КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

Кваліфікаційна робота
на здобуття ступеня бакалавра

За спеціальністю 122 Комп'ютерні науки
на тему:

РОЗРОБКА ТА ОПТИМІЗАЦІЯ C++ ПРОГРАМ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ CUDA ДЛЯ ГЕТЕРОГЕННИХ СИСТЕМ З ГРАФІЧНИМ ПРОЦЕСОРОМ

Виконав студент 4-го курсу
Віталій ДЕМЕДЮК _____
(підпис)

Науковий керівник:
асистент, кандидат фізико-математичних наук
Олексій ФЕДОРУС _____
(підпис)

Засвідчую, що в цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань

Студент _____
(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри математичної інформатики

Протокол № ____ «____» _____ 2023 р.

Завідувач кафедри
Василь ТЕРЕЩЕНКО _____
(підпис)

Київ — 2023

РЕФЕРАТ

Робота містить 69 сторінок, 13 рисунків, 16 таблиць, 20 використаних джерел, 12 додатків.

Ключові слова: CUDA, C++, GPU, ГРАФІЧНИЙ ПРОЦЕСОР, ГЕТЕРОГЕННА СИСТЕМА, ОПТИМІЗАЦІЯ, ВИСОКОПРОДУКТИВНІ ОБЧИСЛЕННЯ

Об'єктом дослідження є методи покращення швидкодії алгоритмів з різними часовими та просторовими складностями, використовуючи технологію CUDA та переваги графічних процесорів. **Мета роботи** – ознайомитись з архітектурою графічного процесора та програмно-апаратною архітектурою паралельних обчислень CUDA, порівнюючи його з фреймворком OpenCL, дослідити сучасні методи оптимізації C++ програм, використовуючи технологію CUDA. **Новизна** роботи полягає у розробці покращених по швидкодії алгоритмів, використовуючи переваги графічного процесора та сучасних мов і технологій. Порівнянні цих реалізацій, з різними підходами до оптимізацій, на сучасній гетерогенній системі. **Сфера застосування** є дуже широкою, адже графічні процесори стають все доступніші, а потреба у високопродуктивних обчисленнях з кожним роком ростуть, у зв'язку зі збільшенням об'єму даних та розвитком сфер, де необхідно моделювати складні системи. Шляхів подальшого **розвитку** роботи є багато: подальше ознайомлення з технологією CUDA та пошук інших методів для покращення швидкодії алгоритмів, порівняння реалізацій на інших гетерогенних системах з різними типами та кількостями графічних процесорів, включаючи й системи з інтегрованими відеокартами та фізично спільною пам'яттю хоста і пристроїв, використання результатів роботи для покращення швидкодії сучасного програмного забезпечення.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

EOM — електронна обчислювальна машина.

CPU — центральний процесор (англ. *Central Processing Unit*).

GPU — графічний процесор (англ. *Graphics Processing Unit*).

ALU — арифметико-логічний пристрій (англ. *Arithmetic Logic Unit*).

API — прикладний програмний інтерфейс (англ. *Application Programming Interface*).

MSVC — Microsoft Visual C++, компілятор мови програмування C++ від компанії Microsoft.

GCC — колекція компіляторів, розроблених проєктом GNU.

G++ — компонент GCC, який використовується спеціально для компіляції програм на мові C++.

LLVM — віртуальна машина низького рівня (англ. *Low Level Virtual Machine*), проєкт програмної інфраструктури для створення компіляторів і супутніх їм утиліт.

RTTI — динамічна ідентифікація типу (англ. *Run-Time Type Identificator*).

ВСТУП

Оцінка сучасного стану об'єкта дослідження або розробки. У сучасному світі зростає популярність використання гетерогенних систем з графічними процесорами для виконання обчислювальних задач. Вони виявляють великий потенціал у розробці та оптимізації програм. Зараз уже існують декілька основних технологій, що надають можливість розробки та оптимізації програм, які використовують переваги графічних процесорів. Одна з найпоширеніших – програмно-апаратна архітектура паралельних обчислень CUDA. Вона дозволяє розробникам ефективно використовувати паралельні обчислення й отримувати значні прирости продуктивності для графічних процесорів компанії NVDIA. CUDA надає програмістам можливість розробляти ефективні програми, використовуючи останні стандарти мови програмування C++, яка є одним з найпоширеніших і потужних інструментів для розробки програмного забезпечення, що дозволяє розробляти ефективні та масштабовані програми. Для цього технологія CUDA представляє спеціальний набір розширень для паралельного програмування на графічних процесорах.

Актуальність роботи та підстави для її виконання. Розвиток технології CUDA не стоїть на місці, а самі її розробники представляють нові можливості для оптимізації програм. Вони потребують дослідження їх ефективності, та порівняння з уже використовуваними технологіями, чим і займаються дослідники з високопродуктивних обчислень у всьому світі.[1] [2] [3] Також незалежно від технології програмування для графічних процесорів, розвивається й сама мова програмування C++. А підтримка її останніх стандартів технологією CUDA, потребує нових реалізацій алгоритмів, які будуть ефективніші та безпечніші при роботі з пам'яттю.[4] У зв'язку росту популярності графічних процесорів для обчислень загального призначення, розвиваються й самі апаратні можливості відеокарт. Це потребує додаткових досліджень ефективності їх використання для окремих класів задач.

Мета й завдання роботи. Ознайомитись з архітектурою графічного процесора та програмно-апаратною архітектурою паралельних обчислень CUDA. Дослідити сучасні методи розробки та оптимізації C++ програм, використовуючи переваги гетерогенних систем з графічним процесором. Для досягнення цієї мети поставлено такі завдання.

- Ознайомитись з архітектурою графічного процесора в порівнянні з центральним процесором.
- Ознайомитись з технологією CUDA, та провести порівняльний аналіз з його найпопулярнішим аналогом – OpenCL.
- Дослідити основні сучасні методи та засоби для розробки та профілювання програм, що використовують платформу CUDA та сучасні стандарти мови C++, для максимально ефективного використання переваг графічних процесорів.
- Застосувати на практиці здобуті знання для розробки нових чи уже наявних алгоритмів, використовуючи новітні стандарти мови C++. Порівняти різні підходи для покращення часу виконання на справжній сучасній гетерогенній системі.

Об'єкт, методи й засоби дослідження. Об'єктом дослідження є сучасні методи покращення швидкодії алгоритмів з різними часовими та просторовими складностями, використовуючи сучасні стандарти мови C++, технологію CUDA та переваги гетерогенних систем з графічним процесором. У роботі проводиться аналіз літературних джерел, наукових робіт, документацій та спеціалізацій технологій. Методи дослідження включають проведення аналізу та порівняння різних алгоритмів для визначення їх швидкодії та ефективності. Для цього використовуються такі засоби, як вимірювання часу виконання й профілювання програм за допомогою спеціальних утиліт, та оцінка часової та просторової складності алгоритмів.

Можливі сфери застосування. Сфера застосування є дуже широкою,

адже графічні процесори стають все доступніші, а потреба у високопродуктивних обчисленнях з кожним роком ростуть, у зв'язку зі збільшенням об'єму даних та розвитком сфер, де необхідно моделювати складні системи. Результати роботи можуть бути використані при досягненні таких цілей: порівняння сучасних гетерогенної системи для розв'язання конкретних проблем, вибір потенційного методу для оптимізації власних реалізацій алгоритмів, розробка сучасних бібліотек та застосунків, що використовують останні стандарти мови C++ та сучасні технології розробки алгоритмів для систем з графічним процесором.

РОЗДІЛ 1. ОГЛЯД ТЕХНОЛОГІЙ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ НА ГРАФІЧНИХ ПРОЦЕСОРАХ

1.1 Архутектура графічних процесорів

1.1.1 Класифікація архітектур ЕОМ за Флінні

Майклом Флінні у 1966 році була запропонована[5], а в 1977 році розширена[6] загальна класифікація ЕОМ за ознаками наявності паралелізму в потоках команд і даних. Вона являє собою всі чотири комбінації наявності чи не наявності паралелізму.

SISD (single instruction, single data) – один потік команд і даних. Це ЕОМ з відсутнім будь-яким паралелізмом. Як правило, мають на увазі, що це такі обчислювальні системи, що можуть виконувати рівно одну інструкцію в один дискретний момент часу. Наочним прикладом є звичайний комп'ютер з одноядерним процесором.

SIMD (single instruction, multiple data) – один потік команд, множинний потік даних. У таких архітектурах паралелізм полягає у виконанні однієї або послідовності одних і тих самих операцій одразу над багатьма елементами даних. На цей час її імплементація у сучасних комп'ютерах проявляється або спеціальним набором інструкцій(MMX, SSE, 3DNow! тощо) для прискорень окремих обчислень, або окремими платами розширеннями, які є представниками архітектури SIMD(наприклад, відеокарти).

MISD (multiple instruction, single data) – декілька потоків команд, один потік даних. Фахівці дотепер не прийшли до єдиної думки щодо того, які архітектури відносити до даного класу. Деякі включають у нього конвеєрні обчислювачі, в яких цілісна операція розбивається на послідовність простіших етапів з суміщенням різних етапів для різних порцій даних у часі. Таким чином, в певному кожна порція даних від моменту завантаження з пам'яті до запису результату обробки проходить через декілька етапів обробки (стадій

конвеєра). Подібні принципи покладені в основу систолічних архітектур, які також належать до цього класу.

MIMD (multiple instructions, multiple data) – декілька потоків команд і даних. Зараз це найпопулярніший клас паралелізму. Він поділяється на 3 типи: із розподіленою пам'яттю – кожен процесор має доступ лишень до своєї локальної пам'яті, а отримувати й відсилати дані він може через мережу; із загальною пам'яттю – усі процесори спільно звертаються до загальної пам'яті; зі спільною віртуальною пам'яттю – у процесора є своє локальна пам'ять, а ще він може звертатись до спільної пам'яті за допомогою глобальної адреси, якщо це не є локальною пам'яттю іншого процесора. Типовим прикладом цього класу ЕОМ є комп'ютер з багатоядерним процесором.

1.1.2 Архітектура GPU та її порівняння з CPU

Звичайний CPU спроєктовано таким чином, щоб якнайшвидше виконувати послідовний потік інструкцій з якомога меншою затримкою, зберігаючи при цьому можливість швидкого перемикання між операціями (рисунок 1). Також у сучасних багатоядерних процесорах є можливість розбивати їх на декілька паралельних потоків інструкцій, з метою у правильному порядку об'єднати результат їх виконання. Отже, за класифікацією Флінні, у CPU реалізовують SISD або MIMD, у випадку наявності одного чи декількох ядер відповідно. Також імплементуючи алгоритм, який буде виконуватись на центральному процесорі, програміст має можливість безпосередньо звертатись до будь-якої ділянки лінійної пам'яті.

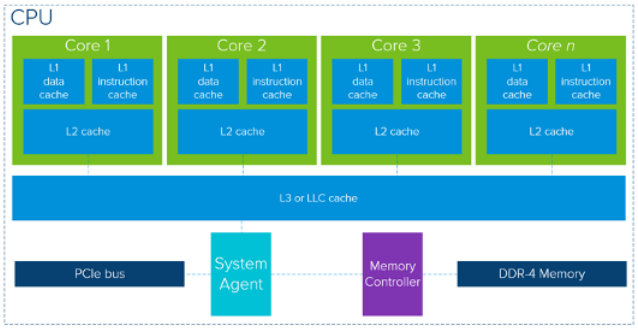


Рисунок 1 – Ілюстрація архітектури багатоядерного центрального процесора

GPU розвинувся як доповнення до свого близького брата, CPU. У той час як центральні процесори продовжують забезпечувати підвищення продуктивності завдяки архітектурним інноваціям, вищим тактовим частотам і додаванню ядер, графічні процесори спеціально розроблені для прискорення робочих навантажень комп'ютерної графіки, тому одразу проєктувався для масових паралельних обчислень. На відмінно від CPU, у якому одне або невелика кількість ядер, у GPU їх у декілька порядків більше. Але коли ми говоримо про ядра в графічному процесорі, ми маємо на увазі ALU, на відмінно від CPU, у якому ще знаходиться не один рівень кешу.

Один графічний процесор складається з кількох кластерів процесорів, які містять кілька потокових мультипроцесорів. Кожен потоковий мультипроцесор має свій рівень кешу інструкцій першого рівня із пов'язаним з ним ядрами (рисунок 2). Як правило, один мультипроцесор використовує виділений кеш першого рівня та спільний кеш другого рівня перед отриманням даних з глобальної пам'яті. На відмінно від CPU, архітектура GPU толерантна до затримки пам'яті, оскільки він працює з меншою її кількістю. Графічний процесор має більше транзисторів призначених для обчислення, тому йому не так важливо, скільки часу потрібно для отримання даних з пам'яті. Потенційна затримка маскується, доки графічний процесор має достатньо обчислень під рукою. Тому зазвичай графічні процесори підходять для високопродуктивних обчислень, які демонструють паралелізм даних для використання архітектури SIMD за Флінні.

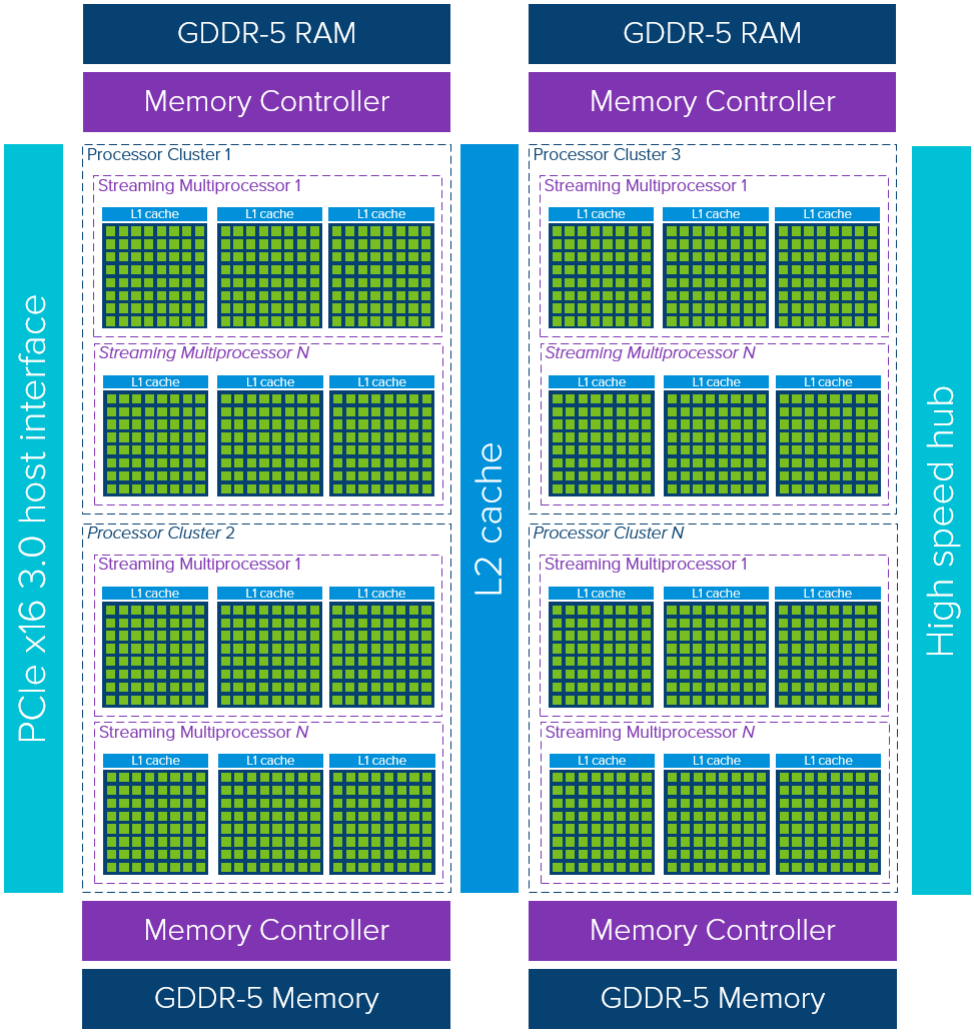


Рисунок 2 – Ілюстрація архітектури графічного процесора

1.2 Технологія паралельних обчислень OpenCL

1.2.1 Опис технології

OpenCL (англ. *Open Computing Language*) – це відкритий безплатний стандарт для кросплатформного паралельного програмування різноманітних прискорювачів, які є в суперкомп'ютерах, хмарних серверах, персональних комп'ютерах, мобільних пристроях і вбудованих платформах. OpenCL значно покращує швидкість і реакцію широкого спектра програм у численних ринкових категоріях, включаючи професійні творчі інструменти, наукове та медичне програмне забезпечення, обробку зору, а також навчання нейронних мереж[7]. За допомогою цієї технології можна розробляти програми для гетерогенних

колекцій процесорів, графічних процесорів тощо. OpenCL – це не просто мова, це ціла структура для паралельного програмування, яка містить мову, API, бібліотеки, та систему виконання для підтримки розробки програмно забезпечення. Наприклад, за допомогою цієї технології програміст може виконувати обчислення на графічних процесорах не зводячи свої алгоритми на API 3D-графіки[9]. Для опису основної ідеї OpenCL використовують ієрархію моделей: модель платформи, модель виконання, модель пам'яті.

1.2.2 Модель платформи

Модель платформи OpenCL (рисунок 3) складається з хоста, підключеного до одного або кількох пристроїв цієї платформи. Пристрій OpenCL поділяється на один або кілька обчислювальних блоків, які далі поділяються на один або більше елементів обробки. Обчислення на пристрої відбуваються в елементах обробки. В OpenCL програмі реалізовується як і код хоста, так і код ядра пристрою. Частина коду хоста програми OpenCL працює на головному процесорі відповідно до моделей, властивих хост-платформі. Код хоста програми OpenCL надсилає код ядра як команди від хоста до пристроїв OpenCL. Пристрій OpenCL виконує обчислення команд на елементах обробки всередині пристрою[9].

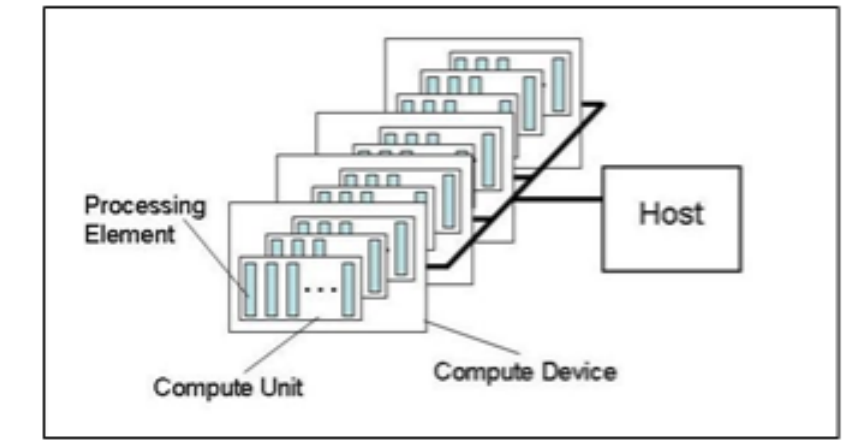


Рисунок 3 – Модель платформи OpenCL

Хоч і програми розроблені на даній технології можуть виконуватись на **гомогенних** системах, де всі компоненти мають однакову архітектуру і

призначені для виконання однакових завдань, це на практиці не дасть відчутного покращення ефективності. Модель платформи OpenCL передбачає виконання ядер та хост-програми на **гетерогенних** обчислювальних системах, що містять обчислювальні блоки різних типів та архітектур.

1.2.3 Модель виконання

Двома основними блоками виконання в OpenCL є ядра та хост-програма. Ядра виконуються на так званому пристрої OpenCL, а хост-програма виконується на головному комп'ютері. Її основна мета – створювати й запитувати атрибути платформи й пристрою, визначати контекст для ядер, створювати й керувати їх виконанням. Коли хост надсилає ядро на пристрій, створюється N-вимірний індексний простір. N дорівнює принаймні 1 і не перевищує 3. Для кожної координати цього індексного простору створюється окремий екземпляр ядра, який називається робочим елементом. На рисунку 4 показано три сценарії для 1, 2 і 3-вимірного NDRange.[8]

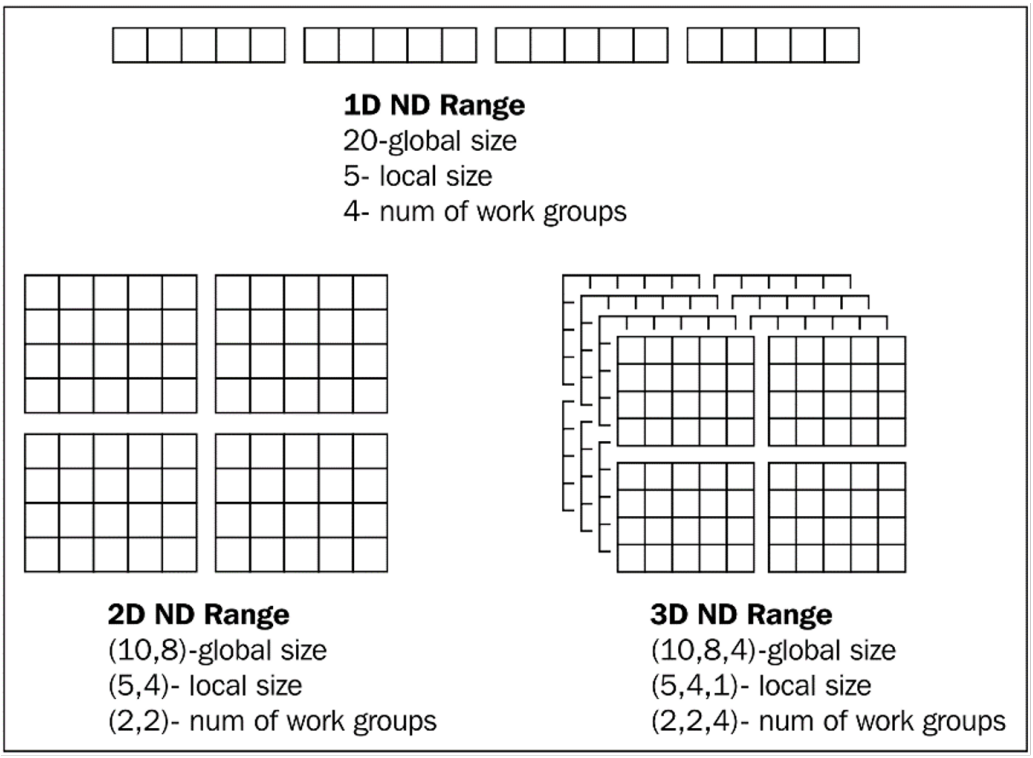


Рисунок 4 – OpenCL NDRange

Усі робочі елементи в OpenCL організовуються в робочі групи, таким

чином забезпечують грубу декомпозицію NDRange. Весь індексний простір можна визначити трьома масивами довжини N:

– обсяг простору індексу (або глобальний розмір) у кожному вимірі;
– індекси зміщення F, що вказують початкове значення індексів у кожному вимірі (нуль за замовчуванням);
– розмір робочої групи (локальний розмір) у кожному вимірі.

Глобальний ідентифікатор кожного робочого елемента можна представити у вигляді N-вимірного кортежу. На рисунку 5 зображений приклад двовимірного індексного простору, що показує робочі елементи, їхні глобальні ідентифікатори та їх відображення на пару ідентифікаторів робочої групи та локальних ідентифікаторів [9].

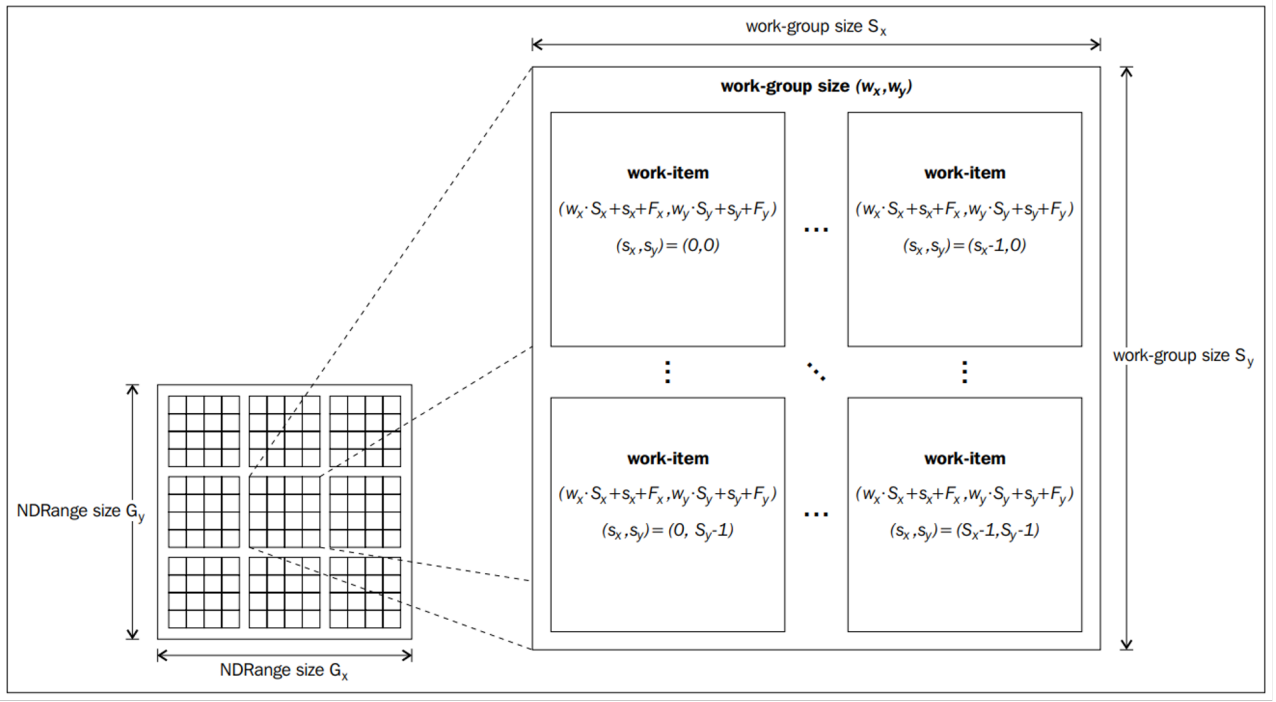


Рисунок 5 – Приклад 2D NDRange

Також хост визначає контекст виконання ядер, який містить такі ресурси:

– пристрої – набір OpenCL пристроїв, які використовує хост;
– ядра – OpenCL функції, які виконуються на пристроях;
– об'єкти програм – вихідні коди й виконувані файли ядер;

– об'єкти пам'яті – набір об'єктів пам'яті, видимого хосту й OpenCL пристрою, що містять значення, з якими працюватимуть ядра.

Контекст створюється та керується за допомогою функцій з API OpenCL. Хост створює структуру даних, яка називається чергою команд, щоб керувати виконанням ядер на пристроях. Хост відправляє команди у чергу, після чого вони встановлюються планувальником для виконання на пристроях у потрібному контексті. Команди можуть бути наступних типів:

– команда виконання ядра: виконати ядро на елементі оброки пристрою;
– команди пам'яті: перемістити дані до об'єктів пам'яті, з них або між ними;
– команди синхронізації: управління порядком виконання команд.

Черга команд планує команди для виконання на пристрої. Вони виконуються асинхронно між хостом та пристроєм. Команди можуть виконуватися один щодо одного двома способами. Перший – виконання послідовно: команди запускаються на виконання в тому порядку, в якому вони розташовані у черзі та завершуються так само послідовно. Тобто команди виконуються послідовно. Другий – непослідовне виконання: команди вирушають на виконання послідовно, але не чекають на завершення попередньої команди перед початком виконання. І тут програміст повинен явно використовувати команди синхронізації. З одним контекстом можна пов'язати кілька черг команд. Ці черги виконуються, конкуруючи між собою, незалежно без будь-яких явних способів синхронізації між ними.

1.2.4 Модель пам'яті

Пам'ять в OpenCL ділиться на два типи. Перший – пам'ять, що безпосередньо доступна хосту, та її детальна поведінка визначається за межами OpenCL. Об'єкти пам'яті переміщаються між хостом і пристроями через функції API, або через спільний інтерфейс віртуальної пам'яті. Друга – пам'ять пристроїв, яка безпосередньо доступна виконуваним ядрам на OpenCL

пристроях. Сама пам'ять пристрою складається з чотирьох іменованих областей пам'яті:

- **Глобальна пам'ять.** Ця область пам'яті надає доступ для читання та запису до всіх робочих елементів у всіх робочих групах, що працюють на будь-якому пристрої в контексті. Робочі елементи можуть читати або записувати в будь-який елемент об'єкта пам'яті. Читання та запис у глобальній пам'яті можуть кешуватися залежно від можливостей пристрою.[9]
- **Константна пам'ять.** Область глобальної пам'яті, яка залишається постійною під час виконання екземпляра ядра. Хост виділяє та ініціалізує об'єкти пам'яті, розміщені в постійній пам'яті.[9]
- **Локальна пам'ять.** Область пам'яті, локальна для робочої групи. Цю область пам'яті можна використовувати для розподілу змінних, які є спільними для всіх робочих елементів у цій робочій групі.[9]
- **Приватна пам'ять.** Область пам'яті, приватна для робочого елемента. Змінні, визначені в приватній пам'яті одного робочого елемента, не видимі для іншого робочого елемента.[9]

1.3 Технологія паралельних обчислень CUDA

1.3.1 Опис технології

CUDA – це універсальна обчислювальна платформа та модель програмування, яка використовує механізм паралельних обчислень у графічних процесорах NVIDIA для більш ефективного вирішення багатьох складних обчислювальних проблем. Використовуючи CUDA, розробник може отримати доступ до GPU для обчислень, як це традиційно робиться на CPU. За допомогою цієї технології програміст може створювати застосунки для безліч систем з графічними процесорами NVIDIA, починаючи від вбудованих пристроїв, планшетів, ноутбуків, настільних ПК і робочих станцій, до кластерних систем високопродуктивних обчислень. Знайомі програмні засоби

програмування на C та C++ було розширено, щоб допомогти редагувати, налагоджувати та аналізувати програми CUDA протягом життєвого циклу проєкту. Дана платформа доступна через прискорені бібліотеки CUDA, директиви компілятора, інтерфейси програмування додатків і розширення стандартних мов програмування, включаючи C, C++, Fortran і Python.[10]

Модель програмування CUDA передбачає виконання програм на **гетерогенних обчислювальних системах**. Це означає, що потоки CUDA виконуються на фізично окремому пристрої, який працює як співпроцесор хосту, на якому виконується C++ програма. У найбільш поширеному випадку код ядра виконується на графічному процесорі, а решта C++ програми виконується на CPU. Модель програмування CUDA також передбачає, що і хост, і пристрій підтримують свої власні окремі області пам'яті, які називаються пам'яттю хоста і пам'яттю пристрою відповідно. Таким чином, програма керує глобальною, константною та текстурною пам'яттю графічного процесора за допомогою викликів середовища виконання CUDA. Це включає виділення та звільнення пам'яті пристрою, а також передачу даних між пам'яттю хоста та пам'яттю пристрою.[11]

Також потрібно зазначити, що загальні характеристики та особливості обчислювального пристрою залежать від його *обчислювальних здатностей* (англ. compute capability). Вони представлені номером версії, яка визначає функції, які підтримуються апаратним забезпеченням графічного процесора, і використовуються програмами під час виконання, щоб визначити, які апаратні функції та інструкції доступні на поточному графічному процесорі. Зараз версія 9.0 є найновішою. Версію обчислювальних можливостей конкретного GPU не слід плутати з версією CUDA, яка є версією саме програмної платформи. Платформа CUDA використовується розробниками додатків для створення додатків, які працюють на багатьох поколіннях архітектур GPU, включаючи майбутні архітектури GPU, які ще не винайдені. У той час як нові версії платформи CUDA часто додають власну підтримку нової архітектури GPU,

підтримуючи версію цієї архітектури з обчислювальними можливостями, нові версії платформи CUDA зазвичай також включають функції програмного забезпечення, які не залежать від покоління апаратного забезпечення.[11]

1.3.2 Ієрархія потоків

Коли функція ядра запускається зі сторони хоста, виконання переміщується на пристрій, де генерується велика кількість потоків, і кожен потік виконує оператори, визначені функцією ядра. Знання того, як організувати потоки, є важливою частиною програмування CUDA. Дана технологія надає абстракцію ієрархії потоків, щоб розробники могли впорядковувати їх. Це дворівнева ієрархія потоків, що складається з блоків потоків, які зі свого боку формують сітку блоків (рисунок 6).[10]

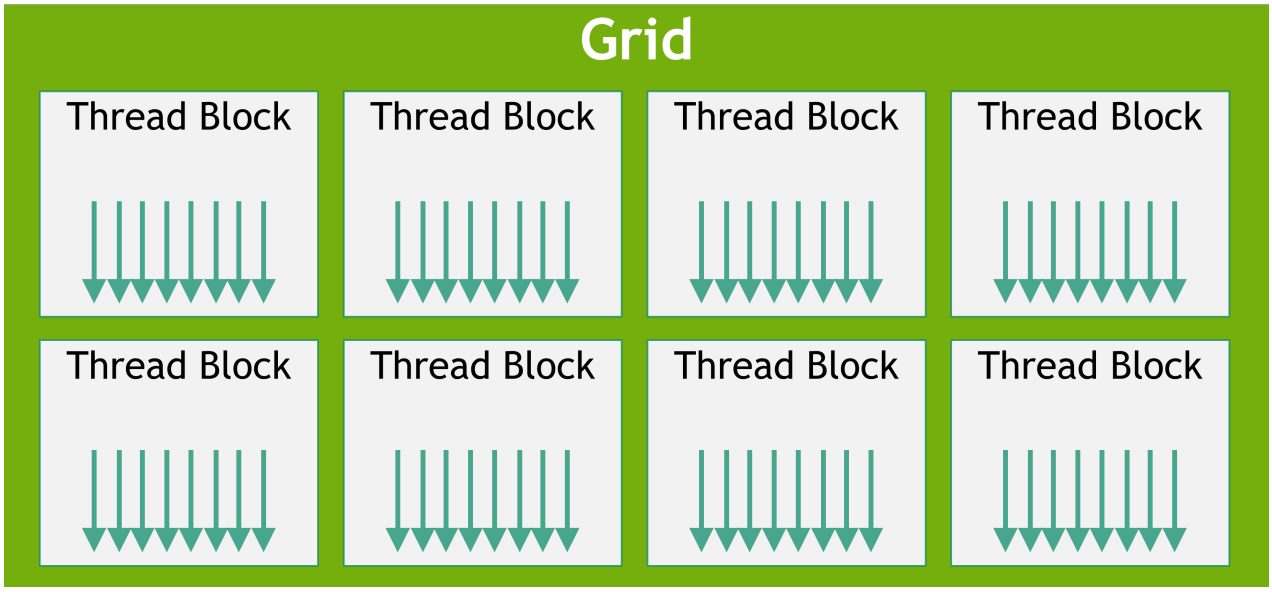


Рисунок 6 – Ілюстрація сітки блоків потоків в CUDA

Потоки покладаються на такі дві унікальні координати, щоб відрізнити їх один від одного: *blockIdx* (індекс блоку в сітці), *threadIdx* (індекс потоку в блоці). CUDA організовує сітки та блоки в трьох вимірах, тому *blockIdx* та *threadIdx* являє собою вектор з трьох компонент x, y та z. Якщо потрібно сітку чи блок меншої розмірності, то розмір невикористовуваних осей ініціалізується

одиницею, а відповідні компоненти векторів індексу блоку та індексу потоку завжди ініціалізуються нулем та ігнорується.

З появою *обчислювальної здатності* 9.0, модель програмування CUDA представляє додатковий рівень ієрархії – кластери, який складається з блоків потоків (рисунок 7). Подібно до того, як потоки в блоці гарантовано спільно плануються на потоковому мультипроцесорі, блоки потоків у кластері також гарантовано спільно плануються в кластері. Подібно до блоків потоків, кластери також мають одновимірну, двовимірну або тривимірну організацію. Кількість блоків потоків у кластері може бути визначена користувачем, але максимум 8 блоків потоків у кластері підтримується як портативний розмір кластера у CUDA.[11]

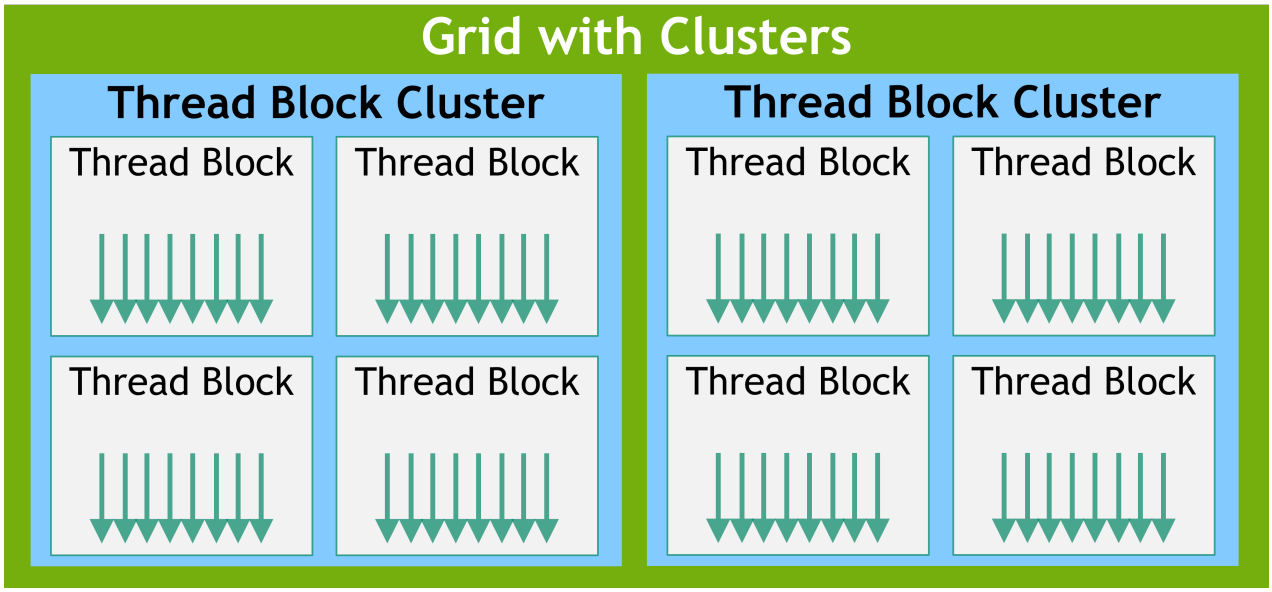


Рисунок 7 – Ілюстрація сітки кластера в CUDA

1.3.3 Ієрархія пам'яті

Потоки CUDA можуть отримувати доступ до даних з декількох областей пам'яті під час виконання. Кожен потік має приватну локальну пам'ять. Кожен блок потоків має спільну пам'ять, видиму для всіх потоків блоку, і з тим же часом існування, що і блок. Блоки потоків у кластері блоків потоків можуть

виконувати операції читання, запису та атомізації у спільній пам'яті один одного. Всі потоки мають доступ до однієї глобальної пам'яті (рисунок 8). [11]

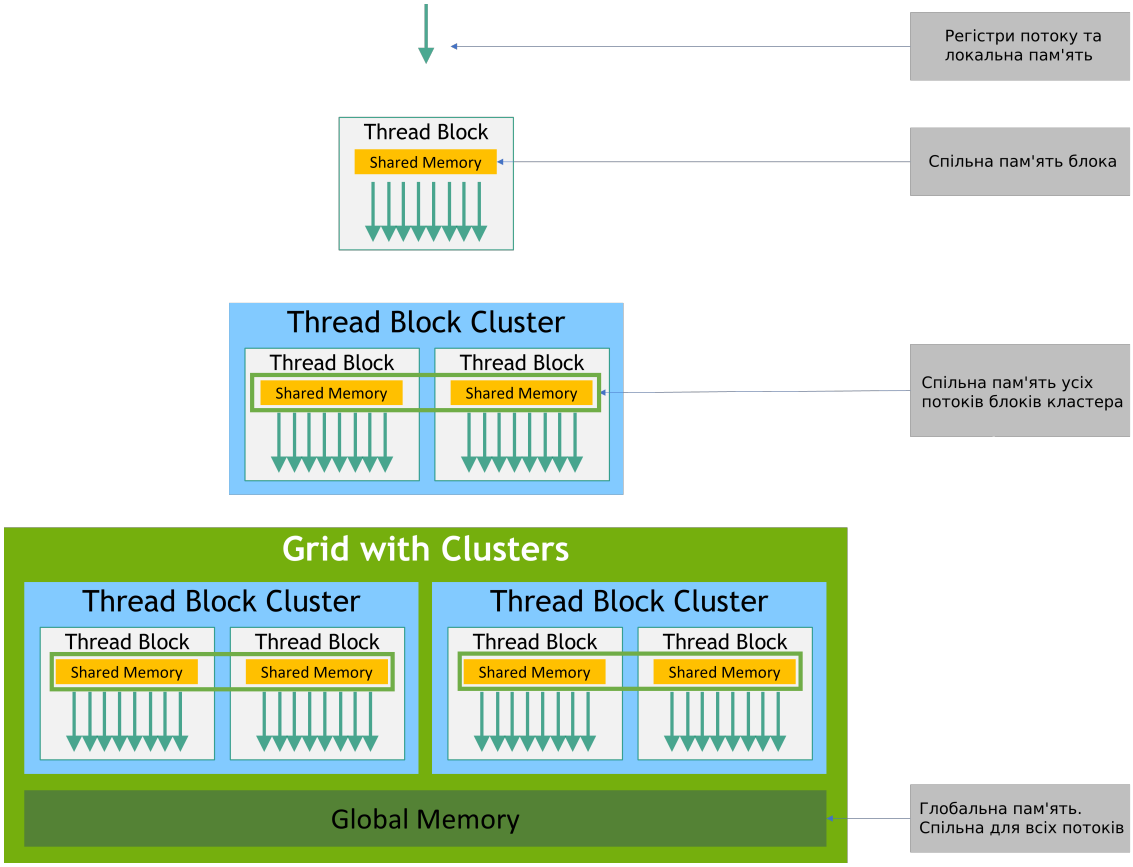


Рисунок 8 – Ієрархія пам'яті в технології CUDA

Існує також два додаткових простори пам'яті, доступних лише для читання, до яких мають доступ усі потоки: простір константної та простір текстурної пам'яті. Константна пам'ять призначений для зберігання даних, які залишаються незмінними протягом тривалого часу в процесі виконання програми на відеокарті. Вона має високу пропускну здатність та низьку затримку доступу. Текстурна пам'ять також є спеціальним типом пам'яті, який використовується для ефективного доступу до двовимірних або тривимірних даних. Вона пропонує особливий механізм кешування та інтерполяції, який дозволяє швидко отримувати дані з текстурної пам'яті за допомогою координат. Текстурна пам'ять часто використовується у задачах обробки зображень та комп'ютерного зору, де потрібний доступ до сусідніх пікселів або використання складних фільтрів.[11]

1.4 Порівняльний аналіз технологій OpenCl та CUDA

Порівняння моделей платформи. Обидві технології мають практично однакові моделі платформи. OpenCL та CUDA передбачають, що код ядра виконується на фізично окремому пристрої, який працює як співпроцесор для хоста, на якому виконується основна програма. Також з самого початку дані технології припускали, що хост і пристрої матимуть окремі області пам'яті, що накладає розходи на виділення та вивільнення пам'яті пристрою, а також передачу даних між хостом і пам'яттю пристрою.

Порівняння моделей виконання. Обидві технології забезпечують схожу ієрархічну декомпозицію простору індексів обчислень, показану в таблиці 1.

Таблиця 1 – Відображення термінологій моделі виконання.

OpenCL	**CUDA**
Grid	ND Range
Thread block	Work group
Thread	Work item
Thread index	Global ID
Block index	Block ID

Причому синхронізація доступна тільки на рівні блоку потоків в CUDA та робочої групи в OpenCL. Але потрібно зазначити, що в останній версії *обчислювальної здатності* 9.0 CUDA запровадили опціональний рівень ієрархії – кластери, які складаються з блоків потоків. Що дає можливість синхронізувати потоки у межах усіх блоків кластера. Завдяки цьому можна реалізовувати складніші та ефективніші обчислення на графічних процесорах.

Порівняння моделей пам'яті. Технологія CUDA, ідентично до OpenCL, підтримує трьох рівневу ієрархію пам'яті. Перший рівень – область пам'яті окремого потоку у випадку технології CUDA чи, відповідно, робочому елементу в OpenCL. Другий рівень – пам'ять доступна усім потокам блока у випадку

CUDA, та усім робочим елементам робочої групи у випадку OpenCL. І третій та останній рівень – глобальна пам'ять, яка доступна на читання та запис усім потокам(робочим елементам). Також обидві технології підтримують константну область пам'яті, яка доступна тільки на читання під час виконання екземпляра ядра, і тільки хост може ініціалізувати об'єкти у ній. Використання даної області пам'яті дає приріст ефективності виконання програм, та унеможливлює деякі типи помилок при реалізації алгоритмів.

Обов'язково потрібно зауважити, як зазначалося вище, в останній версії *обчислювальної здатності* 9.0 CUDA запроваджено опціональний рівень ієрархії – кластери, який дає не тільки можливість синхронізувати виконання ядер усіх блоків кластера, але й доступ до спільної пам'яті усіх блоків. Також у технології компанії NVIDIA є ще один простір пам'яті, яка доступна тільки на читання – простір текстурної пам'яті, яка використовується для оптимізації доступу до даних при роботі з графічними обчислювальними задачами.

Підтримувані платформи та пристрої. CUDA підтримує тільки графічні процесори виробництва NVIDIA, що може бути обмеженням при виборі обладнання для вашого проекту. OpenCL, з іншого боку, підтримує широкий спектр обчислювальних пристроїв, включаючи GPU від різних виробників, процесори та інші спеціалізовані пристрої.

Простота використання. CUDA надає зручний інтерфейс програмування, спеціально призначений для графічних процесорів NVIDIA. Це робить його простим у використанні для розробників, які працюють з цим обладнанням. Через те, що OpenCL надає більш загальний інтерфейс, це досить часто ускладнює розробку алгоритмів на даній технології.

Порівняння продуктивності та ефективності. Оскільки CUDA розроблений спеціально для графічних процесорів NVIDIA, він може забезпечити кращу продуктивність для цього обладнання. CUDA має прямий доступ до апаратних ресурсів графічного процесора та оптимізовані драйвери, що дозволяє досягти високої продуктивності при обчисленнях на GPU. Дана теза є не

просто теоретичним судженням, а підтвердженим на практиці результатом у багатьох дослідженнях. У дослідженні [12] конкретної реальної програми CUDA показав стабільно кращий результат продуктивності виконання коду ядра, ніж OpenCL, попри те, що дві реалізації виконують майже ідентичний код. На їхніх тестах продуктивність ядра OpenCL повільніша приблизно на 13% – 63%, в залежності від розміру вхідних даних. У 2011 році на міжнародній конференції з паралельної обробки даних були представлені результати комплексного порівняння технологій CUDA та OpenCL [13] на 16 тестах, які містять синтетичні й реальні програми. Їхні результати показують, що для більшості програм CUDA працює щонайбільше на 30% краще, ніж OpenCL. Схожий результат переваги в продуктивності було отримано в дослідженні [14], де порівняння відбувалось на класичних криптографічних алгоритмах. Отже, CUDA здається кращим вибором, якщо потрібно досягнути максимальної ефективності алгоритму при її реалізації. Але якщо ми розробляємо програму, яка має запускатись на відмінних від графічних процесорів NVIDIA пристроях, то OpenCL залишається єдиним вибором з цих двох технологій.

Спираючись на цей порівняльний аналіз технологій OpenCL та CUDA, для реалізації алгоритмів в наступних розділах було вибрано технологію CUDA, в силу високої продуктивності, легкості розробки програм та великого розповсюдження графічних процесорів компанії NVIDIA.

РОЗДІЛ 2. РОЗШИРЕННЯ МОВИ ПРОГРАМУВАННЯ CUDA C++

У цьому розділі розглядаються основні доповнення та обмеження CUDA C++ відносно стандартної мови програмування C++, методи збірки програм, що використовують це розширення, які ми будемо використовувати у подальшій реалізації наших алгоритмів.

2.1 Розширення мови C++

Як зазначалось раніше(див. 1.3), CUDA – обчислювальна платформа та модель програмування, що надає можливість використовувати графічні процесори для обчислень загального призначення. Компанією NVIDIA було розроблено API для керування GPU та CUDA C++ – розширення мови C++ для програмування графічних процесорів. Оскільки розробка CUDA застосунків передбачає виконання їх на гетерогенних системах з CPU та GPU, код програми розділений на дві обов'язкові частини:

- код хоста(*англ. host code*) – код, який запускається на центральному процесорі, керує передачею даних між ним і графічним процесором, та координує виконання коду пристрою;
- код пристрою(*англ. device code*) – код, який відповідає за виконання паралельних обчислень на GPU.

Для даних потреб мову C++ було розширено спеціальними специфікатори простору виконання функції: $\_\_global\_\_$, $\_\_device\_\_$ і $\_\_host\_\_$. Вони визначають, де виконується функція(на хості, чи на пристрої), та чи можна її викликати з хосту і з пристрою. CUDA C++ дозволяє програмісту визначати функції C++, що називаються **ядрами**(*англ. kernel*), які при виклику виконуються K разів паралельно K різними потоками CUDA, а не лише один раз, як звичайні функції C++. Для визначення цих функцій використовується спеціальний специфікатор виконання $\_\_\boldsymbol{global}\_\_$. Ядро може бути викликаним з коду хоста за допомогою спеціального синтаксису.

Але для пристроїв *обчислювальної здатності 5.0* і вище функції з даним специфікатором можна викликати з пристрою у рамках динамічного паралелізму. Для виклику ядра з коду хоста використовується новий синтаксис конфігурації виконання <<< ... >>>, завдяки якому можна задати розмір блоку(кількість ядер у кожному блоці та формат їх розміщення), розмір сітки(кількість блоків та формат їх розміщення), розмір спільної пам'яті блоку та потік(англ. *stream*) виконання. Останні два параметри не є обов'язковими, та мають значення за замовчуванням. Кількість потоків у блоці та кількість блоків у сітці може бути типу *int* у випадку одновимірного блоку чи сітки, та *dim*3 у випадку одно-, дво-, тривимірного блоку чи сітки. Кожен потік у сітці може бути ідентифікований одно-, дво- або тривимірним унікальним індексом, доступним у ядрі через вбудовану змінну $threadIdx$. Розмірність блоку потоку доступна в ядрі через вбудовану змінну $threadDim$. Аналогічно, кожен блок у сітці може бути ідентифікований одно-, дво- або тривимірним унікальним індексом, доступним у ядрі через вбудовану змінну $blockIdx$. Розмірність блоку потоку доступна в ядрі через вбудовану змінну $blockDim$.[11] Приклад коду ядра та його виклику з коду хоста для задачі додавання двох матриць:

```
__global__ void addMatrix(double [N][N], double mx2[N][N], double res[N][N]])
{
    const size_t i = blockIdx.x * blockDim.x + threadIdx.x;
    const size_t j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    dim3 blockSize(16, 16);
    dim3 gridSize(N / threadsPerBlock.x, N / threadsPerBlock.y);
    addMatrix<<<blockSize, gridSize>>>(mx1, mx2, res);
    ...
}
```

Функція зі специфікатором виконання ***global*** повинна мати тип повернення *void*, і не може бути методом класу. Також при їх виклику завжди повинна бути вказана її конфігурацію виконання. Важливою деталлю є те, що виклик ядра є асинхронним, тобто хост не буде очікувати кінця виконання цієї функції на пристрої. Специфікатор простору виконання ***device*** оголошує функцію, яка виконується на пристрої й викликається тільки з нього. Специфікатор простору виконання ***host*** оголошує функцію, яка виконується на хості й викликається тільки з нього. Оголошення функції з одним лише специфікатором простору виконання *host* або без специфікаторів еквівалентні. В обох випадках функція буде скомпільована лише для хоста. Специфікатори *global* та *host* не можна використовувати разом. Однак, специфікатори простору виконання *device* та *host* можна використовувати разом, у цьому випадку функція буде скомпільована як для хоста, так і для пристрою.[11]

У CUDA C++ при оголошенні змінних можна використовувати специфікатори простору пам'яті для вказівки їх місця зберігання. Специфікатор області пам'яті ***device*** оголошує змінну, яка знаходиться на пристрої. Він може бути використаний разом з іншими специфікаторами доступу, такими як *constant*, *shared* та. Якщо жодного з них немає, змінна: перебуває у глобальному просторі пам'яті; «живе» впродовж контексту CUDA, у якому її було створено; має окремий об'єкт для кожного пристрою. Цей специфікатор вказує, що змінна зберігається в глобальній пам'яті пристрою. Вона доступна лише на ньому й не може бути прочитана або записана безпосередньо з хоста.[11]

Специфікатор простору пам'яті ***constant***, який необов'язково використовується разом з *device*, оголошує змінну, яка: знаходиться у константній області пам'яті; «живе» впродовж контексту CUDA, у якому її було створено; має окремий об'єкт для кожного пристрою. Цей специфікатор вказує, що змінна зберігається в постійній пам'яті пристрою. Вона доступні для читання як на пристрої, так і на хості. Значення цих змінних не може бути змінено на

пристрої під час виконання ядра.[11]

Специфікатор простору пам'яті ***__shared__***, який необов'язково використовується разом з *__device__*, оголошує змінну, яка: знаходиться у спільному просторі пам'яті блоку потоку; «живе» впродовж існування блоку; має окремий об'єкт для кожного блоку; доступна лише з усіх потоків у блоці; не має постійної адреси.[11] Розподілена пам'ять є обмеженим, але дуже швидким джерелом пам'яті, яке доступне для кожного блоку всередині графічного процесора. Розмір розподіленої пам'яті на кожному блоку обмежений апаратними можливостями конкретного графічного процесора, і зазвичай цей обсяг складається з кількох кілобайтів. Однак, доступ до розподіленої пам'яті є набагато швидшим, ніж доступ до глобальної пам'яті, що знаходиться на графічній карті.[15] Приклад використання спільної для блоку пам'яті, коли заздалегідь відомо її розмір:

```
__global__ void kernel()
{
    // Declaring a shared variable
    __shared__ double sharedData[256];
    // Code that uses shared memory ...
}
```

У CUDA також існує можливість динамічного виділення розподіленої пам'яті, що дозволяє програмістам гнучко управляти розміром розподіленої пам'яті під час виконання програми. Це особливо корисно, коли розмір даних, які потрібно обробити, варіюється в залежності від конкретних умов.Для динамічного виділення розподіленої пам'яті використовується ключове слово *extern* перед специфікатором *__shared__*. А розмір пам'яті у байтах, який потрібно динамічно виділити для кожного блоку, програміст може передати через третій параметр конфігурації виконання ядра.[15] Приклад:

```
__global__ void kernel()
{
    // Declaring a shared variable
    extern __shared__ double sharedData[];
```

```
    // Code that uses shared memory ...
}

int main()
{
    ...
    const size_t dataSize = 256;
    const size_t sharedMemorySize = dataSize * sizeof(double);
    kernel<<<gridSize, blockSize, sharedMemorySize>>>(data, dataSize);
    ...
}
```

Специфікатор простору пам'яті ***__managed__***, який необов'язково використовується разом з *__device__*, оголошує змінну, на яку можна посилатися як з коду пристрою, так і з коду хоста. Тобто можна отримати її адресу або прочитати чи записати її безпосередньо з функції пристрою та функції хоста. Змінна позначена цим специфікатором має час «життя» програми.[11]

Для координації взаємодії між потоками одного блоку використовується функція ***void __syncthreads()***. Коли декілька потоків у блоці звертаються до одних і тих самих адрес у спільній або глобальній пам'яті, існує потенційна небезпека читання після запису, запису після читання або запису після запису для деяких з цих звернень до пам'яті. Цих небезпек для даних можна уникнути, синхронізуючи потоки між цими зверненнями.

Використання *__syncthreads* дозволено в умовному коді, але тільки якщо умова обчислюється однаково у всьому блоці потоків, інакше виконання коду може зависнути або призвести до небажаних побічних ефектів. Пристрої з *обчислювальної здатності* $2.x$ і вище підтримують три варіації *__syncthreads*:

- ***int __syncthreads_count(int predicate)*** – ідентична функції *__syncthreads* з додатковою особливістю, що вона обчислює предикат для всіх потоків блоку, і повертає кількість потоків, для яких предикат має значення відмінне від нуля;

– ***int __syncthreads_and(int predicate)*** – ідентична *__syncthreads* з додатковою особливістю, що вона обчислює предикат для всіх потоків блоку, і повертає ненульове значення тоді й тільки тоді, коли предикат не дорівнює нулю для всіх потоків;
– ***int __syncthreads_or(int predicate)*** – ідентична *__syncthreads* з додатковою особливістю, що вона обчислює предикат для всіх потоків блоку, і повертає ненульове значення тоді й тільки тоді, коли предикат не дорівнює нулю для будь-якого з них.

2.2 Збірка CUDA C++ програм

Вихідні файли для CUDA додатків складаються з суміші звичайного коду хоста на C++ та коду пристрою GPU. Процес компіляції включає виокремлення коду хоста та коду пристрою, компіляцію функцій пристрою за допомогою власного компілятора NVIDIA у PTX або двійковий(*cubin*) код, компіляцію коду хоста за допомогою хост-компілятора C++, та вбудовування скомпільованих функцій GPU в об'єктні хост-файли. На етапі компонування додаються бібліотеки часу виконання CUDA, для підтримки віддаленого виклику процедур SIMD та маніпуляцій з GPU.

Компілятор NVCC, який є драйвером компілятора CUDA, приховує складні деталі збірки програми від розробників, приймаючи звичайні опції компілятора, та керуючи усім процесом. Частину компіляції, що не стосується CUDA, виконує хост-компілятор C++, який підтримується NVCC, а його опції транслюються у відповідні опції хост-компілятора. NVCC використовує встановлений за замовчуванням хост-компілятор, але можна вказати інший за допомогою відповідних опцій. На операційні системі Linux за замовчуванням використовуються GCC і G++, а на Windows – MSVC.[16]

Як зазначалось раніше, код пристрою може бути скомпільований у PTX(англ. *Parallel Thread Execution*) або двійковий(*cubin*) код. PTX надає стабільну модель програмування та набір інструкцій для паралельного

програмування загального призначення. Вона забезпечує стабільний машинно-незалежний набір інструкцій архітектури, який охоплює декілька поколінь графічних процесорів. Будь-який PTX код, завантажений програмою під час виконання, компілюється драйвером пристрою у двійковий код. Такий підхід називається JIT-компіляцією(англ. *Just-in-time compilation*). Вона збільшує час завантаження програми, але дозволяє їх отримувати вигоду від нових покращень компілятора, які з'являються з кожним новим драйвером пристрою. Крім того, це єдиний спосіб запуску програм на пристроях, які не існували на момент компіляції програми. Коли драйвер пристрою вперше збирає певний PTX код для програми, він автоматично кешує копію згенерованого двійкового коду, щоб уникнути повторення компіляції під час наступних викликів програми. Кеш автоматично анулюється при оновленні драйвера пристрою, щоб програми могли скористатися покращеннями нового JIT-компілятора вбудованого у драйвер пристрою.[17]

У деяких пристроях підтримка певних команд PTX залежить від їх версії *обчислювальної здатності*. PTX-код, створений для певної її версії, завжди можна скомпілювати в бінарний код з новою або такою ж *обчислювальною здатністю*. Проте, варто відзначити, що бінарний код, скомпільований з попередньої версії PTX, може не використовувати всі доступні апаратні можливості. Наприклад, бінарний код, створений для пристроїв з *обчислювальною здатністю 7.0* (Volta) з PTX, створеного для *обчислювальної здатності 6.0* (Pascal), не буде використовувати інструкції Tensor Core, які не були доступні у Pascal. Це може призвести до меншої ефективності роботи фінального бінарного файлу, порівняно з використанням останньої версії PTX.[17]

Також вихідний код пристрою може одразу бути скомпільованим у нативний двійковий код, також відомий як *cubin*. Це файл у форматі ELF(англ. *Executable and Linkable Format*), який складається з секцій виконуваного коду CUDA, а також інших секцій, що містять символи, релокатори, налагоджувальну

інформацію тощо. Двійковий код є архітектурно-специфічним, а його сумісність гарантується від одного малого релізу *обчислювальної здатності* до наступного, але не від одного малого релізу до попередньої або між великими релізами. Тобто згенерований для *обчислювальної здатності* $X.y$ код виконуватиметься лише на пристроях з *обчислювальною потужністю* $X.z$, де $z \geq y$.[18]

Різниця між форматами cubin і PTX полягає у рівні абстракції, який вони надають. PTX є проміжним представленням програми, що може бути використане для оптимізації та генерації бінарного коду для конкретної архітектури GPU. З іншого боку, cubin містить вже скомпільований машинний код, який може бути безпосередньо виконаний на підтримуваній архітектурі. Таким чином, PTX надає більшу гнучкість, оскільки може бути перетворений у бінарний код для різних архітектур GPU, а cubin використовується, коли потрібно конкретно виконувати код на підтримуваній архітектурі без необхідності в додатковій оптимізації та генерації бінарного коду.

2.3 Підтримка та обмеження мови C++

Фронтальна частина компілятора відповідає за обробку вихідних файлів CUDA, згідно з правилами синтаксису мови C++. Повний набір можливостей C++ підтримується для коду, що виконується на хості. Однак, для коду, що виконується на пристрої, повністю підтримується лише підмножина цієї мови програмування.

Як було описано у підрозділі 2.2, вихідні файли CUDA, скомпільовані за допомогою NVCC, можуть містити суміш коду для хоста та коду для пристрою. Зовнішній компілятор CUDA прагне емулювати поведінку компілятора для хоста щодо вхідного коду C++. Вхідний код обробляється відповідно до специфікацій C++ ISO/IEC 14882:2003, C++ ISO/IEC 14882:2011, C++ ISO/IEC 14882:2014, C++ ISO/IEC 14882:2017 та C++ ISO/IEC 14882:20120, і намагається емулювати будь-які відхилення хост-компілятора від специфікацій ISO. Крім того, мову розширено специфічними для CUDA конструкціями, які розглянуто у підрозділі

2.1, і на них поширюються низка обмеження. Розглянемо деякі з них, з якими розробники CUDA-програм найчастіше зіштовхуються при реалізації своїх алгоритмів.

- Стандартні бібліотеки підтримуються лише в коді хоста, але не в коді пристрою, якщо це не передбачено спеціальними вимогами.
- Обробка винятків підтримується лише у коді хоста, але не у коді пристрою. Специфікація винятків не підтримується для *__global__* функцій.
- Тип або шаблон не можна використовувати аргументом типу або шаблону шаблону екземпляра шаблону функції *__global__* або екземпляра змінної *__device__*/*__constant__*, якщо вони: визначені в межах *__host__* або *__host__* *__device__*; є членом класу з доступом private або protected, а його батьківський клас не визначений у функції *__device__* або *__global__*; не має імені; є складеним з будь-якого з перерахованих вище типів.
- Використання типу *long double* не підтримується у коді пристрою.
- Функції, пов'язані з RTTI, підтримуються тільки у коді хоста.
- Статичні поля даних класу не підтримуються, за винятком тих, що є константними.
- Статичні методи у класах не можуть бути позначенні специфікатором простору виконання *__global__*.
- Коли функція у похідному класі перевизначає віртуальну функцію у базовому класі, специфікатори простору виконання (*__host__*, *__device__*) у них повинні збігатися. Не допускається передавати аргументом функції зі специфікатором *__global__* об'єкт класу з віртуальними функціями. Якщо об'єкт створено у коді хосту, то виклик віртуальної функції для цього об'єкту у коді пристрою має невизначену поведінку. Якщо об'єкт створено у коді пристрою, виклик віртуальної функції для цього об'єкта у коді хоста має невизначену поведінку.

- Функції $\_\_global\_\_$ не можуть мати змінну кількість аргументів.
- Параметри $\_\_global\_\_$ функції не можуть передаватися за посиланням.
- Адреса функції $\_\_global\_\_$, взята у коді хоста, не може бути використана у коді пристрою (наприклад, для запуску ядра). Аналогічно, адреса функції $\_\_global\_\_$, взята у коді пристрою, не може бути використана у коді хоста. Не можна брати адресу функції $\_\_device\_\_$ у коді хоста.
- Функції зі специфікатором простору виконання $\_\_global\_\_$ не підтримують рекурсивні виклики.
- Не можна визначати дружньою $\_\_global\_\_$ функцію або шаблон функції для класу.

Повну інформацію про підтримку та обмеження мови C++ можна знайти в офіційному посібнику з програмування CUDA C++ від компанії NVIDIA.[11]

2.4 Робота з глобальною пам'яттю GPU та оптимізація передачі даних між хостом та пристроєм

Гетерогенна модель програмування CUDA передбачає, що і хост, і пристрій підтримують свої власні окремі області пам'яті. Таким чином, програма керує глобальною, константною та текстурною пам'яттю за допомогою викликів середовища виконання CUDA. Це включає виділення та звільнення пам'яті пристрою, а також передачу даних між пам'яттю хоста та пам'яттю пристрою. Глобальна пам'ять CUDA – це основна пам'ять, яка використовується в програмах, розроблених для виконання на графічних процесорах з підтримкою CUDA. Це область пам'яті, яка доступна всім потокам, що працюють на графічному процесорі. Глобальна пам'ять використовується для зберігання даних, які можуть бути доступні з будь-якого потоку. Однією з особливостей глобальної пам'яті CUDA є її швидкий доступ з боку керуючого процесора та потоків. Проте, доступ до глобальної пам'яті може бути повільним порівняно зі швидкодією локальної пам'яті. Тому розробнику потрібно уважно планувати доступ до глобальної пам'яті, щоб забезпечити оптимальну продуктивність

програми.

Алокація та управління глобальною пам'яттю є важливими аспектами програмування на CUDA C++. Даний фреймворк надає спеціальні функції та механізми для роботи з пам'яттю на пристрої. Для алокації пам'яті на пристрої використовується функція cudaMalloc з наступною сигнатурою:

```
__host__ __device__ cudaError_t cudaMalloc (void** devPtr, size_t size);
```

Наприклад, для алокації пам'яті розміром 256 елементів типу int, ми можемо використовувати наступний код:

```
int *deviceArray;
cudaMalloc(&deviceArray, 256 * sizeof(int));
```

Для копіювання даних між хостом та графічним процесором, CUDA надає функції *cudaMemcpy* з наступною сигнатурою:

```
__host__ cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
    cudaMemcpyKind kind);
```

Наприклад, для копіювання даних з хоста на пристрій, ми можемо використовувати такий код:

```
int *hostArray;
int *deviceArray;
cudaMalloc(&deviceArray, 256 * sizeof(int));
hostArray = (int*)malloc(256 * sizeof(int));
// Filling the hostArray with data
cudaMemcpy(deviceArray, hostArray, 256 * sizeof(int), cudaMemcpyHostToDevice);
```

Копіювання даних пристрою на хост відбувається аналогічним чином, тільки з *cudaMemcpyDeviceToHost* значенням *cudaMemcpyKind* параметра.

Після використання фрагмента пам'яті на пристрої, його треба звільнити, щоб уникнути витоку пам'яті. Для цього використовується функція *cudaFree* з наступною сигнатурою

```
__host__ __device__ cudaError_t cudaFree ( void* devPtr )
```

Коли ми вирішуємо, чи використовувати класичну версію програми, яка для обчислень використовує тільки CPU, чи яка використовує ядра графічного

процесора, ми повинні робити заміри не тільки часу виконання ядра на GPU, а й часу переміщення інформації. Досить часто передача даних може домінувати у загальному часі виконання, тому варто його відстежувати окремо від часу, витраченого на виконання ядра. Таким чином ми можемо вирішити, який етап виконання програми є пріоритетнішим для оптимізації.

Розподіл даних хоста за замовчуванням є сторінковим. GPU не може отримати доступ до даних безпосередньо з зі сторінкової пам'яті хоста, тому коли викликається передача даних пам'яті хоста в пам'ять пристрою, драйвер CUDA повинен спочатку виділити тимчасовий масив хоста із «закріпленою» пам'яттю, у якій відбувається блокування сторінок, скопіювати дані хоста в «закріплений» масив, а потім перенести дані з закріпленого масиву в пам'ять пристрою.

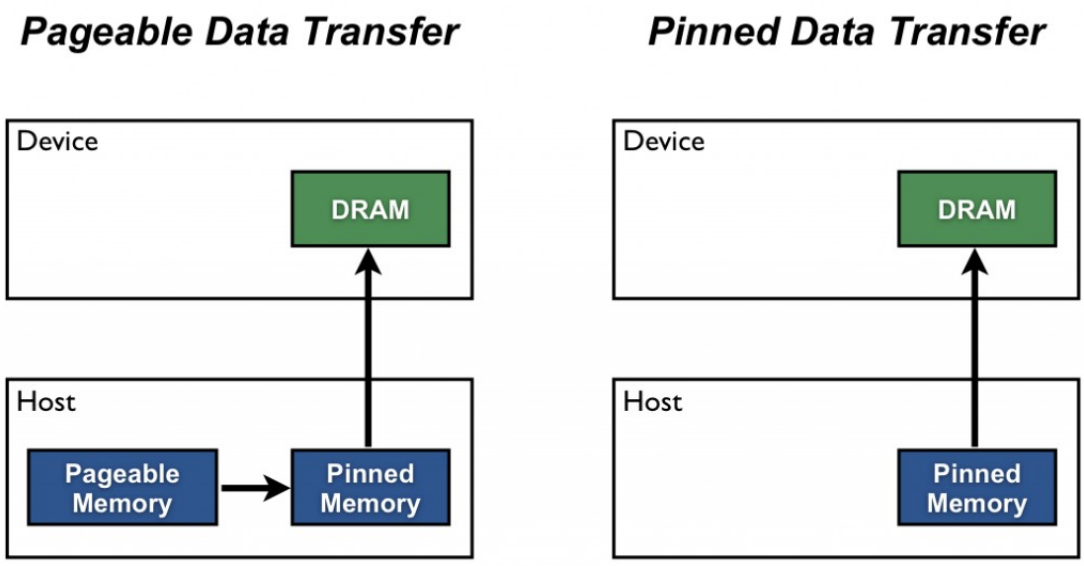


Рисунок 9 – Порівняння передачі «закріпленої» пам'яті та пам'яті зі сторінковим розподілом

Як можна побачити на рисунку 9, закріплена пам'ять використовується як проміжна зона для перенесення даних з пристрою на хост. Можемо уникнути витрат на передачу між масивами хоста, безпосередньо виділяючи його у «закріпленій» пам'яті, та вивільняючи цю пам'ять за допомогою функцій *cudaMallocHost* та *cudaFreeHost* з наступною сигнатурою:

```
__host__ cudaError_t cudaMallocHost (void** ptr, size_t size);
__host__ cudaError_t cudaFreeHost ( void* ptr );
```

Не слід надмірно розподіляти «закріплену» пам'ять. Це може знизити загальну продуктивність системи, оскільки зменшує обсяг фізичної пам'яті, доступної операційній системі та іншим програмам.[15]

Передача даних між хостом і пристроєм іноді може перекриватися виконанням ядра іншими передаваннями даних. Досягнення перекриття між передачею даних та іншими операціями вимагає використання потоків(англ. *stream*) CUDA, які являють собою послідовність операцій, що виконуються на пристрої у тому порядку, в якому вони видаються хостом. Хоча операції у потоці гарантовано виконуються у встановленому порядку, операції у різних потоках можуть чергуватися і, коли це можливо, навіть виконуватися паралельно. Усі операції на пристрої, а саме виконання ядер та передача даних, у CUDA виконуються у потоці. Тому якщо він не вказується, то використовується потік за замовчуванням. Він відрізняється від інших тим, що він є синхронізуючим потоком стосовно операцій на пристрої. Жодна операція в потоці за замовчуванням не почнеться, поки не завершаться всі раніше видані операції в будь-якому потоці на пристрої, і операція в потоці за замовчуванням повинна завершитися до того, як почнеться будь-яка інша операція (в будь-якому потоці на пристрої). Починаючи з сьомої версії CUDA, з'явилася можливість використовувати окремий потік за замовчуванням для кожного хост-потоку, а також поводитися з потоками за замовчуванням для кожного потоку як зі звичайними потоками. Для задання окремих потоків CUDA за замовчуванням для кожного хост-потоку, слід компілювати програму з додатковим параметром *–default-stream per-thread.*[16]

Потоки не за замовчуванням у CUDA C++ оголошуються, створюються та знищуються у хост-коді наступним чином:

```
cudaStream_t nonDefaultstream;
cudaStreamCreate(&nonDefaultstream);
```

```
cudaStreamDestroy(nonDefaultstream);
```

Для передачі даних до потоку не за замовчуванням ми використовуємо функцію *cudaMemcpyAsync*, яка аналогічна функції *cudaMemcpy*, тільки ще приймає ідентифікатор потоку п'ятим аргументом. На відміну від *cudaMemcpy*, *cudaMemcpyAsync* не є блокуючою операцією на хості, тому керування повертається до хост-потоку одразу після її виклику. А щоб видати ядро на потік не за замовчуванням, ми вказуємо ідентифікатор потоку як четвертий параметр конфігурації виконання.

Маємо наступний код програми:

```
__global__ void kernel(double* data, double* result, size_t size) {.... }

int main() {
    ...
    cudaMemcpy(deviceData, hostData, numBytes, cudaMemcpyHostToDevice);
    kernel<<<gridSize, blockSize>>>(deviceArrayA, deviceArrayB, deviceResult, N
       );
    cudaMemcpy(hostResult, deviceResult, numBytes, cudaMemcpyDeviceToHost);
    ...
}
```

Покажемо на його прикладі, як за допомогою потоків можна досягнути перекриття передачі даних та виконання ядра, тим самим покращити її час виконання. У модифікованому коді ми розбиваємо масив розміром N на S частини. Будем вважати, що N кратне S. Оскільки ядро оперує з усіма елементами незалежно, кожен з фрагментів масиву може бути оброблений незалежно. Тому створимо S потоків CUDA, кожен з яких оброблятиме фрагменти масиву розміром N/S. Існує декілька способів реалізувати доменну декомпозицію даних та їх обробки. Перший – зациклити всі операції для кожного фрагмента масиву, як показано у наступному фрагменті коду:

```
const size_t streamSize = N/S;
for (size_t i = 0; i < S; ++i) {
    const size_t offset = i * streamSize;
```

```
    cudaMemcpyAsync(deviceData + offset, hostData + offset, numBytes,
        cudaMemcpyHostToDevice, streams[i]);

    kernel<<<streamSize/blockSize, blockSize, 0, streams[i]>>>(deviceData +
        offset, deviceResult + offset, streamSize;

    cudaMemcpyAsync(hostResult + offset, deviceResult + offset, numBytes,
        cudaMemcpyHostToDevice, streams[i]);
}
```

Де *streams* – масив розміру S із заздалегідь створеними потоками CUDA. Інший підхід полягає у пакетному виконанні подібних операцій, видаючи спочатку всі передачі між хостом і пристроєм, потім всі запуски ядра, а потім всі передачі між пристроєм і хостом, як у наведеному нижче коді:

```
const size_t streamSize = N/S;
for (size_t i = 0; i < S; ++i) {
    const size_t offset = i * streamSize;
    cudaMemcpyAsync(deviceData + offset, hostData + offset, numBytes,
        cudaMemcpyHostToDevice, streams[i]);
}

for (size_t i = 0; i < S; ++i) {
    const size_t offset = i * streamSize
    kernel<<<streamSize/blockSize, blockSize, 0, streams[i]>>>(deviceData +
        offset, deviceResult + offset, streamSize;
}

for (size_t i = 0; i < S; ++i) {
    const size_t offset = i * streamSize;
    cudaMemcpyAsync(hostResult + offset, deviceResult + offset, numBytes,
        cudaMemcpyHostToDevice, streams[i]);
}
```

Обидва асинхронні методи, показані вище, дають правильні результати, але обидва підходи працюють дуже по-різному. Якщо ми виміряємо час виконання кожної версії програми, то побачимо, що на відеокарах з одним рушієм копіювання другий підхід покращує час виконання програми на відмінно від першого. І отримаємо зворотний результат при виконанні цих програм на

графічному процесорі з двома рушіями для копіювання даних. Щоб краще зрозуміти, чому ці два підходи працюють дуже по-різному, поглянемо на часові діаграму виконання кожної версії програми на рисунках 10 та 11. Функція *kernel* навмисне підібрана так, щоб час її виконання був однаковим з часом передачі даних.[19]

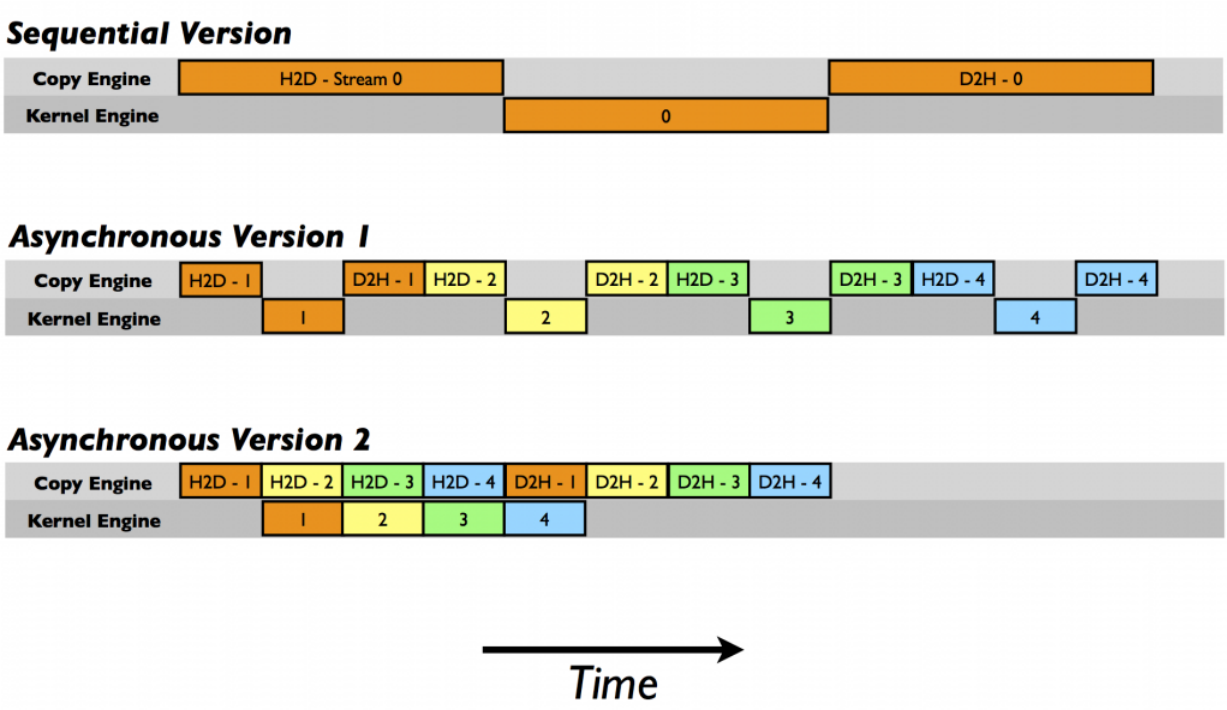


Рисунок 10 – Часові діаграми виконання всіх трьох версій програми GPU з одним рушієм копіювання

Спершу розглянемо випадок GPU з одним рушієм копіювання. Як і очікувалося для послідовної версії, жодна з операцій не перекривається. Для першої асинхронної версії нашого коду порядок виконання в рушії копіювання наступний: H2D потік(1), D2H потік(1), H2D потік(2), D2H потік(2) і так далі. Ось чому ми не бачимо прискорення при використанні першої асинхронної версії. Завдання видавалися рушію копіювання в порядку, що виключає будь-яке перекриття виконання ядра і передачі даних. Однак для другої версії, де всі передачі між хостом і пристроєм видаються перед будь-якою передачею між пристроєм і хостом, перекриття можливе, на що вказує менший час виконання.

Але коли ми маємо два рушії копіювання, один для передавання з хоста

на пристрій, а інший для передавання з пристрою на хост, то ми отримуємо зворотну ситуацію, коли перша версія асинхронної програми працює швидше за другу(рисунок 11).[19]

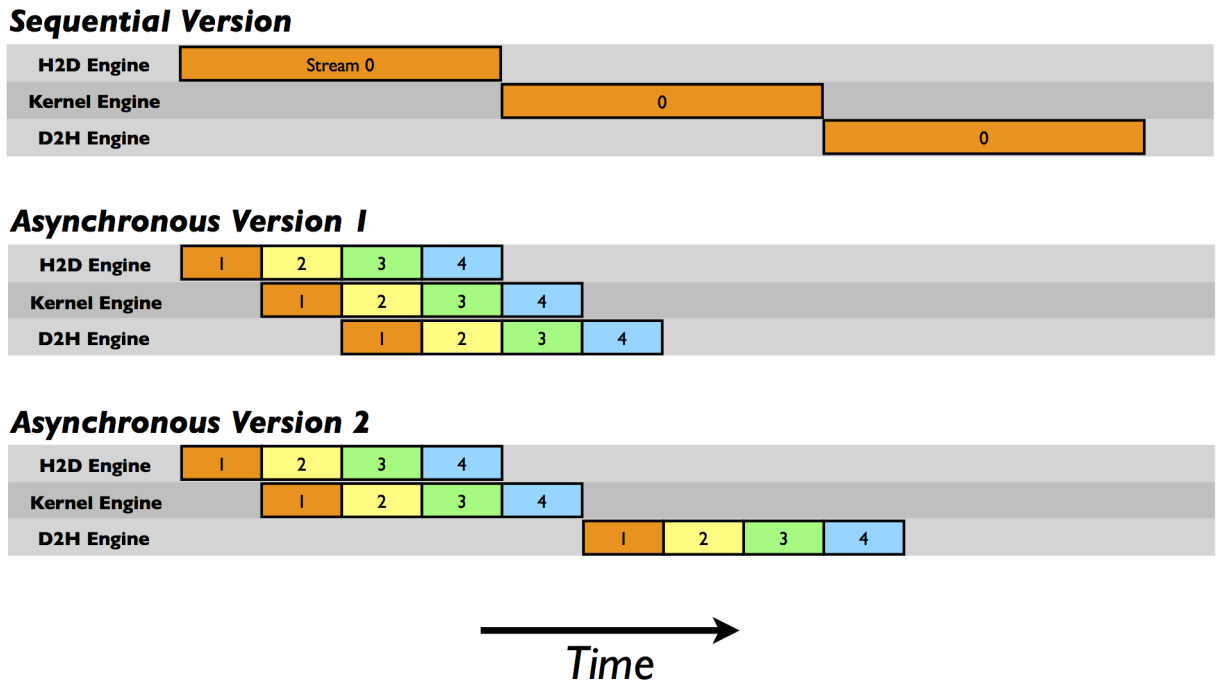


Рисунок 11 – Часові діаграми виконання всіх трьох версій програми на gpu з двома рушіями копіювання

Передача даних між пристроями в потоці i не блокує передачу даних між пристроями в потоці $i + 1$, як це було на GPU з одним рушієм копіювання. Але виникає питання, чому ми спостерігаємо погіршення часу виконання другої версії? Це пов'язано зі здатністю GPU одночасно запускати декілька ядер один за одним у різних не за замовчуванням потоках. Планувальник намагається увімкнути одночасне виконання цих ядер і в результаті затримує сигнал, який зазвичай виникає після завершення кожного ядра, що відповідає за запуск передачі даних між пристроєм і хостом, доки всі ядра не завершать роботу. Отже, хоча у другій версії нашого асинхронного коду є перекриття між передачею даних між хостом і пристроєм та виконанням ядра, немає перекриття між виконанням ядра і передачею даних між пристроєм і хостом.[19]

Також технологія CUDA надає механізм безпосереднього доступу

до пам'яті хоста, використовуючи «закріплену» відображену пам'ять (не підкачувана, з блокуванням сторінок). На дискретних графічних процесорах відображена «закріплену» пам'ять вигідна лише у певних випадках. Оскільки дані не кешуються на графічному процесорі, відображену закріплену пам'ять слід читати або записувати лише один раз, а глобальні завантаження та сховища, які читають і записують пам'ять, мають бути об'єднані. При використанні «закріпленої» відображеної пам'яті, використання декількох потоків(*streams*) можу бути не виправданим, тому що передача даних, створена ядром, автоматично перекриває виконання ядра без накладних витрат на налаштування та визначення оптимальної кількості потоків. Отже, щоб отримати безпосередній доступ до пам'яті хоста, нам потрібно спершу виділити пам'ять за допомогою функції *cudaHostAlloc*, і далі просто отримати вказівник пристрою, який посилатиметься на ту саму пам'ять, за допомогою cudaHostGetDevicePointer. І більше ніякого явного копіювання не знадобиться.[11]

До цього моменту при розробці CUDA-програм у всіх прикладах ми мали справу з розділеними адресними просторами пам'яті для пристрою та хоста(рисунок 12). Це дещо ускладнює розробку застосунків з використанням паралелізму графічних процесорів, та збільшує простір потенційних помилок при написанні коду програми, які можуть призвести до неправильної або невизначеної поведінки. Дана проблема стає ще більш відчутною, коли нам приходиться розробляти програми для гетерогенних систем з декількома графічними процесорами.

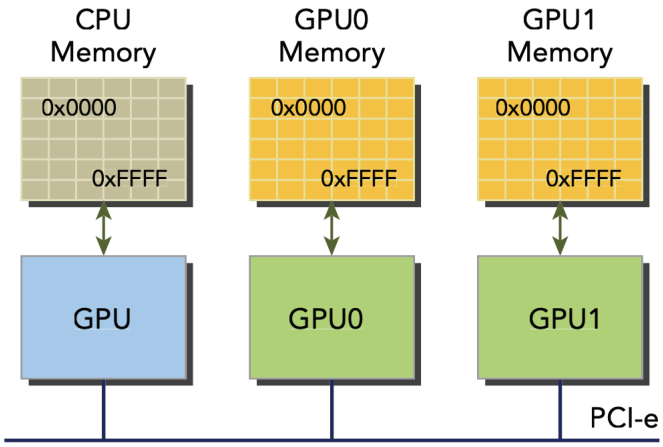


Рисунок 12 – Адресний простір без використання механізму уніфікованої пам'яті.

Дану проблему вирішує механізм уніфікованої пам'яті(англ. *Unified Memory*), який спрощує керування нею між центральним та графічним процесорами. Він дозволяє програмістам працювати з пам'яттю без необхідності явної передачі даних між хостом та пристроєм, що сприяє зручності програмування та може підвищити продуктивність. Це означає, що програміст може створювати та маніпулювати даними, нехтуючи їх розташуванням у пам'яті. CUDA самостійно вирішує, коли копіювати дані між хостом та пристроєм, забезпечуючи автоматичне керування цим процесом. Простими словами, механізм уніфікованої пам'яті усуває необхідність явного переміщення даних за допомогою виклику функції API CUDA *cudaMemcpy*. Переміщення даних, звісно, все одно відбувається, тому час виконання програми зазвичай не зменшується. Натомість уніфікована пам'ять дозволяє писати простіший та легший для супроводу код. Даний механізм пропонує модель спільної адресного простору для хоста та всіх підключених пристроїв до нього(рисунок 13).[10]

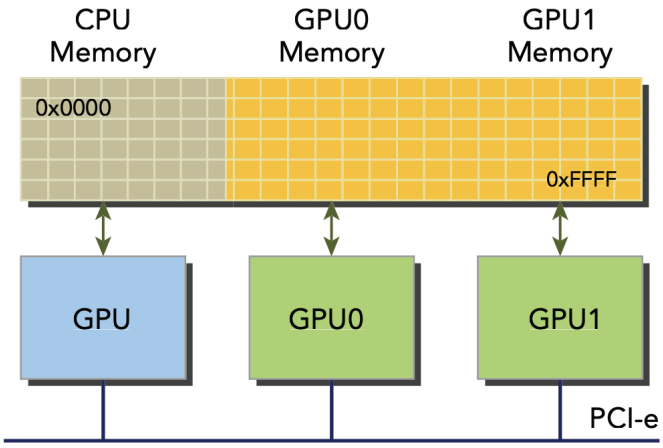


Рисунок 13 – Адресний простір з використанням механізму уніфікованої пам'яті.

Програма виділяє керовану пам'ять одним із двох способів: за допомогою процедури *cudaMallocManaged*, яка семантично подібна до cudaMalloc(); або шляхом визначення глобальної змінної $\_\_managed\_\_$, яка семантично подібна до змінної $\_\_device\_\_$.

Щоб досягти високої продуктивності з використанням механізму уніфікованої пам'яті, важливо:

- уникати помилок доступу, оскільки обробка помилок може значно знизити продуктивність програми, тому потрібно уникати затримок, пов'язаних зі зміною сторінок, міграцією даних та іншими операціями, які можуть зупинити виконання програми;
- забезпечити локалізацію даних, оскільки доступ до даних з місце, близького до процесора, дозволяє зменшити затримки та збільшити пропускну здатність пам'яті, тому необхідно мігрувати дані для забезпечення їх локальності;
- запобігати перевантаженню пам'яті, оскільки якщо дані часто звертаються декількома процесорами й постійно переміщуються, може виникнути надмірна втрата продуктивності, тому слід запобігти перевантаженню пам'яті або вчасно виявляти його та розв'язувати проблеми, які воно створює.

Щоб досягти такого ж рівня продуктивності, як і без використання уніфікованої пам'яті, програма повинна спрямовувати підсистему драйверів даного механізму на уникнення вищесказаних пасток. Варто зазначити, що підсистема драйверів уніфікованої пам'яті може виявляти загальні шаблони доступу до даних, і досягати деяких з цих цілей автоматично, без участі розробника. Але коли шаблони доступу до даних неочевидні, явні підказки від програми мають вирішальне значення. У версії 8.0 CUDA представлено корисний API для надання підказок щодо використання пам'яті під час виконання (функція *cudaMemAdvise*), та для явної попередньої вибірки (функція *cudaMemPrefetchAsync*). Ці інструменти надають ті самі можливості, що і явні API копіювання та закріплення пам'яті, не повертаючись до обмежень явного виділення пам'яті на графічному процесорі.[10]

Обговорюючи відображену «закріплену» та уніфіковану пам'ять, потрібно згадати системи з інтегрованими графічними процесорами. Оскільки використання цих типів пам'яті дозволяють уникнути зайвих копій, бо інтегрована пам'ять графічного процесора і пам'ять центрального процесора є фізично єдиними.

РОЗДІЛ 3. РОЗРОБКА ЕФЕКТИВНИХ АЛГОРИТМІВ ДЛЯ ГЕТЕРОГЕННИХ СИСТЕМ З ВИКОРИСТАННЯМ CUDA

У цьому розділі на практиці розглянуто методи та засоби, за допомогою яких можна ефективніше проводити обчислення загального призначення на гетерогенних системах з графічними процесорами, з використанням платформи CUDA та її основної мови – C++. Також буде проведено теоретичні та експериментальні дослідження кожної версії алгоритму, з метою вибору найкращої, та визначення доцільності використання гетерогенних систем у розв'язні тої чи іншої задачі. Реалізації використовують останні версії технології CUDA та стандарту мови C++, які підтримується в момент написання даної роботи. А саме CUDA 12.0 та C++20. Тому й для збірки коду, представленому у цьому розділі, необхідні відповідні версії компіляторів. Вся експериментальна складова цього розділу(тестування та засікання часу виконання) проводиться на справжній гетерогенній системі, у склад якої входить: два потужних центральних процесори **AMD EPYC 7713** з шістдесятьма чотирма фізичними ядрами, та підтримкою технології *Simultaneous Multithreading*, що в сумі дає 128 фізичних та 256 віртуальних ядер; новітнім графічним процесором **NVIDIA A100**, який спроєктований для високоефективних обчислень.

3.1 Алгоритм знаходження скалярного добутку двох векторів

Почнемо з нескладної, для розуміння та реалізації на гомогенних обчислювальних системах, задачі – знаходження скалярного добутку двох векторів.

Для кращої читабельності та зрозумілості коду програми, було введено два псевдоніми типів:

```
using Number = double;
using Vector = std::vector<Number>;
```

Отже, маємо два n-вимірні вектори в просторі $\mathbb{R}^n$ $\overrightarrow{a} = (a_1, a_2, \cdots, a_{n-1}, a_n)$ та $\overrightarrow{b} = (b_1, b_2, \cdots, b_{n-1}, b_n)$. Потрібно знайти $\overrightarrow{a} \cdot \overrightarrow{b} = \sum_{i=1}^{n} a_i b_i$.

Для початку приведемо реалізацію примітивного однопотокового алгоритму знаходження скалярного добутку двох векторів на CPU (код наведено в додатку А). Не складно помітити, що часова складність даного алгоритму $O(n)$. За допомогою бібліотеки *chrono*, яка входить до набору стандартної бібліотеки C++, були проведенні заміри часу виконання даної функції на нашій системі для даних різної величини (табл. 2).

Таблиця 2 – Час виконання примітивної однопотокової реалізації алгоритму знаходження скалярного добутку двох векторів

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній
10^2	<0	<0	<0	<0	<0	< 0
10^3	2	2	2	2	2	2
10^4	24	24	24	24	24	24
10^5	244	250	244	243	243	244,8
10^6	2435	2460	2444	2442	2435	2443,2
10^7	24408	24410	24400	24396	24404	24403,6

Одразу помітимо, що теоретична лінійна часова складність була підтверджена на практиці. Візьмемо нашу примітивну реалізацію алгоритму, будемо намагатись покращити час її виконання за допомогою можливостей, які нам надає технологія CUDA та наша гетерогенна система. Перед тим як переходити до використання CUDA та GPU, для досягнення нашої цілі, згадаємо, що гетерогенна система – це все ж таки система, що складається з кількох обчислювальних пристроїв різного типу. У нашому випадку це два центральних та один графічний процесори. Оскільки у нашому випадку CPU

мають 64 фізичних ядра та підтримку технології Simultaneous Multithreading, що в сумі дає 128 фізичних та 256 віртуальних ядер, для початку можемо спробувати реалізувати паралельну версію з використанням тільки центральних процесорів (код наведено у додатку Б). Розпаралелення нашої примітивної версії на CPU було проведено загальновідомою технікою *map-reduce*, яка поділяє вхідні дані на декілька частин, після чого для обробки кожного фрагменту створює окремий потік. Після завершення виконання всіх додаткових потоків, основний отримує частковий результат кожного, та об'єднує їх в загальний. Припустимо, у нашій системі є C ядер, та на керування кожним затрачається q одиниць часу. Тоді можемо вважати, що часова складність алгоритму $O(\frac{n}{C} + q)$. Але оскільки q та C є постійними величинами, то при великих n, ними можемо нехтувати при аналізі алгоритму, оскільки час виконання потоку почне домінувати. Але саме такий підхід до аналізу нашої програми може пояснити збільшення часу виконання при малому розмірі даних. Отже, щоб порівняти ці дві реалізації на практиці, проведемо часові заміри виконання даного алгоритму, аналогічні до замірів примітивної однопотокової версії (табл. 3).

Таблиця 3 – Час виконання паралельної реалізації алгоритму знаходження скалярного добутку двох векторів

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	7486	7488	6255	6923	7586	7147,6
10^3	5941	7030	7450	6297	6718	6687,2
10^4	5505	7350	7296	5768	6186	6421
10^5	6694	5872	5890	5796	6617	6173,8
10^6	6943	6610	6566	7162	6481	6752,4
10^7	7597	7567	7607	8120	8183	7814,8

Як зазначалось раніше, у паралельних алгоритмах на CPU можливе погіршення часу виконання на даних малої розмірності, оскільки у цьому випадку відбувається домінування часу керування потоками над часом їх виконання. Але уже на достатньо великих даних ми можемо спостерігати пришвидшення більше ніж у чотири рази.

Тепер наведемо приклад реалізації алгоритму знаходження скалярного добутку двох векторів на графічному процесорі з використанням технології CUDA. Код ядра наведено у додатку В, та далі у цьому розділі змінюватись не буде. А код хоста представлений в додатку Г. Як зазначалось в розділі 2, кожна CUDA-програма повинна складатись з двох частин: коду хоста та коду пристрою. Так і в нашому прикладі є функція *dot* в просторі імен *kernel*, яка позначена спеціальним специфікатором простору виконання $\_\_global\_\_$, що означає, що вона буде виконана K разів на K паралельних потоках. А більша частина коду хоста (крім введення даних), знаходиться в функції *dotProduct* в просторі імен *gpu_single_stream*. У ній реалізовано перенесення даних з хоста на пристрій, виклик ядра, перенесення часткових результатів виконання кожного блоку з пристрою на хост, та їх об'єднання. Оскільки в CUDA є обмеження на кількість потоків у кожному блоці, яке залежить від версії *обчислювальної здатності* пристрою, тому було вирішено використовувати величину 1024 як розмір блоку, через яку точно не виникне помилка при виклику ядра на хості. Далі ми обчислюємо кількість блоків для виклику ядра таким чином, щоб на кожен індекс вектора припадав свій потік. Але оскільки дана величина також обмежена, то зменшуємо кількість блоків до верхнього ліміту, у випадку його перевищення. Максимальну кількість блоків у сітці отримуємо за допомогою власної простої шаблонної функції *getMaxGridSize*, яка як параметр шаблону отримує індекс осі, а сама «дістає» інформацію про обмеження пристрою за допомогою виклику API CUDA *cudaGetDeviceProperties*.

Тепер детально розглянемо код ядра, який містить наступні етапи:

- створення спільної пам'яті для кешування результатів обчислень кожного потоку;
- отримання унікального ідентифікатора потоку tid, який обчислюється на основі threadIdx.x, blockIdx.x та blockDim.x;
- обчислення часткової суми в циклі *while*, шляхом множення відповідних елементів з v1 і v2 та додавання результату до локальної змінної temp;
- збереження часткової суми в спільній пам'яті *cache*;
- синхронізація всіх потоків за допомогою $\_\_syncthreads$, щоб дочекатися завершення запису до спільної пам'яті перед продовженням виконання;
- згортка часткових сум для отримання кінцевого результату;
- збереження кінцевого результату (часткової суми блоку) у векторі *partSum* під індексом blockIdx.x, тільки першим потоком у блоці.

Часова складність виконання ядра складає $O(\frac{n}{K}) + O(log(b))$, де K – це загальна кількість потоків графічного процесора, а b – кількість потоків одного блоку. $O(\frac{n}{K})$ операцій виконує алгоритм при обчисленні часткових сум у циклі while, а $O(log(b))$ операцій – при виконанні згортки часткових сум. Але не потрібно забувати, що перед тим, як виконувати будь-які обчислення на графічному процесорі, хост повинен передати дані на пристрій, і після виконання ядра отримати результат. У цьому випадку хост надсилає на пристрій повністю два вектори розмірності n, і приймає масив часткових сум величиною $\frac{n}{b}$. Останнім кроком алгоритм знаходить суму всіх елементів масиву *partialSum*, що і буде фінальним результатом алгоритму. Тому часову складність передачі даних становить $O(n)$, а знаходження суми елементів масиву *partialSum* – $O(\frac{n}{b})$. Отже, як не складно помітити, фінальна часова складність алгоритму – $O(\frac{n}{K}) + O(log(b)) + O(n) + O(\frac{n}{b}) = O(n)$, що з першого погляду не дає надії на покращення часу виконання нашої програми при використанні графічного процесора. Але якщо передача даних на практиці відбуватиметься достатньо швидко, то і ціла реалізація теоретично може показати кращу продуктивність, ніж попередні.

Тому проведемо часові заміри всієї реалізації цього алгоритму та його окремих фрагментів (таблиці 4, 5, 6). Важливо зазначити, що для правильності заміру часу виконання ядра, використовувався спеціальний механізм подій CUDA, оскільки виклик ядра є асинхронним, та використання бібліотеки chrono не підходить для даної задачі.

Таблиця 4 – Час виконання ядра CUDA для задачі знаходження скалярного добутку двох векторів

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	20	19	20	20	22	20,2
10^3	12	11	12	11	12	11,6
10^4	10	8	10	10	10	9,6
10^5	13	11	13	13	11	12,2
10^6	25	23	25	27	25	25
10^7	356	367	356	361	362	360,4

Таблиця 5 – Час знаходження суми елементів масиву *partialSum*

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	<0	<0	<0	<0	<0	<0
10^3	<0	<0	<0	<0	<0	<0
10^4	<0	<0	<0	<0	<0	<0
10^5	1	1	1	1	1	1
10^6	9	9	9	9	9	9
10^7	94	94	95	95	96	94,8

Таблиця 6 – Час виконання CUDA-програми знаходження скалярного добутку двох векторів з одним потоком команд за замовчуванням

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	354	355	360	355	362	357,2
10^3	296	280	354	294	300	304,8
10^4	306	302	296	304	307	303
10^5	450	443	306	450	444	418,6
10^6	2218	2177	450	2171	2211	1845,4
10^7	15323	15313	15323	15460	15349	15353,6

Аналізуючи таблиці 2, 3 та 6, можемо помітити, що використання переваг графічного процесора для даної задачі все-таки може давати приріст в ефективності для деяких даних. Використання CUDA та GPU незначно покращило час виконання нашої програми порівняно з однопотоковою реалізацією для векторів розмірності 10^6 та 10^7, і в порівнянні з паралельною версією – для всіх розмірностей, крім 10^7. Також проаналізувавши таблиці 4 та 5, бачимо, що самі обчислення займають мало часу від виконання всієї функції. Отже, розуміємо, що основний час виконання функції йде на передачу даних між хостом і пристроєм, що і передбачалось при теоретичній оцінці алгоритму. Для остаточного підтвердження нашої гіпотези, скористаємось утилітою Nsight Systems, за допомогою якої виконаємо профілювання нашої програми (звіт наведено в додатку Д). Чітко можна помітити, що 98.8% часу пристрою йде на операції з даними, і тільки 1.2% він виконує обчислення. Отже, в першу чергу нам потрібно оптимізувати передачу векторів між хостом і пристроєм.

У підрозділі 2.4 ми розглядали різні методи для зменшення часу на передачу даних. Один з найпростіших і найдієвіших методів – це використання

«закріпленої» пам'яті, замість пам'яті зі сторінковим розподілом. Ним і скористаємось у першу чергу (код наведений у додатку Е). Дана реалізація має відмінності тільки у функції *main*. Тепер для зберігання векторів використовуються вказівники, які зберігають адрес на початок фрагмента даних, який ми виділили за допомогою функції API CUDA – *cudaMallocHost*. Проведемо часові заміри цієї реалізації (табл. 7).

Таблиця 7 – Час виконання CUDA-програми знаходження скалярного добутку двох векторів з одним потоком команд за замовчуванням та «закріпленою» пам'яттю хоста

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	361	351	357	368	358	359
10^3	304	300	286	298	302	298
10^4	310	302	300	305	302	303,8
10^5	366	369	373	377	373	371,6
10^6	1520	1542	1449	1535	1519	1513
10^7	7538	7430	7592	7591	7588	7547,8

Бачимо, що використання «закріпленої» пам'яті покращило більше ніж у двічі час виконання нашої CUDA-програми, у порівнянні з версією зі сторінковою організацією пам'яті на хості. І на найбільших, у нашому випадку, розмірностях векторів (10^6 та 10^8) має найкращий результат серед всіх версій, включаючи однопотокову та паралельну реалізацію на CPU. До цього моменту ми використовували тільки один потік(*stream*) за замовчуванням для операцій. Якщо уважно подивитись на нашу реалізацію функції-ядра, то стає зрозуміло, що кожен блок працює тільки зі своїм індексним простором векторів, і вони не перетинаються між собою. Тобто, не виникає такої ситуації, що два потоки з

різних блоків використовуватимуть один і той самий адрес пам'яті, щоб зчитати чи записати інформацію. А це означає, що ми можемо спробувати поділити наш індексний простір на декілька не перехресних частин, та для роботи з кожною створити свій потік команд. Тоді внаслідок асинхронного виконання ядер та передачі даних, ми можемо отримати пришвидшення від перекриття цих операцій, як це розглядалось в підрозділі 2.4. Код CUDA-програми представлено у додатку Ж. Проведемо часові заміри цієї реалізації (табл. 8)

Таблиця 8 – Час виконання CUDA-програми знаходження скалярного добутку двох векторів з використанням декількох потоків команд

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	310	308	295	305	300	303,6
10^3	310	313	300	311	314	309,6
10^4	606	587	588	593	606	596
10^5	541	526	522	524	533	529,2
10^6	1560	1502	1520	1516	1514	1522,4
10^7	7527	7569	7517	7609	7537	7551,8

Видно, що використання декількох потоків практично не дало ніякого результату. Щоб краще розібратися чому, знову проведемо профілювання даної реалізації (звіт представлено у додатку И). За результатами бачимо, що перекривання все-таки відбувається, але оскільки час виконання ядра наскільки малий, порівняно з часом виконання операцій пам'яті, що цей підхід не дав бажаного результату. Але це був би дуже хороший інструмент, якби у нашому випадку ядро виконувало більше обчислень.

Якщо основною нашою проблемою для покращення продуктивності програми – є необхідність виконання операцій передачі даних між хостом і

пристроєм, які домінують у часі виконання обчислень, то ми можемо спробувати використати механізм «закріпленої» відображеної пам'яті (підрозділ 2.4). Оскільки він не потребує перенесення будь-якої інформації між хостом і пристроєм, а просто надає можливість графічному процесору отримувати дані напряму з хоста. Код даної реалізації ідентичний коду версії з просто «закріпленою» пам'яттю. Тільки замість її алокації на пристрої, за допомогою виклику CUDA API *cudaMalloc*, використовується *cudaHostGetDevicePointer*, що отримати відображену адресу на пам'ять хоста. Виконаємо заміри ядра і цілої функції для цієї версії (таблиці 9 та 10).

Таблиця 9 – Час виконання CUDA-ядра знаходження скалярного добутку двох векторів з використанням «закріпленої» відображеної пам'яті

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	27	21	24	23	23	23,6
10^3	15	15	16	16	16	15,6
10^4	23	23	23	23	25	23,4
10^5	95	96	97	86	97	94,2
10^6	652	653	644	644	647	648
10^7	6019	6014	6018	6019	6021	6018,2

Таблиця 10 – Час виконання CUDA-програми знаходження скалярного добутку двох векторів з використанням «закріпленої» відображеної пам'яті

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	83	81	80	85	80	81,8
10^3	46	46	46	47	48	46,6
10^4	54	53	54	54	56	54,2
10^5	128	128	130	118	131	127
10^6	2105	2112	2069	2093	2066	2089
10^7	7688	7620	7733	7718	7673	7686,4

На жаль, використання «закріпленої» відображеної пам'яті не дало ніякого прискорення. По збільшеному у рази часу виконання ядра зрозуміло, що вся причина у великому часі доступу до пам'яті хоста, який у тому числі й зумовлений відсутністю кешування даного виду пам'яті.

3.2 Алгоритм множення прямокутних матриць

Розглянемо звичайний ітеративний алгоритм множення двох прямокутних матриць, який і буде надалі оптимізувати. На вхід програми маємо дві матриці: A розмірності $n \times m$, B розмірності $m \times p$. Потрібно віднайти матрицю D розмірності $n \times p$, де $d_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj}$. Реалізуємо примітивний однопотоковий і паралельний алгоритм, що розбиває простір індексів рядків матриці D на неперетинних частин, де C – кількість CPU-ядер (код реалізації цих функцій представлено у додатку К). Можна помітити, що складність однопотокової версії алгоритму – $O(nmp)$, або $O(n^3)$ у разі квадратних матриць. Тоді як паралельна версія пришвидшує час обчислень приблизно у C разів, але додає накладних часових витрат q на керування кожного потоку. Отже, для паралельного алгоритму маємо складність $O(\frac{nmp}{C} + qC)$ (або $O(\frac{n^3}{C} + qC)$ для квадратних

матриць). Але як уже зазначалось у попередньому розділі, при великих розмірах матриць можна нехтувати доданком qC, який складається з добутку констант. Виконаємо заміри для даних реалізацій. Для простоти й наочності будемо використовувати квадратні матриці. У силу відносно великого часу виконання CPU-реалізацій алгоритму множення двох матриць, було проведено тільки по одному заміру. Для однопотокової версії результати такі: $n = 10$ – 9 мікросекунд, $n = 100$ – 7685 мікросекунд, $n = 1000$ – 7577733 мікросекунд (7.6 с), $n = 2000$ – 60931159 мікросекунд (1 хв), $n = 5000$ – 986853872 мікросекунд (16 хв), $n = 10000$ – > 1 год. Для паралельної версії результати такі: $n = 10$ – 6368 мікросекунд, $n = 100$ – 13010 мікросекунд, $n = 1000$ – 1816413 мікросекунд (1.8 с), $n = 2000$ – 7402146 мікросекунд (7 с), $n = 5000$ – 122146485 мікросекунд (2 хв), $n = 10000$ – 952118269 мікросекунд (16 хв).

Отримані результати показують, що як і передбачалося, паралельна версія на даних малої розмірності працює гірше, ніж однопотокова. Але на довгій дистанції він все-таки його «обганяє». Попри це, швидкість паралельного алгоритму залишається незадовільною. Тому спробуємо покращити продуктивність програми, використовуючи переваги графічних процесорів. У додатку Л запропоновано реалізацію CUDA-програми множення двох матриць, у якому логіку третього вкладеного циклу, з однопотокової версії, переносимо у функцію ядра. А два зовнішніх цикли заміняємо на виклик ядра з двовимірною сіткою та двовимірними блоками. Це працює, оскільки обчислення елементів шуканої матриці повністю незалежне одне від одного, та не потребують жодної синхронізації, бо матриці A та B використовують тільки для читання.

З хоста на пристрій передаються дві матриці розмірності $n \times m$ та $m \times p$, а з пристрою на пам'ять передається масив розмірності $n \times p$. Тому, враховуючи заміну перших двох зовнішніх циклів однопотокового алгоритму множення двох матриць на виклик ядра, загальна складність цього алгоритму становить $O(nm) + O(mp) + O(np) + O(\frac{np}{K}m)$. Або $O(n^2) + O(\frac{n^2}{K}n)$, у випадку квадратних

матриць. А це означає, на меншу, ніж квадратичну складність алгоритму, на практиці очікувати не можна, якщо використовується гетерогенна система з дискретною відеокартою. Проведемо вимірювання часу виконання нашої CUDA-реалізації алгоритму множення двох матриць (таблиці 11 та 12).

Таблиця 11 – Час виконання ядра CUDA-реалізації алгоритму множення двох матриць

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	26	21	28	24	30	25,8
10^3	34	35	41	42	42	38,8
10^4	1089	1136	1377	1374	1379	1271
10^5	17835	18052	18041	18116	17904	17989,6
10^6	265690	274092	272599	263252	263327	267792
10^7	2173195	2174666	2166431	2173078	2170586	2171591,2

Таблиця 12 – Час виконання CUDA-реалізації алгоритму множення двох матриць

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	341	351	361	366	354	354,6
10^3	293	301	310	312	313	305,8
10^4	5825	5986	6286	6249	6219	6113
10^5	32663	32869	32938	32833	31929	32646,4
10^6	343709	352498	351815	344983	341198	346840,6
10^7	2472626	2474966	2467973	2489895	2470899	2475271,8

Помітно очікуваний приріст швидкодії CUDA-реалізації алгоритму при

збільшенні розмірності матриць. Використання переваг графічного процесора на нашій гетерогенній системі знизив час виконання алгоритму, порівняно з попередніми реалізаціями на CPU, у сотні разів. Також можна помітити, що зі збільшенням розмірності вхідних даних, час виконання ядер все більше і більше переважає виконання іншої частини функції. Це означає, що подальші дії щодо покращення швидкодії цієї реалізації алгоритму не має бути пов'язаним з оптимізацією передачі даних.

Також у рамках даного дослідження були намагання об'єднати переваги SIMD та MIMD паралелізму нашої гетерогенної системи, для задачі множення двох матриць (код представлено у додатку М). Ідея полягала у тому, щоб адаптувати найуспішнішу CUDA-реалізацію знаходження скалярного добутку з підрозділу 3.1 під рядки матриць. Тепер функція ядра приймає дві матриці, індекси рядка першої та стовпця другої матриці. А повертає скалярний добуток рядка і стовпця. Далі уже на хості створюються C потоків для фрагментів матриці, і в кожному потоці, за допомогою викликів ядра, швидко знаходяться значення елементів відповідного фрагмента матриці. Передбачалось, що програма буде компілюватись зі спеціальним параметром *–default-stream per-thread*, що забезпечить кожен потік хоста своїм потоком(stream) команд за замовчуванням, що б гарантувало асинхронне виконання виклику ядер з різних CPU-потоків. Але дана реалізація на практиці показала погані показники продуктивності (на розмірності 2000×2000 час виконання становив 1 хв), тому була відкинута.

3.3 Алгоритм Флойда – Воршелла

Алгоритм Флойда – Воршелла є методом знаходження найкоротших шляхів між усіма парами вершин у зваженому орієнтованому графі. Цей алгоритм базується на динамічному програмуванні й забезпечує розв'язання проблеми найкоротших шляхів у графі з часовою складністю $O(|V|^3)$, де $|V|$ – кількість вершин у графі. Передбачається, що граф не містить циклів від'ємної

ваги (тоді відповіді між деякими парами вершин може просто не існувати – вона буде нескінченно маленькою). Основна ідея алгоритму полягає в тому, щоб поступово покращувати найкоротші шляхи між парами вершин, використовуючи проміжні вершини як посередники. Алгоритм працює у кілька етапів, на кожному з яких розглядаються шляхи, які можуть бути скорочені з використанням проміжних вершин. На вхід алгоритм приймає матрицю суміжності графа, а результатом його виконання є матриця довжин найкоротших шляхів між вершинами.[20] Фрагмент C++ коду однопотокової версії алгоритму:

```
Matrix floydWarshallAlg(const Matrix& graph)
{
    auto dis = graph;
    const size_t numVertices = graph.size();
    for (size_t k = 0; k < numVertices; ++k) {
        for (size_t i = 0; i < numVertices; ++i) {
            for (size_t j = 0; j < numVertices; ++j) {
                if (dis[i][j] > dis[i][k] + dis[k][j]) {
                    dis[i][j] = dis[i][k] + dis[k][j];
                }
            }
        }
    }
    return dis;
}
```

Очевидно зі структури вкладених циклів, що часова складність цього алгоритму $O(|V|^3)$. Одразу реалізуємо паралельну версію алгоритму, розділивши індексний простір k на однакових неперетинних частин, де C - це кількість ядер на CPU. Фрагмент коду розбиття індексного простору:

```
const size_t blockSize = std::ceil(static_cast<double>(numVertices) /
   numThreads);
std::vector<std::thread> threads(numThreads);
for (size_t k = 0; k < numVertices; ++k) {
    for (size_t t = 0; t < numThreads; ++t) {
        const size_t start = t * blockSize;
        const auto end = std::min<size_t>((t + 1) * blockSize, numVertices);
```

```
            threads[t] = std::thread([&, k, start, end]() {
                for (size_t i = start; i < end; ++i) {
                    for (size_t j = 0; j < numVertices; ++j) {
                        if (dis[i][j] > dis[i][k] + dis[k][j]) {
                            dis[i][j] = dis[i][k] + dis[k][j];
                        }
                    }
                }
            });
        }
        for (auto& thread : threads) {
            thread.join();
        }
    }
```

Варто зауважити, що у нас нема потреби синхронізувати між потоками читання та запис у масив *dic*, оскільки наш алгоритм намагається покращити відстань між двома вершинами графа, на основі попередньої ітерації покращень. Але якщо для будь-яких двох вершини в іншому потоці уже відбулося покращення величини відстані між ними, то це ні в якому разі не погіршить результат на даній ітерації. Що грає нам на руку при досягненні нашої мети в оптимізації алгоритму. Аналогічно аналізу алгоритму множення матриць, можемо прийти до того, що оцінка часової складності нашої паралельної версії – $O(\frac{|V|^3}{C} + qC) = O(n^3)$. Хоч константи C та q ми ігноруємо при асимптотичному аналізі, вони можуть пояснити причину покращення чи погіршення продуктивності на практиці. Проведемо вимірювання його часу виконання цих двох реалізацій на нашій системі (таблиці 13, 14).

Таблиця 13 – Час виконання однопотокової реалізації алгоритму Флойда – Воршелла

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	906	905	909	910	905	907
10^3	100591	100563	100205	100414	100470	100448,6
10^4	750685	750440	750246	751079	752943	751078,6
10^5	6165509	6186122	5889505	6006957	5864430	6022504,6
10^6	91992893	92041917	91770207	92295687	91828346	91985810
10^7	717019218	717521090	717551388	717116862	717338952	717309502

Таблиця 14 – Час виконання паралельної реалізації алгоритму Флойда – Воршелла

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	635682	635744	648527	643872	637819	640328,8
10^3	3226873	3178291	3257496	3269018	3229756	3232286,8
10^4	6419849	6489726	6442135	6463211	6402104	6443405
10^5	13256811	13569842	13296592	13198749	13310752	13326549,2
10^6	74895818	76052417	74587215	75264918	74536982	75067470
10^7	443353628	437142372	441927336	449622371	441872156	442783572,6

Як і передбачувалось, паралельний алгоритм не ефективним на даних малої розмірності через домінантні, в часі виконання, затрати на керування потоками. Але на матрицях більшої розбірності, уже спостерігаємо прискорення в 1.6 рази.

Тепер спробуємо ще пришвидшити наш алгоритм, використовуючи

переваги графічних процесів. Для цього скористаємось властивістю про відсутність необхідності синхронізувати запис і читання в масиві з дистанціями, яку ми згадували при реалізації паралельної версії. Знаючи цей факт, ми можемо в кожній ітерації покращення, виконувати спробу зменшення довжини шляху для кожної пари вершин в окремому потоці незалежно. Що дає нам можливість замінити подвійний внутрішній цикл на виклик одного ядра з двовимірною сіткою з двовимірними блоками (код представлений в додатку Н). Як і в попередньому розділі, одразу буде приведена реалізація із використанням «закріпленої» пам'яті, з метою отримання максимальної швидкодії. З хоста на пристрій передається граф у виді матриці суміжності, а з пристрою на хост у результаті роботи алгоритму передається матриця розмірності $|V|$ з всіма відстанями між ребрами. Отже, на обмін даних між хостом і пристроєм потрібно $O(|V|^2)$ додаткових операцій. На виклик одного ядра потрібно $O(\frac{|V|^2}{K})$, де – це кількість потоків графічного процесора. Ну й оскільки ядро викликається $|V|$ разів, то асимптотична складність усього алгоритму становить $O(|V|\frac{|V|^2}{K} + |V|^2)$. Це означає, що на практиці у найкращому випадку (коли $K \geq |V|$) ми зможемо сподіватись на $O(|V|^2)$ операцій на гетерогенних системах з дискретною відеокартою. Проведемо вимірювання часу виконання нашої реалізації (таблиці 15 та 16).

Таблиця 15 – Сумарний час виконання всіх $|V|$ викликів ядра CUDA-реалізації алгоритму Флойда – Воршелла

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	1983	1970	1966	1953	1882	1950,8
10^3	6913	6844	6917	6855	6880	6881,8
10^4	19791	17204	20125	17140	17406	18333,2
10^5	146014	146861	146197	146499	145380	146190,2
10^6	2572294	2570234	2571398	2571114	2571457	2571299,4
10^7	20377976	20378640	20375720	20365522	20376980	20374967,6

Таблиця 16 – Сумарний час виконання всіх $|V|$ викликів ядра CUDA-реалізації алгоритму Флойда – Воршелла

	Час (мілісекунди)					
Розмірність векторів	Запуск №1	Запуск №2	Запуск №3	Запуск №4	Запуск №5	Середній час
10^2	2281	2286	2277	2278	2188	2262
10^3	7944	7863	7953	7883	7916	7911,8
10^4	22909	20295	23262	20271	20556	21458,6
10^5	157378	158160	157577	157892	156810	157563,4
10^6	2640015	2643442	2639447	2639224	2639891	2640403,8
10^7	20643960	20644768	20641981	20632643	20645382	20641746,8

Можемо помітити очікуваний приріс швидкодії CUDA-реалізації алгоритму при збільшенні кількості вершин в графі. Алгоритм Флойда – Воршелла, з використанням переваг графічного, процесора на нашій гетерогенній системі знизив час виконання, порівняно з попередніми реалізаціями на CPU, у 20-30

разів. Також можна помітити, що зі збільшенням вершин графа, час виконання ядер все більше і більше переважає виконання іншої частини функції. Що означає, що подальші дії щодо покращення швидкодії цієї реалізації алгоритму не має бути пов'язаним з оптимізацією передачі даних.

ВИСНОВКИ

У першому розділі роботи було розглянуто архітектуру графічного процесора, порівнюючи її з усім відомою архітектурою центрального процесора. Для кращого розуміння відмінностей, спершу було наведено класифікацію ЕОМ за Флінні. У хоті порівняння графічного та центрального процесорів, було вияснено, що CPU реалізовують SISD або MIMD, у випадку наявності одного чи декількох ядер відповідно, коли GPU реалізовують SIMD архітектуру. А це означає, що вони найкращим чином підходять для однотипних обчислень над великою кількістю даних. Далі у цьому розділі було описано дві технології для розробки програм для гетерогенних систем з графічними процесорами: CUDA та її найпопулярніший аналог - OpenCL. У ході порівняльного аналізу вияснили, що вони дуже схожі між собою, але кожна з них має ряд нюансів, які можуть бути вирішальними при виборі між ними. А саме, якщо ви для своїх обчислень використовуєте графічні процесори компанії NVIDIA та вам необхідно максимальна продуктивність, то технологія CUDA – це ваш вибір. Як показали роботи[12] [13] [14] інших дослідників, у більшості випадків реалізація алгоритмів на CUDA дають кращий результат часу виконання програми. Але якщо потрібно запускати ваші програми на пристроях відмінних від графічних процесорів компанії NVIDIA, то OpenCL залишається єдиним вибором з цих двох технологій. Оскільки він підтримує широкий спектр обчислювальних пристроїв, включаючи GPU від різних виробників, процесори та інші спеціалізовані пристрої.

У другому розділі розглянуто розширення мови C++ від компанії NVIDIA – CUDA C++, яке дозволяє звичним нам способом розробляти застосунки, які використовують переваги графічних процесорів. Дізналися, що для відділення коду пристрою від коду хоста, були додані спеціальні специфікатори простору виконання. Також компанія NVIDIA пропонує свій компілятор NVCC, який уміє:

– відділяти код хоста від коду ядра, замінювати специфічні синтаксичні конструкції у коді хоста на стандартні;
– передавати код хоста на компіляцію стандартному компілятору системи;
– компілювати код пристрою;
– об'єднувати результати попередніх компіляцій в одну програму.

У результаті чого, при написанні коду хоста, ми маємо можливість використовувати мовні конструкції та стандартні бібліотеки останньої версії C++. Своєю чергою, розглянуто ряд основних обмежень, які накладає компілятор NVCC на код пристрою, недотримання яких може призвести до помилки компіляції або невизначеній поведінці при виконанні. У кінці цього розділу розглянули різні методи керування й доступу до пам'яті, а також різні техніки для оптимізації передачі даних між хостом та графічним процесором.

У третьому розділі ми на практиці розглянули різні методи оптимізації алгоритмів, використовуючи усі переваги нашої гетерогенної системи, яка складається з двох 64 ядерних центральних та одного графічного NVIDIA A100 процесорів. Для кожного реалізації були проведені заміри часу їх виконання, і де потрібно для кращого розуміння таких показників, провели повне профілювання програми за допомогою утиліти Nsight Systems. Для кожної задачі спершу приводились однопотокові та паралельні реалізації, які для обчислень використовували тільки CPU. А далі намагалися покращити час виконання, використовуючи переваги графічних процесорів.

Через простоту та наочність реалізації, першим було вибрано алгоритм знаходження скалярного добутку двох векторів. Початкова версія з використанням технології CUDA уже показала покращення у часі виконання, порівнюючи її з однопотоковою реалізацією алгоритму, але не показала кращі показники продуктивності, ніж у паралельної версії на CPU. Хоча потрібно помітити, що наша гетерогенна система містить два потужних центральних процесорів, які у сумі мають 128 фізичних ядер, та 256 – віртуальних. Що не є

розповсюдженим явищем для середньостатистичних гетерогенних систем. Це означає, що на інших системах наша перша CUDA-версія алгоритму може показати кращий результат, ніж паралельна CPU реалізація. Далі ми провели профілювання нашої версії, та визначили, що 98.8% часу GPU затрачає на переміщення даних з хоста. Тому було прийнято рішення замінити на хості пам'ять зі сторінковою організацією на «закріплену», що прискорило виконання нашої програми удвічі, наблизивши її продуктивність (і навіть трішки обігнавши) до паралельної реалізації на нашій системі. Далі наводяться реалізації з використанням декількох потоків операцій у зв'язці з асинхронною передачею даних, та версії алгоритму з «закріпленою» відображеною пам'яттю. Але вони не дали в результаті відчутного пришвидшення. Тому було зроблено висновок, що для задач, у яких складність часу виконання еквівалентна кількості вхідних даних, все-таки можливо отримати кращу продуктивність, використовуючи переваги графічного процесора. Але це покращення не є колосальним. Але потрібно помітити, що також існують гетерогенні системи з інтегрвоними графічнимим процесорами, які мають фізично спільну пам'ять між хостом і пристроє. У таких системах теоретично можна досягти колосального пришвидшення для цієї задачі, у силу відсутності неодбхідності передачі даних. У даному випадку для таких систем потрібно використовувати або «закріплену» відображену пам'ять, або уніфіковану. Тестування часу виконання реалізації алгоримів у цій роботі на таких системах може стати логічним продовження даної роботи.

Наступні дві задачі уже мали кубічну часову складність та квадратичну простору складність вхідних даних, що давало надію на кращі результи пришвидшення програми. Перший задача – множення двох прямокутних матриць, яка є дуже класичною та наглядною у для гетерогенних систем з графічним процесором. Замінивши два зовнішні цикли ітеративної версії, на виклик ядра у двовимірні сітці з двовимірних блоків, було досягнуто пришвидчення часу виконання більше ніж у сто разів. Показано, що із

зібльшенням об'єму вхідних даних, час виконання ядра домінував над часом передачі даних. Тому й втрачається сенс покращення продуктивності даногої реалізації методами оптимізаці передачі даних. Також була спроба використання як і SIMD, так і MIMD паралелізму у програмі. Але дана реалізація алгоритму не показала покращення часу виконання, тому була відкинута.

Останнім було покращено алгоритм Флойда – Воршелла, для уже менш класичної пролеми для відеокарт – задачі про найкоротший шлях в орієнтованому зваженому граф. Для неї уже шляхом заміни двох внутрішніх циклів у алгоримі Флойда – Воршелла, на виклик ядра у двовимірні сітці з двовимірних блоків, було досягнуто прискорення у 20-30 разів відносно паралельного алгоритму на CPU.

Отже, задачі, де часова складніст виконання більша за просторову складність вхідних даних, мають більший потенціал до покращення на гетерогенних системах з дискретними відеокартами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Al-Mouhamed M. A review of CUDA optimization techniques and toolsfor structured grid computing / M. Al-Mouhamed, A. Khan, N. Mohammad. // Springer. – 2019.

2. Kruliš M. Detailed Analysis and Optimization of CUDA K-means Algorithm / M. Kruliš, M. Kratochvíl. // ICPP'20: Proceedings of the 49th International Conference on Parallel Processing. – 2020. – С. 1–11.

3. Chen L. Research on Programming Model and Compilation Optimization Technology of Multi-Core GPU / Liyan Chen. // Journal of Physics. – 2022.

4. The Celerity High-level API: C++20 for Accelerator Clusters / P.Thoman, F. Tischler, P. Salzmann, P. Fahringer. // International Journal of Parallel Programming. – 2022. – №50. – С. 341–359.

5. Flynn M. Very High-speed Computing Systems / Michael J. Flynn. // Proceedings of the IEEE. – 1966. – №54. – С. 1901–1909.

6. Flynn M. Some Computer Organizations and Their Effectiveness / Michael J. Flynn. // IEEE Transactions on Computers. – 1972. – С. 948 – 960.

7. Сайт консорціуму Khronos Group. Головна сторінка технології OpenCL. [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/opencl/.

8. Banger R. OpenCL Programming by Example / R. Banger, K. Bhattacharyya., 2013. – 304 с.

9. Сайт «Khronos Registry»: Сторінка специфікації OpenCL. [Електронний ресурс]. Режим доступу: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html

10. Grossman M. Professional CUDA C Programming / M. Grossman, J. Cheng, T. McKercher., 2014. – 528 с.

11. CUDA C++ Programming Guide [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

12. Karimi K. A Performance Comparison of CUDA and OpenCL / K. Karimi, N. Dickson, F. Hamze. – 2011.

13. Fang J. A Comprehensive Performance Comparison of CUDA and OpenCL / J. Fang, A. Varbanescu, H. Sips. // IEEE. – 2011.

14. Mahapatra Manas. Computer Science and Engineering: Магістерська кваліфікаційна робота / [Mahapatra Manas]; [Korra Sathya Babu]. - Odisha: National Institute of Technology Rourkela, 2015. - 39.

15. Sanders J. CUDA by Example: An Introduction to General-Purpose GPU Programming / J. Sanders, E. Kandrot., 2010. – 320 с. – (1).

16. NVIDIA CUDA Compiler Driver NVCC [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

17. Parallel Thread Execution ISA [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nvidia.com/cuda/parallel-thread-execution/.

18. CUDA Binary Utilities [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nvidia.com/cuda/cuda-binary-utilities/.

19. How to Overlap Data Transfers in CUDA C/C++ [Електронний ресурс] – Режим доступу до ресурсу: https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/.

20. Introduction to Algorithms / T.Cormen, C. Leiserson, R. Rivest, C. Stein., 2009. – 1292 с. – (3).

Додаток А

Код реалізації однопотокового алгоритму знаходження скалярного добутку двох векторів на CPU

```
using Number = long double;
using Vector = std::vector<Number>;

namespace cpu_single_thread
{
    Number dotProduct(const Vector& v1, const Vector& v2)
    {
        if (v1.size() != v2.size()) {
            throw std::runtime_error("Vectors must have the
                                      same dimensions.");
        }

        Number result = 0;
        for (size_t i = 0; i < v1.size(); i++) {
            result += v1[i] * v2[i];
        }

        return result;
    }

} // cpu_single_thread

int main ()
{
    Vector v1, v2;

    // entering data ...

    auto result = dotProduct(v1, v2);

    // ...
}
```

Додаток Б

Код реалізації паралельного алгоритму знаходження скалярного добутку двох векторів на CPU

```
using Number = long double;
using Vector = std::vector<Number>;

namespace cpu_multi_thread
{

    Number dotProduct(const Vector& v1, const Vector& v2)
    {
        const size_t numThreads = std::thread::hardware_concurrency();
        const size_t chunkSize = v1.size() / numThreads;

        std::vector<Number> partialResults(numThreads, 0);

        auto threadFunc = [&](size_t begin, size_t end, size_t threadId) {
            auto& sum = partialResults[threadId];
            for (size_t i = begin; i < end; i++) {
                sum += v1[i] * v2[i];
            }
        };

        std::vector<std::thread> threads;
        for (size_t i = 0; i < numThreads; i++) {
            const size_t begin = i * chunkSize;
            const size_t end = std::min(begin + chunkSize, v1.size());
            threads.emplace_back(threadFunc, begin, end, i);
        }

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

        Number result = 0;

        for (auto partialResult : partialResults) {
            result += partialResult;
        }
```

```
        return result;
    }

} // cpu_multi_thread

int main ()
{
    Vector v1, v2;

    // entering data ...

    auto result = cpu_multi_thread::dotProduct(v1, v2);

    // ...
}
```

Додаток В

Код реалізації функції CUDA-ядра для знаходження скалярного добутку двох векторів

```
constexpr size_t threadsPerBlock = 1024;

namespace kernel
{
    __global__ void dot(const Number* v1, const Number* v2, Number* partSum,
       size_t vectorDim)
    {
        __shared__ Number cache[threadsPerBlock];
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        int cacheIndex = threadIdx.x;

        Number temp = 0;
        while (tid < vectorDim) {
            temp += v1[tid] * v2[tid];
            tid += blockDim.x * gridDim.x;
        }

        cache[cacheIndex] = temp;

        __syncthreads();

        int i = blockDim.x/2;
        while (i != 0){
            if (cacheIndex < i)
                cache[cacheIndex] += cache[cacheIndex + i];
            __syncthreads();
            i /= 2;
        }

        if (cacheIndex == 0)
            partSum[blockIdx.x] = cache[0];
    }
} // kernel
```

Додаток Г

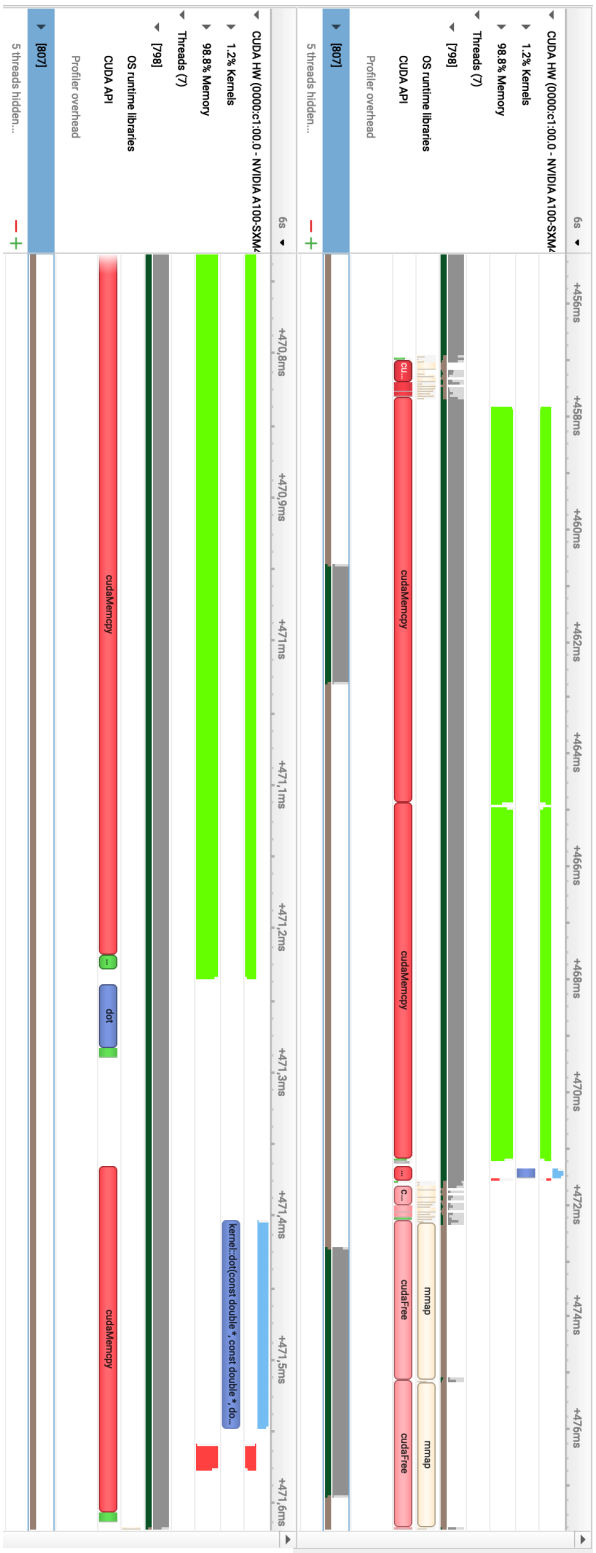
Код реалізації функції CUDA-ядра для знаходження скалярного добутку двох векторів

```
const size_t maxGridSize = getMaxGridSize<0>();

namespace gpu_single_stream
{

    Number dotProduct(const Number* v1, const Number* v2, size_t vectorDim)
    {
        using defer = std::shared_ptr<void>;

        const size_t blocksPerGrid = std::min<size_t>(maxGridSize,
                                                      (vectorDim +
                                                         threadsPerBlock-1) /
                                                         threadsPerBlock);

        Number *dev_v1, *dev_v2, *dev_partSum;

        cudaMalloc((void**)&dev_v1, vectorDim*sizeof(Number));
        cudaMalloc((void**)&dev_v2, vectorDim*sizeof(Number));
        cudaMalloc((void**)&dev_partSum, blocksPerGrid*sizeof(Number));

        defer _(nullptr, [&] (...) {
            cudaFree(dev_v1);
            cudaFree(dev_v2);
            cudaFree(dev_partSum);
        });

        cudaMemcpy(dev_v1, v1, vectorDim*sizeof(Number), cudaMemcpyHostToDevice
           );
        cudaMemcpy(dev_v2, v2, vectorDim*sizeof(Number), cudaMemcpyHostToDevice
           );

        kernel::dot<<<blocksPerGrid, threadsPerBlock>>>(dev_v1, dev_v2,
           dev_partSum, vectorDim);

        std::vector<Number> partialSum(blocksPerGrid);
```

```
        cudaMemcpy(partialSum.data(), dev_partSum, blocksPerGrid*sizeof(Number)
            , cudaMemcpyDeviceToHost);

        Number res = 0;
        for(size_t i = 0; i < blocksPerGrid; ++i) {
            res += partialSum[i];
        }

        return res;
    }

} // gpu_single_stream

int main ()
{
    Vector v1, v2;

    // entering data ...

    auto result = gpu_single_stream::dotProduct(v1, v2, v1.size());

    // ...
}
```

Додаток Д

Звіт профілювання утилітою Nsight Systems CUDA-програми знаходження скалярного добутку двох векторів



Додаток Е

Код хоста CUDA-програми знаходження скалярного добутку двох векторів з використанням та «закріпленої» пам'яті

```
const size_t maxGridSize = getMaxGridSize<0>();

namespace gpu_single_stream
{

    Number dotProduct(const Number* v1, const Number* v2, size_t vectorDim)
    {
        using defer = std::shared_ptr<void>;

        const size_t blocksPerGrid = std::min<size_t>(maxGridSize,
                                                      (vectorDim +
                                                         threadsPerBlock-1) /
                                                         threadsPerBlock);

        Number *dev_v1, *dev_v2, *dev_partSum;

        cudaMalloc((void**)&dev_v1, vectorDim*sizeof(Number));
        cudaMalloc((void**)&dev_v2, vectorDim*sizeof(Number));
        cudaMalloc((void**)&dev_partSum, blocksPerGrid*sizeof(Number));

        defer _(nullptr, [&] (...) {
            cudaFree(dev_v1);
            cudaFree(dev_v2);
            cudaFree(dev_partSum);
        });

        cudaMemcpy(dev_v1, v1, vectorDim*sizeof(Number), cudaMemcpyHostToDevice
           );
        cudaMemcpy(dev_v2, v2, vectorDim*sizeof(Number), cudaMemcpyHostToDevice
           );

        kernel::dot<<<blocksPerGrid, threadsPerBlock>>>(dev_v1, dev_v2,
           dev_partSum, vectorDim);

        std::vector<Number> partialSum(blocksPerGrid);
```

```
        cudaMemcpy(partialSum.data(), dev_partSum, blocksPerGrid*sizeof(Number)
            , cudaMemcpyDeviceToHost);

        Number res = 0;
        for(size_t i = 0; i < blocksPerGrid; ++i) {
            res += partialSum[i];
        }

        return res;
    }

} // gpu_single_stream

int main ()
{
    using defer = std::shared_ptr<void>;

    size_t vectorDim;

    // entering vector dimension ...

    Number *v1Pinned, *v2Pinned;

    cudaMallocHost((void**)&v1Pinned, vectorDim * sizeof(Number));
    cudaMallocHost((void**)&v2Pinned, vectorDim * sizeof(Number));

    auto _ = defer(nullptr, [&] (...) {
        cudaFreeHost(v1Pinned);
        cudaFreeHost(v2Pinned);
    });

    // entering vectors ...

    auto result = gpu_single_stream::dotProduct(v1Pinned, v2Pinned, vectorDim);

    // ...
}
```

Додаток Ж

Код хоста CUDA-програми знаходження скалярного добутку двох векторів з використанням декількох потоків команд

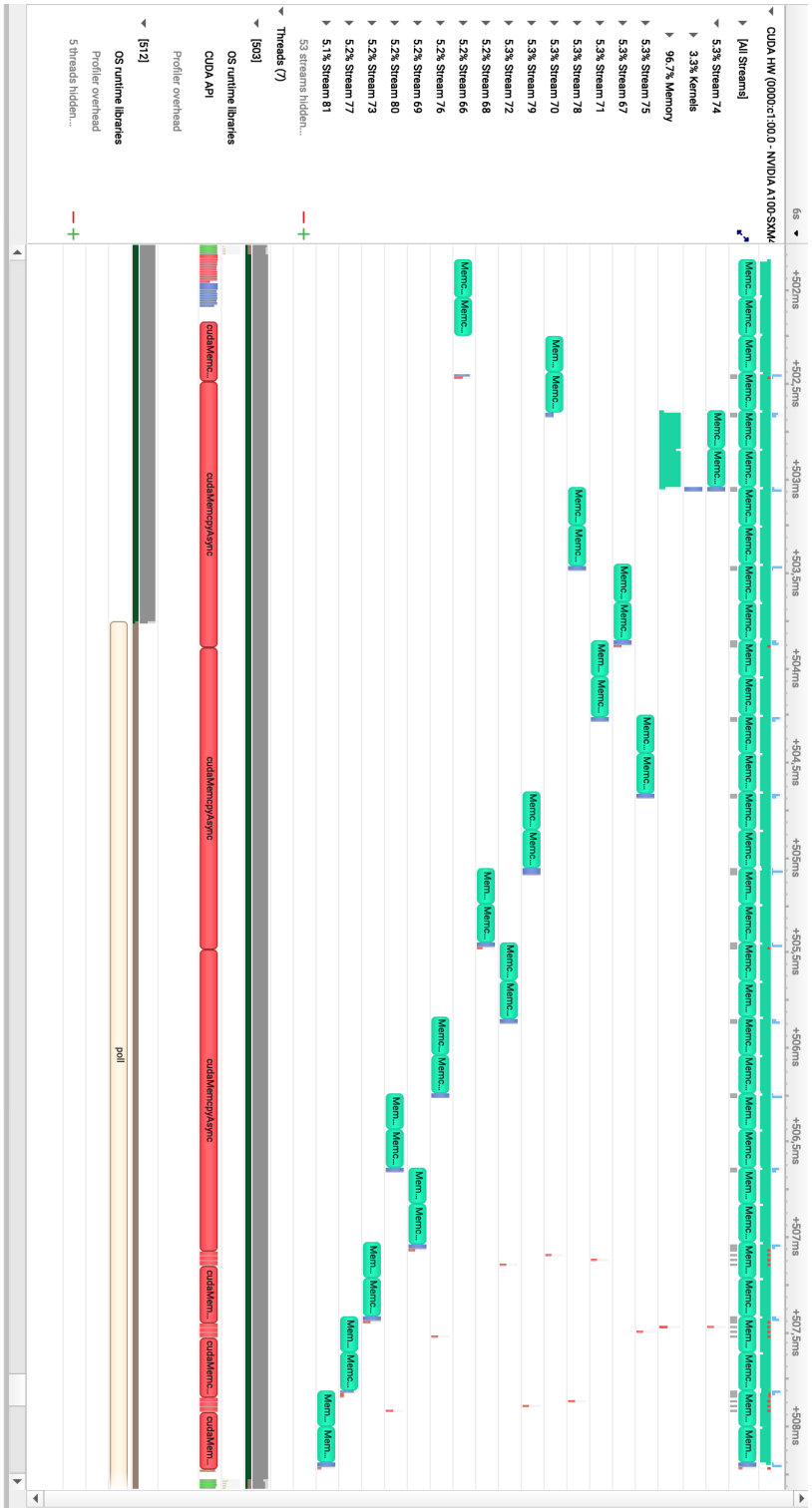
```
class CudaStreamWrapper
{
public:
    CudaStreamWrapper()
        { cudaStreamCreate(&fStream); }
    ~CudaStreamWrapper()
        { cudaStreamDestroy(fStream); }
    auto get() -> const cudaStream_t&
        { return fStream; }
private:
    cudaStream_t fStream;
};

const size_t maxGridSize = getMaxGridSize<0>();

namespace gpu_multiple_streams
{
    Number dotProduct(const Number* v1, const Number* v2, size_t vectorDim)
    {
        using defer = std::shared_ptr<void>;

        const size_t blocksPerGrid = std::min<size_t>(maxGridSize, (vectorDim +
            threadsPerBlock-1) / threadsPerBlock);

        Number *dev_v1, *dev_v2, *dev_partSum;

        cudaMalloc((void**)&dev_v1, vectorDim*sizeof(Number));
        cudaMalloc((void**)&dev_v2, vectorDim*sizeof(Number));
        cudaMalloc((void**)&dev_partSum, blocksPerGrid*sizeof(Number));

        defer _(nullptr, [&] (...) {
            cudaFree(dev_v1);
            cudaFree(dev_v2);
            cudaFree(dev_partSum);
        });
```

```
const size_t nStreams = std::min<size_t>(blocksPerGrid, 16);
const size_t nGeneralElemPerStream = vectorDim / nStreams;
std::vector<CudaStreamWrapper> streams(nStreams);

for (size_t i = 0; i < nStreams; ++i) {
    const size_t offset = i * nGeneralElemPerStream;
    const size_t nElemPerStream = i == (nStreams - 1) ? vectorDim -
       nGeneralElemPerStream * (nStreams - 1) : nGeneralElemPerStream;
    const size_t nBytes = nElemPerStream * sizeof(Number);
    cudaMemcpyAsync(dev_v1 + offset, v1 + offset, nBytes,
       cudaMemcpyHostToDevice, streams[i].get());
    cudaMemcpyAsync(dev_v2 + offset, v2 + offset, nBytes,
       cudaMemcpyHostToDevice, streams[i].get());
}

const size_t nGeneralBlocksPerStream  = blocksPerGrid / nStreams;
for (size_t i = 0; i < nStreams; ++i)
{
    const size_t offset = i * nGeneralElemPerStream;
    const size_t blockOffset = i * nGeneralBlocksPerStream;
    const size_t nBlocksPerStream = i == (nStreams - 1) ? blocksPerGrid
        - nGeneralBlocksPerStream * (nStreams - 1) :
       nGeneralBlocksPerStream;
    const size_t nElemPerStream = i == (nStreams - 1) ? vectorDim -
       nGeneralElemPerStream * (nStreams - 1) : nGeneralElemPerStream;
    kernel::dot<<<nBlocksPerStream, threadsPerBlock, 0, streams[i].get
       ()>>>(dev_v1 + offset, dev_v2 + offset, dev_partSum +
       blockOffset, nElemPerStream);
}

std::vector<Number> partialSum(blocksPerGrid);
for (size_t i = 0; i < nStreams; ++i)
{
    const size_t blockOffset = i * nGeneralBlocksPerStream;
    const size_t nBlocksPerStream = i == (nStreams - 1) ? blocksPerGrid
        - nGeneralBlocksPerStream * (nStreams - 1) :
       nGeneralBlocksPerStream;
    const size_t nBytes = nBlocksPerStream * sizeof(Number);
    cudaMemcpyAsync(partialSum.data() + blockOffset, dev_partSum +
       blockOffset, nBytes, cudaMemcpyDeviceToHost, streams[i].get());
```

```
        }

        cudaDeviceSynchronize();

        Number res = 0;
        for(size_t i = 0; i < blocksPerGrid; ++i) {
            res += partialSum[i];
        }

        return res;
    }
} // gpu_multiple_streams

int main ()
{
    using defer = std::shared_ptr<void>;

    size_t vectorDim;

    // entering vector dimension ...

    Number *v1Pinned, *v2Pinned;

    cudaMallocHost((void**)&v1Pinned, vectorDim * sizeof(Number));
    cudaMallocHost((void**)&v2Pinned, vectorDim * sizeof(Number));

    auto _ = defer(nullptr, [&] (...) {
        cudaFreeHost(v1Pinned);
        cudaFreeHost(v2Pinned);
    });

    // entering vectors ...

    auto result = gpu_multiple_streams::dotProduct(v1Pinned, v2Pinned,
       vectorDim);

    // ...
}
```

Додаток И

Звіт профілювання утилітою Nsight Systems CUDA-програми знаходження скалярного добутку двох векторів з використанням декількох потоків команд



Додаток К

Код однопотокової та паралельної функції множення двох матриць

```
namespace cpu_single_thread_iterative
{
    Matrix matrixMultiplication(const Matrix& mx1, const Matrix& mx2)
    {
        // m1 - NxM, m2 - MxP, res - NxP
        const size_t n = mx1.size();
        const size_t m = mx1[0].size();
        const size_t p = mx2[0].size();

        Matrix res(n, std::vector<Number>(p));
        for (size_t i = 0; i < n; ++i) {
            for (size_t j = 0; j < p; ++j) {
                size_t sum = 0;
                for (size_t k = 0; k < m; ++k) {
                    sum += mx1[i][k] * mx2[k][j];
                }
                res[i][j] = sum;
            }
        }
        return res;
    }
} // cpu_single_thread_iterative

namespace cpu_multi_thread_iterative
{

    Matrix matrixMultiplication(const Matrix& mx1, const Matrix& mx2)
    {
        // m1 - NxM, m2 - MxP, res - NxP
        const size_t n = mx1.size();
        const size_t m = mx1[0].size();
        const size_t p = mx2[0].size();

        Matrix res(n, std::vector<Number>(p));

        const size_t numThreads = 256;
```

```
        const size_t rowsPerThread = n / numThreads;

        std::vector<std::thread> threads;
        threads.reserve(numThreads);

        for (size_t t = 0; t < numThreads; ++t) {
            threads.emplace_back([&mx1, &mx2, &res, n, m, p, rowsPerThread, t
               ]() {
                const size_t startRow = t * rowsPerThread;
                const size_t endRow = (t == numThreads - 1) ? n : startRow +
                   rowsPerThread;

                for (size_t i = startRow; i < endRow; ++i) {
                    for (size_t j = 0; j < p; ++j) {
                        size_t sum = 0;
                        for (size_t k = 0; k < m; ++k) {
                            sum += mx1[i][k] * mx2[k][j];
                        }
                        res[i][j] = sum;
                    }
                }
            });
        }

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

        return res;
    }

} // cpu_multi_thread_iterative
```

Додаток Л

Код CUDA-програми для множення двох матриць

```
const size_t maxGridSize = getMaxGridSize<0>();

constexpr auto blockSize2d = dim3(32, 32);

namespace kernel
{
    __global__ void matrixMultiplication(const Number* mx1, const Number* mx2,
       Number* res,
                                         size_t n, size_t m, size_t p)
    {
        const size_t starRow = blockIdx.y * blockDim.y + threadIdx.y;
        const size_t starCol = blockIdx.x * blockDim.x + threadIdx.x;

        size_t row = starRow;
        while (row < n) {
            size_t col = starCol;
            while (col < p) {
                Number sum = 0.0f;
                for (size_t k = 0; k < m; ++k) {
                    sum += mx1[row * m + k] * mx2[k * p + col];
                }
                res[row * p + col] = sum;
                col += gridDim.x * blockDim.x;
            }
            row += gridDim.y * blockDim.y;
        }
    }
} // kernel

namespace gpu_pinned
{
    void matrixMultiplication(const Number* mx1, const Number* mx2, Number* res
       ,
                              size_t n, size_t m, size_t p)
    {
        using defer = std::shared_ptr<void>;
```

```
        Number* d_mx1;
        Number* d_mx2;
        Number* d_res;
        cudaMalloc((void**)&d_mx1, n * m * sizeof(Number));
        cudaMalloc((void**)&d_mx2, m * p * sizeof(Number));
        cudaMalloc((void**)&d_res, n * p * sizeof(Number));

        defer _(nullptr, [&] (...) {
            cudaFree(d_mx1);
            cudaFree(d_mx2);
            cudaFree(d_res);
        });

        cudaMemcpy(d_mx1, mx1, n * m * sizeof(Number), cudaMemcpyHostToDevice);
        cudaMemcpy(d_mx2, mx2, m * p * sizeof(Number), cudaMemcpyHostToDevice);

        const size_t gridX = std::min(maxGridXSize, (p + blockSize2d.x - 1) /
            blockSize2d.x);
        const size_t gridY = std::min(maxGridYSize, (n + blockSize2d.y - 1) /
            blockSize2d.y);
        const auto gridSize = dim3(gridX, gridY);

        kernel::matrixMultiplication<<<gridSize, blockSize2d>>>(d_mx1, d_mx2,
            d_res, n, m, p);

        cudaDeviceSynchronize();

        cudaMemcpy(res, d_res, n * p * sizeof(Number), cudaMemcpyDeviceToHost);
    }
} // gpu_pinned

int main ()
{
    using defer = std::shared_ptr<void>;

    size_t n, m, p;

    // entering n, m, p ...

    Number *mx1Pinned, *mx2Pinned, *resPinned;
```

```
    cudaMallocHost((void**)&mx1Pinned, n * m * sizeof(Number));
    cudaMallocHost((void**)&mx2Pinned, m * p * sizeof(Number));
    cudaMallocHost((void**)&resPinned, n * p * sizeof(Number));
    auto _ = defer(nullptr, [&] (...) {
        cudaFreeHost(mx1Pinned);
        cudaFreeHost(mx2Pinned);
        cudaFreeHost(resPinned);
    });

    // entering matrix ...

    gpu_pinned::matrixMultiplication(mx1Pinned, mx2Pinned, resPinned, n, m, p);
    // ...
}
```

Додаток М

Код CUDA-програми для множення двох матриць з використанням CPU паралелізму

```
const size_t maxGridSize = getMaxGridSize<0>();

constexpr auto blockSize2d = dim3(32, 32);

namespace kernel
{
    __global__ void dotProduct(const Number* mx1, const Number* mx2, Number*
        partialSum,
                               size_t i, size_t j, size_t n, size_t m, size_t p
                                  )
    {
        __shared__ Number cache[blockSize1d];
        size_t k = threadIdx.x + blockIdx.x * blockDim.x;
        const size_t cacheIndex = threadIdx.x;

        Number temp = 0;
        while (k < m) {
            temp += mx1[i * m + k] * mx2[k * p + j];
            k += blockDim.x * gridDim.x;
        }

        cache[cacheIndex] = temp;

        __syncthreads();

        size_t workingCacheIndex = blockDim.x/2;
        while (workingCacheIndex != 0){
            if (cacheIndex < workingCacheIndex)
                cache[cacheIndex] += cache[cacheIndex + workingCacheIndex];
            __syncthreads();
            workingCacheIndex /= 2;
        }

        if (cacheIndex == 0)
            partialSum[blockIdx.x] = cache[0];
    }
```

```
} // kernel

namespace multithread_cpu_gpu_pinned
{
    Matrix matrixMultiplication(const Number* mx1, const Number* mx2,
                                size_t n, size_t m, size_t p)
    {
        using defer = std::shared_ptr<void>;

        // Allocate device memory
        Number* d_mx1;
        Number* d_mx2;
        cudaMalloc((void**)&d_mx1, n * m * sizeof(Number));
        cudaMalloc((void**)&d_mx2, m * p * sizeof(Number));

        defer _(nullptr, [&] (...) {
            cudaFree(d_mx1);
            cudaFree(d_mx2);
        });

        cudaMemcpy(d_mx1, mx1, n * m * sizeof(Number), cudaMemcpyHostToDevice);
        cudaMemcpy(d_mx2, mx2, m * p * sizeof(Number), cudaMemcpyHostToDevice);

        Matrix res(n, std::vector<Number>(p));

        const size_t gridSize1d = std::min<size_t>(maxGridXSize, (m +
           blockSize1d - 1) / blockSize1d);
        auto threadFunc = [&] (size_t begin, size_t end) {
            Number* partialSum, *d_partialSum;
            cudaMallocHost((void**)&partialSum, gridSize1d * sizeof(Number));
            cudaMalloc((void**)&d_partialSum, gridSize1d * sizeof(Number));
            defer _(nullptr, [&] (...) {
                cudaFreeHost(partialSum);
                cudaFree(d_partialSum);
            });

            for (size_t i = begin; i < end; ++i) {
                for (size_t j = 0; j < p; ++j) {
                    kernel::dotProduct<<<gridSize1d, blockSize1d>>>(mx1, mx2,
                       d_partialSum, i, j, n, m, p);
```

```
                    cudaMemcpy(partialSum, d_partialSum, gridSize1d * sizeof(
                        Number), cudaMemcpyDeviceToHost);
                    res[i][j] = 0;
                    for (size_t g = 0; g < gridSize1d; ++g) {
                        res[i][j] += partialSum[g];
                    }
                }
            }
        };

        const size_t rowsPerThread = std::max<size_t>(n / numThreads, 1);

        std::vector<std::thread> threads;
        size_t offset = 0;
        while (offset < n) {
            const size_t begin = offset;
            const size_t end = std::min<size_t>(offset + rowsPerThread, n);
            threads.emplace_back(threadFunc, begin, end);
            offset = end;
        }

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

        return res;
    }

} // multithread_cpu_gpu_pinned

int main ()
{
    using defer = std::shared_ptr<void>;

    size_t n, m, p;

    // entering n, m, p ...

    Number *mx1Pinned, *mx2Pinned, *resPinned;
    cudaMallocHost((void**)&mx1Pinned, n * m * sizeof(Number));
```

```
    cudaMallocHost((void**)&mx2Pinned, m * p * sizeof(Number));
    cudaMallocHost((void**)&resPinned, n * p * sizeof(Number));
    auto _ = defer(nullptr, [&] (...) {
        cudaFreeHost(mx1Pinned);
        cudaFreeHost(mx2Pinned);
        cudaFreeHost(resPinned);
    });

    // entering matrix ...

    multithread_cpu_gpu_pinned::matrixMultiplication(mx1Pinned, mx2Pinned,
       resPinned, n, m, p);
    // ...
}
```

Додаток Н

Код оптимізації алгоритму Флойда – Воршелла за допомогою CUDA

```
constexpr auto blockSize2d = dim3(16, 16);
const size_t maxGridXSize = getMaxGridSize<0>();
const size_t maxGridYSize = getMaxGridSize<1>();

namespace kernel
{
    __global__ void floydWarshall(Number* dis, size_t numVertices, size_t k)
    {
        const size_t starRow = blockIdx.y * blockDim.y + threadIdx.y;
        const size_t starCol = blockIdx.x * blockDim.x + threadIdx.x;

        size_t i = starRow;
        while (i < numVertices) {
            size_t j = starCol;
            while (j < numVertices) {
                if (dis[i * numVertices + j] > dis[i * numVertices + k] + dis[k
                    * numVertices + j]) {
                    dis[i * numVertices + j] = dis[i * numVertices + k] + dis[k
                        * numVertices + j];
                    j += gridDim.x * blockDim.x;
                }
            }
            i += gridDim.y * blockDim.y;
        }
    }
} // kernel

namespace gpu
{
    void floydWarshallAlg(const Number* graph, Number* res, size_t numVertices)
    {
        using defer = std::shared_ptr<void>;

        Number* d_res;
        cudaMalloc((void**)&d_res, numVertices * numVertices * sizeof(Number));

        defer _(nullptr, [&] (...) {
```

```
            cudaFree(d_res);
        });

        cudaMemcpy(d_res, graph, numVertices * numVertices * sizeof(Number),
            cudaMemcpyHostToDevice);

        const size_t gridX = std::min(maxGridXSize, (numVertices + blockSize2d.
            x - 1) / blockSize2d.x);
        const size_t gridY = std::min(maxGridYSize, (numVertices + blockSize2d.
            y - 1) / blockSize2d.y);
        const auto gridSize = dim3(gridX, gridY);

        for (size_t k = 0; k < numVertices; ++k) {
            kernel::floydWarshall<<<gridSize, blockSize2d>>>(d_res, numVertices
                , k);
        }
        cudaDeviceSynchronize();

        cudaMemcpy(res, d_res, numVertices * numVertices * sizeof(Number),
            cudaMemcpyDeviceToHost);
    }
} //gpu

int main ()
{
    using defer = std::shared_ptr<void>;
    size_t v;
    // entering v ...
    Number *graphPinned, *disPinned;
    cudaMallocHost((void**)&graphPinned, v * v * sizeof(Number));
    cudaMallocHost((void**)&disPinned, v * v * sizeof(Number));
    auto _ = defer(nullptr, [&] (...) {
        cudaFreeHost(graphPinned);
        cudaFreeHost(disPinned);
    });
    // entering graph ...
    gpu::floydWarshallAlg(graphPinned, disPinned, v);
    // ...
}
```