури та досліджень у галузі паралельних обчислень і технологій GPGPU. Цей етап дозволив окреслити основні підходи та методи оптимізації обчислень з використанням SIMD-операцій. Другий етап присвячено проектуванню алгоритмів. Було розроблено концепції розподілених обчислень для множення матриць, скалярного добутку векторів та ділення матриці на число, враховуючи можливості паралелізації на графічних процесорах. Створено модульну архітектуру, що забезпечує ефективну синхронізацію потоків та оптимальний розподіл обчислювального навантаження між локальними підсистемами. На третьому етапі було реалізовано програмний застосунок із використанням технології CUDA. Розроблене ПЗ дозволяє проводити обчислення за допомогою паралельних алгоритмів, що імплементують SIMD-операції, з використанням різних кількостей потоків.

Четвертий етап охоплює експериментальне тестування розроблених алгоритмів на різних конфігураціях вхідних даних, зокрема для векторів та квадратних матриць із розмірністю. Проведено порівняльний аналіз продуктивності виконання задач на центральному та графічних процесорах, що дозволило визначити оптимальні параметри розпаралелювання та ідентифікувати ключові фактори, що впливають на ефективність обчислень.

Сформовано висновки та рекомендації щодо практичного застосування технологій GPGPU для реалізації високопродуктивних обчислювальних задач.

6. Консультанти по кваліфікаційній роботі, із зазначенням розділів, що їх стосуються

Розділ	Консультант	ПІДПИС	
		Завдання видав	Завдання прийняв
Основний розділ	*Кривченко А.А.*		
Розділ охорони праці	*Чорновол Н.І.*		
Нормоконтроль	*Петрашова В.І.*		
Старший консультант	*Кривченко Ю.В.*		

7. Дата видачі завдання _____

Керівник роботи *Кривченко А.А.* _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

Пор. №	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	*Вступ. Постановка задачі проектування*	19.05.2025	Виконав
2.	*Порівняння архітектури CPU та GPU*	19.05.2025	Виконав
3.	*Дослідження технологій GPGPU*	21.05.2025	Виконав
4.	*Можливості NVIDIA CUDA. Переваги і обмеження*	22.05.2025	Виконав
5.	*Апаратне забезпечення технології CUDA. Склад CUDA*	24.05.2025	Виконав
6.	*Модель програмування CUDA. Модель пам'яті CUDA*	26.05.2025	Виконав
7.	*Аналіз технології GPGPU для виконання SIMD-операцій*	27.05.2025	Виконав
8.	*Огляд засобів програмування GPGPU CUDA*	28.05.2025	Виконав
9.	*Реалізація SIMD-операцій з матрицями*	30.05.2025	Виконав
10.	*Тестування продуктивності обчислення скалярного*	02.06.2025	Виконав
	добутку векторів, множення та ділення матриць	04.06.2025	Виконав
11.	*Реалізація паралельного математичного алгоритму та*	09.06.2025	Виконав
	аналіз результатів		
12.	*Розробка питань з охорони праці*	10.06.2025	Виконав
13.	*Оформлення слайдів мультимедійної презентації*	12.06.2025.	Виконав

Здобувач освіти _____
(підпис)

Керівник роботи _____
(підпис)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

Відділення *Комп'ютерних систем* Кафедра *Комп'ютерної інженерії*

Спеціальність *123 «Комп'ютерна інженерія»*

Освітньо-професійна програма *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ:

Заст. дир. з НВР *Беркань І.В.*

"28" 05 2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу бакалавра

здобувачеві освіти *Бухтєєву Максиму Ігоровичу*

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Дослідження можливостей технологій GPGPU при виконанні SIMD-операцій*

затверджена наказом по коледжу від "24" 11 2024 р. № 246

2. Термін здачі студентом кваліфікаційної роботи 20.06.2025

3. Вихідні дані до роботи *1. Дослідити коефіцієнт прискорення виконання SIMD-операцій в залежності від кількості ядр для математичної задачі обробки матриць; 2. Тестування проводити з використанням розмірності векторів і квадратних матриць: N = 1024; N = 2048; N = 4096; 3. Обчислення проводити на CPU і на GPU з налаштуванням кількості паралельних потоків у режимах 32, 128 та 256; 4. Специфікації технології GPGPU CUDA*

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
Порівняння архітектури CPU та GPU; Аналіз технології GPGPU для виконання SIMD-операцій; Огляд засобів програмування GPGPU CUDA; Реалізація SIMD-операцій з матрицями за допомогою GPGPU CUDA; Тестування продуктивності обчислення матриці MX; Тестування продуктивності обчислення скалярного добутку векторів, множення матриць та ділення матриці на число; Питання охорони праці та техніки безпеки

5. Перелік графічного матеріалу (слайдів мультимедійної презентації) *Архітектура багатоядерного CPU; Архітектура GPU з SIMD-модулями; Порівняльний аналіз архітектури CPU та GPU; Архітектура CUDA; Ієрархія засобів програмування CUDA; Етапи компіляції застосунків GPGPU CUDA; Схема рішення задачі з використанням GPU; Схема розподілених обчислень для реалізації SIMD-операцій множення матриць за допомогою GPGPU CUDA; Алгоритм математичних операцій з матрицями, що реалізується з використанням GPGPU CUDA; Графіки коефіцієнту прискорення операцій з матрицями в залежності від кількості ядр; Графіки коефіцієнту ефективності операцій з матрицями в залежності від кількості ядр*

Формат	Зона	Поз.	Позначення	Назва	Кіл.	Примітка
				Документація		
			БКС 29. 03 000. 00 КРБ	Випускна робота		
А4			БКС 29. 03 000. 00 КРБ ПЗ	Пояснювальна записка	1	

Зм.	Арк.	№ докум.	Підпис	Дата	БКС 29. 03 000. 00 КРБ		
Розробив		Бухтєєв М.І.		16.06.25	Дослідження можливостей технологій GPGPU при виконанні SIMD-операцій	Літ.	Аркуш 4 / Аркушів 68
Перевірив		Кривченко А.А.		16.06.25		Н Д П	
Н. Контр.		Петрашова В.І.		16.06.25	ВСП "ОТФК ОНТУ" гр.2БКС-29		
Затверд.		Іванова Л.В.					

ЗМІСТ

ВСТУП

У сучасну епоху стрімкого розвитку обчислювальних технологій зростає потреба у високопродуктивних рішеннях, здатних ефективно опрацьовувати великі обсяги даних у режимі реального часу. Особливо актуальним стає використання паралельних обчислень у різноманітних галузях науки та інженерії, де необхідно скорочувати час виконання складних алгоритмічних задач. Графічні процесори (GPU) із їхньою високою паралельною обчислювальною потужністю завоювали популярність завдяки можливості реалізації загальнокорисних задач (GPGPU – General-Purpose Computing on Graphics Processing Units) поза межами традиційної графіки. Сучасна технологія NVIDIA CUDA дозволяє використовувати GPU не лише для візуальної обробки, а й для реалізації масштабованих паралельних алгоритмів, зокрема на основі SIMD-операцій (Single Instruction, Multiple Data). Це відкриває нові горизонти для оптимізації обчислювальних процесів, серед яких особливе місце займають задачі над матрицями, що є невід'ємною частиною широкого спектру наукових розрахунків та інженерних застосувань.

Метою даної роботи є дослідження та аналіз можливостей технологій GPGPU при виконанні типових SIMD-операцій, зокрема операцій над матрицями, з порівнянням часу виконання та ефективності розв’язань на CPU і GPU. Досягнення цієї мети передбачає розробку програмного продукту, що реалізує матричні обчислення – такі як множення матриць, додавання матриць, скалярний добуток векторів, додавання із коефіцієнтом, а також ділення матриці на число – в режимі паралельного виконання. Важливим аспектом даного проекту є впровадження механізмів синхронізації потоків, що включають бар’єри та атомарні операції, а також застосування системи прапорців у глобальній пам’яті для забезпечення взаємодії потоків із різних блоків.

У роботі буде проведений комплексний аналіз продуктивності обраних алгоритмів як у послідовному виконанні на центральному процесорі, так і у паралельному режимі на GPU. Результати дослідження планується представити у вигляді графічних залежностей.

1 ОСНОВНА ЧАСТИНА

1.1 Порівняння архітектури CPU та GPU

Центральний процесор (CPU) та графічний процесор (GPU) спроектовані з різними акцентами, що відображається у їх архітектурних особливостях та завданнях, для яких вони оптимізовані. Розуміння відмінностей між цими архітектурами є ключовим для ефективного застосування технологій GPGPU, зокрема при виконанні SIMD-операцій.

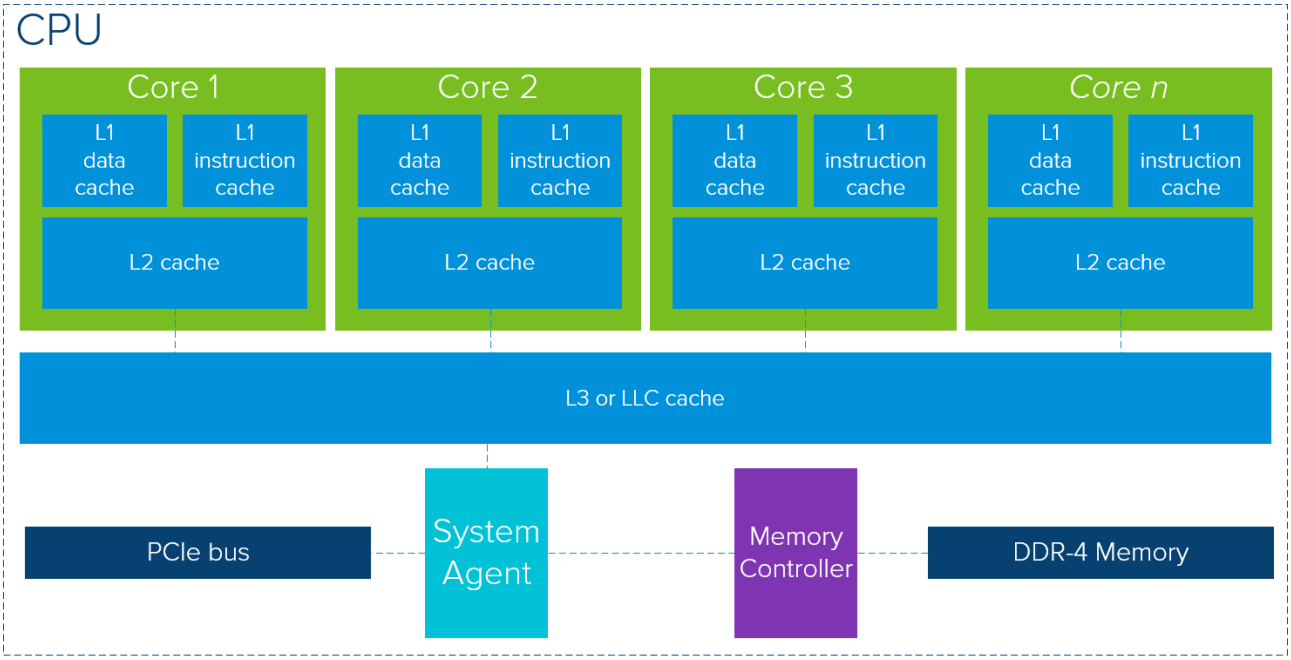


Рисунок 1.1. Архітектура CPU на прикладі багатоядерного процесора з DDR4

CPU (рис.1.1) розроблено для узагальненого виконання широкого спектра алгоритмів, що вимагають високої продуктивності при послідовних обчисленнях. Основні характеристики архітектури CPU включають:

- Невелика кількість високопродуктивних ядер: Сучасні центральні процесори зазвичай містять від 4 до 16 ядер, кожне з яких здатне виконувати складні обчислення завдяки високій тактовій частоті та розвиненій логіці керування;
- Складна ієрархія кеш-пам’яті: CPU оснащені багаторівневими кешами (L1, L2, L3), що дозволяє мінімізувати затримки при доступі до оперативної пам’яті та забезпечує високу продуктивність при обчисленнях із

нерегулярним доступом до даних;

- Розгалуження та прогнозування переходів: Завдяки складним алгоритмам прогнозування, CPU здатний ефективно обробляти умовні оператори та розгалужені інструкції, що суттєво впливає на продуктивність у програмному забезпеченні загального призначення.

Ці особливості роблять CPU оптимальним для завдань з високою варіативністю обчислювальних сценаріїв, де домінує послідовне виконання інструкцій.

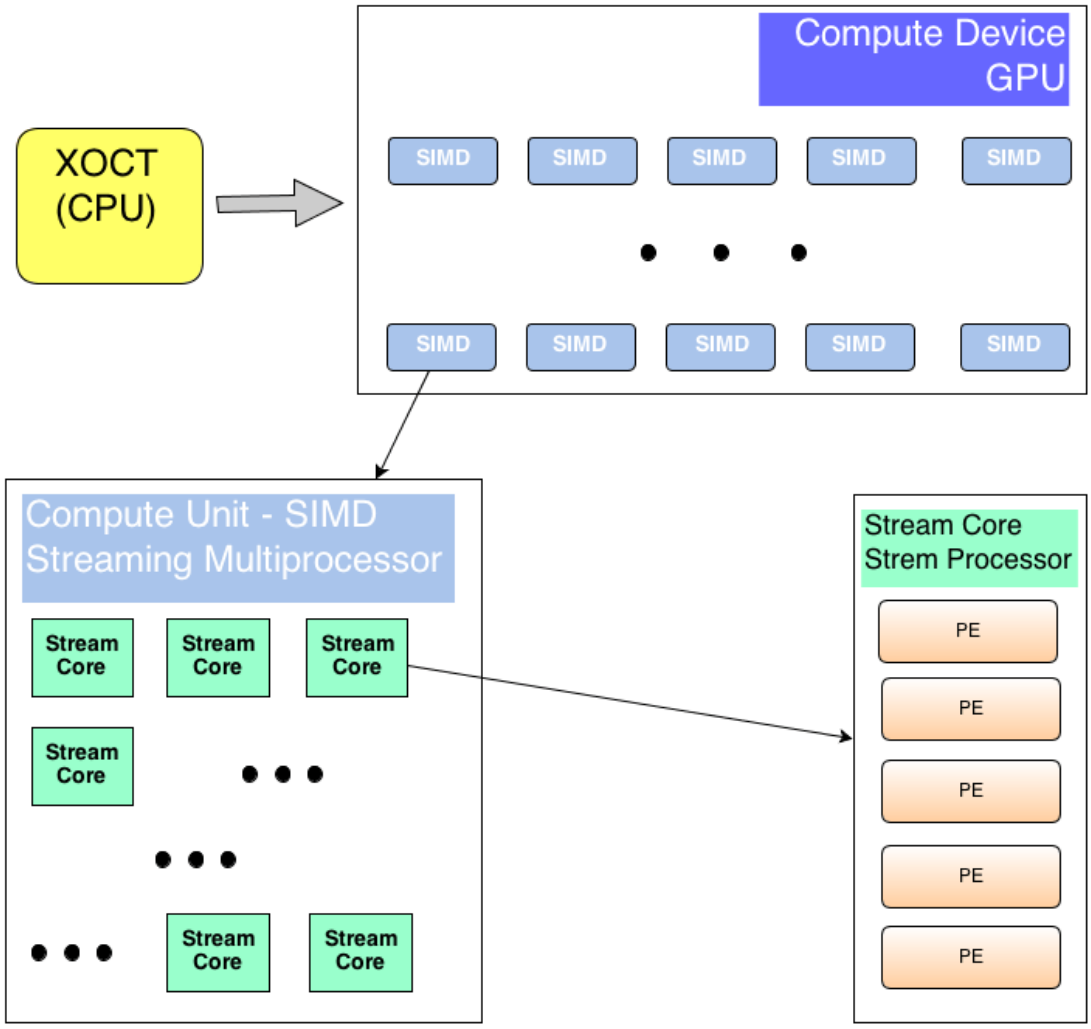


Рисунок 1.2. Архітектура GPU з SIMD-модулями

GPU (рис.1.2) спроектовано з акцентом на масовий паралелізм, що робить його ідеальним у середовищах, де виконується велика кількість однотипних обчислень. Основні аспекти архітектури GPU включають:

- Велика кількість простих ядер: Графічний процесор містить сотні і навіть тисячі спрощених обчислювальних ядер, організованих у блоки або

мультипроцесори. Така організація дозволяє одночасно запускати тисячі потоків, використовуючи модель SIMT (Single Instruction, Multiple Threads);

- Спеціалізоване керування пам'яттю: GPU використовують багаторівневий підхід до організації пам'яті, що включає регістри, спільну пам'ять (shared memory) всередині мультипроцесорних блоків, а також глобальну пам'ять із високою пропускною здатністю. При цьому кешування організоване менш гнучко, але з акцентом на забезпечення високого обсягу переданих даних;
- Масовий паралелізм замість високої тактової частоти: Хоча окреме ядро GPU працює на нижчих тактових частотах порівняно з CPU, загальна продуктивність системи забезпечується завдяки численним паралельним обчисленням, що дозволяє значно скоротити час виконання однотипних операцій.

Таблиця 1.1. Порівняльний аналіз архітектури CPU та GPU

Характеристика	**CPU**	**GPU**
Кількість ядр	Невелика (4–16 високопродуктивних ядр)	Велика (сотні-тисячі простих обчислювальних ядр)
Призначення	Загальне призначення, оптимізація для послідовних обчислень та складних алгоритмів	Оптимізований для масових паралельних обчислень, підходить для задач типу SIMD/SIMT
Тактова частота	Висока; забезпечує швидке послідовне виконання	Низька; компенсується великим числом потоків
Система кешування	Розвинена ієрархія кеш-пам'яті (L1, L2, L3)	Менша складність кешування, з високою пропускною здатністю глобальної пам'яті
Фокус	Гнучкість та обробка розгалужених інструкцій	Ефективність при виконанні одноманітних обчислювальних завдань

Завдяки таким характеристикам, GPU є незамінним інструментом для обчислювальних задач, що легко піддаються векторизації, як-от операції над матрицями, де кожна операція може виконуватися одночасно для великої кількості елементів.

Основна різниця між CPU та GPU полягає не стільки в швидкості окремих

обчислень, скільки в підході до організації обчислень: CPU орієнтується на послідовне виконання складних інструкцій із високою точністю, тоді як GPU спеціалізований на одночасному виконанні великої кількості простих операцій. Ця особливість GPU є критичною для реалізації паралельних алгоритмів, таких як множення матриць, що лежать в основі SIMD-операцій, і забезпечує значне прискорення обчислювальних задач порівняно з традиційним центральним процесором.

1.2 Аналіз технології GPGPU для виконання SIMD-операцій

Технологія GPGPU (General-Purpose Computing on Graphics Processing Units) дозволяє використовувати графічні процесори для виконання обчислювальних задач загального призначення, що виходять за рамки обробки графіки. Однією з ключових особливостей даного підходу є можливість масового паралелізму за допомогою SIMD-операцій (Single Instruction, Multiple Data), при якому одна інструкція виконується над великою кількістю даних одночасно. Це дозволяє значно пришвидшувати обчислення, зокрема у задачах над матрицями, векторних обчисленнях та інших обчислювально інтенсивних алгоритмах.

Найпоширенішими технологіями для реалізації GPGPU є NVIDIA CUDA та OpenCL. Проте в рамках даного дослідження основна увага приділяється платформі NVIDIA CUDA за її широкою підтримкою, розвиненою екосистемою інструментів та спроможністю ефективно працювати з великою кількістю потоків. Архітектура CUDA реалізує модель SIMT (Single Instruction, Multiple Threads), яка концептуально є подібною до SIMD, але дозволяє кожному потоку отримувати окремий доступ до даних. Це дає змогу адаптувати алгоритми до специфічних потреб паралельних обчислень та оптимізувати роботу з пам’яттю.

При виконанні SIMD-операцій за допомогою технологій GPGPU основною перевагою є можливість одночасного виконання тисяч потоків. При цьому концепції синхронізації набувають великого значення. Внутрішньо CUDA забезпечує синхронізацію потоків у межах одного блоку за допомогою вбудованих бар’єрів (функція __syncthreads()), що дозволяє координувати

обчислення всередині мультипроцесора. Для задач, де необхідна синхронізація потоків із різних блоків, розробляються спеціальні методи, такі як використання системи прапорців у глобальній пам’яті, що дозволяє забезпечити правильну послідовність виконання операцій та уникнути ситуацій "гонок" даних. Таким чином, можливості синхронізації є критично важливими для ефективної реалізації паралельних алгоритмів на GPU.

Ефективне використання технологій GPGPU часто залежить від правильної організації пам’яті. У рамках CUDA розрізняють декілька типів пам’яті: регістри, спільну (shared) пам’ять, глобальну пам’ять та кеші певного рівня. Використання спільної пам’яті дозволяє суттєво зменшити затримки при доступі до даних, що є надзвичайно важливим при виконанні SIMD-операцій, де кожна операція виконується паралельно над елементом великого обсягу даних. Водночас робота із глобальною пам’яттю вимагає додаткових зусиль для оптимізації, оскільки доступ до неї може бути повільнішим через високий рівень латентності.

Серед основних переваг використання GPGPU для SIMD-операцій можна виділити:

- Масовий паралелізм. GPU здатний одночасно обробляти тисячі потоків, що дозволяє досягти значного прискорення при обчисленнях над великими масивами даних;

- Висока пропускна здатність пам’яті. Завдяки спеціалізованій архітектурі, GPU забезпечує паралельний доступ до пам’яті, що є важливим фактором при роботі з великими обчислювальними задачами.

Проте, технології GPGPU мають і певні обмеження:

- Ситуації з нерегулярним доступом до даних. Алгоритми, що сильно залежать від нерегулярної роботи з пам’яттю, можуть не отримувати повної вигоди від паралелізму;

- Складність реалізації та налагодження. Розробка оптимізованих алгоритмів для GPU може вимагати додаткових зусиль щодо гармонізації синхронізації потоків і налагодження продуктивності;

- Механізми синхронізації. Хоча в межах одного блоку синхронізація

організована ефективно, керування синхронізацією між блоками вимагає спеціальних підходів, що може ускладнювати розробку та впровадження алгоритмів.

З позиції розробника, графічний конвеєр представляє собою послідовність етапів обробки, де кожен крок відповідає за певні трансформації даних. Наприклад, модуль геометричної обробки формує трикутникові примітиви, а модуль растеризації перетворює їх на піксельні дані, готові для виведення на екран (рис.1.3). Традиційна схема GPGPU-програмування базується на використанні графічних процесів для реалізації загальних обчислень, що вимагало обертання навіть елементарних операцій, таких як поелементне складання векторів, через механізми візуалізації.

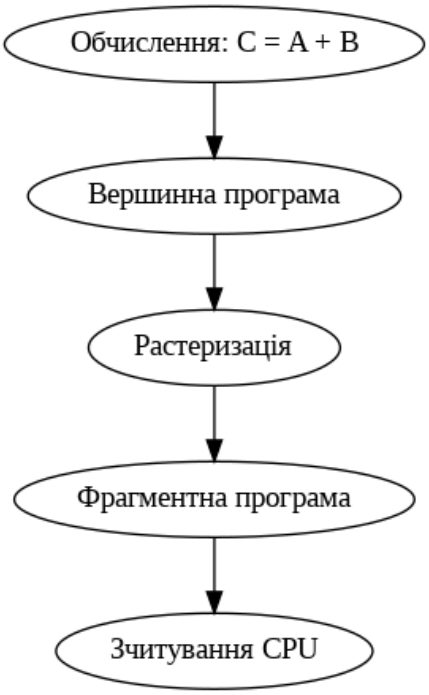


Рисунок 1.3. Послідовність виконання програми з використанням GPGPU

У класичній моделі, щоб перенести обчислення на GPU, необхідно інтегрувати їх у графічний конвеєр. Зрештою, навіть якщо мова йде про просте додавання двох векторів, система повинна ініціювати процес побудови зображення: визначається фігура або об’єкт, що потім растеризується, а колір кожного пікселя обчислюється за допомогою програмного коду, що називається піксельним шейдером. Шейдер зчитує дані з відповідних текстурних блоків, проводить необхідні арифметичні операції і записує результат у вихідний буфер.

Такий підхід додає зайві рівні абстракції, призводячи до значного ускладнення як розробки, так і налагодження програм, адже піксельний шейдер є лише засобом вираження математичних залежностей кольору від координат, записаним мовою з С-подібним синтаксисом.

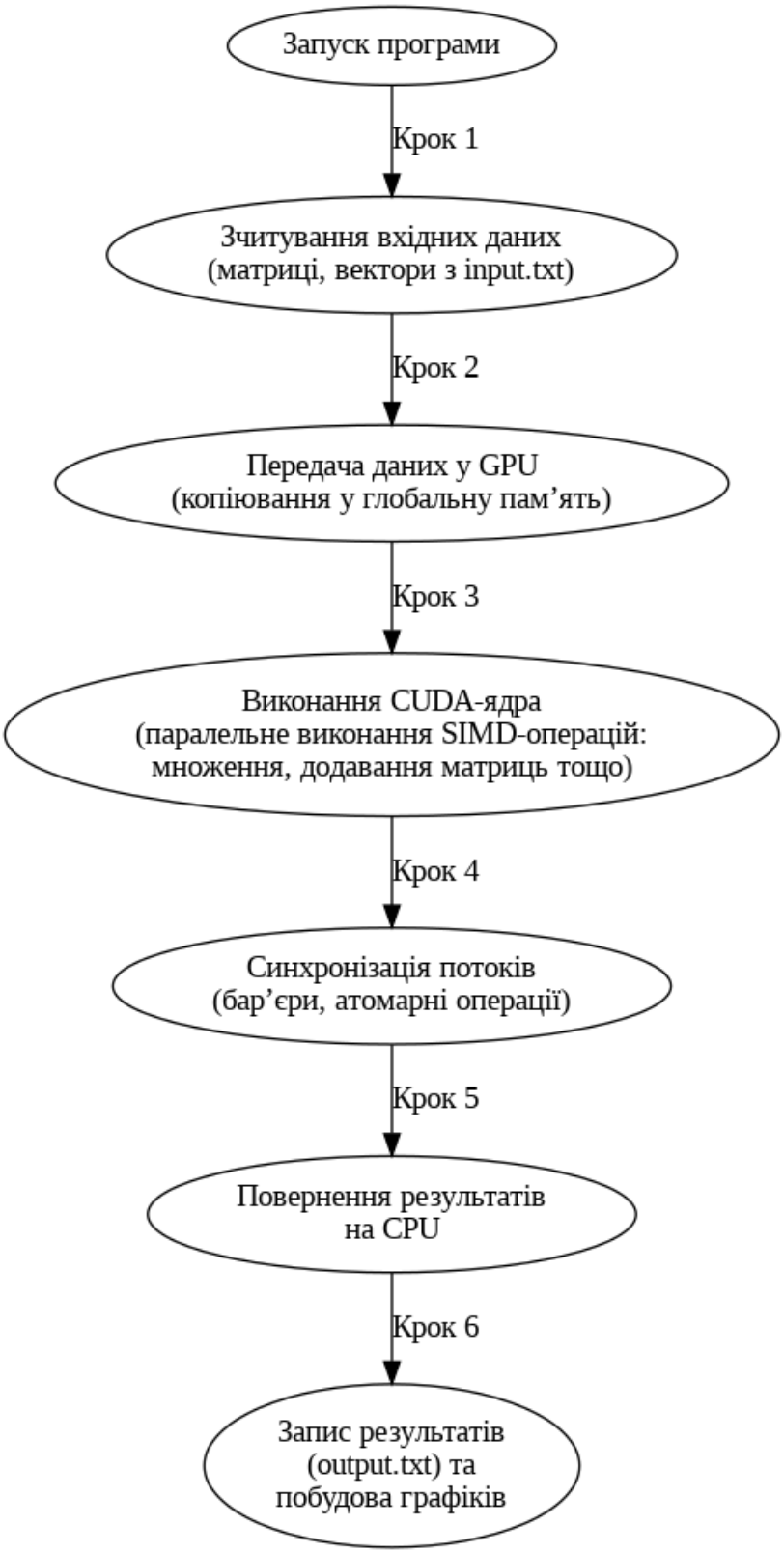


Рисунок 1.4. Схема рішення задачі з використанням SIMD-операцій GPGPU

Реалізація обчислень засобами GPGPU дозволяє значно знизити обсяг передач даних через глобальну пам’ять відеочіпа (рис.1.4) і оптимізувати загальну продуктивність системи. Цей підхід відкриває нові можливості для ефективного вирішення обчислювальних задач загального призначення на основі технологій GPGPU.

Аналіз технологій GPGPU показує, що використання графічного процесора для виконання SIMD-операцій відкриває значний потенціал для оптимізації обчислень, особливо при роботі з великими обчислювальними задачами, такими як операції над матрицями. Розуміння особливостей синхронізації, організації пам’яті та можливостей масового паралелізму дозволяє ефективно використовувати платформу CUDA для реалізації високопродуктивних алгоритмів. У подальших розділах роботи буде розглянуто конкретні приклади застосування даних технологій та проведено порівняльний аналіз продуктивності обчислювальних систем на базі CPU і GPU.

1.3 Огляд засобів програмування GPGPU CUDA

Сучасна парадигма обчислень на графічних процесорах, що реалізована за допомогою технології NVIDIA CUDA, суттєво відрізняється від традиційних підходів до GPGPU. Завдяки можливості програмування на стандартній мові Сі з мінімальними розширеннями, розробники отримують прямий доступ до обчислювальних ресурсів GPU, не обтяжуючись використанням графічних API. Це дозволяє створювати високопродуктивні програми для виконання SIMD-операцій без необхідності обходити графічний конвеєр, що використовувався у попередніх моделях GPGPU.

Основні переваги використання засобів програмування CUDA включають:

- Знайомий синтаксис. Програмування в CUDA базується на мові Сі, що є звичним і широко розповсюдженим серед розробників. Додаткові розширення для доступу до ресурсів GPU є інтуїтивно зрозумілими, що значно сприяє швидкому освоєнню технології;

- Доступ до спільної пам’яті. На кожному мультипроцесорі відеочіпа

доступно до 16 Кб спеціально організованої пам’яті для обміну даними між потоками. Ця спільна пам’ять може використовуватися як високошвидкісний кеш, що забезпечує стабільну смугу пропускання і робить обчислення більш ефективними порівняно з класичними текстурними вибірками;

- Оптимізована передача даних. Завдяки ретельно продуманій організації зв’язку між системною оперативною пам’яттю та пам’яттю GPU, CUDA мінімізує затримки при передачі даних, що є критично важливим при обробці великих наборів даних;

- Незалежність від графічних API. Відсутність необхідності використовувати інтерфейси, орієнтовані на рендеринг, дозволяє звести до мінімуму накладні витрати, пов’язані з обробкою графіки, і сфокусувати ресурси саме на паралельних обчисленнях;

- Гнучка адресація пам’яті. Підтримка лінійної адресації, а також операцій типу gather та scatter, дозволяє здійснювати читання і запис даних за довільними адресами, що розширює можливості алгоритмічних рішень;

- Апаратна підтримка цілочисельних та бітових операцій. Закріплена апаратна реалізація таких операцій дозволяє ефективно виконувати широкий спектр математичних завдань, необхідних для високопродуктивних додатків.

Проте, незважаючи на численні переваги, технологія CUDA має і певні обмеження, які впливають на розробку додатків:

- Недоступність рекурсії. Функції, що виконуються на GPU, не підтримують рекурсивні виклики, що обмежує застосування ряду алгоритмічних конструкцій, що потребують рекурсивної логіки;

- Фіксована організація потоків. Потоки в CUDA групуються у варпи по 32 елементи, що є мінімальним об’ємом даних, який може бути оброблений в межах одного мультипроцесора. Крім того, розміри блоків, що використовуються для організації потоків, зазвичай компонуються у діапазоні від 64 до 512 потоків, що накладає певні обмеження на можливу масштабованість паралельних обчислень;

- Профілювання та оптимізація. Першим етапом перенесення існуючих алгоритмів на платформу CUDA є детальне профілювання коду з метою

ідентифікації «вузьких місць». Виявлені фрагменти, що характеризуються високою обчислювальною інтенсивністю, переробляються із застосуванням специфічних Сі-розширень CUDA та виконуються у вигляді CUDA-ядра (kernel).

У процесі компіляції застосовується спеціалізований компілятор від NVIDIA, який генерує код як для CPU, так і для GPU. Під час виконання програми центральний процесор займається послідовною обробкою менш інтенсивних завдань, тоді як GPU виконує паралельні обчислення, що реалізовані через CUDA-ядра. Усередині ядра кожному елементу даних відповідає окремий потік, з власними регістрами та внутрішнім станом, що дозволяє обробляти мільйони елементів одночасно.

Потоки організовуються в варпи, які розподіляються між мультипроцесорами GPU з метою максимізації паралелізму та мінімізації затримок при перемиканні контексту.

Засоби програмування на базі технології CUDA створюють нові можливості для розробки високопродуктивних паралельних алгоритмів, що особливо актуально при вирішенні задач типу SIMD-операцій. Подальші підпункти даного підрозділу детально розглянуть ключові компоненти середовища CUDA, механізми синхронізації та методи оптимізації коду для досягнення максимальної ефективності обчислень на GPU.

1.3.1 Рівні обробки даних GPGPU CUDA

Технологія NVIDIA CUDA побудована на багаторівневій архітектурі, що дозволяє організувати доступ до ресурсів GPU у вигляді чітко структурованих шарів. На базовому рівні знаходяться два основні API: високорівневий runtime і низькорівневий Driver API. Оскільки одночасне застосування обох у одному проєкті є неможливим, розробник повинен обрати один із підходів. Високорівневий API виконує роль обгортки, трансформуючи виклики з середовища виконання у прості інструкції, які потім обробляються драйвером. Проте навіть цей рівень вимагає розуміння специфіки роботи апаратного забезпечення NVIDIA, оскільки повна абстракція від апаратної деталізації відсутня (рис. 1.5).

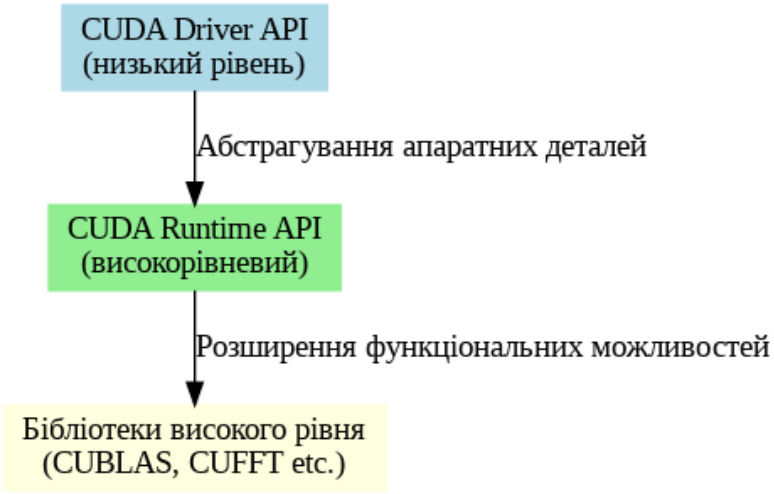


Рисунок 1.5. Ієрархія засобів програмування CUDA

Окрім базових API, CUDA пропонує ще один шар – набір високопродуктивних бібліотек, що значно спрощують розробку складних обчислювальних алгоритмів. Серед таких засобів особливо виділяються:

- CUBLAS – це реалізація стандарту BLAS (Basic Linear Algebra Subprograms) для GPU, яка оптимізована для задач лінійної алгебри. Бібліотека забезпечує швидке виконання операцій над векторами та матрицями, починаючи від простих векторно-векторних обчислень і закінчуючи повноцінними матричними операціями. Робота з CUBLAS зводиться до створення об’єктів (матриць, векторів) у пам’яті GPU, заповнення їх вихідними даними, виклику відповідних функцій і повернення результатів у системну пам’ять. Завдяки спеціальним процедурам для створення та знищення об’єктів, а також для читання і запису даних, бібліотека дозволяє ефективно використовувати апаратні можливості відеочіпів;

- CUFFT – це CUDA-адаптована бібліотека для реалізації алгоритмів швидкого перетворення Фур’є. Вона забезпечує простий і оптимізований інтерфейс для обчислення FFT на GPU, що є надзвичайно важливим при аналізі сигналів, фільтрації даних та інших завданнях із обробки частотної інформації. Бібліотека підтримує 1D, 2D та 3D перетворення як для комплексних, так і для дійсних даних, дозволяючи виконувати пакетні обчислення для кількох трансформацій одночасно. При цьому CUFFT дозволяє працювати з різними типами перетворень, зокрема complex→complex (C2C), real→complex (R2C) та

complex→real (C2R), що робить її універсальним інструментом для широкого спектру завдань.

Таким чином, багаторівнева модель обробки даних в CUDA включає низькорівневий Driver API, що забезпечує прямий доступ до апаратних ресурсів, поряд із високорівневим середовищем runtime, яке спрощує розробку, але не приховує всю специфіку графічних процесорів. Додатковий шар високопродуктивних бібліотек, таких як CUBLAS і CUFFT, ще більше розширює можливості розробників, дозволяючи використовувати перевірені алгоритмічні рішення для найпоширеніших задач обчислювальної математики та аналізу даних.

1.3.2 Паралельна модель обчислень GPGPU CUDA

Паралельна модель обчислень, запропонована технологією CUDA, базується на принципі одночасного виконання однакових інструкцій над різними наборами даних, що відповідає парадигмі SIMD. У цьому контексті GPU виступає як спеціалізований співпроцесор центрального процесора (host), який має власну пам’ять і здатний паралельно обробляти мільйони елементів завдяки надзвичайно великій кількості логічних потоків.

Основна одиниця обчислення в середовищі CUDA – це ядро (kernel). Воно представляє собою функцію, яку запускає GPU, при цьому кожен окремий потік виконує цю функцію над своїм набором даних. Відмінною особливістю є те, що кожен потік працює в режимі скалярних обчислень, що дозволяє уникнути примусової упаковки даних у вектори з чотирма компонентами. Такий підхід є більш зручним і природним для вирішення широкого спектру завдань, що вимагають обробки окремих елементів.

Організація обчислень у CUDA базується на групуванні потоків у так звані блоки (thread blocks). Ці блоки можуть утворювати одновимірні або двовимірні сітки, а кожен блок забезпечує взаємодію потоків через спільну (shared) пам’ять і точки синхронізації. Саме виклик ядра організовують над сіткою, або grid, що складається з численних блоків. Ілюстрація (рис. 1.6) демонструє стекову структуру: всі потоки, об’єднані в блоки, разом утворюють сітку, що забезпечує виконання обчислювального завдання.

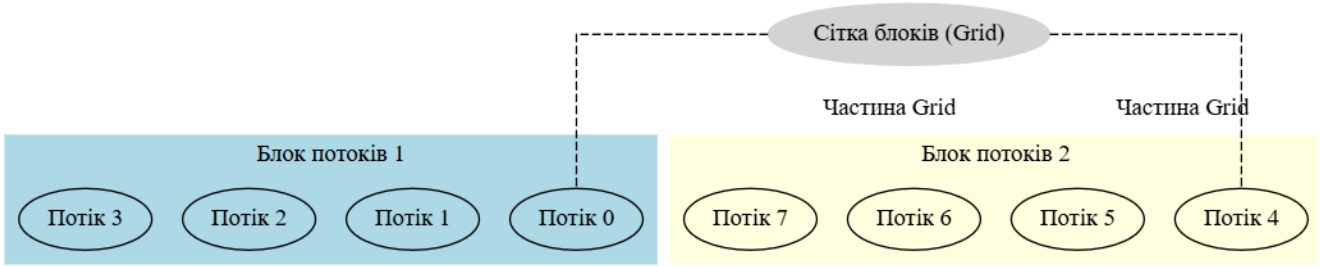


Рисунок 1.6. Організація потоків в паралельній моделі обчислень GPGPU

Гнучка організація блоків дає змогу запускати значну кількість потоків за один виклик ядра, що є суттєвим для масштабування обчислень. Якщо апаратна платформа не може одночасно обробити всю сітку блоків, система автоматично виконує їх поступово, у послідовному режимі. Проте при наявності достатніх ресурсів блоки можуть обчислюватися паралельно, що дозволяє досягти високої продуктивності навіть на GPU різних категорій – від мобільних до високопродуктивних інтегрованих рішень.

Паралельна модель обчислень у CUDA – це ефективний механізм організації великомасштабної обробки даних, який дозволяє розробникам оптимально розподіляти обчислювальне навантаження і використовувати апаратні можливості графічного процесора для реалізації високопродуктивних SIMD-операцій.

1.3.3 Модель доступу до пам'яті GPGPU CUDA

Архітектура пам’яті у CUDA побудована з урахуванням потреб високопродуктивних паралельних обчислень, забезпечуючи гнучкий, побайтовий доступ до даних та підтримку операцій типу gather і scatter (рис. 1.7). Така модель дозволяє розробникам максимально ефективно організовувати обчислення, враховуючи особливості різних типів пам’яті, що працюють у тісній взаємодії.

Глобальна пам’ять. Це найбільший об’єм доступної пам’яті, який використовується всіма мультипроцесорами GPU. Доступ до глобальної пам’яті характеризується великими затримками, що можуть сягати сотень тактів. Типові операції load і store виконуються за допомогою звичайних вказівників, проте відсутність власного кешування негативно позначається на швидкодії.

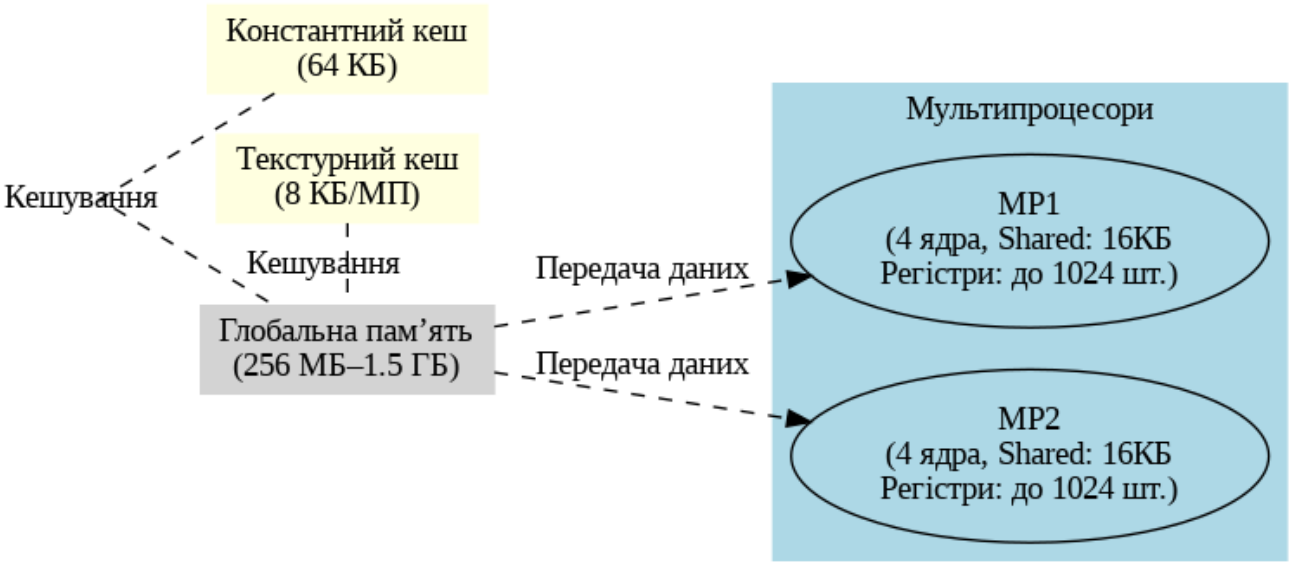


Рисунок 1.7. Розподіл пам'яті та обчислювальних ресурсів GPGPU CUDA

Локальна пам'ять. Цей невеликий сегмент пам'яті відведено для кожного потоку окремо, використовується як розширення регістрів, коли їхній обсяг вичерпано. Хоча логічно розділена для кожного потоку, локальна пам'ять працює на тих же характеристиках затримки, що й глобальна, що робить її менш оптимальною для операцій, де критично важлива швидкість.

Розділювана пам'ять (Shared Memory). Цей тип пам'яті зазвичай має розмір близько 16 КБ на мультипроцесор сучасних GPU. Вона забезпечує надзвичайно низькі затримки, подібними до регістрів, і організована таким чином, щоб усі потоки в одному блоці могли оперативно взаємодіяти, обмінюючись даними. Розділювана пам'ять часто слугує керованим програмістом кешем першого рівня, що дозволяє значно зменшити кількість звернень до повільнішої глобальної пам'яті.

Пам'ять констант. Цей блок пам'яті обсягом приблизно 64 КБ доступний для читання всіх мультипроцесорів. Завдяки власному кешуванню (до 8 КБ на кожний мультипроцесор), константна пам'ять підходить для зберігання даних, які не змінюються протягом виконання ядра. Однак при відсутності даних у кеші доступ може бути досить повільним.

Текстурна пам'ять. Також надається спеціальний сегмент пам'яті для отримання даних із застосуванням текстурних вибірок. Хоча цей тип пам'яті, зокрема кешований (до 8 КБ на мультипроцесор), забезпечує додаткові

можливості, як-от лінійна інтерполяція, він характеризується затримками, схожими на глобальну пам’ять у випадках невдалого кешування.

Варто зазначити, що глобальна, локальна, текстурна і константна пам’ять фізично представляють одну й ту ж відеопам’ять, проте використовуються за різними схемами доступу та кешування. Ззовні CPU може взаємодіяти безпосередньо лише з глобальною, константною та текстурною пам’яттю.

В умовах високопродуктивних обчислень правильна організація адресації пам’яті має вирішальне значення. Розробник повинен враховувати, що доступ до глобальної та локальної пам’яті є значно повільнішим у порівнянні з регістрами чи розділюваною пам’яттю. Для оптимізації обчислювальних задач типово застосовують наступний алгоритм розподілу даних:

- Декомпозиція задачі: основна задача розбивається на менші підзадачі, що дозволяє працювати з позитивно обмеженими обсягами даних;

- Розбиття даних: вхідні дані діляться на блоки, які можна розмістити у розділюваній пам’яті;

- Розподіл обчислень: кожен блок обробляється окремим набором потоків, що працюють у межах одного блоку (thread block);

- Копіювання в розділювану пам’ять: дані підблоку завантажуються з глобальної пам’яті до швидкодіючої розділюваної пам’яті;

- Обчислення: виконання арифметичних операцій здійснюється безпосередньо у розділюваній пам’яті;

- Повернення результатів: після завершення обчислень результати копіюються з розділюваної пам’яті назад у глобальну пам’ять для подальшої обробки чи відображення.

Таке планування (рис.1.8) дозволяє мінімізувати затримки при доступі до даних і використовувати високошвидкісні регістри та розділювану пам’ять для знаходження максимальної продуктивності обчислень. У цьому контексті спеціальний підхід до розробки алгоритмів з використанням CUDA є обов’язковим, враховуючи значну різницю в затримках доступу між різними типами пам’яті та особливостями їх кешування.

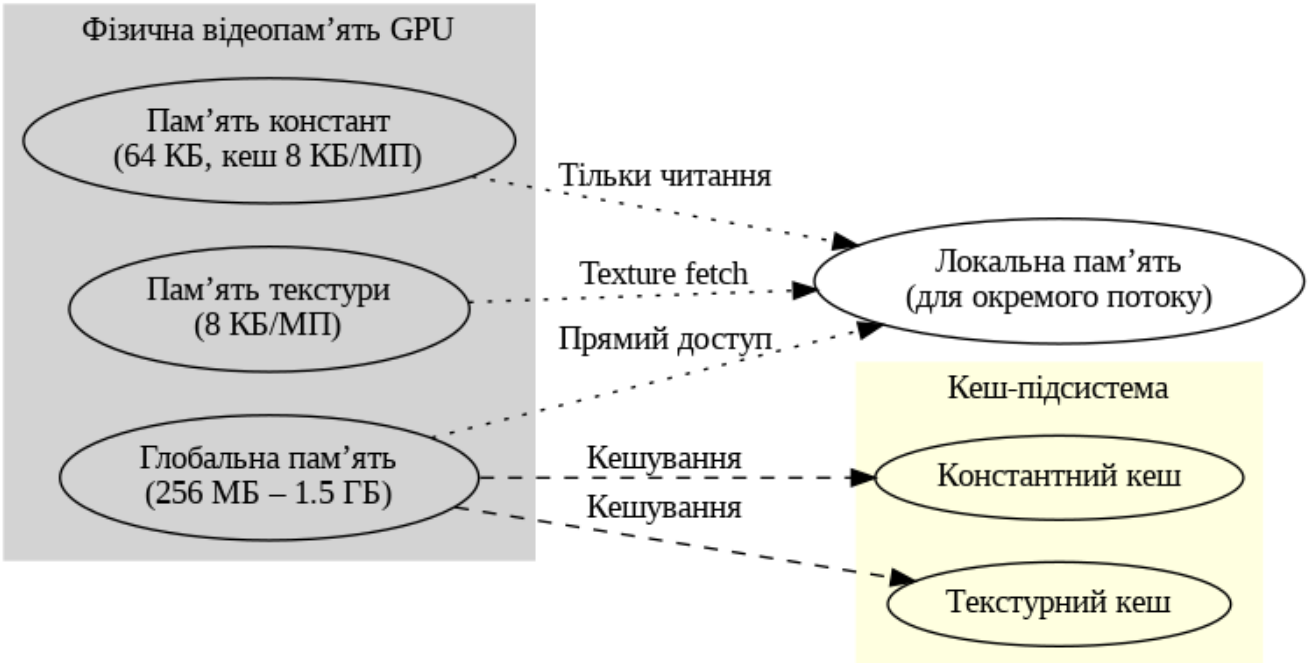


Рисунок 1.8. Доступ до фізичної відеопам'яті GPU по сегментам

1.3.4 Огляд засобів розробки застосунків GPGPU CUDA

Сучасне програмування на базі CUDA спирається на розвинений комплекс засобів розробки, який забезпечує підтримку всіх етапів створення застосунків – від написання коду та налагодження до профілювання продуктивності і оптимізації. У цьому підрозділі розглядаються основні інструменти, що входять до складу CUDA Toolkit, та їх роль у розробці високопродуктивних GPGPU-додатків. Фази компіляції GPGPU CUDA-застосунку показані на рис.1.9.

Основним засобом трансляції коду є компілятор NVCC, який дозволяє розробникам писати програми на мові Сі з додатковими розширеннями для GPU. NVCC автоматично генерує код для обчислювального пристрою, виконуючи всі необхідні трансляції і оптимізації, що дозволяє ефективно використовувати паралелізм GPU.

Набір засобів розробки включає два основні API – високорівневий CUDA Runtime та низькорівневий Driver API. Хоча одночасне використання обох не підтримується, Runtime API забезпечує більш зручний інтерфейс для створення застосунків, у той час як Driver API дозволяє здійснювати більш детальний контроль над апаратними ресурсами GPU.

CUDA Toolkit містить численні оптимізовані бібліотеки, такі як CUBLAS

(для операцій лінійної алгебри), CUFFT (для швидкого перетворення Фур'є), CUSPARSE (для розріджених матриць) та інші. Ці бібліотеки надають готові рішення для розповсюджених обчислювальних задач, що дозволяє розробникам зосередитися на алгоритмічному аспекті проблеми, не витрачаючи додатковий час на реалізацію стандартних процедур.

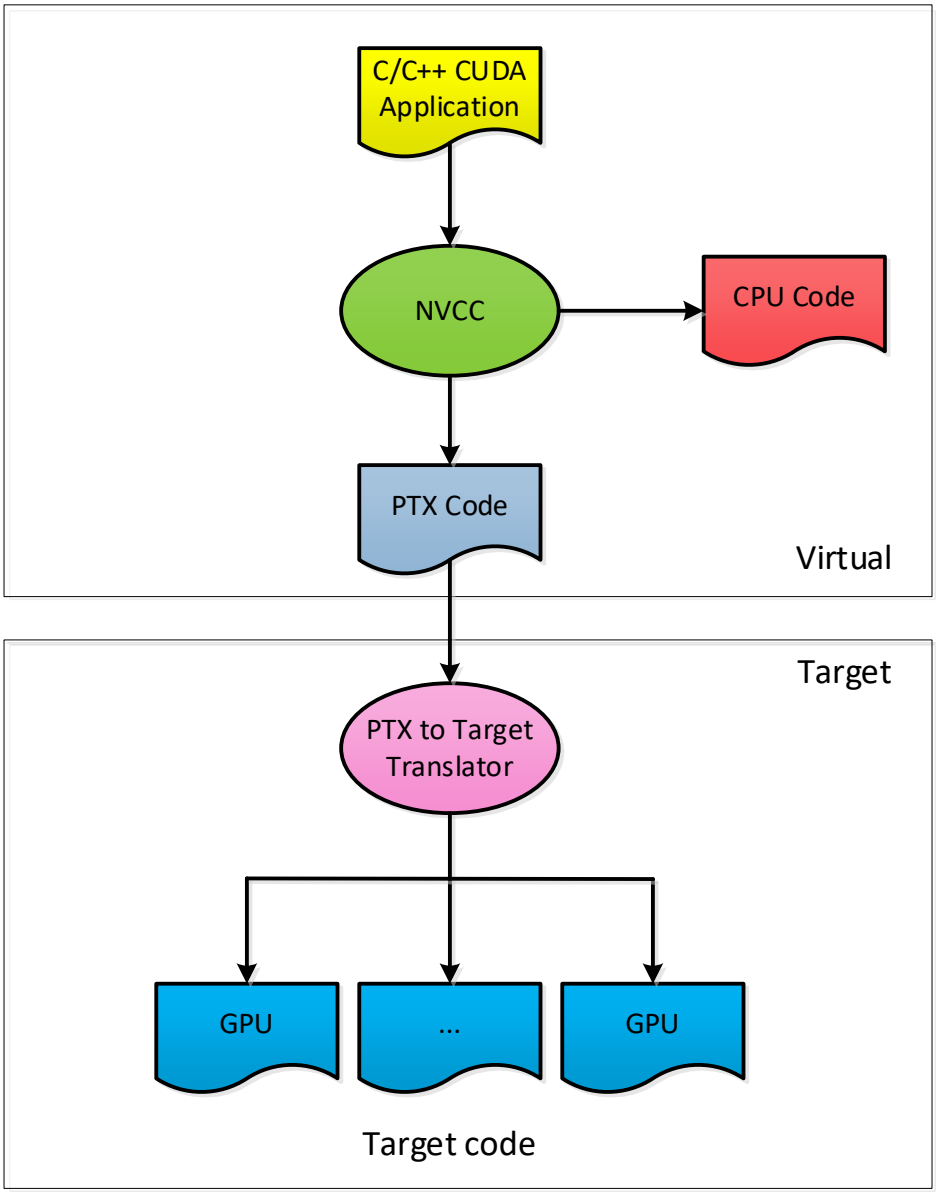


Рисунок 1.9. Етапи компіляції застосунків GPGPU CUDA

Для полегшення навчання та розробки NVIDIA надає великий набір прикладів (CUDA Samples), які демонструють типові алгоритми та підходи до розробки GPGPU-застосунків. Розгорнута документація, що входить до складу SDK, забезпечує необхідну інформацію щодо можливостей та обмежень

використання апаратних ресурсів.

CUDA-Memcheck є інструментом, що використовується для виявлення помилок пам’яті – від неправильного доступу до неініціалізованих регістрів до перевищення виділених меж. Подібні засоби налагодження є незамінними при розробці високопродуктивних паралельних алгоритмів, де будь-яка помилка може призвести до критичних збоїв.

Розробники можуть використовувати стандартні інтегровані середовища розробки (IDE) такі, як Visual Studio, Eclipse або CLion, які мають плагіни для підтримки CUDA. Інтеграція з IDE дозволяє зручно писати, налагоджувати та оптимізувати код в одній робочій системі. Крім того, автоматизовані засоби збірки та тестування забезпечують безперервну інтеграцію з сучасними workflow, що полегшує підтримку масштабних проектів.

Засоби розробки CUDA надають повний набір інструментів, що дозволяє створювати, налагоджувати та оптимізувати високопродуктивні застосунки, використовуючи паралельні можливості графічних процесорів. Цей комплекс включає компілятор NVCC, різноманітні API, готові бібліотеки, а також інструменти для налагодження та профілювання, що роблять розробку GPGPU-додатків з високою ефективністю і мінімальними витратами часу.

1.3.5 Оптимізація застосунків GPGPU CUDA

Для максимального використання можливостей GPU за допомогою CUDA необхідно ретельно планувати розміщення даних. Це означає — знайти оптимальне місце для зберігання інформації, вибираючи між регістрами, розділюваною (shared) пам’яттю та іншими типами пам’яті, а також мінімізувати накладні витрати при передачі даних між CPU та GPU. Використання буферизації даних також сприяє зниженню затримок і дозволяє організувати ефективний обмін інформацією між різними компонентами системи.

Ще одним важливим аспектом є баланс між розміром та чисельністю блоків. Збільшення числа потоків у блоці може зменшити ефект затримок при доступі до пам’яті, проте це одночасно зменшує кількість доступних регістрів для кожного потоку. Саме тому, замість блоків з 512 потоків, компанія NVIDIA рекомендує

оптимальні конфігурації з 128 або 256 потоків, що дозволяє знайти компроміс між затримками доступу до пам’яті та доступністю регістрів.

Серед ключових моментів оптимізації варто відзначити наступне:

- Інтенсивне використання shared memory. Розділювана пам’ять має набагато вищу швидкодію порівняно з глобальною відеопам’яттю, тому активне її застосування є критичним для прискорення обчислень;

- Об’єднання операцій читання та запису. Для досягнення максимальної пропускної здатності операції доступу до глобальної пам’яті слід організовувати широко (coalesced). Це досягається завдяки використанню спеціальних типів даних, що дозволяють одночасно читати або записувати 32, 64 або 128 біт однією операцією;

- Використання текстурних вибірок. Якщо об’єднання операцій читання у глобальній пам’яті виявляється складним, варто розглянути застосування текстурних вибірок, що іноді забезпечують додаткове пришвидшення завдяки апаратній оптимізації.

1.4 Реалізація SIMD-операцій з матрицями за допомогою GPGPU CUDA

У цьому підрозділі розглядається створення та впровадження програми, що реалізує операції з матрицями (множення та ділення) із використанням паралелізму GPGPU на базі технології CUDA. Програма побудована за моделлю, представленою на рис. 1.10, яка демонструє розподіл обчислювальних потоків у середовищі GPU. Основна мета – продемонструвати можливості паралельного обчислення великих обчислювальних завдань при застосуванні SIMD-підходу.

Для аналізу обчислювальної потужності згідно технічному завданню передбачено математичний вираз:

$$MX = e \cdot MC \cdot MZ \cdot MK + max(R) \cdot MT \qquad (1.1)$$

де кожна з матриць (або векторів) і скалярний коефіцієнт є частинами алгоритму, що включає множення матриць, додавання та обчислення максимальної величини. Таке завдання добре підходить для розбиття арифметичних операцій на тисячі потоків, що може виконуватись паралельно на

GPU.

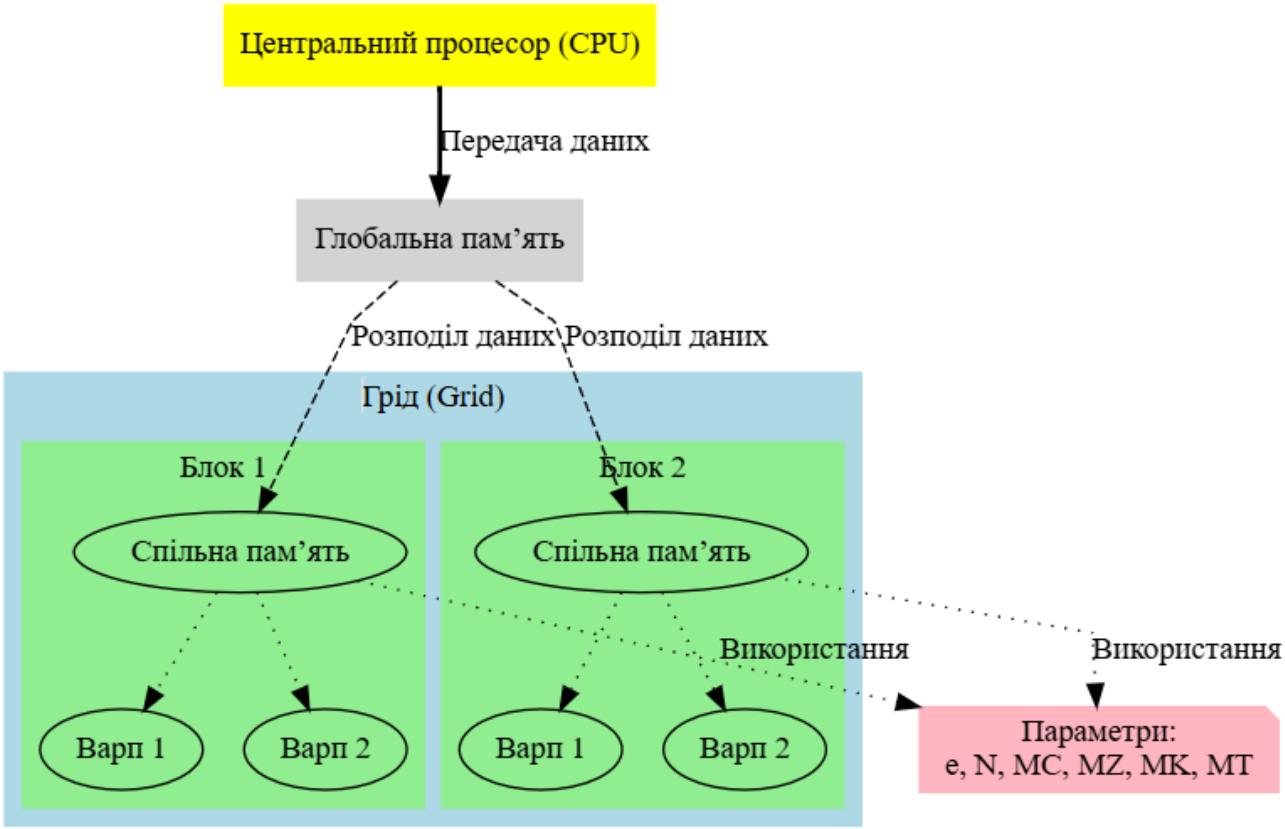


Рисунок 1.10. Схема розподілених обчислень для реалізації SIMD-операцій множення матриць за допомогою GPGPU CUDA

Програма написана мовою C із застосуванням CUDA-розширень. Особлива увага приділяється управлінню міжпотоковою взаємодією, зокрема використанню бар'єрів синхронізації та атомарних операцій для забезпечення коректного порядку обчислень. Ці засоби дозволяють координувати виконання паралельних обчислень таким чином, щоб уникнути проблем взаємного виключення та гонок при доступі до спільних даних.

Структура програми включає:

- Попередню обробку: завантаження вхідних даних (матриць та векторів) з файлу, що визначає розмірність і параметри обчислень;
- Запуск CUDA-ядра: розподіл обчислювальних завдань між потоками, де кожен потік виконує обчислення для частини елементів результатної матриці;
- Синхронізацію: використання бар'єрів для узгодження даних між потоками, а також атомарних операцій для обчислення глобально важливих значень

(наприклад, визначення max(R));

- Постобробку: збереження результатів обчислень у вихідний файл, що дозволяє провести подальший аналіз та побудову графіків для порівняльної оцінки ефективності роботи GPU і CPU.

Цей підхід дозволяє максимально використовувати можливості GPU, демонструючи справжній приріст продуктивності при застосуванні SIMD-операцій для множення матриць. У подальших пунктах будуть розглянуті етапи розробки, налаштування параметрів і оптимізації обчислювального процесу з використанням зазначеної моделі.

1.4.1 Створення алгоритму обчислень для операцій з матрицями

Паралельний математичний алгоритм для математичних операцій з матрицями, що реалізується з використанням технології CUDA, можна умовно поділити на три основних етапи.

1. Обчислення локальних максимумів: Вхідний вектор R ділиться на P рівних сегментів, кожен з яких містить H = N/P елементів (де N – розмірність векторів і матриць, а P – кількість потоків). Для кожного сегмента, який обробляється окремим потоком, визначається локальне максимальне значення:

$$m_i = \max(R_H), \quad i = 1,2,\dots,P \qquad (1.2)$$

Таким чином, кожен потік знаходить максимальне значення у своєму підмножині вектору *R*;

2. Формування глобального максимуму: З отриманих локальних значень m_i формується загальне максимальне значення m за допомогою порівняння значень, отриманих усіма потоками: $m = \max m_1, m_2, \dots, m_P$. Це значення є критично важливим для наступного етапу обчислень;
3. Обчислення результатної матриці: Остаточна обчислювальна операція полягає у поєднанні добутку певних матриць із скалярними коефіцієнтами. В даному алгоритмі результуюча матриця MX_H визначається за наступною формулою:

$$MX_H = e \cdot (MC_H \cdot MZ \cdot MK) + m \cdot MT_H, \qquad (1.3)$$

де:

- *e, m, MZ, MK* – це спільні, незмінні параметри чи ресурси, що використовуються при обчисленнях;
- MC_H, MT_H, MX_H – відповідно, підмножини рядків матриць *MC, MT* та *MX*;
- масштабування та агрегування обчислень проводиться паралельно за допомогою техніки SIMD, що дозволяє розпаралелити навіть такі інтенсивні математичні операції.

Алгоритм спирається на ефективний розподіл вхідних даних між потоками, де спочатку відбувається локальне визначення максимуму у частині вектору, потім — глобальна агрегація цих значень, а в завершальному етапі результат обчислюється як комбінація матричних операцій з використанням отриманого глобального максимуму. Цей підхід дозволяє реалізувати масштабоване розпаралелювання обчислень, забезпечуючи високу продуктивність за рахунок використання апаратних можливостей GPU у виконанні SIMD-операцій.

1.4.2 Синхронізація та керування взаємодією потоків

Оскільки розроблювана програма має високий рівень масштабованості та орієнтована на систему з $P \geq 2$ процесорами, для керування взаємодією між потоками було створено єдиний універсальний алгоритм, представлений на рис. 1.11. Враховуючи особливості роботи з CUDA – де стандартні методи бар'єрної синхронізації застосовуються виключно до потоків, що знаходяться в одному блоці – для забезпечення повноцінної взаємодії між потоками з різних блоків було впроваджено додаткові механізми синхронізації.

Алгоритм взаємодії потоків показаний на рис. 1.11. Основні кроки алгоритму можна узагальнити наступним чином:

1. Ініціалізація даних: Створюються і завантажуються вхідні параметри застосунку (*e, R, MC, MZ, MK, MT*);

2. Передача даних у глобальну пам'ять: Всі вхідні значення копіюються у глобальну пам'ять пристрою для подальшої обробки;

3. Сигналізація про завершення завантаження: Потік з ідентифікатором *tid = 1* надсилає сигнал про те, що дані введені, у той час як інші потоки очікують цей сигнал;

4. Обчислення локальних максимумів: Кожен потік обчислює максимальне значення свого сегменту вектора $R(mi = max(RH))$;

5. Глобальна агрегація максимумів: Значення mi порівнюються з глобальною змінною m для оновлення останньої ($m = max(m, mi)$);

6. Поширення результату: Потоки обмінюються сигналами про завершення етапу обчислення m таким чином, що кожен потік отримує оновлене значення;

7. Копіювання спільних ресурсів: Кожному потоку повторно присвоюються значення *e, m, MZ, MK* відповідно до свого порядку виконання;

8. Основне обчислення: Виконується операція обчислення MX_H за формулою

$$MX_H = e \times (MC_H \times MZ \times MK) + m \times MT_H, \qquad (1.4)$$

де результат розраховується паралельно для кожного блоку даних;

9. Завершальна синхронізація: Потоки, окрім першого, надсилають сигнал про закінчення обчислень, а потік з *tid = 1* чекає на їх отримання;

10. Передача результату: Останній, завершальний потік, передає фінальний результат у оперативну пам’ять, після чого здійснюється виведення матриці MX.

Використані методи синхронізації та обміну даними:

- atomicCopy(): Атомарна операція для безпечного копіювання спільних ресурсів з глобальної пам’яті;
- atomicMax(): Забезпечує послідовний доступ до змінної m при оновленні результату локальних максимумів;
- __syncthreads(): Функція внутрішньоблочної синхронізації, що гарантує коректний порядок виконання операцій;
- Інтегровані функції, що реалізують паралельні обчислення:

  main(): Ініціює виконання паралельних потоків;

  vectMax(): Проводить пошук максимального значення в окремому сегменті вектора;

matrSum(): Реалізує операцію складання матриць;

– scalOnMatr(): Виконує множення матриці на скаляр;
– matrOnMatr(): Забезпечує множення однієї матриці на іншу;

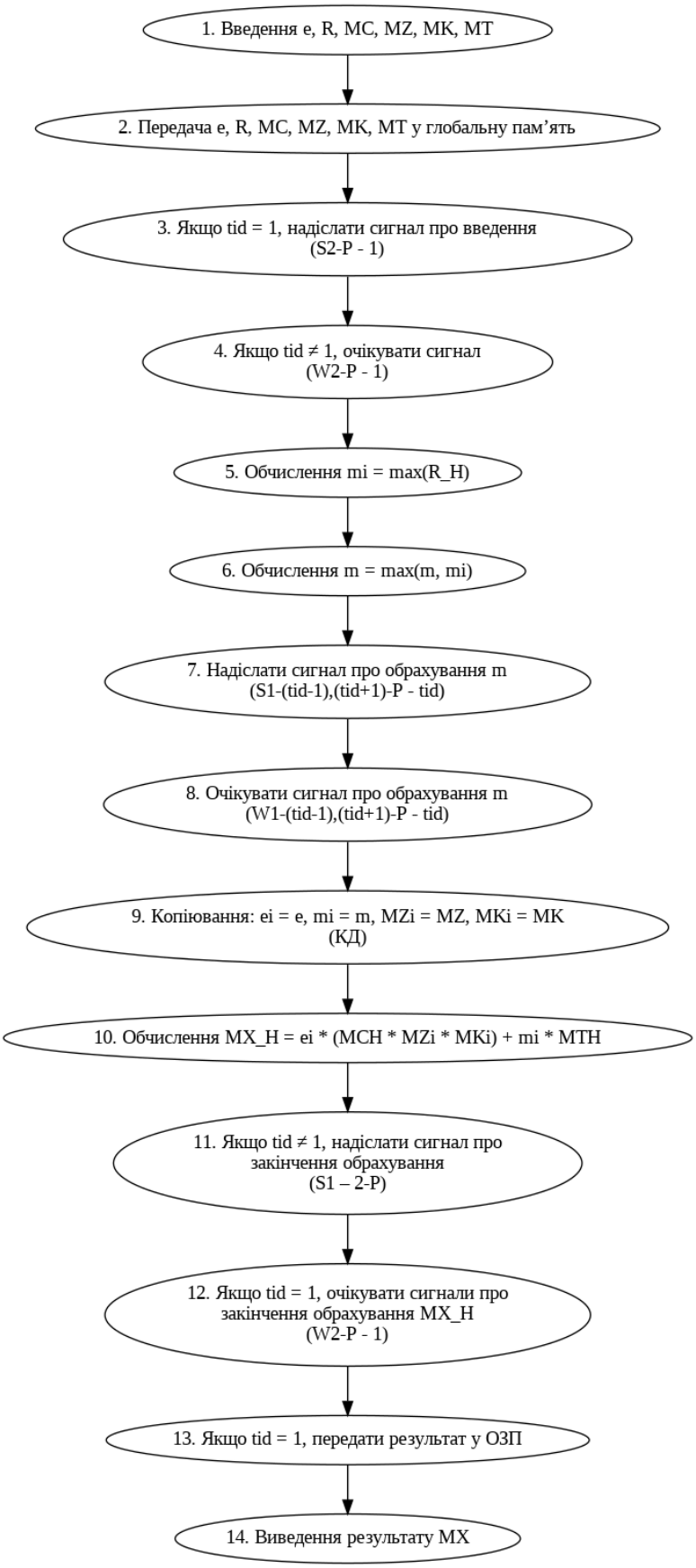


Рисунок 1.11. Алгоритм синхронізації та керування взаємодією потоків

Запропонований алгоритм дозволяє уніфікувати та масштабувати процес взаємодії потоків у паралельних обчисленнях, вирішуючи проблеми синхронізації як у межах одного блоку, так і між різними блоками, що є критично важливим для реалізації високопродуктивних SIMD-операцій у середовищі CUDA.

1.4.3 Реалізація застосунку для тестування продуктивності операцій з матрицями

Програмний застосунок, розроблений для вимірювання швидкодії паралельного алгоритму множення та ділення матриць із використанням технології CUDA, поєднує кілька ключових модулів, що забезпечують як завантаження даних, так і їх паралельну обробку, а також вимірювання часу виконання і розрахунок коефіцієнтів прискорення та ефективності.

Для тестування застосунку було обрано такі значення розмірності векторів і матриць:

- N = 1024;
- N = 2048;
- N = 4096.

Програма запускалася із різною кількістю паралельних потоків (або ядр): 32, 128 та 256. Для порівняння ефективності реалізовано аналогічну послідовну програму, яка працює на однопроцесорній системі.

Опис використовуваного програмного середовища

- Операційна система: Microsoft Windows 11;
- Середовище розробки: Microsoft Visual Studio 2022;
- Комплект розробки: CUDA Toolkit.

Програмний застосунок включає такі ключові методи та модулі:

1. Ініціалізація даних (initData()): Цей метод відповідає за завантаження вхідних даних – матриць і векторів. Дані зчитуються з файлів або генеруються програмно, після чого копіюються у глобальну пам’ять GPU. Метод забезпечує правильне формування матриць:

- MC, MZ, MK, MT – основні матриці;

- R – вектор, з якого буде обчислюватися максимальне значення для подальшого масштабування;
- e – скалярний коефіцієнт, що використовується при множенні;

2. Реалізація операцій множення матриць: Ключову роль відіграє функція CUDA-ядра, де кожен потік обробляє певну частину матриці. Основні методи тут включають:

- matrOnMatr() – функція для множення матриць;
- scalOnMatr() – множення матриці на скаляр;
- matrSum() – функція для додавання матриць.

Ці методи комбінуються всередині ядра для обчислення виразу:

$$MX_H = e \times (MC_H \times MZ \times MK) + m \times MT_H, \qquad (1.5)$$

де m – глобальний максимум, отриманий із вектору R;

3. Обчислення максимального значення (vectMax()): Для отримання локальних максимумів із сегментів вектора R використовується функція vectMax(), яка працює паралельно на кожному потоці. Локальні максимуми агрегіруються функціями атомарного порівняння, зокрема з використанням операції atomicMax();

4. Синхронізація потоків: Для забезпечення коректного порядку виконання використовується внутрішньоблочна синхронізація з допомогою функції __syncthreads(), а також додаткові механізми синхронізації за допомогою атомарних операцій (наприклад, atomicCopy()) для координування обчислень між потоками, що розташовані в різних блоках;

5. Вимірювання часу виконання (measureTime()): Засіб вимірювання часу реалізовано за допомогою стандартної функції мови C clock(). Метод фіксує початковий та кінцевий моменти виконання основного алгоритму, що дозволяє розрахувати тривалість роботи обчислень як для паралельного, так і для послідовного виконання;

6. Розрахунок коефіцієнтів: Після отримання часу виконання обчислюється коефіцієнт прискорення K_n за співвідношенням:

$$K_n = \frac{T_1}{T_P}, \quad (1.6)$$

де T_1 – час виконання послідовного алгоритму, а T_P – час виконання паралельного алгоритму на *P* процесорах. Також розраховується коефіцієнт ефективності K_e:

$$K_e = \frac{K_n}{P} \times 100\% \quad (1.7)$$

Код застосунку організовано у вигляді модульної системи. Головна функція main() викликає послідовно:

- initData() для ініціалізації і завантаження всіх необхідних даних у визначені області пам'яті;
- Виклик CUDA-ядра, яке розподіляє обчислювальні завдання між потоками за допомогою конфігурації блоків по 128 або 256 потоків;
- vectMax() для визначення локальних максимумів і подальшої агрегації глобального максимального значення;
- Функції matrOnMatr(), scalOnMatr() та matrSum() всередині CUDA-ядра для обчислення результату за заданою формулою;
- measureTime(), який визначає тривалість виконання обчислень;
- Розрахунок коефіцієнтів K_n та K_e, результати яких виводяться на екран або записуються у файл.

Розроблений програмний застосунок дозволяє комплексно оцінити продуктивність паралельного алгоритму операцій з матрицями за допомогою CUDA. Завдяки використанню розмірностей N = 1024, 2048, 4096 та різним конфігураціям потоків, передбачається масштабованість рішення високопродуктивних обчислень на GPU. Код застосунку, наданий у додатку А, містить основні методи ініціалізації даних, обчислення максимумів, операцій з матрицями, синхронізації потоків та вимірювання часу виконання.

1.5 Тестування продуктивності обчислення матриці MX

У цьому підрозділі проведено комплексне тестування застосунку, розробленого для тестування продуктивності виконання операцій з матрицями за

допомогою CUDA, з метою оцінки продуктивності паралельного алгоритму обчислення матриці $MX_H = e \cdot (MC_H \cdot MZ \cdot MK) + m \cdot MT_H$. Для дослідження використовувалася система із сучасними апаратними засобами, що відрізняються високою обчислювальною потужністю та ефективною підтримкою паралельних обчислень.

Апаратні характеристики тестової системи:

- Процесор: Intel® Core™ i7-12700H з базовою частотою 2.4 ГГц та режимом Turbo Boost до 4.9 ГГц;
- Графічний процесор: Nvidia® GeForce® RTX 3060 (6 ГБ GDDR6, базова частота ~1300 МГц, 192-бітна шина даних);
- Оперативна пам’ять: 16 ГБ DDR4.

Для аналізу продуктивності застосунку тестування проводилося з використанням трьох варіантів розмірності векторів і квадратних матриць:

- N = 1024;
- N = 2048;
- N = 4096.

Обчислення виконувалися як послідовно (на CPU), так і паралельно (на GPU) із налаштуванням кількості паралельних потоків (ядр) у трьох режимах: 32, 128 та 256.

Нижче наведено результати вимірювання часу виконання обчислювального алгоритму для заданих розмірностей (табл. 1.2), розрахунки коефіцієнтів прискорення K_n (табл. 1.3) та коефіцієнтів ефективності K_e (табл. 1.4).

Таблиця 1.2. Час виконання програми (в мілісекундах)

N	**CPU T_1 (ms)**	**GPU T_2 (ms, P=32)**	**GPU T_3 (ms, P=128)**	**GPU T_4 (ms, P=256)**
1024	4100	1950	1350	1020
2048	65000	33000	22000	17000
4096	180000	80000	55000	42000

Таблиця 1.3. Значення коефіцієнта прискорення K_n

N	CPU	GPU (P=32)	GPU (P=128)	GPU (P=256)
1024	1	2.10	3.04	4.02
2048	1	1.97	2.95	3.82
4096	1	2.25	3.27	4.29

Таблиця 1.4. Значення коефіцієнта ефективності K_e

N	CPU	GPU (P=32)	GPU (P=128)	GPU (P=256)
1024	100	6.56%	2.38%	1.57%
2048	100	6.16%	2.30%	1.49%
4096	100	7.03%	2.55%	1.68%

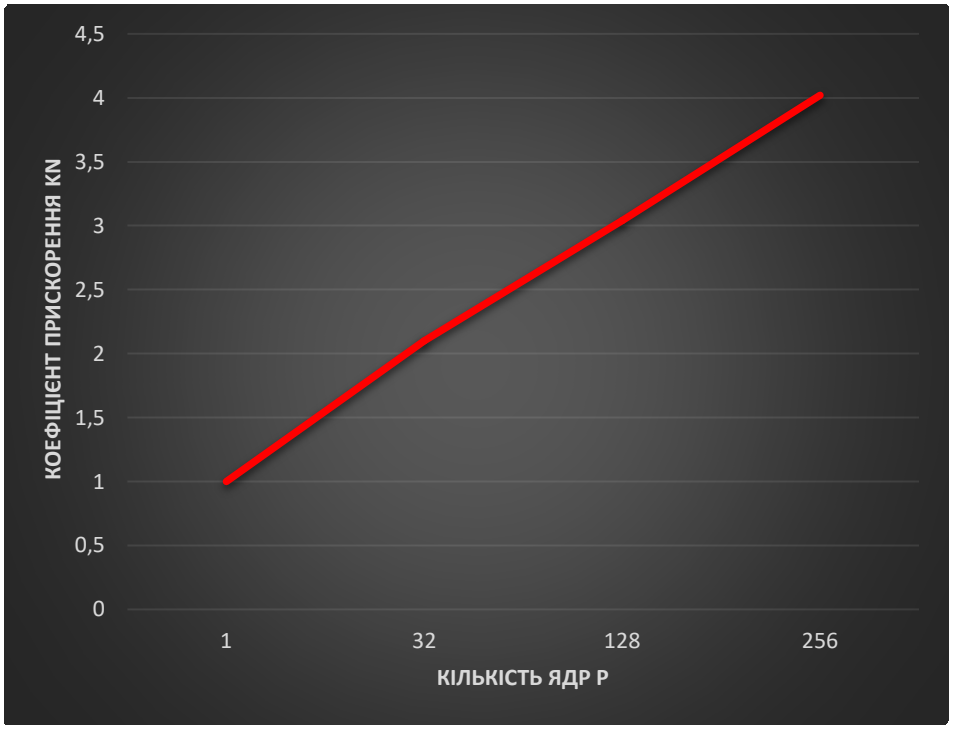


Рисунок 1.12. Коефіцієнт прискорення в залежності від кількості ядр (N=1024)

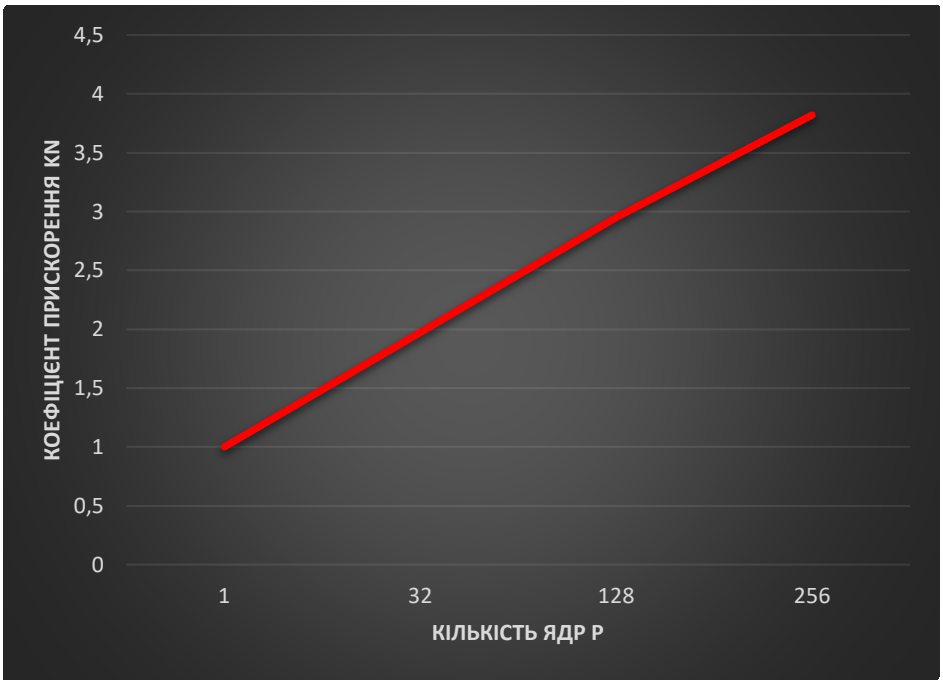


Рисунок 1.13. Коефіцієнт прискорення в залежності від кількості ядр (N=2048)

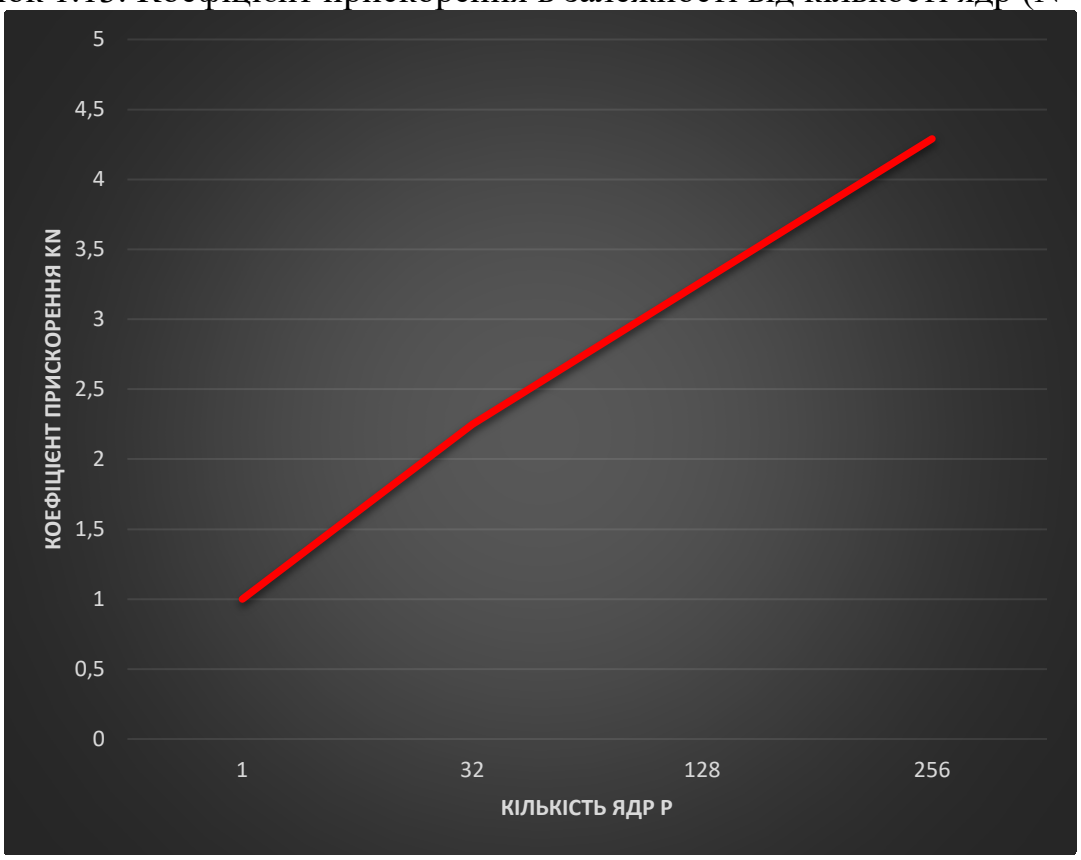


Рисунок 1.14. Коефіцієнт прискорення в залежності від кількості ядр (N=4096)

Аналіз отриманих результатів:

- Прискорення обчислень: Тестування показало, що використання технології CUDA дає суттєве скорочення часу виконання завдання порівняно із

послідовним виконанням на CPU. Значення коефіцієнта прискорення K_n варіюються від приблизно 2.10 до 4.29 залежно від розмірності задачі та кількості потоків;

- Ефективність масштабування: Значення коефіцієнта ефективності K_e демонструють, що при збільшенні кількості потоків спостерігається зниження ефективності використання кожного додаткового процесора. Наприклад, для (N = 1024) ефективність падає від 6.56 % при 32 потоках до 1.57% при 256 потоках. Це пов'язано з накладними витратами на синхронізацію та обмеженнями доступу до спільних ресурсів;
- Графічна візуалізація: На основі даних з таблиць 1.3 та 1.4 було побудовано графіки залежності коефіцієнта прискорення та ефективності від кількості потоків для кожної обраної розмірності (N). Графіки (рис. 1.12–1.17) демонструють нелінійність приросту швидкодії із збільшенням (P) і висвітлюють компроміс між збільшенням кількості потоків і зниженням ефективності через додаткові накладні витрати.

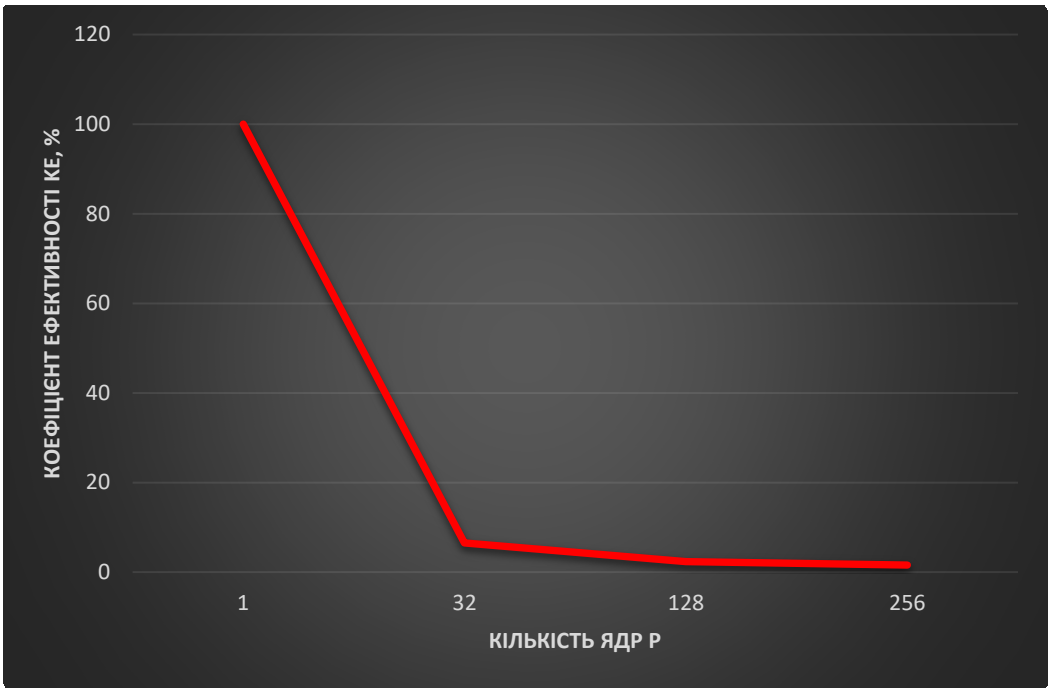


Рисунок 1.15. Коефіцієнт ефективності в залежності від кількості ядр (N=1024)

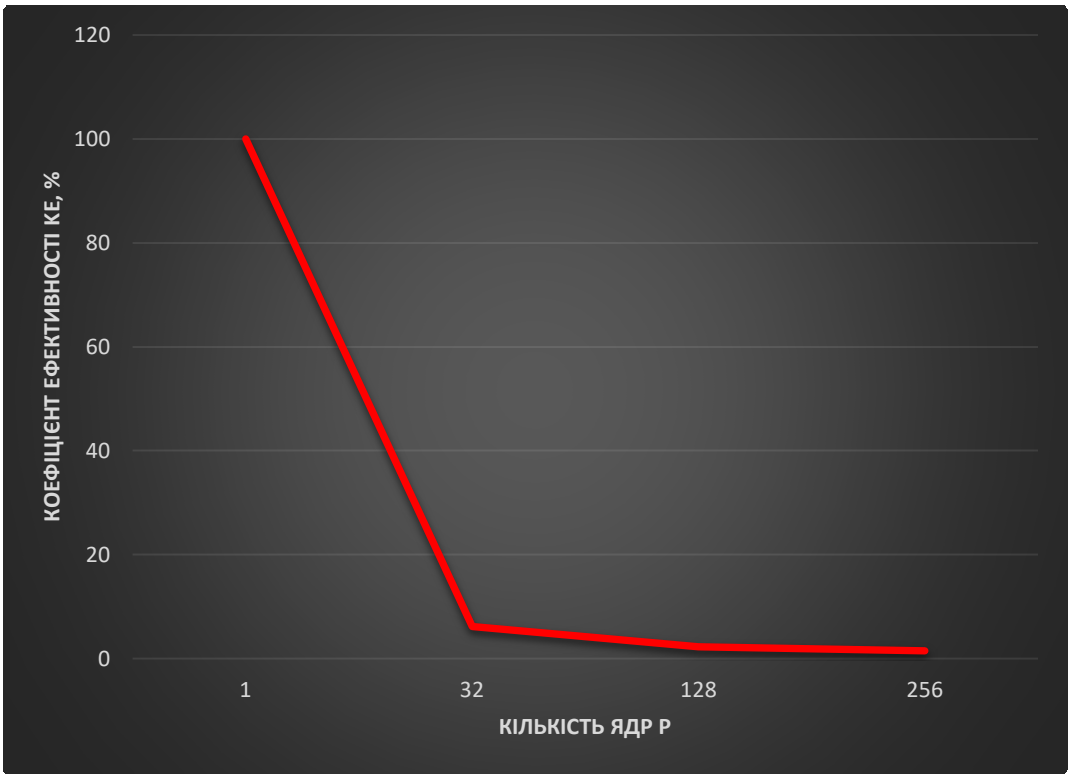


Рисунок 1.16. Коефіцієнт ефективності в залежності від кількості ядр (N=2048)

Рисунок 1.17. Коефіцієнт ефективності в залежності від кількості ядр (N=4096)

Використання технології CUDA дозволяє значно скоротити час виконання матричних обчислень, що є критично важливим для інтенсивних чисельних задач. Проте, збільшення числа потоків веде до нелінійного зростання продуктивності, оскільки частина часу витрачається на синхронізацію та управління спільним доступом. Найбільше прискорення спостерігається при задачах високої розмірності N = 4096, де максимальний $K_n \approx 4.29$ для 256 потоків, тоді як

ефективність використання ресурсів залишається на рівні 1–2.55%.

Проведене тестування демонструє, що, незважаючи на певні накладні витрати, використання паралельного алгоритму на базі CUDA є ефективним для масштабування обчислювальних задач множення матриць. Результати підтверджують значну перевагу GPU над послідовним виконанням на CPU, а також дозволяють оцінити компроміс між швидкодією та ефективністю використання доступних ресурсів.

1.6 Тестування продуктивності обчислення скалярного добутку векторів, множення матриць та ділення матриці на число

Цей підрозділ має на меті порівняти ефективність роботи центрального процесора та графічних процесорів у наступних задачах:

1. Обчислення скалярного добутку векторів;
2. Множення матриць;
3. Ділення матриці на число.

Як було встановлено в попередніх дослідженнях, ключовими параметрами, що впливають на продуктивність, є кількість потоків та розмір вхідних даних. Тому в цьому дослідженні тестування проводилося для різних розмірностей:

- Вектори: від 256 до 81,920,000 елементів;
- Квадратні матриці: розмірність від 402 до 90,642.

Характеристики пристроїв, використаних для тестів, є такими:

- CPU: Процесор: Intel® Core™ i7-12700H з базовою частотою 2.4 ГГц та режимом Turbo Boost до 4.9 ГГц;
- GPU: NVIDIA GeForce 845M, NVIDIA Quadro P6000 та NVIDIA GeForce RTX 3060.

Для кожної з задач проводилися серії тестів із заміром часу виконання, причому результати було усереднено за 10 запусків. Це дозволило побудувати графіки, що демонструють ефективність виконання, зокрема, «Ефективність виконання множення матриць», «Ефективність виконання скалярного добутку векторів».

Методологія тестування була такою:

1. Скалярний добуток векторів. Обчислення виконується за формулою $a \cdot b = a_1 \cdot b_1 + a_2 \cdot b_2 + \cdots + a_n \cdot b_n$. Оскільки кінцевим результатом є єдине число, для синхронізації використовується атомарне додавання (через шаблонний клас *std::atomic<float>* з методом *fetch_add*), що забезпечує безпечне накопичення результатів з усіх потоків;

2. Множення матриць. Для спрощення реалізації матриці на CPU та GPU представлені як одновимірні масиви із коректною індексацією за допомогою зсуву, що дозволяє зменшити накладні витрати на управління пам'яттю. У CPU обчислення здійснюється через послідовний обхід рядків і стовпців, а на GPU – завдяки використанню параметрів *blockIdx* та *threadIdx* для емулювання двовимірної сітки, що дозволяє виконувати операції паралельно для кожного елемента;

3. Ділення матриці на число. Операція виконується незалежно для кожного елемента матриці, що забезпечує повний паралелізм і мінімальні накладні витрати, адже додаткові засоби синхронізації не використовуються.

На рис. 1.18 подано результати виконання операції скалярного добутку векторів на CPU для різної кількості ядр процесора (від 1 до 16). Час вимірюється в мілісекундах, а розмірність відображає повний обсяг досліджуваного вектора.

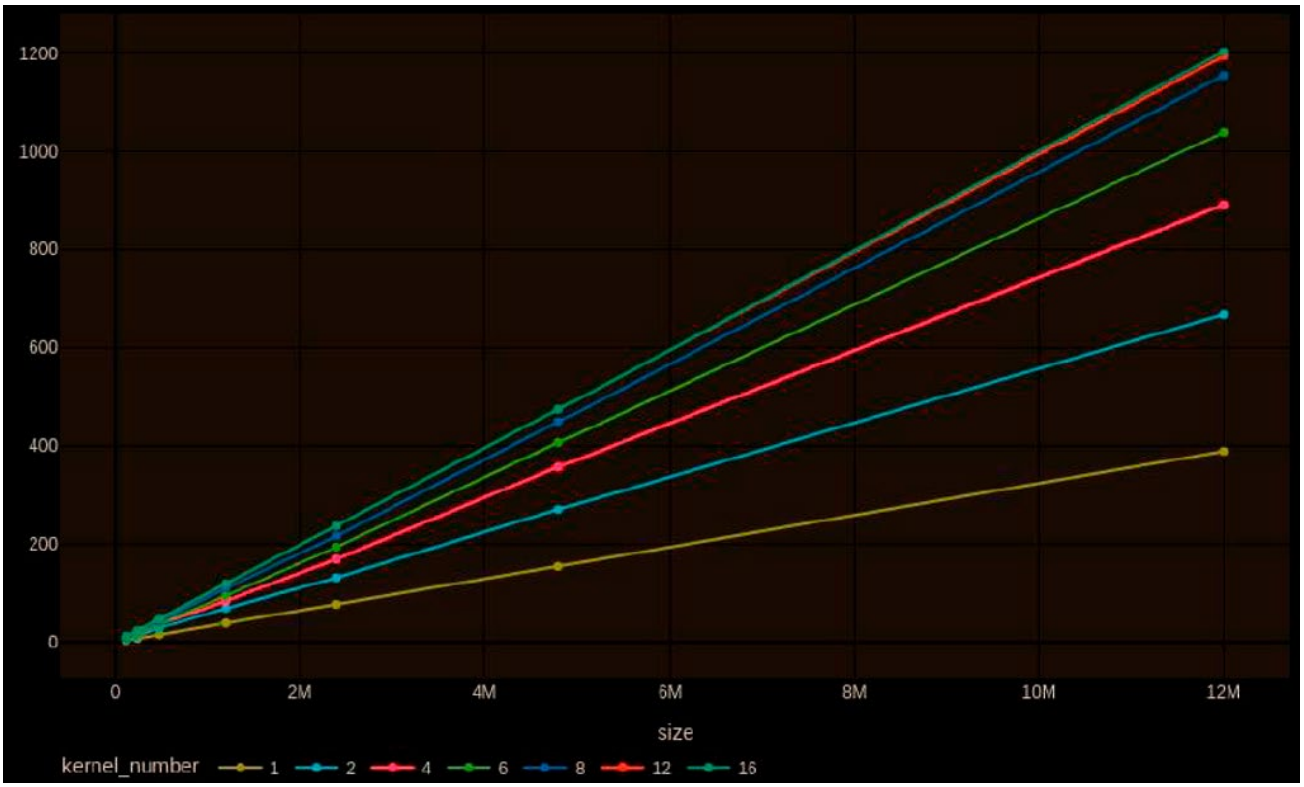


Рисунок 1.18. Продуктивність виконання скалярного множення векторів на CPU

З отриманих результатів видно, що найкращий час виконання забезпечується при використанні лише одного потоку. Це свідчить про те, що навіть застосування атомарних операцій як засобу синхронізації не завжди сприяє прискоренню — у цьому випадку воно, навпаки, сповільнює виконання. Таке явище пояснюється тим, що після обробки кожної пари елементів векторів здійснюється оновлення спільного кінцевого результату, що вимагає виконання атомарної операції. Часте викликання цієї операції призводить до виникнення значних затримок. Для вирішення цієї проблеми функцію, що виконується у потоках, було переписано таким чином, що кожен потік спершу використовує власну проміжну змінну для акумуляції результатів додавання (оскільки обхід елементів у межах функції відбувається послідовно), а в кінці роботи потоку отримане значення додається до загальної спільної змінної. Нижче наведено адаптований опис роботи функції, яка реалізує обчислення часткового скалярного добутку двох векторів:

```
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);
}
```

Функція *vec_sum* приймає два вказівники на об'єкти типу *std::vector<float>* (тобто, вектори), з яких будуть братися значення; посилання на атомарну змінну типу *std::atomic<float>*, яка використовується для збереження підсумкового результату; два цілі параметри *start* і *end*, що визначають діапазон індексів, для якого необхідно обчислити скалярний добуток. Функція оголошує локальну змінну *interm*, яка буде використовуватися для акумуляції добутків відповідних елементів векторів. В циклі від індексу *start* до *end - 1* обчислюється добуток значень, взятих із першого вектора *(vec1->at(i))* і другого вектора *(vec2->at(i))*, після чого отриманий результат додається до локального акумулятора *interm*. Після завершення циклу, значення *interm* додається до спільної атомарної змінної *res* за допомогою методу *fetch_add* з режимом пам’яті *std::memory_order_relaxed*. Це дозволяє безпечно акумулювати часткові результати, обчислені в окремих потоках, без ризику виникнення гонок за умов конкурентного доступу. Таким чином, дана функція розбиває процес обчислення скалярного добутку векторів на менші частини, які можуть обчислюватися паралельно. Кожен потік обчислює свій локальний підсумок (збережений у змінній *interm*), а потім цей результат додається до загальної атомарної змінної *res*. Це дозволяє зменшити накладні витрати, пов’язані з частим оновленням спільного ресурсу, передаючи обчислення локальним потокам, які лише наприкінці синхронізуються атомарним додаванням.

Маючи оптимізований варіант алгоритму багатопотокового обчислення скалярного добутку векторів, було повторно здійснено вимірювання із використанням різних конфігурацій потоків. Отримані результати представлені на рис. 1.19.

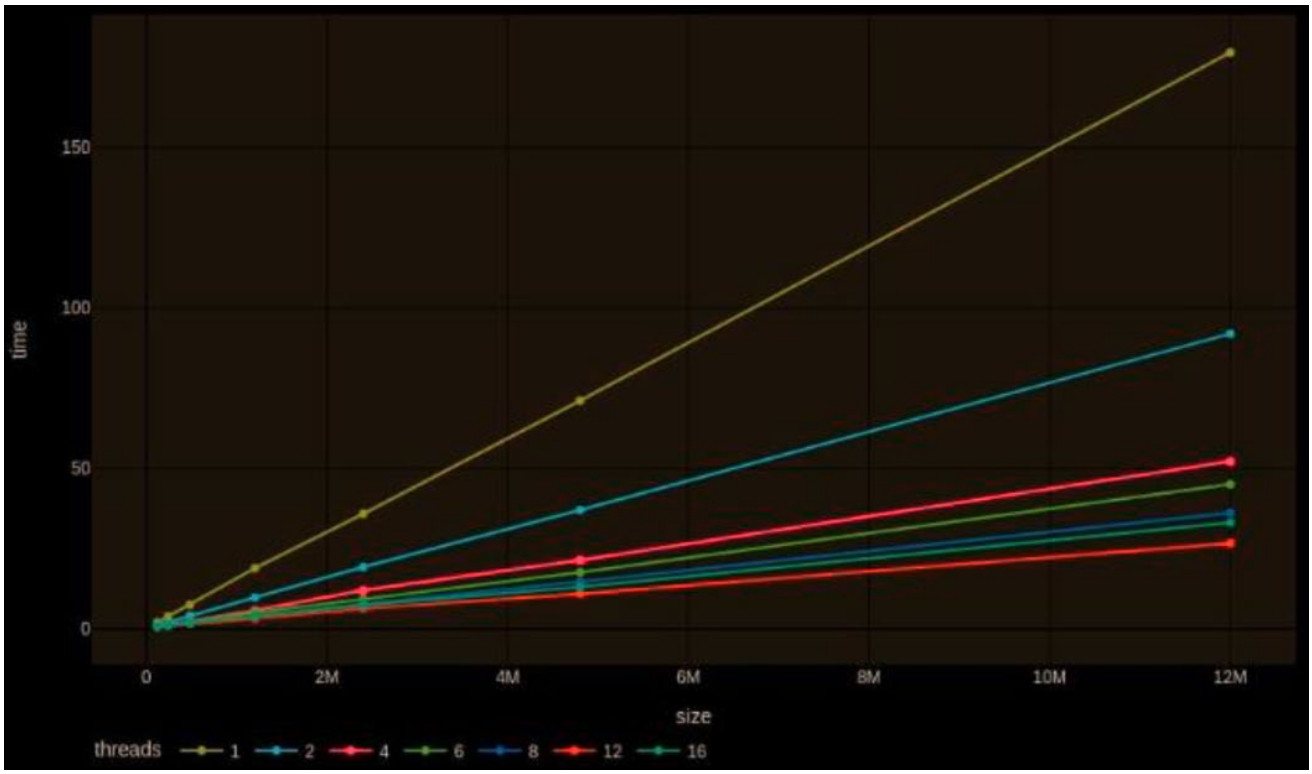


Рисунок 1.19. Продуктивність виконання скалярного множення векторів на CPU за оптимізованим алгоритмом багатопотокового обчислення

Для порівняння продуктивності CPU та GPU, тести на CPU виконуватимуться у 12 потоків. Цю конфігурацію обрано з огляду на апаратні можливості процесора, що дозволяє досягнути оптимального балансу між використанням доступних ядр і мінімізацією накладних витрат на синхронізацію. У даному підході обчислювальне навантаження розподіляється між 12 незалежними потоками, кожен з яких обробляє свою частину вхідних даних. Такий розподіл дозволяє максимально використовувати можливості багатоядерної архітектури CPU для точного вимірювання часу виконання задач, що в свою чергу створює основу для порівняльного аналізу з продуктивністю GPU. Тести обчислення результатів ділення матриць та скалярного добутку (рис.1.20, 1.21) проводились за середнім значенням, отриманим з 10 окремих запусків.

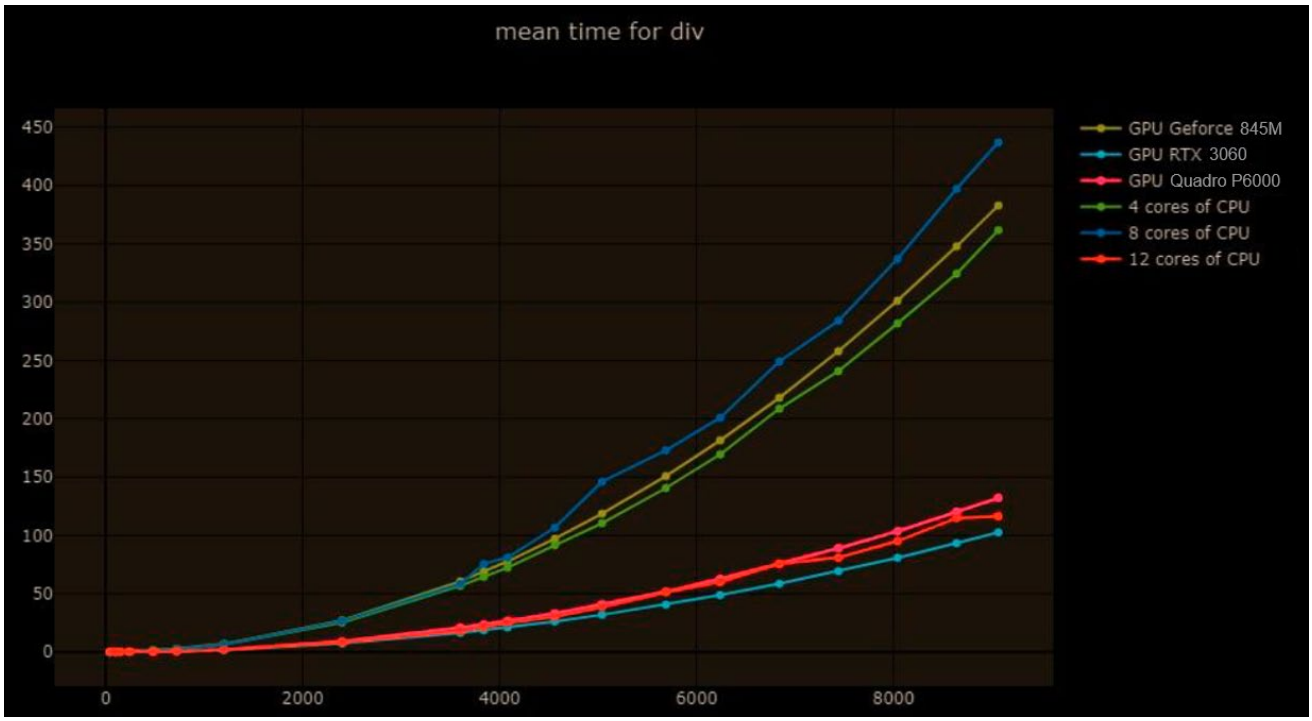


Рисунок 1.20. Продуктивність виконання ділення матриць на CPU та GPU

З рис.1.20, 1.21 можна помітити, що найкращі результати показує RTX3060, а CPU працює краще за NVIDIA GeForce 845M і NVIDIA Quadro P6000.

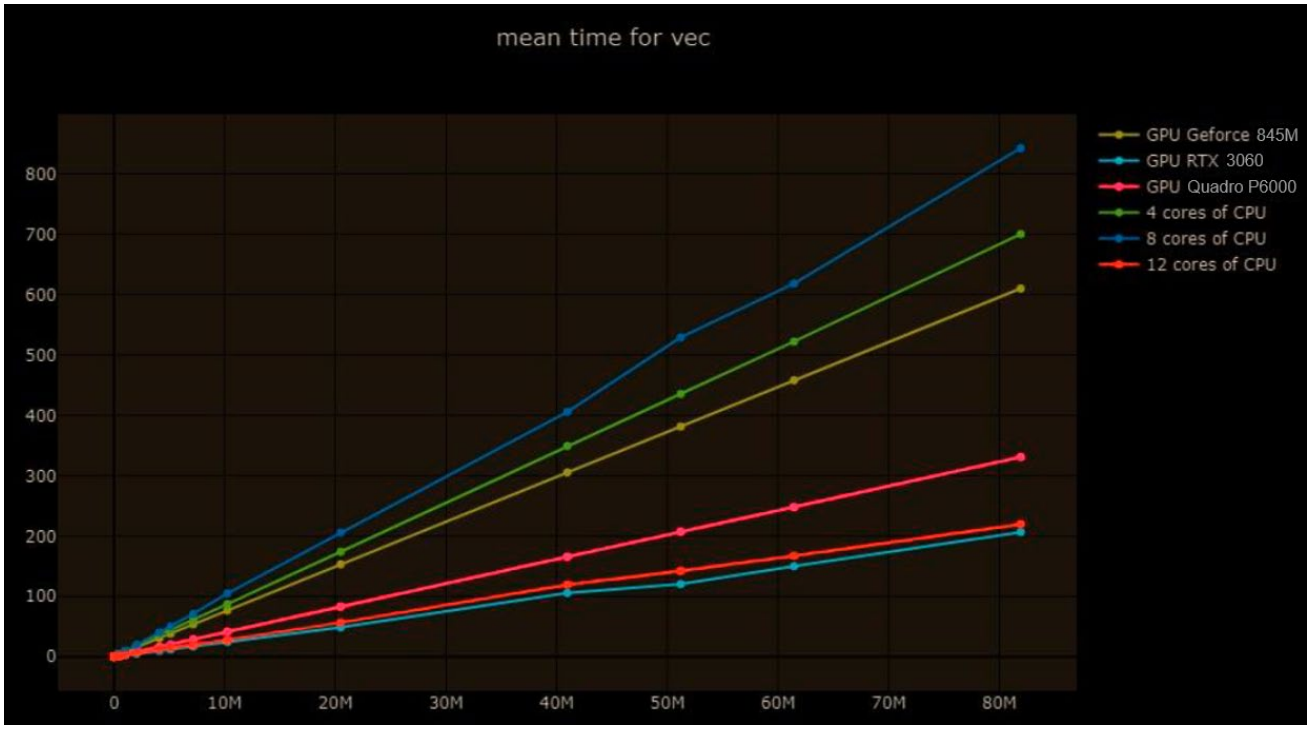


Рисунок 1.21. Продуктивність виконання скалярного множення векторів на CPU та GPU

Треба відзначити, що у CUDA-потоках також використовується атомарне додавання, аналогічне до того, як було реалізовано в початковій імплементації на

CPU. Проте, навіть при повторному застосуванні атомарних операцій, вплив на загальну продуктивність є мінімальним, і не спостерігається значного сповільнення виконання.

На рис. 1.22 зображено графік продуктивності при множенні двох матриць. За даними графіка видно, що центральні процесори демонструють значно гірші результати, тоді як графічні процесори працюють набагато ефективніше. Це пояснюється різницею в обчислювальній складності: у реалізації на C++ із використанням класичних потоків алгоритм має складність $O(n^3)$ через необхідність вкладених циклів – спершу по рядках, всередині яких перебираються усі стовпці, а потім ще один цикл по елементах кожного рядка. Натомість у CUDA застосовується 2D-індексація, яка дозволяє уникнути перших двох вкладених циклів, і таким чином зводить складність до O(n). Завдяки цьому стандартне множення матриць завжди ефективніше реалізовується на CUDA, що відзначається на поданому графіку.

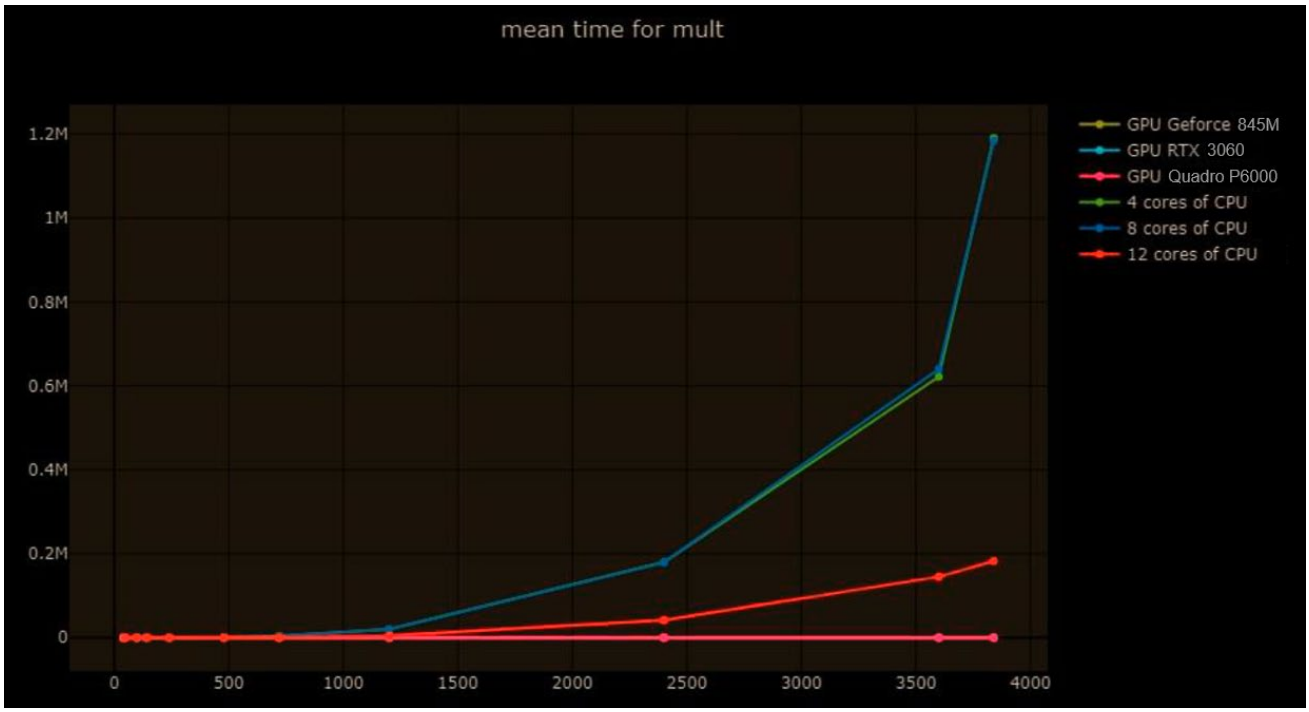


Рисунок 1.22. Продуктивність виконання множення двох матриць на CPU та GPU

На основі аналізу інших графіків видно, що CPU не завжди поступається GPU. Зокрема, у трьох з п’яти тестів GeForce 845M показала гірші результати, ніж центральний процесор. Це ще раз підтверджує, що швидкість передачі даних між

компонентами гетерогенної системи є вирішальним чинником при оптимізації обчислень.

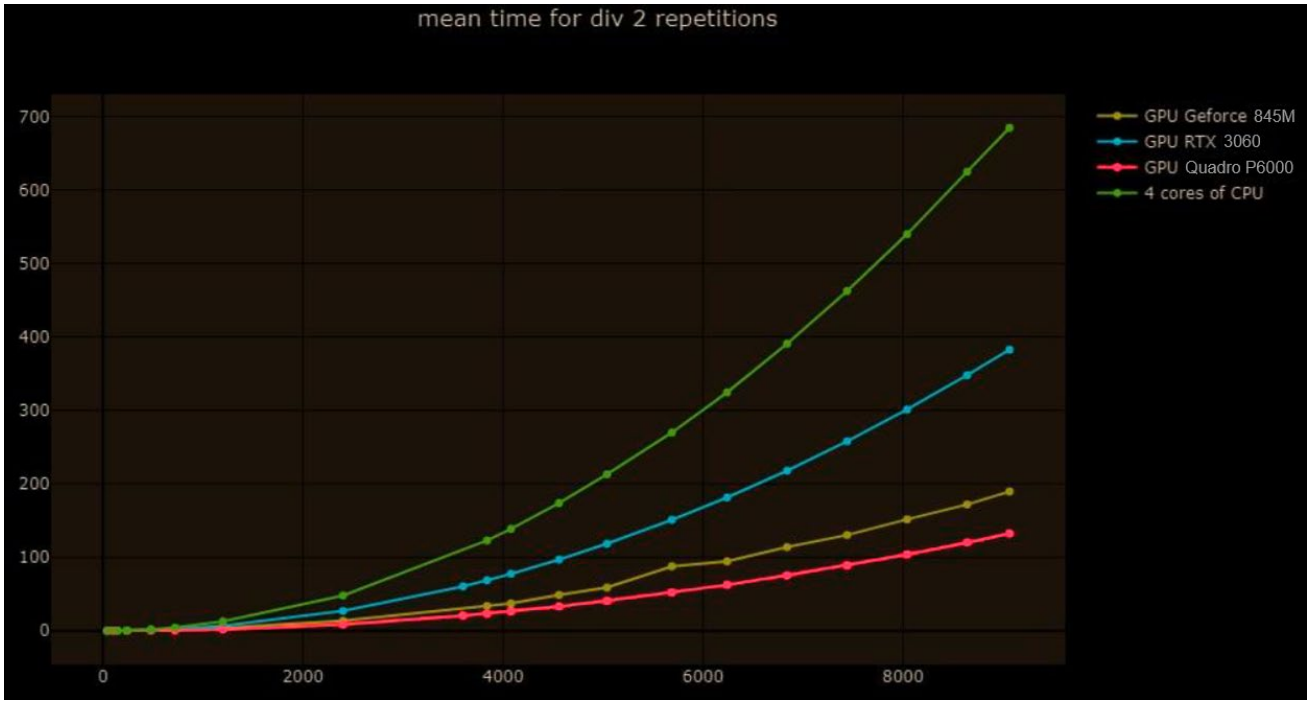


Рисунок 1.23. Продуктивність виконання ділення матриць на CPU та GPU із повторенням операції два рази всередині потоку

При переході на використання GPU замість CPU важливо, щоб індивідуальні задачі, що виконуються всередині кожного потоку, мали достатню обчислювальну складність для компенсації накладних витрат на трансфер даних.

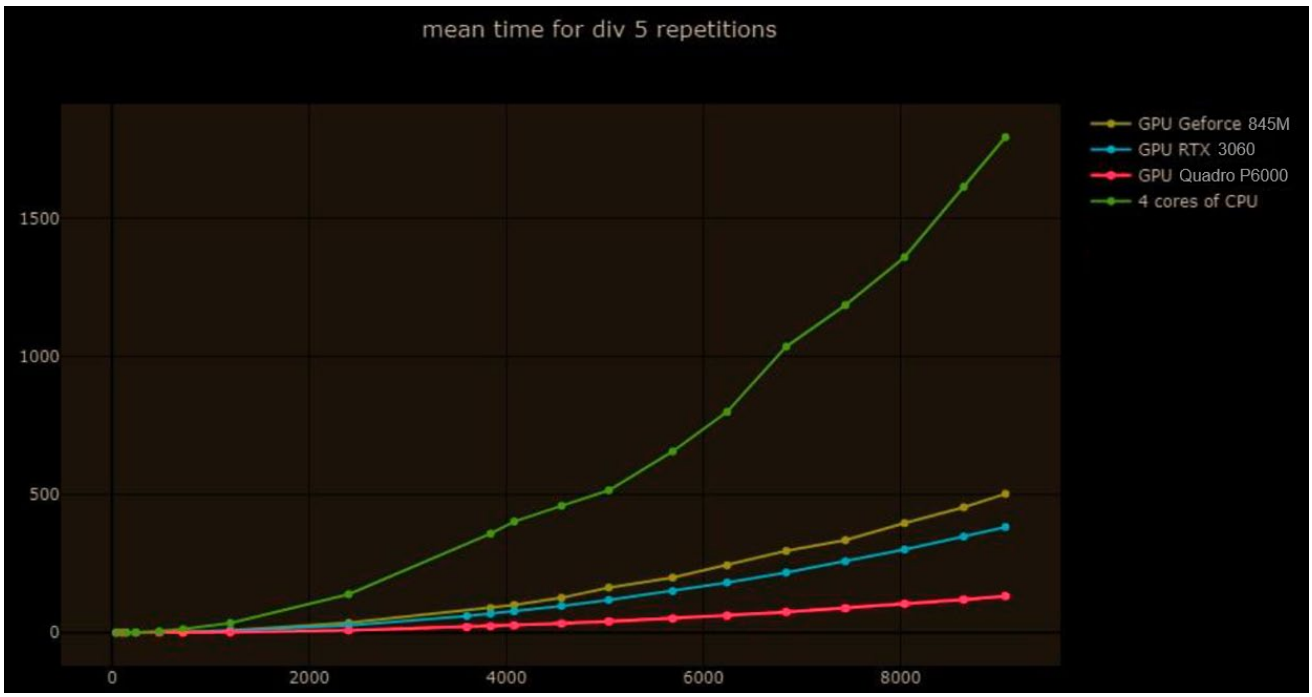


Рисунок 1.24. Продуктивність виконання ділення матриць на CPU та GPU із повторенням операції п’ять разів всередині потоку

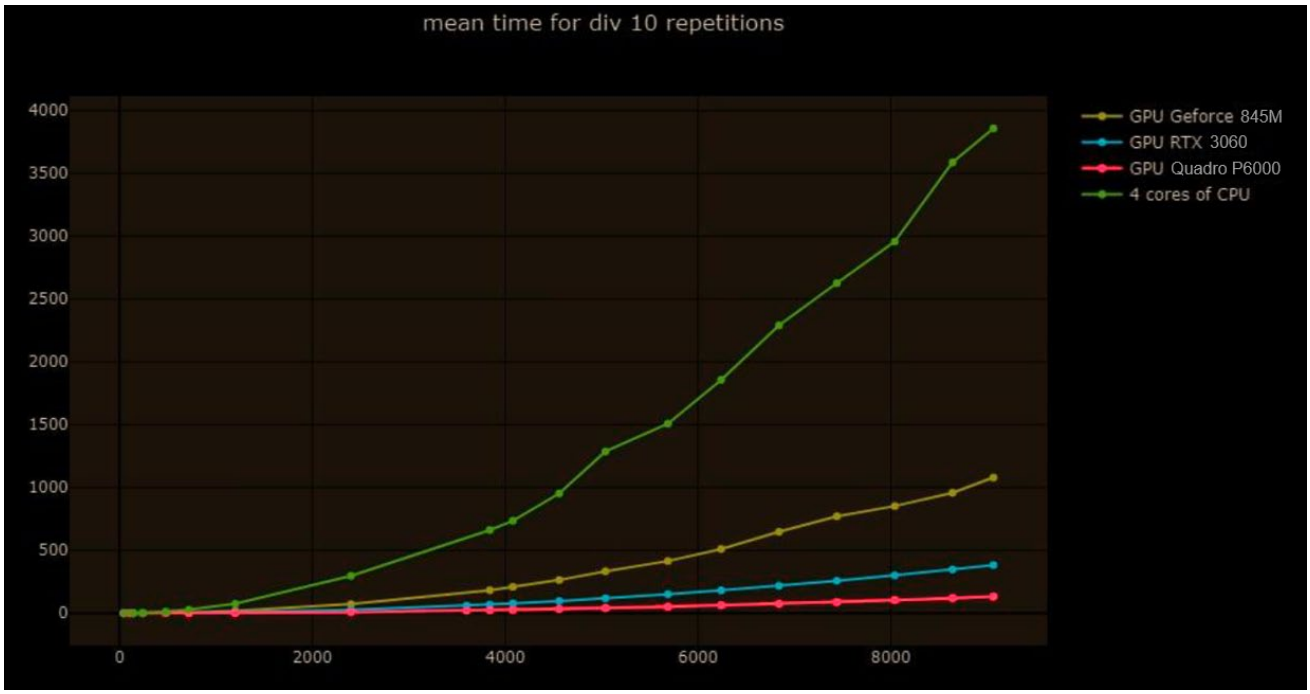


Рисунок 1.25. Продуктивність виконання ділення матриць на CPU та GPU із повторенням операції десять разів всередині потоку

Щоб проілюструвати це, тести для завдань, у яких GPU GeForce 845M демонстрував найнижчу продуктивність, продовжено. Замість виконання однієї операції в окремому потоці, її було повторено два, п’ять та десять разів. Оскільки попередні виміри показали незначну різницю у часі виконання для двох варіантів додавання матриць, природньо, що графіки для тестів (рис. 1.23–1.25) є схожими між собою. Можна помітити з рис. 1.23, що GPU GeForce 845M працює швидше за CPU. Відповідно графіку на рис.1.24, ситуація по відношенню до випадку з двома потоками змінилась, GPU працює швидше ніж CPU.

Після аналізу отриманих результатів (рис.1.18–1.25) можна зробити наступні висновки з тестування:

- При збільшенні кількості потоків і розміру вхідних даних час виконання задач зменшується, проте оптимальне співвідношення досягається лише для задач з високою обчислювальною складністю;
- Для обчислення скалярного добутку векторів використання атомарних операцій може створювати додаткові затримки, що знижує продуктивність при надто частому оновленні спільного результату, тому для цих задач важлива оптимізація внутрішнього акумулювання результатів усередині

кожного потоку;

- Множення матриць стандартним методом на CPU має обчислювальну складність $O(n^3)$, тоді як реалізація на GPU, завдяки двовимірній індексації, дозволяє скоротити кількість виконуваних циклів і значно покращити час виконання;
- Тестування на діленні матриці на число показало, що розподілене виконання цієї операції є ефективним для GPU, особливо при великих розмірах матриць, проте накладні витрати на трансфер даних між CPU та GPU можуть впливати на загальну продуктивність при малих розмірностях.

Таким чином, результати тестування демонструють, що оптимальне використання GPU під час виконання SIMD-операцій залежить від балансу між складністю обчислень у потоках та витратами на передачу даних. Графічні процесори, зокрема NVIDIA GeForce RTX 3060, показали кращу продуктивність порівняно з CPU при обробці великих масивів даних, тоді як для менш вимогливих задач чи при використанні застарілих GPU (NVIDIA GeForce 845M) ефективність може бути нижчою. Аналіз продуктивності для задач скалярного добутку векторів, множення матриць та ділення матриці на число підтверджує, що вибір архітектури обчислень має здійснюватися з урахуванням обсягу вхідних даних і складності задачі, а також особливостей передачі даних між хостом і GPU.

2 РОЗДІЛ ОХОРОНИ ПРАЦІ ТА ТЕХНІКИ БЕЗПЕКИ

Охорона праці спрямована на забезпечення безпечних і нешкідливих умов для робітників. На сучасному етапі технічного розвитку вона набуває дедалі більшого значення.

Забезпечення безпеки на робочому місці ґрунтується на досягненнях науки про організацію праці, ергономіки, гігієни та фізіології людини, а також психофізіологічних факторів. Крім того, рівень охорони праці залежить від швидкості впровадження новітніх технологій, рівня механізації та автоматизації виробничих процесів.

Забезпечення безпеки праці можливо лише за умови суворого дотримання вимог трудового законодавства, державних стандартів України та нормативних актів, що регламентують збереження здоров’я працівників. Особливу роль у цьому відіграють організаційні заходи з охорони праці, а також трудова дисципліна.

2.1 Аналіз небезпечних та шкідливих чинників, що впливають на працівника

Під час роботи за комп’ютером на людину можуть впливати такі негативні фактори:

Випромінювання електромагнітного поля;

Коливання яскравості екрану;

Тривале перебування в статичному положенні.

Сукупний ефект цих факторів може зменшувати енергетичний потенціал організму, погіршувати імунітет, викликати м’язову втому та інші негативні наслідки.

2.2 Розробка заходів з охорони праці

Зменшити вплив перерахованих факторів ризику і зберегти здоров'я людині, яка постійно використовує в роботі ПК, дозволяє дотримання всіх заходів і засобів, передбачених охороною праці.

2.2.1 Мікроклімат робочої зони працівників, вентиляція

Освітлення робочого місця має бути рівномірним, а рівень шуму в межах допустимих значень, щоб запобігти перевтомі.

Рівні позитивних і негативних іонів в повітрі приміщень з ВДТ повинні задовольняти санітарно-гігієнічним нормам № 2152-80.

2.2.2 Освітлення робочого місця, шум, вібрація

Штучне освітлення в приміщеннях з робочими місцями, обладнаними ЕОМ і ПЕОМ, має здійснюватись системою загального рівномірного освітлення. Значення освітленості на поверхні робочого столу в зоні розміщення документів має становити 300 - 500 лк.

Як джерело світла при штучному освітленні застосовуються переважно люмінесцентні лампи.

Значення освітленості на поверхні робочого столу в зоні розміщення документів має становити 300 - 500 лк.

Система загального освітлення має становити суцільні або переривчасті лінії світильників, розташовані збоку від робочих місць (переважно зліва), паралельно лінії зору працюючих Застосування світильників без розсіювачів та екрануючих грат заборонено.

Рівні звукового тиску в октавних смугах частот мають відповідати вимогам СН 3223-85, ГОСТ 12.1.003-83, ГР 2411-81.

2.2.3 Організація робочого місця користувача ПК

- Важливо, щоб офісний працівник сидячи за комп'ютером знаходився за добре освітленим робочим столом. Найчастіше саме погане освітлення робочого місця надає більш згубний для зору вплив, ніж сам факт перебування за комп'ютером.
- Робочі столи слід розміщувати таким чином, щоб монітори були орієнтовані бічною стороною до світлових прорізів, щоб природне світло падало переважно ліворуч.

– При розміщенні робочих місць відстань між робочими столами повинна бути не менше 2,0 м, а відстань між бічними поверхнями відеомоніторів - не менше 1,2 м.
– Конструкція робочого столу повинна забезпечувати оптимальне розміщення на робочій поверхні використовуваного обладнання.
– Конструкція робочого стільця або крісла повинна забезпечувати підтримку раціональної робочої пози працівника.
– Клавіатуру слід розташовувати на поверхні столу на відстані 100..300 мм від краю, зверненого до користувача, або на спеціальній поверхні, відокремленій від основної стільниці.
– Екран відеомонітора повинен знаходитися від очей користувача на відстані 600-700 мм, але не ближче 500мм.

Рисунок 2.1. Правильне положення оператора ПК на робочому місці

Безпека праці при роботі за комп’ютером передбачає, що тривалість безперервної роботи за комп'ютером без регламентованої перерви не повинна перевищувати 2 години.

Не рекомендується працювати за комп'ютером більше 6 годин за зміну. Рекомендується робити перерви в роботі за ПК тривалістю 10 хвилин через кожні 50 хвилин роботи. Під час регламентованих перерв доцільно виконувати комплекси вправ.

При нерегламентованій роботі підвищеної інтенсивності можливі головні болі, нервові зриви та інше.

2.3 Пожежна безпека

Належна пожежна безпека – це інтегральна складова організації виробничих приміщень, що ґрунтується на дотриманні чинних законодавчих норм. Сфера пожежного захисту регулюється Правилами пожежної безпеки, затвердженими наказом Міністерства внутрішніх справ України, із періодичними доповненнями та уточненнями, що відображають сучасні вимоги.

Навіть при оснащенні будівель різноманітними системами пожежогасіння, сигналізації та внутрішніми пожежними кранами, офісні приміщення повинні бути додатково обладнані основними засобами гасіння пожеж. До них відносяться вогнегасники, кошми (спеціальні негорючі покривала), ящики з піском, бочки з водою, пожежні відра, багри, ломи, сокири та інші пристосування, причому вогнегасники вважаються найбільш зручними у використанні.

За своєчасне та повне забезпечення об’єктів заходами пожежного захисту відповідає роботодавець спільно з керівниками відповідних підрозділів. Вони зобов’язані розробити та впровадити комплекс заходів, що включає створення протипожежного режиму та інструкцій, відповідно до вимог чинних нормативних актів.

Рисунок 2.2. Засоби пожежогасіння

Режим пожежного захисту повинен містити детальний опис зон спеціального призначення та встановлювати правила їх експлуатації та обслуговування, зокрема:

- шляхи евакуації;
- спеціально відведені зони для куріння (так звані «курилки»);
- приміщення для зберігання продукції та сировини;
- території для стоянки автомобілів.

Ключовим елементом протипожежного режиму є розробка алгоритму дій на випадок пожежі. Обов’язковим є створення чіткого плану евакуації, в якому буде описано порядок відключення електроустановок та визначено послідовність дій співробітників у надзвичайній ситуації.

ВИСНОВКИ

У випускній роботі проаналізовані можливості технологій GPGPU при виконанні SIMD-операцій, розглянуті питання розробки та оптимізації алгоритмів для обчислювально-інтенсивних задач із використанням графічних процесорів. Основною метою роботи було дослідити, як технологія GPGPU CUDA дозволяє реалізувати паралельне множення матриць, обчислення скалярного добутку векторів та ділення матриці на число, а також провести порівняльний аналіз продуктивності між CPU та GPU у контексті використання SIMD-операцій.

Результати тестування показали, що при збільшенні обсягу обчислень перевага GPU над CPU стає очевидною, проте оптимальна продуктивність залежить від балансу між кількістю арифметичних операцій у потоці і витратами на передачу даних між хостом і графічним пристроєм. З одного боку, для задач з високою обчислювальною складністю, таких як множення матриць, використання CUDA забезпечує суттєве прискорення завдяки ефективному розпаралелюванню. З іншого боку, при обчисленні скалярного добутку векторів, де оновлення загального результату здійснюється атомарним способом, накладні витрати на синхронізацію можуть іноді призводити до того, що CPU демонструє кращі показники, особливо на попередніх моделях GPU.

Аналіз отриманих результатів також підтвердив, що продуктивність гетерогенних систем значною мірою визначається швидкістю передачі даних між пристроями. При недостатньому обсязі обчислень перевага GPU може бути зменшеною, оскільки витрати на трансфер даних не компенсуються високою швидкістю паралельних обчислень. Результати дипломного проекту засвідчили, що технології GPGPU, зокрема CUDA, є потужним інструментом для реалізації SIMD-операцій у обчислювально-інтенсивних задачах. Водночас, ефективність використання GPU залежить від специфіки задачі, обсягу даних і архітектурних особливостей системи. Оптимізація алгоритмів, розмежування обчислень на локальні частини із подальшим атомарним злиттям результатів, а також правильне налаштування передачі даних – усе це має вирішальне значення для досягнення максимальної продуктивності.

ПЕРЕЛІК ВИКОРИСТАНИХ ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. Коваль М. І. «Технології GPGPU та їх застосування в моделюванні». – Дніпро: Електроніка, 2019.
2. Сидоренко В. Г. «SIMD-операції в сучасних обчислювальних системах». – Львів: Видавництво «Світ науки», 2018.
3. Орлова Т. Ю., Захарченко І. О. «Паралельне програмування на базі CUDA: навчальний посібник». – Київ: Вид-во «Основи», 2020.
4. Демченко Л. П. «Методи оптимізації алгоритмів на GPU». – Тернопіль: Інформаційні технології, 2021.
5. Мельник С. В. «Високопродуктивні обчислення для паралельних систем». – Київ: Академічне видавництво, 2022.
6. Гончаренко І. Л. «Архітектура GPU в обчислювальних системах: теорія та практика». – Одеса: Літопис, 2018.
7. Білик Н. О. «Підходи до розробки алгоритмів з використанням SIMD-операцій». – Львів: Університетське видавництво, 2019.
8. Федоренко П. С. «Моделювання обчислювальних процесів з використанням технологій GPGPU». – Київ: Наука і життя, 2020.
9. Єрмаков Р. А. «Паралельна обробка даних: сучасні тенденції». – Вінниця: Технічна література, 2021.
10. Кучма Ю. О. «Порівняльний аналіз CPU та GPU в обчислювальних задачах». – Харків: Професіонал, 2022.
11. Соловйов О. М. «Технології високопродуктивних обчислень». – Чернівці: Видавництво «Енциклопедія», 2020.
12. Романенко І. П. «Сучасні методи розпаралелювання обчислень». – Дніпро: Інноваційні технології, 2021.
13. Горобець В. І. «Оптимізація обчислювальних алгоритмів». – Київ: Основи, 2022.
14. Офіційна документація CUDA Toolkit. [Електронний ресурс]. – Режим доступу: https://developer.nvidia.com/cuda-toolkit, дата звернення: 15 квітня 2023.

ДОДАТОК А

Ключовий код мовою C / CUDA реалізації множення та ділення матриць

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <float.h>
#include <cuda.h>

#define N 1024          // Розмірність матриць і вектора (можна змінити на 1024, 2048, 4096)
#define BLOCK_SIZE 16   // Розмір блоку для ядра множення

// Допоміжна функція: атомарне обчислення maximum для типу float
__device__ float atomicMaxFloat(float* addr, float val) {
   int* addr_as_int = (int*) addr;
   int old = *addr_as_int, assumed;
   do {
      assumed = old;
      if (__int_as_float(assumed) >= val)
         break;
      old = atomicCAS(addr_as_int, assumed, __float_as_int(val));
   } while (assumed != old);
   return __int_as_float(old);
}

// Ядро для знаходження максимального значення вектора R
__global__ void vectMaxKernel(const float* R, float* max_val, int n) {
   __shared__ float sdata[256]; // Припускаємо, що blockDim.x = 256
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

   float x = (i < n) ? R[i] : -FLT_MAX;
   sdata[tid] = x;
   __syncthreads();

   // Редукція для знаходження максимуму у межах блоку
   for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
      if (tid < s)
         sdata[tid] = fmaxf(sdata[tid], sdata[tid+s]);
      __syncthreads();
   }

   // Результат блоку записується у глобальну пам'ять
   if (tid == 0)
      atomicMaxFloat(max_val, sdata[0]);
}

// Ядро для множення матриць із використанням формули:
// MX = e * (MC * MZ) + m * MT
```

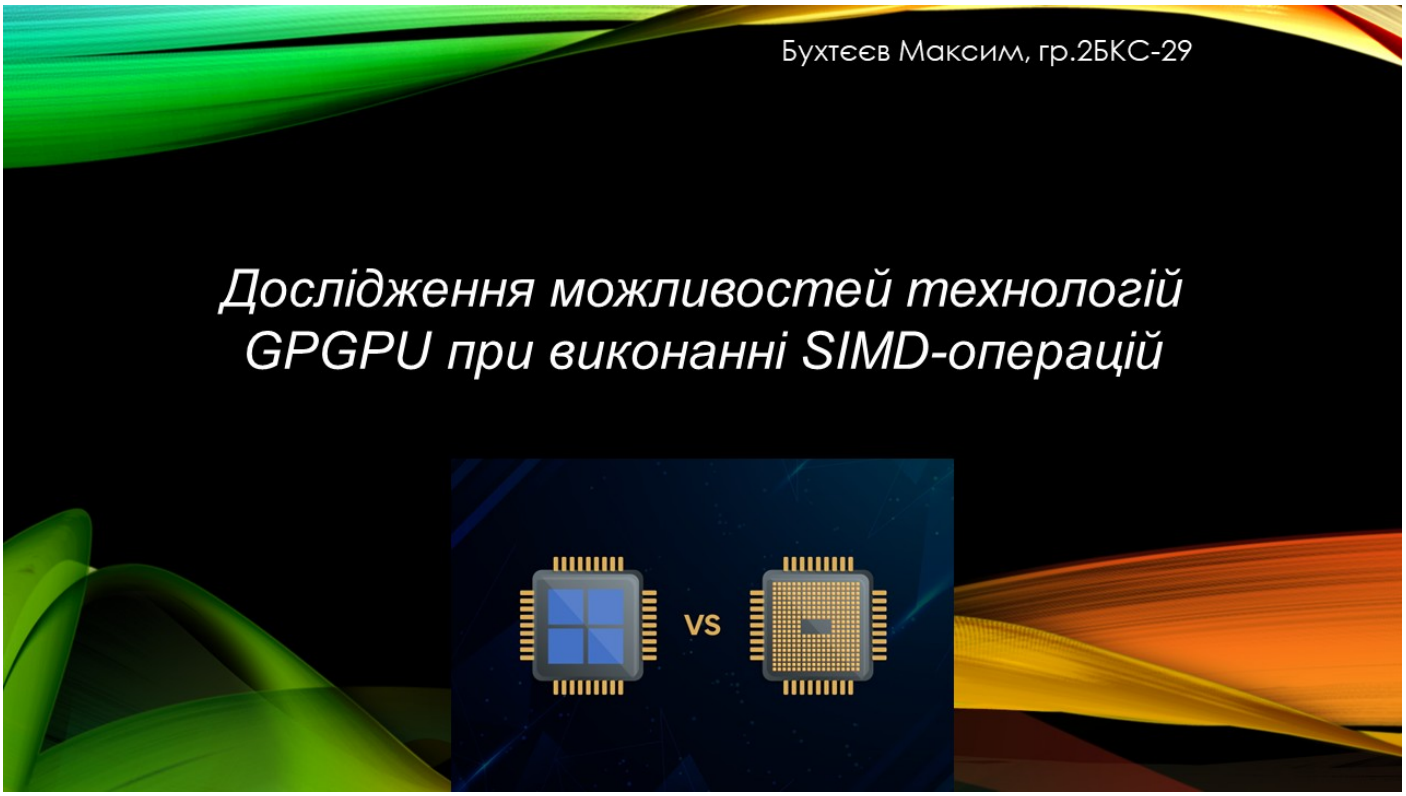
```
// Для спрощення прикладу виконується звичайне множення матриць MC та MZ,
// а потім комбінується з MT.
__global__ void matrixMultiplyKernel(const float* MC, const float* MZ, const float* MT,
                                     float* MX, float e, float m, int n) {
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   if (row < n && col < n) {
      float sum = 0.0f;
      for (int k = 0; k < n; k++) {
         sum += MC[row * n + k] * MZ[k * n + col];
      }
      MX[row * n + col] = e * sum + m * MT[row * n + col];
   }
}

int main() {
   int n = N;
   size_t matrixSize = n * n * sizeof(float);
   size_t vectorSize = n * sizeof(float);

   // Виділення пам'яті на хості
   float *h_MC = (float*) malloc(matrixSize);
   float *h_MZ = (float*) malloc(matrixSize);
   float *h_MT = (float*) malloc(matrixSize);
   float *h_MX = (float*) malloc(matrixSize);
   float *h_R  = (float*) malloc(vectorSize);

   // Ініціалізація матриць та вектора (наприклад, заповнюємо одиницями)
   for (int i = 0; i < n * n; i++) {
      h_MC[i] = 1.0f;
      h_MZ[i] = 1.0f;
      h_MT[i] = 1.0f;
   }
   for (int i = 0; i < n; i++) {
      h_R[i] = (float)(i % 100);
   }

   // Виділення пам'яті на пристрої (GPU)
   float *d_MC, *d_MZ, *d_MT, *d_MX, *d_R, *d_m;
   cudaMalloc(&d_MC, matrixSize);
   cudaMalloc(&d_MZ, matrixSize);
   cudaMalloc(&d_MT, matrixSize);
   cudaMalloc(&d_MX, matrixSize);
   cudaMalloc(&d_R,  vectorSize);
   cudaMalloc(&d_m, sizeof(float));

   // Копіювання даних з хоста у глобальну пам'ять GPU
   cudaMemcpy(d_MC, h_MC, matrixSize, cudaMemcpyHostToDevice);
   cudaMemcpy(d_MZ, h_MZ, matrixSize, cudaMemcpyHostToDevice);
   cudaMemcpy(d_MT, h_MT, matrixSize, cudaMemcpyHostToDevice);
   cudaMemcpy(d_R,  h_R,  vectorSize, cudaMemcpyHostToDevice);
```

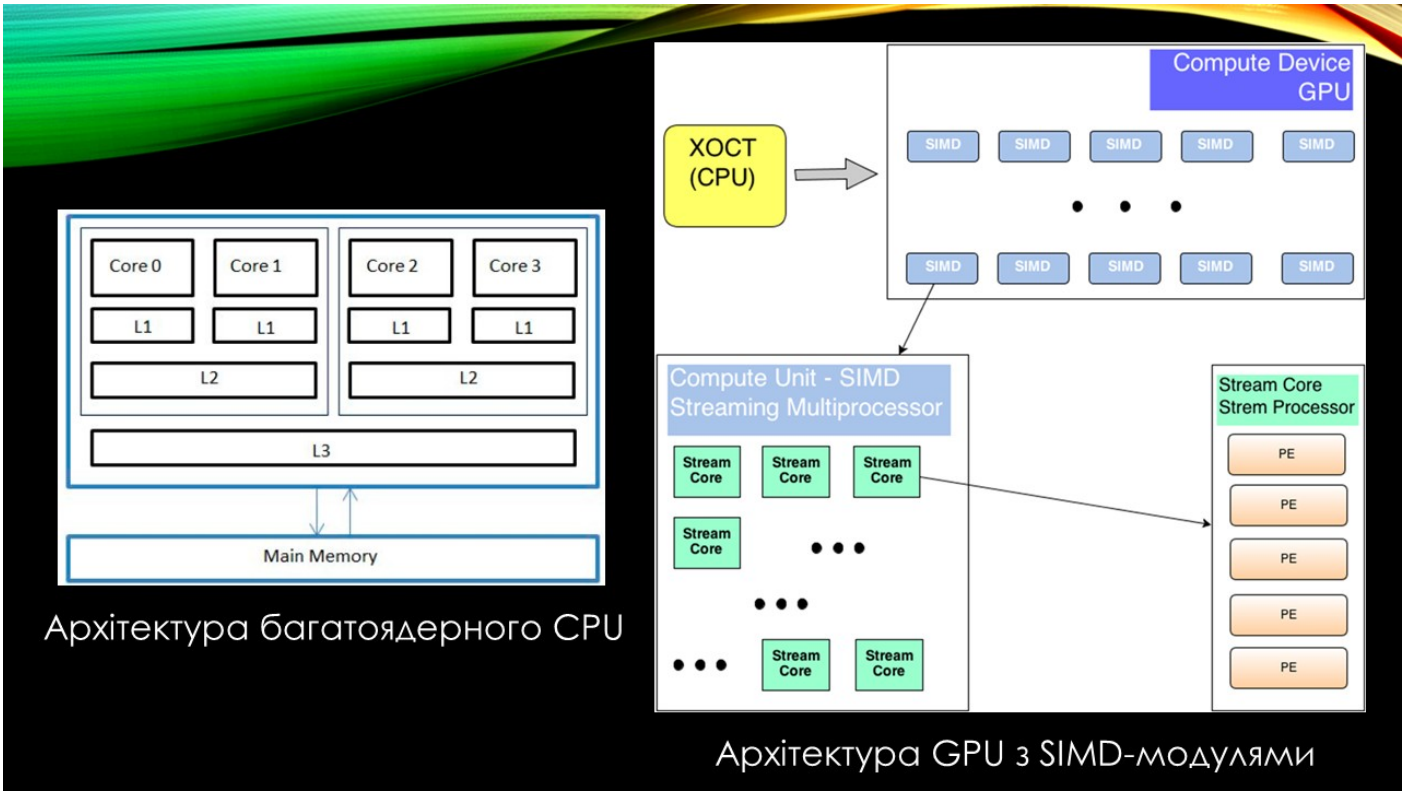
```
    // Ініціалізація глобальної змінної для максимуму (встановлюється дуже мала величина)
    float initVal = -FLT_MAX;
    cudaMemcpy(d_m, &initVal, sizeof(float), cudaMemcpyHostToDevice);

    // Вимірювання часу виконання за допомогою clock()
    clock_t start = clock();

    // Запуск ядра для знаходження максимального значення вектора R
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    vectMaxKernel<<<blocksPerGrid, threadsPerBlock>>>(d_R, d_m, n);
    cudaDeviceSynchronize();

    // Копіювання отриманого максимального значення з GPU на хост
    float h_m;
    cudaMemcpy(&h_m, d_m, sizeof(float), cudaMemcpyDeviceToHost);

    // Параметр e як скаляр
    float e = 2.0f;

    // Налаштування сітки для ядра множення матриць
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 gridDim((n + BLOCK_SIZE - 1) / BLOCK_SIZE, (n + BLOCK_SIZE - 1) / BLOCK_SIZE);
    matrixMultiplyKernel<<<gridDim, threads>>>(d_MC, d_MZ, d_MT, d_MX, e, h_m, n);
    cudaDeviceSynchronize();

    clock_t end = clock();
    double elapsedTime = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Час виконання: %f секунд\n", elapsedTime);
    printf("Обчислене максимальне m: %f\n", h_m);

    // Копіювання результатної матриці MX з GPU на хост для подальшої перевірки
    cudaMemcpy(h_MX, d_MX, matrixSize, cudaMemcpyDeviceToHost);
    printf("MX[0][0] = %f\n", h_MX[0]);  // Вивід першого елементу для перевірки

    // Звільнення пам'яті на пристрої
    cudaFree(d_MC); cudaFree(d_MZ); cudaFree(d_MT); cudaFree(d_MX); cudaFree(d_R);
cudaFree(d_m);
    // Звільнення пам'яті на хості
    free(h_MC); free(h_MZ); free(h_MT); free(h_MX); free(h_R);

    return 0;
}

#define N 3 // Розмір матриці

// Ядро для приведення матриці до одиничної форми
__global__ void inverseMatrixKernel(float* A, float* I, int n) {
    int row = threadIdx.x;

    if (row < n) {
```

```
        // Нормалізація поточного рядка
        float diag = A[row * n + row];
        for (int col = 0; col < n; col++) {
            A[row * n + col] /= diag;
            I[row * n + col] /= diag;
        }

        // Занулення інших рядків
        for (int r = 0; r < n; r++) {
            if (r != row) {
                float factor = A[r * n + row];
                for (int col = 0; col < n; col++) {
                    A[r * n + col] -= factor * A[row * n + col];
                    I[r * n + col] -= factor * I[row * n + col];
                }
            }
        }
    }
}

// Ядро для множення двох матриць
__global__ void matrixMultiplyKernel(const float* A, const float* B, float* C, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < n) {
        float sum = 0.0f;
        for (int k = 0; k < n; k++) {
            sum += A[row * n + k] * B[k * n + col];
        }
        C[row * n + col] = sum;
    }
}

int main() {
    int n = N;
    size_t matrixSize = n * n * sizeof(float);

    float h_A[N][N] = { {2, -1, 0}, {1, 3, -1}, {0, 1, 2} }; // Вхідна матриця
    float h_B[N][N] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };   // Друга матриця
    float h_I[N][N] = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };   // Одинична матриця
    float h_C[N][N];  // Результат

    // Виділення пам'яті на GPU
    float *d_A, *d_B, *d_I, *d_C;
    cudaMalloc(&d_A, matrixSize);
    cudaMalloc(&d_B, matrixSize);
    cudaMalloc(&d_I, matrixSize);
    cudaMalloc(&d_C, matrixSize);

    // Копіювання даних в GPU
    cudaMemcpy(d_A, h_A, matrixSize, cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(d_B, h_B, matrixSize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_I, h_I, matrixSize, cudaMemcpyHostToDevice);

    // Запуск ядра для оберненої матриці
    inverseMatrixKernel<<<1, n>>>(d_A, d_I, n);
    cudaDeviceSynchronize();

    // Запуск ядра множення для отримання результату ділення
    dim3 threadsPerBlock(N, N);
    dim3 blocksPerGrid(1, 1);
    matrixMultiplyKernel<<<blocksPerGrid, threadsPerBlock>>>(d_I, d_B, d_C, n);
    cudaDeviceSynchronize();

    // Копіювання результату на хост
    cudaMemcpy(h_C, d_C, matrixSize, cudaMemcpyDeviceToHost);

    // Вивід результату
    printf("Результат ділення матриць:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%f ", h_C[i][j]);
        }
        printf("\n");
    }

    // Звільнення пам'яті
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_I);
    cudaFree(d_C);

    return 0;
}
```

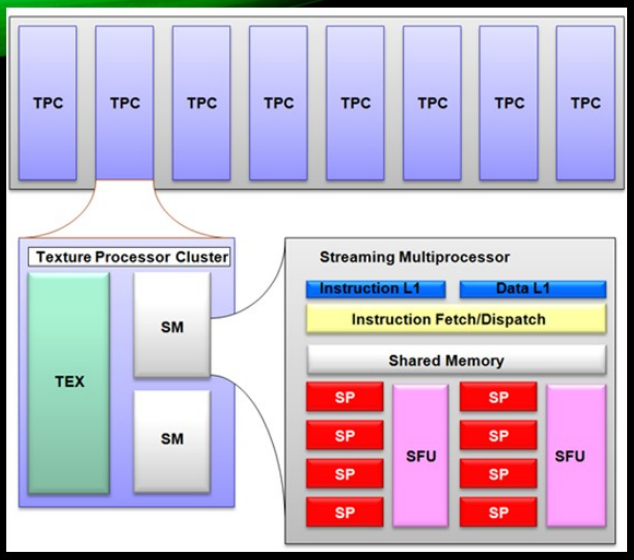
ДОДАТОК Б. Слайди мультимедійної презентації



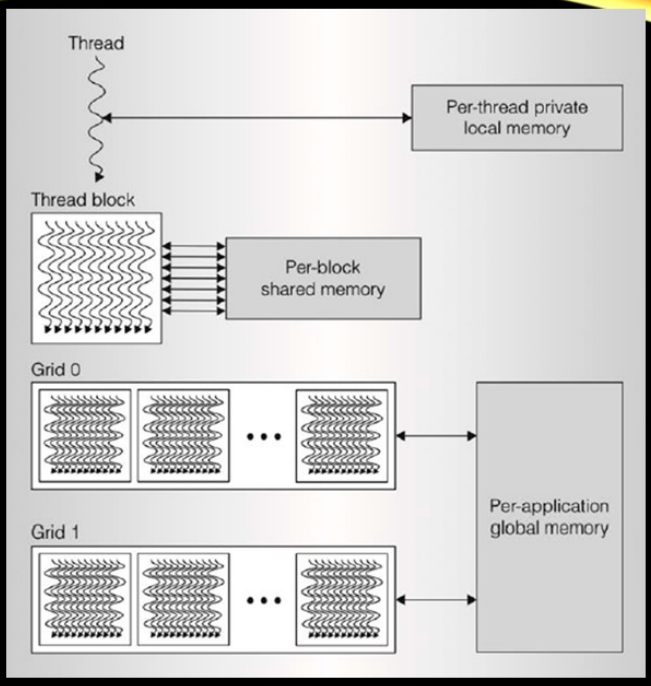


Характеристика	CPU	GPU
Кількість ядр	Невелика (4–16 високопродуктивних ядр)	Велика (сотні-тисячі простих обчислювальних ядр)
Призначення	Загальне призначення, оптимізація для послідовних обчислень та складних алгоритмів	Оптимізований для масових паралельних обчислень, підходить для задач типу SIMD/SIMT
Тактова частота	Висока; забезпечує швидке послідовне виконання	Низька; компенсується великим числом потоків
Система кешування	Розвинена ієрархія кеш-пам'яті (L1, L2, L3)	Менша складність кешування, з високою пропускною здатністю глобальної пам'яті
Фокус	Гнучкість та обробка розгалужених інструкцій	Ефективність при виконанні одноманітних обчислювальних завдань

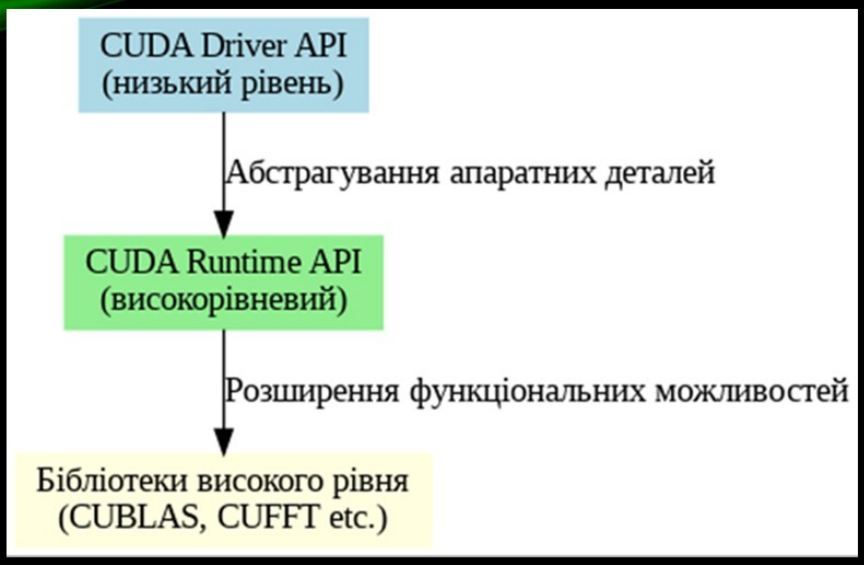
Порівняльний аналіз архітектури CPU та GPU



Структура відеоадаптера NVIDIA

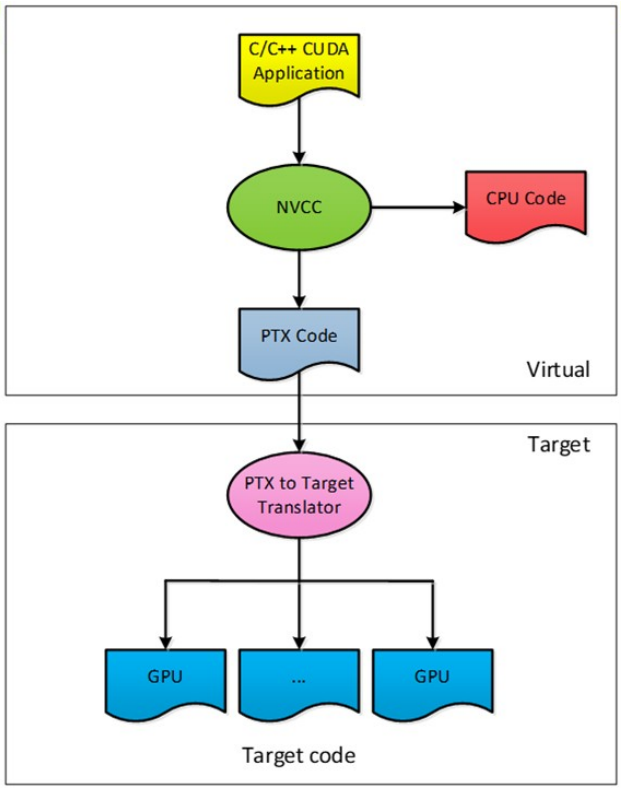


Архітектура CUDA



Ієрархія засобів програмування CUDA

Етапи компіляції застосунків GPGPU CUDA



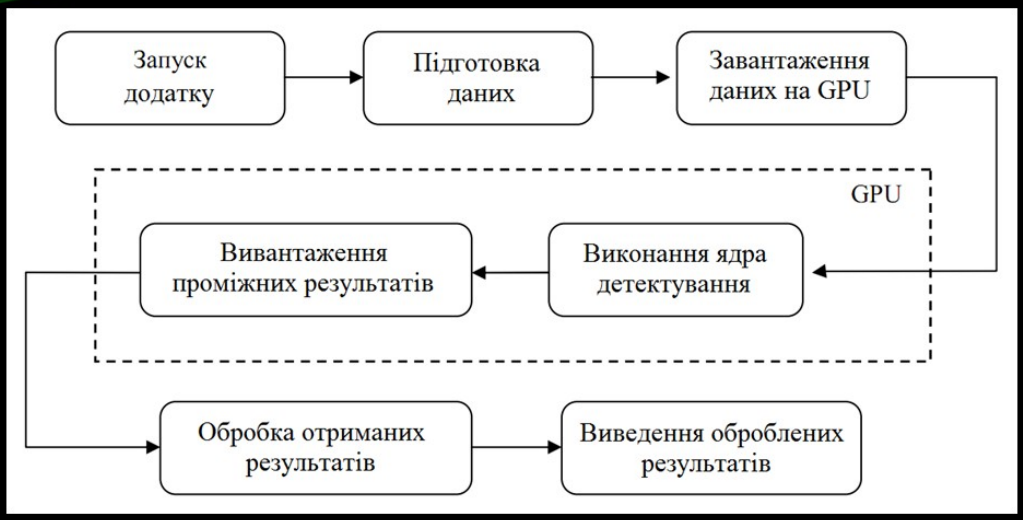


Схема рішення задачі з використанням GPU

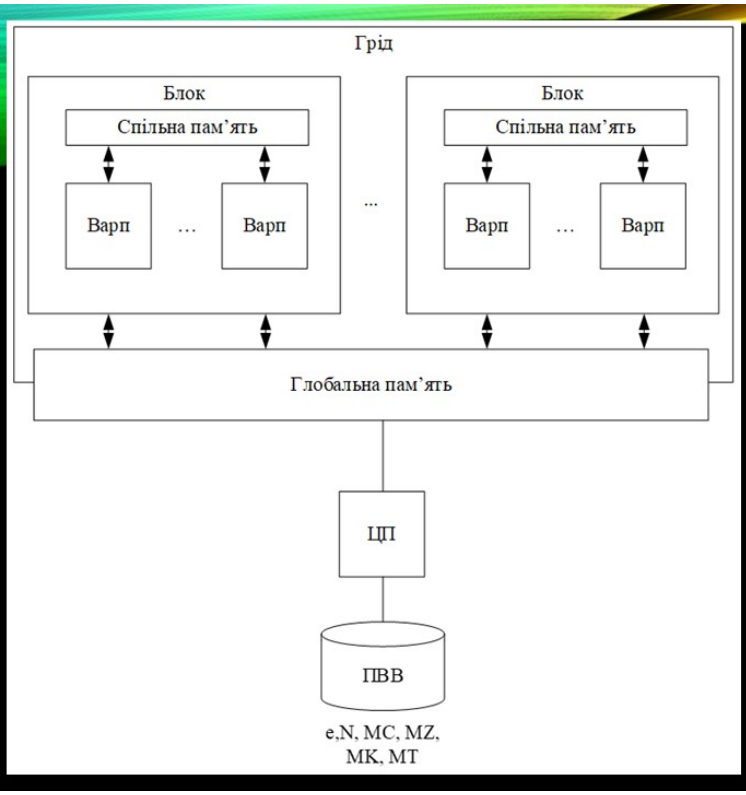


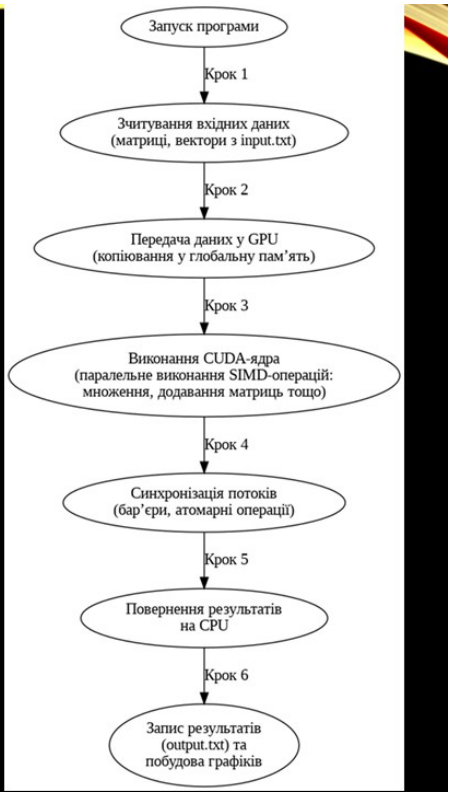
Схема розподілених обчислень для реалізації SIMD-операцій множення матриць за допомогою GPGPU CUDA

1. $m_i = \max(R_H), i = \overline{1,P}$
2. $m = \max(m, m_i), i = \overline{1,P}$
3. $MX_H = e * (MC_H * MZ * MK) + m * MT_H$

де

- e, m, MZ, MK – спільні ресурси;
- P – кількість потоків;
- N – розмірність векторів та матриць;
- H = N/P;
- R_H – H елементів вектору R;
- MC_H – рядків матриці MC;
- MT_H – рядків матриці MT;
- MX_H – рядків матриці MX;

Алгоритм математичних операцій з матрицями, що реалізується з використанням GPGPU CUDA



Назва	*Бар'єри, КД*
1. Введення e, R, MC, MZ, MK, MT	
2. Передача e, R, MC, MZ, MK, MT у глобальну пам'ять	
3. Якщо tid = 1, надіслати сигнал про введення e, R, MC, MZ, MK, MT	$S_{2\text{-}P\text{ - }1}$
4. Якщо tid != 1, очікувати сигнал про введення e, R, MC, MZ, MK, MT	$W_{2\text{-}P\text{ - }1}$
5. Обчислення $m_i = \max(R_H)$	
6. Обчислення $m=\max(m, m_i)$	
7. Надіслати сигнал про обрахування m	$S_{1\text{-}(tid\text{-}1),(tid+1)\text{-}P\text{ - }tid}$
8. Очікувати сигнал про обрахування m	$W_{1\text{-}(tid\text{-}1),(tid+1)\text{-}P\text{ - }tid}$
9. Копіювати $e_i = e$, $m_i = m$, $MZ_i = MZ$, $MK_i = MK$	КД
10. Обчислення $MX_H=e_i*(MC_H*MZ_i*MK_i) + m_i*MT_H$	
11. Якщо tid != 1, надіслати сигнал про закінчення обрахування	$S_{1\text{ – }2\text{-}P}$
12. Якщо tid = 1, очікувати сигнали про закінчення обрахування MX_H	$W_{2\text{-}P\text{ - }1}$
13. Якщо tid = 1, передати результат у ОЗП	
14. Виведення результату MX	

Алгоритм синхронізації та керування взаємодією потоків

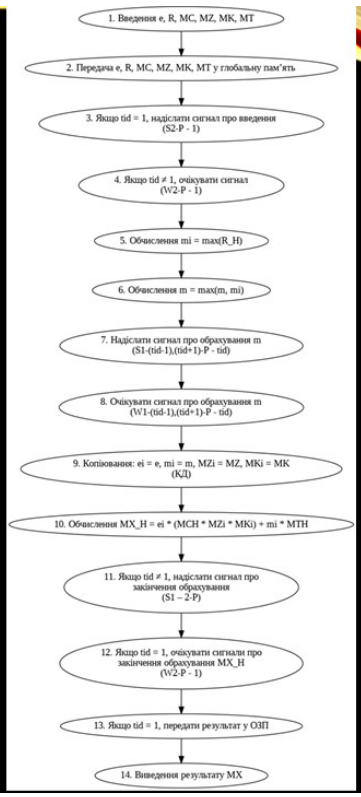
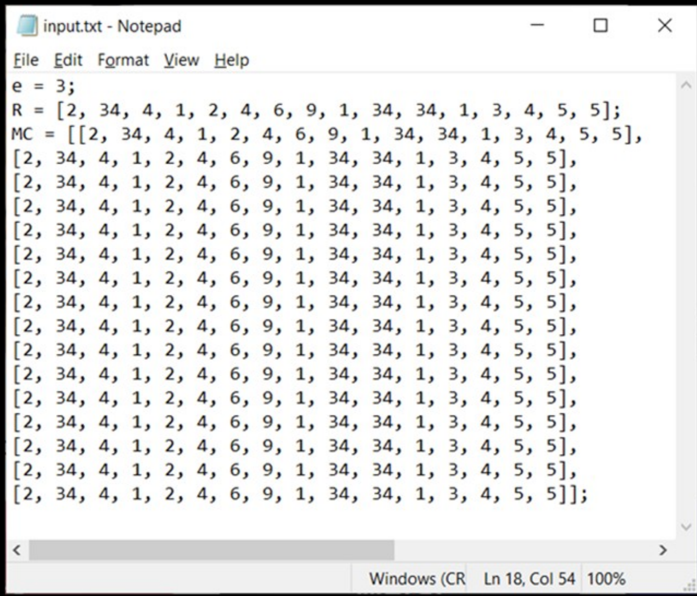


Схема процесу вводу-виведення даних



```
e = 3;
R = [2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5];
MC = [[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5],
[2, 34, 4, 1, 2, 4, 6, 9, 1, 34, 34, 1, 3, 4, 5, 5]];
```

Приклад заповнення вхідного файлу

Апаратні характеристики тестової системи:

- CPU: Intel® Core™ i7-12700H з базовою частотою 2.4 ГГц та режимом Turbo Boost до 4.9 ГГц;
- GPU: Nvidia® GeForce® RTX 3060 (6 ГБ GDDR6, базова частота ~1300 МГц, 192-бітна шина даних);
- Оперативна пам’ять: 16 ГБ DDR4.

Для аналізу продуктивності застосунку тестування проводилося з використанням трьох варіантів розмірності векторів і квадратних матриць:

- N = 1024;
- N = 2048;
- N = 4096.

Обчислення виконувалися як послідовно (на CPU), так і паралельно (на GPU) із налаштуванням кількості паралельних потоків (ядер) у трьох режимах: 32, 128 та 256.

N	CPU T_1 (ms)	GPU T_2 (ms, P=32)	GPU T_3 (ms, P=128)	GPU T_4 (ms, P=256)
1024	4100	1950	1350	1020
2048	65000	33000	22000	17000
4096	180000	80000	55000	42000

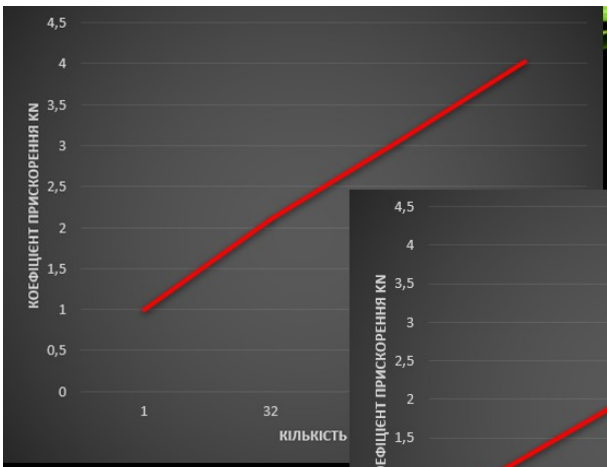
Час виконання програми (в мілісекундах)

N	CPU	GPU (P=32)	GPU (P=128)	GPU (P=256)
1024	1	2.10	3.04	4.02
2048	1	1.97	2.95	3.82
4096	1	2.25	3.27	4.29

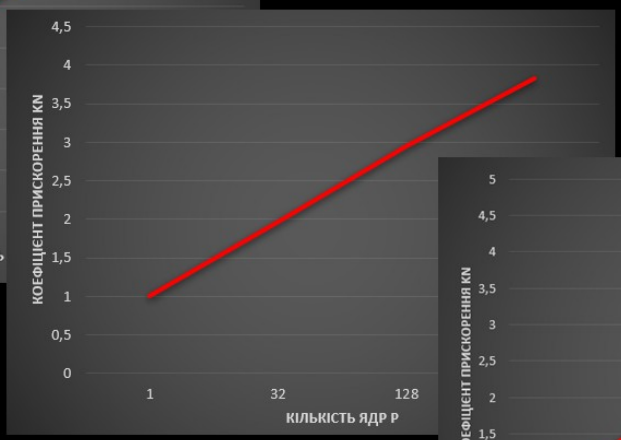
Значення коефіцієнта прискорення K_n

N	CPU	GPU (P=32)	GPU (P=128)	GPU (P=256)
1024	100	6.56%	2.38%	1.57%
2048	100	6.16%	2.30%	1.49%
4096	100	7.03%	2.55%	1.68%

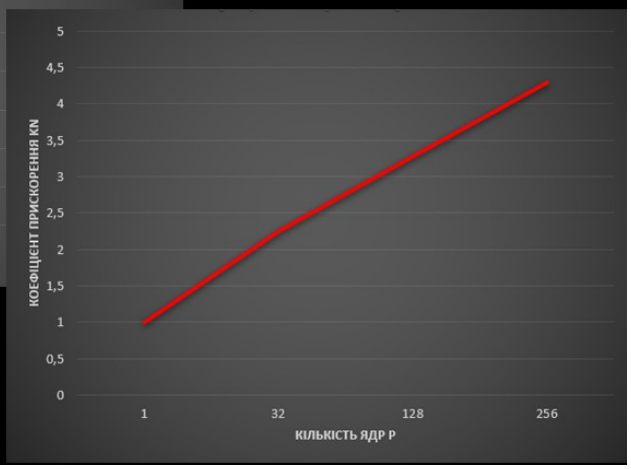
Значення коефіцієнта ефективності K_e



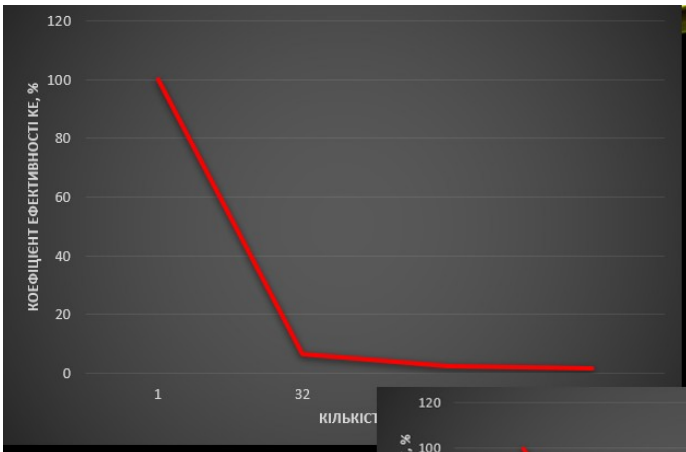
Коефіцієнт прискорення в залежності від кількості ядр (N=1024)



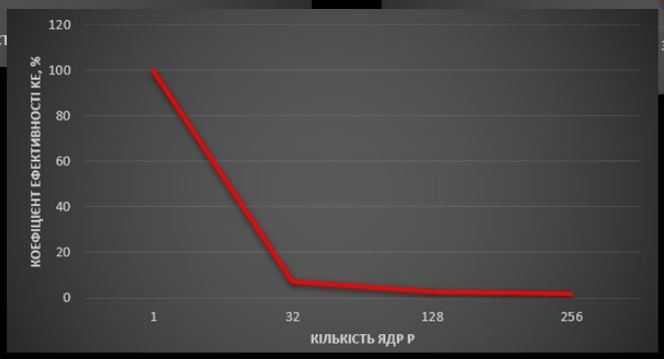
Коефіцієнт прискорення в залежності від кількості ядр (N=2048)



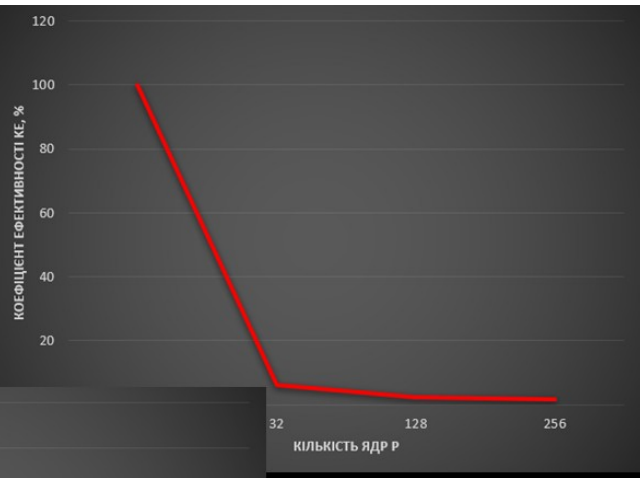
Коефіцієнт прискорення в залежності від кількості ядр (N=4096)



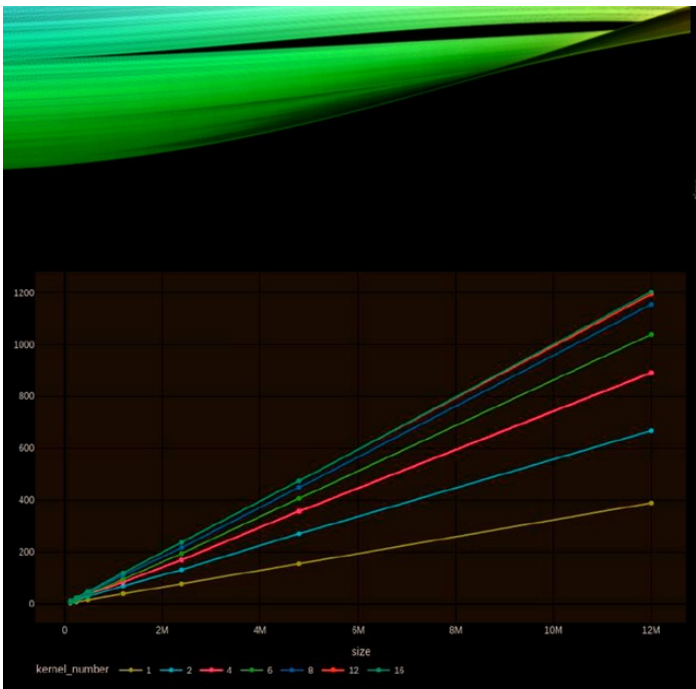
Коефіцієнт ефективності в залежності від кількості ядр (N=1024)



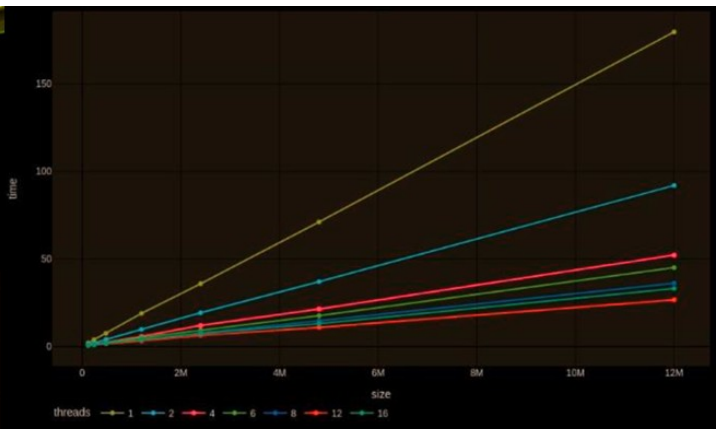
Коефіцієнт ефективності в залежності від кількості ядр (N=4096)



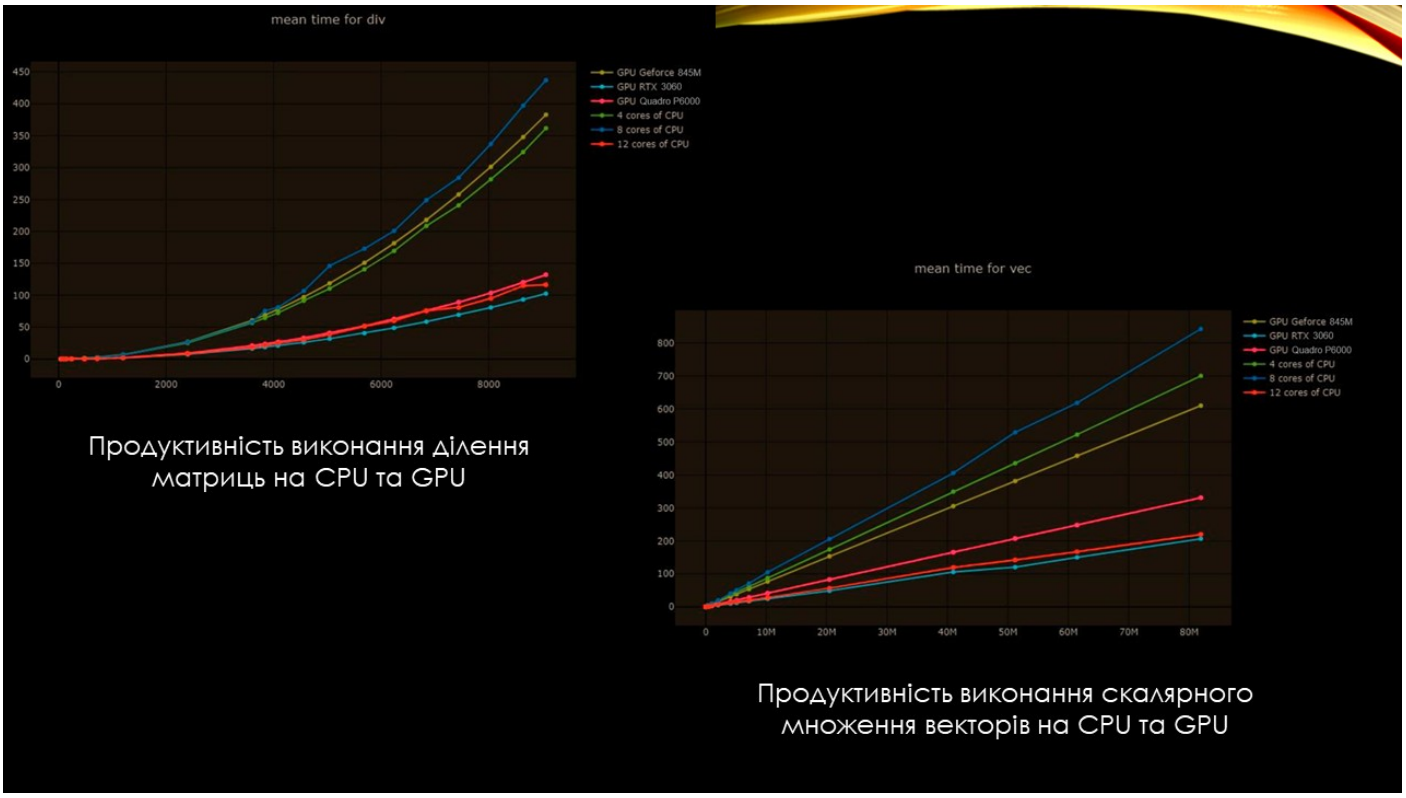
Коефіцієнт ефективності в залежності від кількості ядр (N=2048)



Продуктивність виконання скалярного множення векторів на CPU

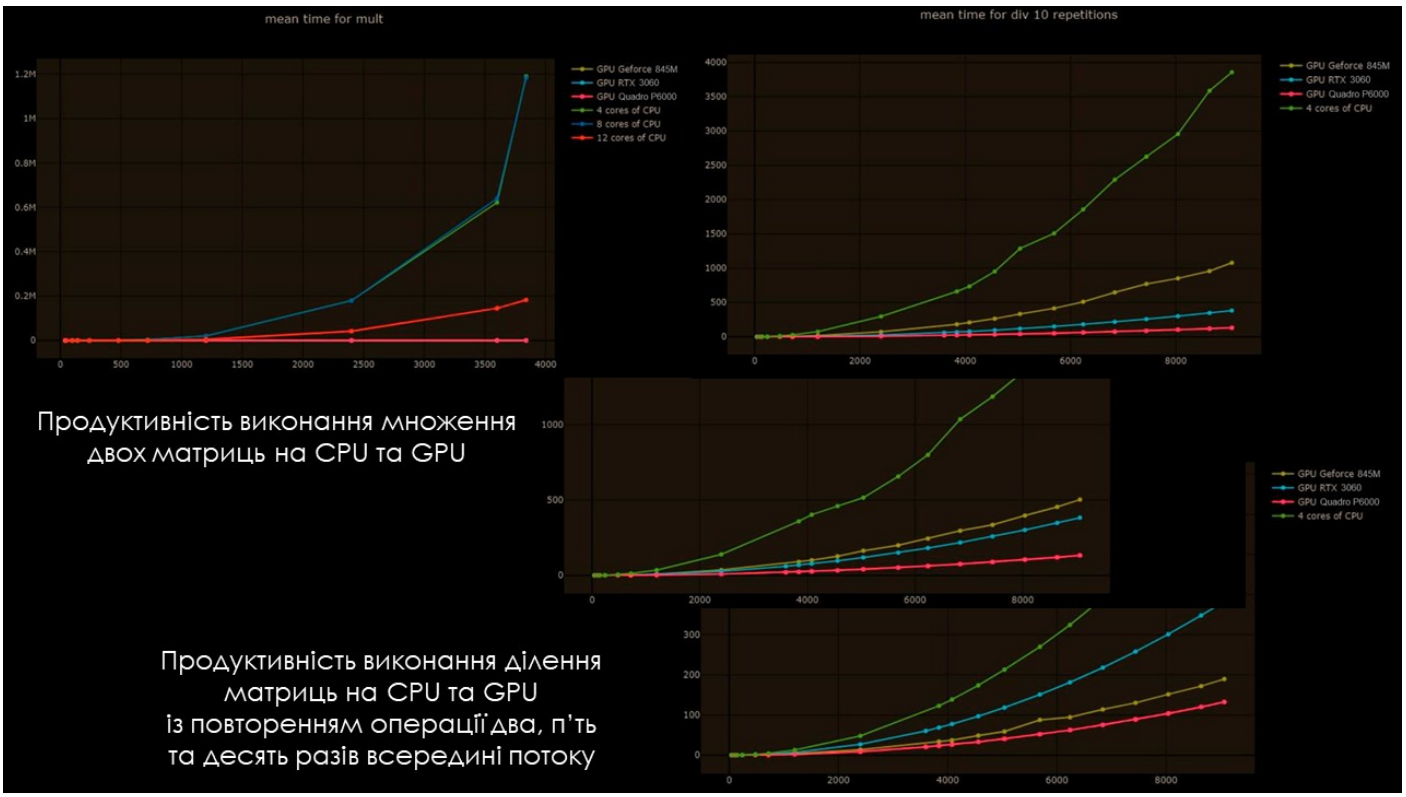


Продуктивність виконання скалярного множення векторів на CPU за оптимізованим алгоритмом багатопотокового обчислення



Продуктивність виконання ділення матриць на CPU та GPU

Продуктивність виконання скалярного множення векторів на CPU та GPU



Продуктивність виконання множення двох матриць на CPU та GPU

Продуктивність виконання ділення матриць на CPU та GPU із повторенням операції два, п'ть та десять разів всередині потоку

ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

РЕЦЕНЗІЯ

на кваліфікаційну роботу здобувача (здобувачки) освіти

відділення комп'ютерних систем

Бухтєєва Максима Ігоровича

(прізвище, ім`я та по батькові)

Спеціальність *123 "Комп'ютерна інженерія"*

Освітньо-професійна програма *«Комп'ютерна інженерія»*

Керівник кваліфікаційної роботи *Кривченко Анастасія Анатоліївна*

(прізвище, ім`я та по батькові)

Тема кваліфікаційної роботи *Дослідження можливостей технологій GPGPU при виконанні SIMD-операцій*

Обсяг розрахунково-пояснювальної записки *68* сторінок

Обсяг графічної (презентаційної) частини *18* аркушів (слайдів)

ХАРАКТЕРИСТИКА КВАЛІФІКАЦІЙНОЇ РОБОТИ

а) заключення про ступінь відповідності виконаного кваліфікаційної роботи завданню

Представлена на рецензію кваліфікаційна робота бакалавра повністю відповідає меті випускної роботи та технічному завданню. Тематика кваліфікаційної роботи є актуальною для своєї галузі та присвячена моделюванню та аналізу можливостей технологій GPGPU при виконанні SIMD-операцій

б) характеристика виконання кожного розділу кваліфікаційної роботи

Кваліфікаційна робота складається зі вступу, двох розділів, висновків, переліку використаних джерел. У основному розділі виконано порівняння архітектури CPU та GPU; Аналіз технології GPGPU; Огляд засобів програмування GPGPU CUDA; Реалізація SIMD-операцій з матрицями за допомогою GPGPU; Тестування продуктивності обчислення матриці MX; Тестування продуктивності множення матриць; Питання охорони праці та техніки безпеки

в) оцінка якості виконання пояснювальної записки та графічної частини кваліфікаційної роботи

Графічна частина виконана на достатньо високому рівні у вигляді презентації із використанням офісного пакету Microsoft PowerPoint та Visio. Пояснювальна записка виконана охайно та у відповідності до норм оформлення документів із використанням офісного пакету Microsoft Word. Загальна якість виконання документації – добра, академічного плагіату ідей у роботі не виявлено

г) перелік позитивних якостей кваліфікаційної роботи

Наведені ключові фрагменти коду CUDA-ядер, показано, як реалізувати пошук максимуму, синхронізацію потоків, матричні операції та оптимізувати скалярний добуток. Проведено серію вимірювань на трьох різних розмірах (1024, 2048, 4096) і трьох-рівневому блоці (32, 128, 256), побудовано таблиці та графіки коефіцієнтів прискорення і ефективності для CPU і кількох GPU.

д) основні недоліки кваліфікаційної роботи

Невисока фактична ефективність використання потоків - значні накладні витрати на синхронізацію й трансфер, але обговорення причин обмежене й не містить рекомендацій для покращення.

Тестування лише на рівних квадратних матрицях та векторах, без експериментів із нерівномірними розмірами або реальних наукових задач.

Оцінка розрахункової частини *Відмінно*

Оцінка графічної частини *Добре*

Загальна оцінка *Відмінно*

Прізвище, ім'я, по батькові рецензента *к.т.н. Рудніченко Микола Дмитрович*

Місце роботи і посада рецензента *Національний університет «Одеська політехніка», доцент кафедри інформаційних технологій*

Підпис

« 23 » 06 2025 р.

ВСП «Одеський технічний фаховий коледж ОНТУ»

ВІДГУК

керівника про кваліфікаційну роботу бакалавра

Бухтєєва Максима Ігоровича

(прізвище, ім`я та по батькові)

Спеціальність *123 "Комп'ютерна інженерія"*

Тема кваліфікаційної роботи *Дослідження можливостей технологій GPGPU при виконанні SIMD-операцій*

ХАРАКТЕРИСТИКА КВАЛІФІКАЦІЙНОЇ РОБОТИ

а) Обсяг і якість виконання роботи (графічного матеріалу і розрахунково-пояснювальної записки) *Випускна робота виконана відповідно технічному завданню. Пояснювальна записка до випускної роботи містить 68 сторінок. У пояснювальній записці викладено аналіз можливостей технологій GPGPU при виконанні SIMD-операцій, які дозволяють використовувати обчислювальні ресурси відеоадаптера для рішення обчислювальних задач широкого кола, зокрема, реалізовано математичні задачі обчислення матриць. Графічна частина складається з 18 слайдів, оформлених у вигляді презентації, передбачених технічним завданням. Якість виконання пояснювальної записки та слайдів добра, розробку виконано у повному обсязі.*

б) Самостійність роботи *Протягом виконання випускної бакалаврської роботи Бухтєєв Максим поступово та послідовно виконував всі етапи, проявив ініціативу у створенні загальної концепції та реалізації випускної роботи. Всі роботи він виконував самостійно, з оглядом на рекомендації керівника.*

в) Теоретична підготовка здобувача освіти

Бухтєєв Максим під час роботи над випускною бакалаврською роботою вивчив і опрацював достатню кількість літературних джерел за даною тематикою.

Вважаю, що теоретична підготовка здобувача освіти достатня і він готовий до захисту роботи.

г) Вміння розв`язувати виробничі і конструкторські питання на базі останніх досліджень науки і техніки, передових методів виробництва

Під час виконання роботи Бухтєєв Максим мав змогу самостійно приймати окремі рішення з виконання програмної частини роботи та показав вміння організовано працювати над поставленою задачею, складати та оформлювати презентацію проекту, користуючись сучасними комп'ютерними програмними засобами, такими як Microsoft Visual Studio 2022; CUDA Toolkit.

Оцінка розрахункової частини *Добре*

Оцінка графічної частини *Відмінно*

Загальна оцінка *Відмінно*

Прізвище, ім`я, по батькові *Кривченко Анастасія Анатоліївна*

Місце роботи і посада керівника роботи *ВСП "Одеський технічний фаховий коледж ОНТУ", викладач кафедри комп'ютерної інженерії, голова обласної методичної комісії викладачів комп'ютерної інженерії*

Підпис

« 20» 06 2025 р.

ДОЗВІЛ
НА РОЗМІЩЕННЯ
ВИПУСКНОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
В ЕЛЕКТРОННОМУ РЕПОЗИТАРІЇ ВСП «ОТФК ОНТУ»

Ми, що нижче підписалися,

Бухтєєв М.І.,
здобувач освіти гр. 2БКС-29, та

Кривченко А.А.,
керівник дипломного проекту,

не заперечуємо щодо розміщення електронного варіанту пояснювальної записки до випускної кваліфікаційної роботи бакалавра на тему:

«Дослідження можливостей технологій GPGPU при виконанні SIMD-операцій» (автор роботи – Бухтєєв М.І., керівник роботи – Кривченко А.А.)

виконаного у ВСП «Одеський технічний фаховий коледж Одеського національного технологічного університету» в 2025 році, у повному обсязі в електронному репозитарії ВСП «ОТФК ОНТУ» для вільного доступу через мережу Інтернет.

Несемо відповідальність за ідентичність електронного та друкованого варіантів випускної кваліфікаційної роботи, і даємо згоду на обробку персональних даних.

Виконавець _____ / Бухтєєв М.І. /

Керівник _____ / Кривченко А.А. /

«16» червня 2025 р.

ДОВІДКА

кафедри комп'ютерної інженерії
про допуск до захисту кваліфікаційної роботи
здобувача (здобувачки) освіти II курсу
відділення комп'ютерних систем групи 2БКС-29

Бухтєєва Максима Ігоровича

на тему *Дослідження можливостей технологій GPGPU при виконанні SIMD-операцій*

Висновок відповідальної особи за проведення нормоконтролю:
пояснювальна записка до кваліфікаційної роботи виконана з некритичними порушеннями ДСТУ та оформлена відповідно до вимог Положення про дипломне проєктування

(підпис) *20.06.2025* (дата) *Петрашова В.І.* (П.І.Б.)

Висновок відповідальної особи за перевірку роботи на наявність академічного плагіату *згідно звіту про перевірку від 31.05.2025 р. значення коефіцієнту подібності в роботі становить 13,29%, коефіцієнт цитування – 1,27%.*

(підпис) *20.06.2025* (дата) *Краснокутська К.Г.* (П.І.Б.)

Попередня експертиза (малий захист) кваліфікаційної роботи

здобувача (здобувачки) освіти *Бухтєєва М.І.* (П.І.Б.)

проведена « *20* » *червня* 2025 р.

Висновки *Пояснювальна записка до кваліфікаційної роботи виконана у повному обсязі. Випускна кваліфікаційна робота відповідає вимогам Положення про дипломне проєктування та рекомендована до захисту.*

Зав. кафедри КІ (підпис) *Іванова Л.В.* (П.І.Б.)

StrikePlagiarism.com

Дата звіту 5/31/2025
Дата редагування 5/31/2025

Документ прийнятий

Звіт подібності

метадані

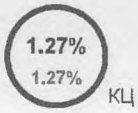
Назва організації
Odesa Technical Professional College of Odesa National University of Technology
Заголовок
Дослідження можливостей технологій GPGPU при виконанні SIMD-операцій

Автор
Бухтєєв Максим Ігорович
Науковий керівник / Експерт
Кривченко Анастасія Анатоліївна

підрозділ
Відокремлений структурний підрозділ "Одеський технічний фаховий коледж Одеського національного технологічного університету"

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



25
Довжина фрази для коефіцієнта подібності 2

11706
Кількість слів

95874
Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	2
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	412
Парафрази (SmartMarks)	a	65

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://card-file.ontu.edu.ua/server/api/core/bitstreams/d5a3d14f-d5cb-460f-9c49-cba3f9d50554/content	106 0.91 %
2	https://card-file.ontu.edu.ua/bitstreams/361286d7-8a03-4221-ad05-db5133ab5f79/download	50 0.43 %
3	https://revolution.allbest.ru/management/00769018_0.html	39 0.33 %
4	https://card-file.ontu.edu.ua/bitstreams/361286d7-8a03-4221-ad05-db5133ab5f79/download	37 0.32 %
5	https://revolution.allbest.ru/management/00769018_0.html	27 0.23 %

6	Пошук оптимального розміру вхідного набору даних для ефективного прискорення базових матричних операцій на GPU [redacted] 3/15/2025 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute)	24 0.21 %
7	Пошук оптимального розміру вхідного набору даних для ефективного прискорення базових матричних операцій на GPU [redacted] 3/15/2025 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute)	23 0.20 %
8	https://conferences.vntu.edu.ua/public/files/itpf/conf_itpf-2016_all.pdf	23 0.20 %
9	https://medium.com/@iphoenix179/running-cuda-c-c-in-jupyter-or-how-to-run-nvcc-in-google-colab-663d33f53772	22 0.19 %
10	2017_6050903_Sobechko_Rostyslav_Olehovych_8074 [redacted] 10/26/2024 National University "Lviv Politechnika" (National University Lviv Politechnika)	22 0.19 %

з домашньої бази даних (0.05 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	Створення web-застосунку цифрового помічника з використанням Open AI [redacted] 5/28/2025 Odesa Technical Professional College of Odesa National University of Technology (Відокремлений структурний підрозділ "Одеський технічний фаховий коледж Одеського національного технологічного університету")	6 (1) 0.05 %

з програми обміну базами даних (1.38 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	Пошук оптимального розміру вхідного набору даних для ефективного прискорення базових матричних операцій на GPU [redacted] 3/15/2025 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute)	72 (4) 0.62 %
2	Диплом_Січкаренко [redacted] 5/28/2024 Interregional Academy of Personnel Management (Інститут комп'ютерно-інформаційних технологій та дизайну)	33 (4) 0.28 %
3	2017_6050903_Sobechko_Rostyslav_Olehovych_8074 [redacted] 10/26/2024 National University "Lviv Politechnika" (National University Lviv Politechnika)	22 (1) 0.19 %
4	Програмна система паралельних обчислень на базі технології Nvidia Cuda [redacted] 3/16/2025 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute)	18 (2) 0.15 %
5	Високопродуктивний програмно-апаратний комплекс для операцій з комплексними матрицями [redacted] 3/16/2025 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute)	11 (2) 0.09 %
6	Сортування фото та відео по теках [redacted] 6/6/2024 Zhytomyr Agricultural Technical Professional College (Zhytomyr Agricultural Technical Professional College)	6 (1) 0.05 %

з Інтернету (11.86 %)

ПОРЯДКОВИЙ НОМЕР	ДЖЕРЕЛО URL	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://card-file.ontu.edu.ua/bitstreams/0e6c3361-ffbf-4469-86a1-fe84a1fe21cd/download	469 (42) 4.01 %
2	**https://card-file.ontu.edu.ua/bitstreams/361286d7-8a03-4221-ad05-db5133ab5f79/download**	**107 (5) 0.91 %**
3	**https://card-file.ontu.edu.ua/server/api/core/bitstreams/d5a3d14f-d5cb-460f-9c49-cba3f9d50554/content**	**106 (1) 0.91 %**
4	https://medium.com/@iphoenix179/running-cuda-c-c-in-jupyter-or-how-to-run-nvcc-in-google-colab-663d33f53772	97 (6) 0.83 %
5	https://card-file.ontu.edu.ua/bitstreams/53ed22ad-8700-4162-b97a-082a1ad472d6/download	96 (9) 0.82 %
6	https://conferences.vntu.edu.ua/public/files/itpf/conf_itpf-2016_all.pdf	89 (7) 0.76 %
7	**https://revolution.allbest.ru/management/00769018_0.html**	**78 (3) 0.67 %**
8	https://studopedia.org/6-120929.html	60 (5) 0.51 %
9	https://card-file.ontu.edu.ua/server/api/core/bitstreams/44c16132-5f53-48e2-b6c0-61e9a2f0fd75/content	58 (5) 0.50 %
10	https://blog.csdn.net/buuliuda/article/details/145890719	54 (6) 0.46 %
11	https://card-file.ontu.edu.ua/server/api/core/bitstreams/e86ba9fc-9135-43bb-922a-10bf0bce46b5/content	44 (4) 0.38 %
12	https://www.ydma.com/article-34900-1.html	34 (3) 0.29 %
13	https://card-file.ontu.edu.ua/bitstreams/d832942b-e631-471e-b716-8e52c042ba32/download	22 (2) 0.19 %
14	https://card-file.ontu.edu.ua/bitstreams/549ee9fe-7574-4ae5-b500-9fe2711f33e6/download	19 (1) 0.16 %
15	http://econ2016.krasalex.com/wp-content/uploads/2019/03/%D0%9C%D0%BE%D0%BD%D0%BE%D0%B3%D1%80%D0%B0%D1%84.-2019-%D0%98%D0%97%D0%94%D0%90%D0%A2%D0%95%D0%9B%D0%AC.pdf	18 (2) 0.15 %
16	https://card-file.ontu.edu.ua/bitstreams/d42aac6d-ab01-4a74-b9cb-ced2a9eff719/download	17 (2) 0.15 %
17	http://dspace.wunu.edu.ua/jspui/bitstream/316497/20079/1/%D0%92%D0%B5%D1%80%D0%B3%D1%83%D0%BD%20%D0%9B.%D0%86.%20%D0%A1%D0%9A%D0%9E%D0%A0%D0%9E%D0%A7%D0%95%D0%9D%D0%9D%D0%AF%20%D0%92%20%D0%90%D0%9D%D0%93%D0%9B%D0%86%D0%99%D0%A1%D0%AC%D0%9A%D0%86%D0%99%20%D0%A2%D0%95%D0%A0%D0%9C%D0%86%D0%9D%D0%9E%D0%A1%D0%98%D0%A1%D0%A2%D0%95%D0%9C%D0%86%20%E2%80%9C%D0%9E%D0%A1%D0%92%D0%86%D0%A2%D0%90%E2%80%9D%20%D0%A2%D0%90%20%D0%87%D0%A5%20%D0%9F%D0%95%D0%A0%D0%95%D0%9A%D0%9B%D0%90%D0%94%20%D0%A3%D0%9A%D0%A0%D0%90%D0%87%D0%9D%D0%A1%D0%AC%D0%9A%D0%9E%D0%AE%20%D0%9C%D0%9E%D0%92%D0%9E%D0%AE.pdf	10 (1) 0.09 %
18	https://card-file.ontu.edu.ua/bitstreams/21173711-5b67-4b87-b17f-6302c25e7a31/download	5 (1) 0.04 %
19	https://blog.csdn.net/baishuiniyaonulia/article/details/123023666	5 (1) 0.04 %

Список прийнятих фрагментів (немає прийнятих фрагментів)

ПОРЯДКОВИЙ НОМЕР	ЗМІСТ	КІЛЬКІСТЬ ОДНАКОВИХ СЛІВ (ФРАГМЕНТІВ)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВСП «ОДЕСЬКИЙ ТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ ОНТУ»

Спеціальність: 123 «Комп'ютерна інженерія»_
Освітньо-професійна програма: **«Комп'ютерна інженерія» Група:** 2БКС- 29