Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

Галузь знань 12 – Інформаційні технології

Спеціальність 123 –Комп'ютерна інженерія

на тему «Метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA»

КвРКІП. 2202133.22.02.30 ПЗ

Виконав: студент 2 курсу, група КІ2м-22-2 _____ Дзюбчик О.Л.
Підпис Ініціали, прізвище

Керівник доктор техн. наук, професор _____ Лисенко С.М.
Науковий ступінь, вчене звання Підпис Ініціали, прізвище

До захисту допускаю:
Зав. кафедри КІС, д.т.н., проф.
Т.О. Говорущенко
14 05 2024 р.

Хмельницький, 2024

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Освітній рівень МАГІСТР

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма ОСВІТНЬО-НАУКОВА ПРОГРАМА «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ

Зав. кафедри Т.О.Говорущенко

"01" 09 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Дзюбчику Олександру Леонідовичу

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA

Керівник проекту (роботи) Лисенко С.М., д.т.н., професор

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 01.01.2024 р. № 1

2. Строк подання студентом проекту (роботи) на кафедру 01.05.2024 р.

3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Аналіз відомих методів паралельної обробки даних, зокрема з використанням CUDA

Моделювання оцінки використання ресурсів та часу виконання ядра CUDA

Синтез методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA

Дослідження застосування методу оптимізації паралельної обробки даних з використанням CUDA

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

6. Консультанти розділів кваліфікаційної роботи магістра

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Лисенко С.М., професор кафедри КІІС		
Антиплагіат	Нічепорук А.О., доцент кафедри КІІС		

7. Дата видачі завдання «_01_»_____09_____2023р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) кваліфікаційної роботи магістра	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики КвРМ з керівником	01.09.2023	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.10.2023	виконано
3	Робота над розділом 1 – аналіз відомих моделей, методів за темою; постановка задачі	01.11.2023	виконано
4	Робота над розділом 2 – розробка моделей для вирішення поставленої задачі	01.12.2023	виконано
5	Робота над науковою статтею	01.02.204	виконано
6	Робота над розділом 3 – розробка методів для вирішення поставленої задачі	15.02.2024	виконано
7	Робота над розділом 4 – проектування та розробка ПЗ для вирішення поставленої задачі, експериментальна частина	01.04.204	виконано
8	Оформлення пояснювальної записки згідно вимог	18.04.2024	виконано
9	Попередній захист ДРМ	29.04.2024	виконано
10	Захист ДРМ на засіданні ЕК	До 15.05.2024	

Студент _____ Дзюбчик О. Л.
Підпис Ініціали, прізвище

Керівник роботи _____ Лисенко С. М.
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема кваліфікаційної роботи магістра: Метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA

Автор роботи: Дзюбчик Олександр Леонідович

Керівник роботи: Лисенко Сергій Миколайович

Пояснювальна записка: 82 с., 5 рис., 3 табл., 5 дод., 80 джерел.

АРХІТЕКТУРА CUDA, ОПТИМІЗАЦІЯ, ПАРАЛЕЛЬНА ОБРОБКА ДАНИХ

Об’єктом дослідження є оптимізація паралельної обробки даних.

Предметом дослідження є метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA.

Метою кваліфікаційної роботи магістра є оптимізація паралельної обробки даних згідно з кластерною архітектурою CUDA.

Для розв’язання поставлених задач використовуються основні методи та положення теорії комп’ютерних мереж та систем, системного аналізу, моделювання, методів аналізу даних, теорії дискретної математики, теорії паралельної обробки даних.

Наукова новизна отриманих результатів:

1. Набув подальшого розвитку метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA, який на відміну від відомих методів пропонує генералізований підхід без прив’язки до конкретного алгоритму, що дозволяє оптимізувати паралельної обробки даних.

2. Набули подальшого розвитку програмно-технічні засоби оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

Практична цінність отриманих результатів полягає у розробці апаратно-програмного засобу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

У першому розділі було проведено дослідження використання існуючих методів послідовної і паралельної обробки даних в різних доменних областях. У другому розділі представлено моделі, що дозволяють репрезентувати

використання ресурсів і час виконання ядра. Зокрема було визначено модель, яка описує правила оцінки ядра при відстеженні вартості виконання одного варпу. У третьому розділі було синтезовано метод оптимізації паралельної обробки даних з використанням кластерної архітектури CUDA та описано його застосування на основі алгоритмів FFT, kNN, і Дейкстри. У четвертому розділі подано дослідження застосування синтезованого методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA, відповідно до описаних кроків.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

CUDA - Compute Unified Device Architecture (програмно-апаратна архітектура паралельних обчислень).

FFT - Fast Fourier Transform (швидкий алгоритм обчислення дискретного перетворення Фур'є).

kNN - k-nearest neighbor method (метод k-найближчих сусідів).

CPU - Central processing unit (центральний процесор).

GPU - Graphics processing unit (графічний процесор).

SM - Streaming Multiprocessor (потоковий мультипроцесор).

ВСТУП

У сучасному світі обчислювальні технології розвиваються стрімкими темпами, а потреба в ефективній обробці великих обсягів даних стає дедалі актуальнішою. Інноваційні підходи в паралельній обробці даних, як-от кластерна архітектура CUDA від NVIDIA, відіграють ключову роль у досягненні нових висот в обчислювальній ефективності. Ці технології відкривають перед науковою спільнотою та індустрією нові можливості, сприяючи прогресу в різноманітних дослідницьких та промислових галузях. Використання GPU для оптимізації та паралельної обробки даних показало значні переваги у зменшенні часу виконання складних обчислювальних завдань. Архітектура CUDA стала важливим інструментом у розробці високопродуктивних обчислювальних систем, здатних вирішувати завдання, які раніше вважалися надзвичайно складними.

Дослідження у галузі оптимізації та паралельної обробки даних показує вагому роль архітектури CUDA в розвитку цих технологій. Архітектура CUDA, розроблена NVIDIA, перетворила графічні процесори на масивно-паралельні обчислювальні машини, які відкривають нові можливості для обробки великих даних та вирішення складних обчислювальних задач. Розвиток паралельної обробки даних за допомогою кластерної архітектури CUDA стає ключем до прискорення наукових досліджень та інженерних розробок, оскільки вона дозволяє ефективно використовувати обчислювальні потужності GPU. У зв'язку з цим, систематизація та глибоке розуміння принципів кластерної архітектури CUDA та методів паралельної обробки даних стає вирішальним фактором у вдосконаленні обчислювальних.

Актуальність роботи полягає в розробці методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

Мета даної роботи полягає в оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

Поставлена мета досягається розв’язанням таких основних задач:

– дослідити методи паралельної обробки даних, в тому числі із залученням архітектури CUDA;

– проаналізувати сучасні програмно-технічних засоби оптимізації паралельної обробки даних з використанням CUDA;

– дослідити та описати засоби балансування навантаження в інфраструктурі як послузі;

– удосконалити метод в оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.;

– реалізувати розроблений метод та дослідити результати оптимізації.

Об’єктом дослідження є оптимізація паралельної обробки даних.

Предметом дослідження є метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA.

Наукова новизна отриманих результатів:

1. Набув подальшого розвитку метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA, який на відміну від відомих методів пропонує генералізований підхід без прив’язки до конкретного алгоритму, що дозволяє оптимізувати паралельної обробки даних.

2. Набули подальшого розвитку програмно-технічні засоби оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

Практична цінність отриманих результатів. В результаті виконаного наукового дослідження було розроблено апаратно-програмні засоби оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

Для розв’язання поставлених задач використовуються основні положення теорії комп’ютерних мереж та систем, системного аналізу, моделювання, методів аналізу даних, теорії дискретної математики, теорії паралельної обробки даних.

За темою кваліфікаційної роботи магістра опубліковані тези у матеріалах конференції XXIV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів «Стан, досягнення та перспективи інформаційних систем і технологій» 18-19 квітня 2024 р., Одеса, Україна (Додаток 1).

1 АНАЛІЗ ВІДОМИХ МЕТОДІВ ОПТИМІЗАЦІЇ ПАРАЛЕЛЬНОЇ ОБРОБКИ ДАНИХ З ВИКОРИСТАННЯМ CUDA

1.1 Огляд та поняття паралельної обробки даних

Послідовна і паралельна обробка даних – це два основних підходи в обчислювальній техніці, які використовуються для обробки інформації [1]. Порівняння цих методів [2] важливе для розуміння їхньої придатності до різних типів завдань та обчислювальних середовищ [3]. Послідовна обробка, або серійна обробка, відбувається, коли процесор виконує одну обчислювальну задачу за раз. У цьому режимі [4,5] процеси запускаються один за одним, де кожен наступний процес починається лише після завершення попереднього. Цей підхід є простим у реалізації [6] та управлінні, оскільки не вимагає складної логіки координації між процесами. Проте, послідовна обробка може бути часозатратною, особливо при великих обсягах даних або складних обчислювальних завданнях [7-9], оскільки весь процес обробки залежить від одного виконавчого потоку.

Послідовна обробка даних залишається важливою у ситуаціях, де завдання мають сильні залежності, і результат однієї операції визначає параметри наступної [10-12]. Це особливо актуально в задачах, де необхідна висока точність і послідовність даних, наприклад, у фінансових розрахунках, де помилка в одному етапі може призвести до значних невідповідностей у кінцевих результатів [13]. У цих випадках, послідовна обробка допомагає уникнути помилок, пов'язаних з гонитвою за даними і змаганням за ресурси [14,15], забезпечуючи точне виконання кожної операції. Також послідовна обробка корисна при відладці складних систем, де важливо зрозуміти ланцюжок виконання задач [16-18], щоб ідентифікувати джерела помилок. Вона дозволяє виконувати задачі в контрольованому середовищі, де кожна дія може бути перевірена на коректність перед переходом до наступної [19]. Незважаючи на те, що послідовна обробка може бути повільнішою в порівнянні з паралельною обробкою для обробки великих обсягів даних, вона залишається незамінною в областях, де важлива стабільність та надійність [20,21].

Вона забезпечує простоту розуміння та відтворення процесів, що є ключовим для тестування та гарантування якості обчислювальних систем [22-24].

Паралельна обробка даних – це процес, у якому велике завдання розбивається на декілька менших, які виконуються одночасно на кількох обчислювальних вузлах або процесорах [25]. Це дозволяє значно збільшити швидкість обробки даних та ефективність системи, оскільки паралельне виконання задач зменшує загальний час на обробку інформації [26-28]. У випадку паралельної обробки, великі завдання розбиваються на менші підзадачі, що дозволяє ефективно використовувати ресурси обчислювальної системи, забезпечуючи вищу продуктивність, особливо для завдань, які можуть бути легко розподілені на незалежні частини [29].

Методи паралелізації включають розбиття завдань на менші підзадачі, розподіл даних між обчислювальними вузлами, та визначення залежностей між завданнями для оптимізації послідовності виконання.

Архітектури паралельних систем можуть бути класифіковані за типами: SIMD (Single Instruction, Multiple Data), MIMD (Multiple Instruction, Multiple Data), MISD (Multiple Instruction, Single Data). Кожна з цих архітектур визначає спосіб взаємодії між обчислювальними вузлами та розподіл завдань.

MIMD (Multiple Instruction, Multiple Data):

- Дозволяє різним процесорам виконувати різні інструкції на різних наборах даних.
- Підходить для різноманітних задач обчислень, підтримує багатозадачність та паралелізм на рівні задач.
- Використовується в багатопроцесорних та багатоядерних системах.

Архітектура Single Instruction Multiple Data (SIMD), що підтримується різними високопродуктивними обчислювальними платформами, ефективно використовує паралелізм на рівні даних. Модель SIMD використовується в традиційних процесорах, спеціалізованих векторних системах та прискорювачах, таких як графічні процесори, векторні розширення. Вона забезпечує пропускну здатність в додатках з інтенсивними обчисленнями та паралельною обробкою даних. Незважаючи на схожість принципів обробки даних між цими архітектурами,

перенесення різних моделей програмування між розглянутими платформами є складним завданням. Крім того, покращення програмованості цих архітектур є важливою особливістю для використання їхньої зростаючої обчислювальної потужності та спрощення складності програмування. У статті [30] розглядаються основні принципи оптимізації методів виконання асинхронних багатоінструкційних обчислень з багатьма даними (MIMD) на SIMD прискорювачах. Також розглянуто декілька парадигм програмування для GPU та інтерфейсів прикладного програмування (API) і класифіковано ці фреймворки на різні групи на основі їхніх критеріїв.

MISD (Multiple Instruction, Single Data):

- Рідкісна архітектура, де кілька інструкцій обробляють одні й ті ж дані.
- Теоретично корисна для деяких спеціалізованих застосувань, наприклад, у відмовостійких обчислювальних системах.

У статті [31] досліджується Splitfed Learning (SFL) , що є результатом поєднання федеративного навчання (Federated Learning, FL) [32] і розділеного навчання (Split Learning, SL) [33]. Дозволяючи одночасне виконання локального тренування на клієнтських пристроях і агрегацію на сервері, як у FL, а також розділення процесу навчання між клієнтом і сервером, як у SL, можна забезпечити кращу ефективність і конфіденційність, а також знизити навантаження на мережу, оскільки передається менше даних порівняно з традиційними методами навчання. Таким чином, SFL може вважатися ефективнішим рішенням завдяки паралельній обробці даних і оптимізації обміну інформацією.

Однією з ключових відмінностей між послідовною та паралельною обробкою є швидкість виконання. Паралельна обробка часто є набагато швидшою за послідовну, оскільки вона здатна виконувати кілька задач одночасно. Це робить паралельну обробку ідеальною для великомасштабних обчислень та обробки великих даних.

Ще одна важлива різниця полягає в складності реалізації. Паралельна обробка вимагає більш складного планування та координації між процесами, що може збільшити складність програмування та потенційний ризик виникнення

помилок у коді. Послідовна обробка, з іншого боку, є більш простою у розробці та відладці, але може бути неефективною для завдань, які потребують інтенсивних обчислень.

Масштабованість та гнучкість також є ключовими аспектами, що дозволяють системам адаптуватися до різних обсягів даних і складності обчислень, що забезпечує стабільність та високу продуктивність в різних умовах. Наприклад, моделі глибокого навчання (DL) [34] досягли чудової продуктивності в багатьох сферах застосування, включаючи зір, мову, медицину, комерційну рекламу, розваги тощо. Завдяки швидкому розвитку, як додатки DL, так і апаратне забезпечення, що лежить в їх основі, продемонстрували сильні тенденції до масштабування, тобто масштабування моделей та обчислень, наприклад, нещодавня попередньо навчена модель з сотнями мільярдів параметрів, яка споживає ~ТБ пам'яті, а також новітні прискорювачі GPU, що забезпечують сотні TFLOPS. З обома тенденціями масштабування виникають нові проблеми та виклики в системах обслуговування виводу DL, які поступово рухаються в бік великомасштабних систем обслуговування глибокого навчання (LDS). Є дослідження [35], що узагальнює та класифікує нові виклики та можливості оптимізації для великомасштабних паралельних систем глибокого навчання.

У підсумку, вибір між послідовною та паралельною обробкою залежить від конкретних потреб та обмежень задачі. Для завдань, де швидкість є критичною, і де можлива ефективна розбивка на підзадачі, паралельна обробка є кращим вибором. Для простіших або менш обчислювально інтенсивних задач послідовна обробка може бути більш ніж достатньою.

1.2 Можливості CUDA, як засобу паралельної обробки даних

Архітектура CUDA (Compute Unified Device Architecture) від NVIDIA – це революційний підхід до паралельної обробки даних, що дозволяє значно збільшити обчислювальну потужність за допомогою графічних процесорів (GPU) [36]. CUDA забезпечує розробникам інструментарій та технології для використання масивної

паралельної обчислювальної потужності GPU для загальних обчислювальних завдань, не обмежуючись лише графічними застосунками [37,38].

Однією з ключових особливостей CUDA є її здатність дозволити програмам використовувати множинність обчислювальних ядер GPU для паралельної обробки даних [39]. Це дозволяє розробникам значно прискорити обчислення у широкому діапазоні застосунків, включаючи наукові дослідження, інженерні симуляції, обробку зображень, аналіз великих даних швидше, ніж це було б можливо на традиційних процесорах [40-42].

Основним компонентом архітектури CUDA є GPU, який містить сотні або навіть тисячі малих обчислювальних ядер, здатних виконувати задачі паралельно [43,44]. Ці ядра організовані в більші блоки, звані мультипроцесорами, кожен з яких може незалежно виконувати інструкції та обробляти дані [45,46]. Мультипроцесори спільно використовують пам'ять та ресурси, забезпечуючи високу пропускну спроможність та ефективність обчислень [47,48].

CUDA використовує модель програмування, засновану на ієрархії пам'яті та потоків [49]. Розробники можуть організовувати код на блоки потоків, які виконуються на мультипроцесорах [50]. Кожен блок потоків містить менші потоки, які можуть виконувати код паралельно, оптимізуючи обробку та використання обчислювальних ресурсів.

Важливою особливістю CUDA є її підтримка різних типів пам'яті, включаючи регістри, локальну пам'ять, спільну пам'ять між потоками в межах одного блоку, глобальну пам'ять доступну всім потокам та постійну пам'ять, яка оптимізована для кешування. Це дозволяє програмам CUDA ефективно використовувати пам'ять і ресурси GPU, що призводить до значного підвищення продуктивності при обробці великих обсягів даних та складних обчислень.

Кластерні обчислення на базі CPU-GPU в сучасному світі охоплюють сферу складних та високоінтенсивних обчислень [51]. Для ефективного використання ресурсів кластера традиційної парадигми програмування недостатньо. Тому в статті [52] розглядаються паралельні парадигми програмування, такі як OpenMP на кластері CPU та CUDA на кластері GPU. З результатів експерименту видно, що

розпаралелювання з моделлю програмування OpenMP з використанням графового алгоритму не збільшує продуктивність процесорів CPU, а навпаки, зменшує продуктивність за рахунок додавання накладних витрат, таких як час простою, міжпотокова комунікація та надлишкові обчислення [53]. З іншого боку, паралельне програмування CUDA на GPU дає кращі результати. Реалізація досягає прискорення від 187 до 240 разів порівняно з реалізацією на CPU [54].

Поряд з високопродуктивними комп'ютерними системами, інтерфейс прикладного програмування (API), що використовується, має вирішальне значення для розробки ефективних рішень для сучасних паралельних і розподілених обчислень. CUDA та відкрита мова обчислень (Open Computing Language, OpenCL) [55] - це два популярні API, які дозволяють графічним процесорам загального призначення (GPGPU, скорочено GPU) прискорювати обробку даних у додатках, де вони підтримуються. У статті [56] представлено порівняльне дослідження OpenCL та CUDA та їх вплив на паралельні та розподілені обчислення. Згідно з результатами експериментів, CUDA працює краще, ніж OpenCL (до 7.34x прискорення), однак у більшості випадків OpenCL працює з прийнятною швидкістю (прискорення CUDA не перевищує 2x) [57].

CUDA надає спеціалізоване програмне середовище, що дозволяє використовувати C, C++ та інші мови програмування для розробки програм, які можуть виконуватися на GPU [58]. Це означає, що розробники можуть легко інтегрувати паралельну обробку даних у свої додатки, використовуючи знайомі мови та інструменти. Наприклад, у роботі [59] розроблено методологію та інструмент для генерації коду для GPU шляхом введення нових атрибутів до мови SPar (високорівнева доменно-специфічна мова (DSL), яка дозволяє виражати паралелізм потоків і даних у послідовному коді за допомогою анотацій з використанням атрибутів C++) та правил перетворення до компілятора SPar. Ці нові внески, окрім спрощення та скорочення коду порівняно з CUDA та OpenCL, дозволили SPar досягти вищої пропускної здатності при дослідженні комбінованого паралелізму CPU та GPU, а також при використанні пакетної обробки даних.

Компільовані мови низького рівня, такі як C/C++ та Fortran, використовувалися як інструменти програмування для реалізації додатків для дослідження пристроїв GPU. На противагу цій тенденції, в статті [60] представлено аналіз продуктивності та трудомісткості програмування за допомогою Python, інтерпретованої мови високого рівня, яка була застосована для розробки ядер та додатків паралельних тестів NAS Parallel Benchmarks, орієнтованих на графічні процесори. Результати експериментів показали, що додатки на Python досягли продуктивності, подібної до програм на C++ з використанням CUDA, і кращої, ніж на C++ з використанням OpenACC для більшості бенчмарків NPB [61].

Однією з переваг архітектури CUDA є її масштабованість. Програми, розроблені для CUDA, можуть автоматично масштабуватися, використовуючи доступні обчислювальні ресурси GPU, що означає, що вони можуть ефективно працювати на різних пристроях з різною кількістю обчислювальних ядер. Оскільки арифметична продуктивність зростає швидше, ніж пропускна здатність пам'яті та мережі, оптимізація руху даних стала критично важливою для досягнення значного масштабування в багатьох додатках, що вимагають значних витрат на комунікації. У роботі [62] досліджено обчислювальні аспекти ітераційних трафаретних циклів і реалізовано загальну схему зв'язку з використанням CUDA-орієнтованого MPI, яка використовується для прискорення моделювання магнітогідродинаміки на основі скінченних різниць високого порядку та інтегрування Рунге-Кутта третього порядку. Подана GPU-реалізація добре масштабується від одного до 64 пристроїв з ефективністю 50%-87% від очікуваної на основі теоретичної моделі продуктивності. Порівняно з багатоядерним CPU-розв'язувачем, реалізація демонструє 20-кратне прискорення та 9-кратне покращення енергоефективності у прив'язаних до обчислень бенчмарках на 16 вузлах.

Програмування для досягнення високої продуктивності графічних процесорів NVIDIA з використанням CUDA, як відомо, є складним завданням. Графічний процесор має сотні і тисячі ядер, на яких програма повинна демонструвати достатній паралелізм для досягнення максимального використання графічного процесора. Система з GPU-прискорювачами має гетерогенну і глибоку

систему пам'яті, яку програмісти повинні ефективно і правильно використовувати, щоб повною мірою скористатися можливостями паралелізму GPU. У статті [63] представлено колекцію з чотирнадцяти мікротестів, які демонструють проблеми з продуктивністю при програмуванні на CUDA та методи оптимізації програм на CUDA для вирішення цих проблем. Він також включає приклади і методи використання розширених можливостей CUDA, таких як перестановка даних між потоками, динамічний паралелізм тощо, які можуть допомогти користувачам оптимізувати CUDA-програми для підвищення продуктивності. Мікробенчмарк можна використовувати для оцінки продуктивності архітектур GPU, систем пам'яті самого GPU і системних архітектур в цілому, а також для оцінки ефективності компілятора і інструментів для аналізу продуктивності. Він може бути використаний, щоб допомогти користувачам зрозуміти складність гетерогенних систем з GPU-прискорювачами на прикладах і спрямувати користувачів на оптимізацію продуктивності.

1.3 Відомі методи оптимізації та паралельної обробки даних

У сучасному світі, де обсяги даних зростають з неймовірною швидкістю, потреба в ефективній паралельній обробці даних стає надзвичайно актуальною. Існування численних методів оптимізації відображає складність і різноманіття задач, з якими стикаються дослідники і практики. Кожен метод оптимізації пропонує унікальний підхід до розв'язання проблем, пов'язаних з обробкою великих даних, ефективності обчислень та швидкості виконання. Дослідження цих методів є ключовим для розвитку технологій, що дозволяють максимально використовувати потенціал сучасних обчислювальних систем.

Розуміння різних методів оптимізації та паралельної обробки даних допомагає виявити найбільш ефективні стратегії для конкретних обчислювальних завдань. Вивчення цих методів відіграє критичну роль у підвищенні продуктивності і масштабуванні систем, що в кінцевому підсумку призводить до прискорення наукових досліджень та комерційних розробок. Тому дослідження та

розвиток нових методів паралельної обробки та оптимізації є важливими для того, щоб інформаційні технології могли відповідати зростаючим вимогам сучасної епохи великих даних.

Алгоритм стохастичного градієнтного с пуску (SGD) та його варіації ефективно використовуються для оптимізації нейромережевих моделей. Однак зі стрімким зростанням обсягів великих даних і глибокого навчання SGD більше не є найкращим вибором через його природну поведінку послідовної оптимізації функції помилки. Це призвело до розробки паралельних алгоритмів SGD, таких як асинхронний SGD (ASGD) і синхронний SGD (SSGD) для навчання глибоких нейронних мереж. Однак, це призводить до високої дисперсії через затримку в оновленні параметрів (ваг). В роботі [64] досліджується ця проблема. Швидкість збіжності також подібна до A/SSGD, однак для компенсації затримки потрібна додаткова (паралельна) обробка. Експериментальні результати демонструють, що запропонований підхід здатен пом'якшити вплив затримки на якість точності класифікації. Широко поширеною практикою є навчання моделей глибокого навчання на спеціалізованих апаратних прискорювачах, наприклад, GPU або TPU, завдяки їхній вищій продуктивності в операціях лінійної алгебри. Однак ця стратегія не дозволяє ефективно використовувати великі ресурси процесора та пам'яті, які використовуються лише для попередньої обробки, передачі даних та планування, доступні за замовчуванням на прискорених серверах. У статті [65] досліджуються алгоритми навчання для глибокого навчання на гетерогенних архітектурах CPU+GPU. Мета - максимізувати швидкість збіжності та використання ресурсів одночасно - робить проблему складною. Було розроблено два гетерогенні асинхронні алгоритми стохастичного градієнтного спуску (SGD). Перший алгоритм - CPU+GPU Hogbatch - поєднує невеликі партії на CPU з великими партіями на GPU, щоб максимізувати використання обох ресурсів. Однак це призводить до незбалансованого розподілу оновлень моделі, що перешкоджає статистичній збіжності. Другий алгоритм - Adaptive Hogbatch - призначає партії з розміром, що постійно змінюється, на основі відносної швидкості CPU та GPU. Продемонстровано, що реалізація цих алгоритмів у запропонованому фреймворку

CPU+GPU досягає швидшої збіжності та вищого рівня використання ресурсів, ніж TensorFlow, на кількох реальних наборах даних.

Структурні зміни в напрямку цифрової трансформації онлайн-продажів підвищують важливість методів паралельної обробки в системах рекомендацій, особливо в епоху пандемії та постпандемії. Матрична факторизація (MF) є популярним і масштабованим підходом у колаборативній фільтрації (CF) для прогнозування вподобань користувачів у рекомендаційних системах. Дослідники застосовують стохастичний градієнтний спуск (SGD) як один з найвідоміших методів оптимізації для МФ. Розпаралелювання методів SGD допомагає вирішувати проблеми, пов'язані з великими обсягами даних, через широкий асортимент товарів і розрідженість оцінок користувачів. Однак на швидкість збіжності та точність цих методів впливає залежність між латентними факторами користувача та товару, особливо у великомасштабних задачах. Крім того, продуктивність чутлива до застосованої швидкості навчання. У статті [66] пропонується новий паралельний метод для усунення залежностей і прискорення роботи за рахунок використання дробових обчислень для підвищення точності та швидкості збіжності.. Запропонований метод базується на платформі CUDA. Результати показують, що метод може отримати високу точність і швидкість збіжності на додаток до високого паралелізму.

З розвитком штучного інтелекту та покращенням обчислювальних потужностей апаратного забезпечення, моделі глибокого навчання стали широко використовуватися у сфері Інтернету речей (IoT), особливо для аналізу просторово-часових даних, зібраних бездротовими датчиками. Рекурентні нейронні мережі (RNN), такі як мережі з довгою короткочасною пам'яттю (LSTM), зазвичай використовуються для цих часових рядів. Налаштування гіперпараметрів навчання моделі вважаються важливими факторами для продуктивності моделей глибокого навчання. Оптимізація гіперпараметрів (АОП), як одна з найважливіших галузей автоматизованого машинного навчання, в основному включає в себе сітковий пошук, пошук гіперпараметрів на основі генетичного алгоритму тощо. У статті [67] пропонується автоматизований метод HPO на основі паралельного генетичного

алгоритму (PGA). Порівнюючи запропонований нами метод з іншими основними методами HPO на різних наборах даних, доведено, що наш метод HPO на основі PGA демонструє кращі показники як за часовими витратами, так і за результатами прогнозування. У роботі [68] запропоновано новий підхід до оцінки та аналізу поведінки багатопопуляційних паралельних генетичних алгоритмів (PGA) під час роботи на кластері багатоядерних процесорів. Зокрема, вивчається їхня чисельна та обчислювальна поведінку, пропонуючи математичну модель, що представляє спостережувані криві продуктивності. У них ми обговорюються нові математичні описи продуктивності PGA замість, наприклад, окремих ізольованих результатів, що підлягають візуальному огляду, для кращого розуміння впливу кількості використовуваних ядер (масштабованість), їхньої міграційної політики (міграційний розрив, у цій статті) та особливостей розв'язуваної задачі (тип кодування та розмір задачі). Висновки, зроблені на основі реальних даних і числових моделей, що їм відповідають, представляють новий погляд на прискорення, час виконання та обчислювальні зусилля, дозволяючи порівняти їх на основі кількох значущих числових параметрів. Алгоритм може бути використаний як інструмент оцінки майбутньої продуктивності алгоритмів і спосіб з'ясувати верхню межу продуктивності, якщо кількість використовуваних ядер збільшиться.

Своєчасне збирання та розбирання відпрацьованого електричного та електронного обладнання дозволяє не тільки отримати більшу економічну вигоду, але й зменшити вплив небезпечних речовин на навколишнє середовище. Паралельна лінія розбирання може розбирати різні види обладнання синхронно і підвищити ефективність розбирання. Тому в статті [69] створено модель балансування паралельної лінії часткового розбирання зі стохастичним часом розбирання. Для оптимізації моделі запропоновано новий генетичний алгоритм імітаційного відпалу. До результатів генетичної операції застосовано симуляцію відпалу. Запропонований алгоритм отримує кращі розв'язки, ніж алгоритм пошуку табу в стохастичних задачах балансування паралельної складальної лінії, а запропонований алгоритм має кращу продуктивність, ніж генетичний алгоритм та

імітаційне відпалювання в задачах балансування паралельної розбірної лінії. Нарешті, побудовано паралельну лінію часткового розбирання відпрацьованих телевізорів і холодильників, а продуктивність запропонованого багатоцільового алгоритму перевершує продуктивність п'яти класичних багатоцільових алгоритмів. Результати показують, що запропонована модель має кращі можливості практичного застосування і що запропонований алгоритм може покращити продуктивність ліній розбирання. Також у роботі [70], метод паралельного імітаційного відпалу розглядається як метод оптимізації схеми завантаження, який буде використовуватися в рамках новітньої системи проектування ядерних реакторів Westinghouse NEXUS/ANC9. Було розроблено прототипну версію NEXUS/ANC9, яка включає метод паралельного імітаційного відпалу.

Управління енергією в енергосистемах є складною проблемою оптимізації. Із зростанням розмірів систем централізовані методи оптимізації обмежуються складністю зв’язку, тоді як розподілені методи оптимізації стали потужним інструментом для роботи зі все більш складними системами. Однак показники збіжності деяких широко використовуваних розподілених методів оптимізації, таких як стандартний метод змінного напрямку множників (alternating direction method of multipliers (ADMM)), все ще мають місце для вдосконалення. У роботі [71] пропонується паралельний і розподілений метод оптимізації для керування енергією мікромереж для підвищення швидкості конвергенції без шкоди для точності оптимумів, у яких агенти обчислюють, обмінюються та оновлюють паралельно. Спочатку представлено метод декомпозиції, де цільові функції та обмеження вихідної задачі оптимізації з роздільними змінними розкладаються на локальні цільові функції та обмеження для агентів, що є ключем до нашого методу. Крім того, агенти самостійно вирішують свої проблеми локальної оптимізації, а потім обмінюються визначеними оптимумами зі своїми сусідами. Нарешті, метод оцінюється для вирішення економічної диспетчеризації з реагуванням на попит для мікромереж. Результати моделювання показують, що порівняно зі стандартним ADMM для заданої точності кількість ітерацій у нашому методі становить лише одну третину або навіть менше, ніж у ADMM. Крім того, метод може мінімізувати

функції витрат розподіленої генерації на стороні пропозиції та максимізувати функції прибутку гнучких навантажень на стороні попиту.

Емпіричні дослідження є основоположними в оцінці ефективності реалізацій розгалужених (branch-and-bound) алгоритмів. Складність таких реалізацій ускладнює емпіричне дослідження з багатьох причин. Були зроблені різні спроби розробити та кодифікувати набір стандартних методів для оцінки алгоритмів оптимізації та їх програмних реалізацій; однак більшість попередніх робіт було зосереджено на класичних послідовних алгоритмах. Оскільки паралельні обчислення стають все більш масовими, необхідно переглянути та модернізувати ці практики. У роботі [72] пропонується структура для оцінки, заснована на уявленні про те, що споживання ресурсів лежить в основі того, що зазвичай називають «ефективністю» реалізації. Запропонована структура ретельно розрізняє базову ефективність реалізації, ефективність, з якою вона використовує фіксований розподіл ресурсів, і її масштабованість, міру того, як ефективність змінюється, коли ресурси (як правило, додаткові обчислювальні ядра) додаються або видаляються. Ефективність зазвичай застосовується до послідовних реалізацій, тоді як масштабованість застосовується до паралельних реалізацій. Ефективність і масштабованість є важливими факторами, що визначають загальну ефективність даної паралельної реалізації, але мета підвищення ефективності часто суперечить меті покращення масштабованості.

У статті [73] представлено BiqBin, точний розв’язник бінарних квадратичних задач із лінійними обмеженнями. Підхід ґрунтується на методі точного штрафу, щоб спочатку ефективно перетворити вихідну проблему на екземпляр Max-Cut, а потім вирішити проблему Max-Cut за допомогою алгоритму розгалуження та межі (branch-and-bound). Усі основні інгредієнти ретельно розроблені з використанням нових релаксацій напіввизначеного програмування, отриманих шляхом посилення існуючих релаксацій набором гіперметричних нерівностей, застосування методу розшарування як обмежувальної процедури та використання нових стратегій для дослідження дерева розгалужень і зв’язків. Крім того, представлено ефективну реалізацію C послідовного та паралельного алгоритму розгалуження та межування.

Останній базується на схемі координатор навантаження-працівник з використанням MPI для багатовузлового розпаралелювання та оцінюється на високопродуктивному комп'ютері. Чисельні результати демонструють, що BiqBin є висококонкурентоспроможним розв'язувачем. Також оцінено паралельний розв'язувач і показуємо, що він має хороші властивості масштабування.

Хмарні обчислення є ідеальним способом для великомасштабних розподілених обчислень і паралельної обробки. Хмарні обчислення підтримують велику кількість послуг, які охоплюють велику кількість споживчих послуг, як-от хмарне резервне копіювання зображень, відео на смартфоні тощо. Продуктивність і ефективність послуг, що надаються хмарними обчисленнями, залежать від часу виконання завдань користувача представлені в хмарну систему. Ефективне планування завдань користувача відіграє важливу роль в управлінні фізичними та віртуальними ресурсами з кращою продуктивністю в хмарних службах. Планування завдань — це один із основних типів планування, що виконується в хмарному середовищі, метою якого є мінімізація тривалості обробки завдань. Makespan означає загальний час, потрібний віртуальним машинам для виконання призначених їм завдань. У гетерогенній системі планування різного розміру завдань різної значущості є складною проблемою, яку намагалися вирішити за допомогою різних підходів, напр. FCFS, SJF, Min-Min, Max-Min тощо. У роботі [74] було реалізовано та протестовано алгоритм розгалуження та зв'язку (branch-and-bound) для призначення цих різнорідних завдань, що виконуватимуться паралельно, віртуальним машинам, щоб зменшити тривалість виконання. Результати порівнюються з іншими загальними алгоритмами планування, наприклад FCFS, MIN-MIN, MAX-MIN і SJF.

Оптимізація продуктивності паралельних маніпуляторів (ПМ) привернула належну увагу в останні роки. В основному це стосується показників ефективності, алгоритмів оптимізації та методів оптимізації. Дослідження [75] представляє глибокий, вичерпний, аргументований огляд трьох основних питань. Обговорюються переваги, недоліки, сценарії застосування та рекомендації різних показників ефективності, паралельні алгоритми та методи оптимізації. Також увага

приділяється майбутнім напрямкам досліджень. Результати корисні дослідникам та інженерам для правильного вибору індексів ефективності, алгоритмів оптимізації та методів оптимізації під час проектування ПМ.

У статті [76] розглядається прогрес досліджень платформ обробки великих даних і алгоритмів на основі моделі програмування MapReduce. Платформа обробки великих даних на основі MapReduce аналізує та порівнює принципи реалізації існуючих алгоритмів обробки великих даних та застосовні сценарії, абстрагує їх спільність, а потім представляє засновані на MapReduce алгоритми аналізу великих даних, включаючи алгоритми пошуку, алгоритми очищення/перетворення даних, алгоритми агрегації, алгоритми об'єднання, алгоритми сортування, запити переваг, метод обчислень оптимізації, алгоритм графів, алгоритм інтелектуального аналізу даних, класифікує ці алгоритми відповідно до реалізації MapReduce, аналізує фактори, що впливають на продуктивність алгоритму; нарешті, великі дані. Алгоритм обробки абстрагується як алгоритм зовнішньої пам'яті, а характеристики алгоритму зовнішньої пам'яті сортуються. Запропоновано ідеї та проблеми дослідження методу оптимізації продуктивності універсального алгоритму зовнішньої пам'яті. Зокрема, це включає введення/виведення диска для оптимізації алгоритму зовнішньої пам'яті, оптимізації локальності алгоритму зовнішньої пам'яті та розробки інкрементного ітераційного алгоритму. Існуюча велика платформа обробки даних і дослідження алгоритмів здебільшого зосереджені на оптимізації динамічної продуктивності платформи на основі розподілу ресурсів і планування завдань, розпаралелювання конкретного алгоритму, конкретного алгоритму.

З різким розвитком обчислювальних технологій існує тенденція до зростання обсягу даних. Фахівці з обробки даних перевантажені такою великою та постійно зростаючою кількістю даних, оскільки тепер для цього потрібно більше каналів обробки. Велике занепокоєння, яке виникає тут щодо великомасштабних даних, полягає в забезпеченні підтримки процесу прийняття рішень. У дослідженні [77] застосовано модель програмування MapReduce, пов'язану реалізацію, представлену Google. Ця модель програмування передбачає обчислення двох

функцій; map і reduce. Бібліотеки MapReduce автоматично розпаралелюють обчислення та вирішують складні завдання, включаючи розподіл великих даних, навантаження та відмовостійкість. Ця реалізація MapReduce з вихідним кодом Google і механізмом відкритого вихідного коду Hadoop має на меті обробку обчислень великих кластерів товарів. Ідея MapReduce і Hadoop framework спрямована на обговорення терабайтів і петабайт пам'яті з тисячами машин, паралельних кожній машині та процесу в однаковий час. Таким чином, велика обробка та маніпулювання великими даними підтримується з ефективним орієнтуванням на результат. У цьому дослідженні будуть показані основи програмування MapReduce і застосування структури Hadoop з відкритим кодом. Система Hadoop може прискорити обробку великих даних і дуже швидко відповідати на запити.

Як ефективно обробляти дані та зменшити тиск одночасного доступу до даних, стало рушійною силою постійного розвитку рішень для великих даних. Стаття [78] в основному вивчає структуру паралельних обчислень MapReduce на основі кількох датчиків об'єднання даних і кластерів GPU. У цьому експериментальному середовищі використовується повністю розподілене кластерне середовище Hadoop, а все програмування алгоритму найкоротшого шляху з одним джерелом на основі MapReduce реалізовано мовою Java. Для побудови повністю розподіленого кластера використовуються 8 звичайних фізичних машин, і середовище конфігурації кожного вузла в основному однакове. Фреймворк MapReduce розділяє завдання запиту на кілька завдань відображення та призначає їх різним обчислювальним вузлам. Після процесу відображення створюється певний проміжний файл, який узгоджується з кінцевим форматом файлу. У цей час система згенерує кілька завдань скорочення та розповсюдить ці файли на різні вузли кластера для виконання. Цей експеримент перевірить зміни в часі роботи алгоритму PSON, коли розмір тестового набору даних поступово збільшується, зберігаючи рівень апаратного забезпечення та конфігурацію програмного забезпечення платформи Hadoop незмінними. При збільшенні кількості обчислювальних вузлів з 2 до 4 час роботи значно скорочується. Коли

кількість обчислювальних вузлів продовжує збільшуватися, скорочення часу роботи ставатиме все менш значним. Результати показують, що NESTOR може завершити базовий робочий процес MapReduce і спростити процес розробки користувачем позитивного дерева GPU, що має значне прискорення для додатків із великим обсягом обчислень.

Архітектура обробки в пам’яті (PIM) з її здатністю виконувати паралельну обробку з наднизькою затримкою вважається більш прийнятною альтернативою обчислювальним архітектурам фон Неймана для реалізації додатків із інтенсивним об’ємом даних, таких як глибокі нейронні мережі (DNN) і згорткові нейронні. мережі (CNN). У статті [79] представлено архітектуру PIM на основі таблиці перегляду (LUT), спрямовану на прискорення CNN/DNN, яка замінює логічну обробку попередньо обчисленими результатами, що зберігаються всередині LUT, щоб виконувати складні обчислення на платформі пам’яті DRAM. Архітектура DRAM-PIM на основі LUT забезпечує чудову продуктивність зі значно вищою енергоефективністю порівняно з більш звичайними побітово-паралельними архітектурами PIM, у той же час уникаючи проблем виготовлення, пов’язаних із реалізацією логічних схем у пам’яті. Крім того, елементи обробки можна програмувати та перепрограмувати для виконання практично будь-яких операцій, включаючи операції згорткового, повного з’єднання, об’єднання та активації рівнів CNN/DNN. Крім того, він здатний працювати з декількома комбінаціями бітових ширин даних операнда і, таким чином, пропонує ширший діапазон гнучкості щодо продуктивності, точності та ефективності. Моделювання демонструє, що запропонована архітектура може виконувати задачі майже в 13 разів швидше та в 125 разів ефективніше порівняно з найсучаснішим графічним процесором, а також забезпечує в 1,35 рази вищу пропускну здатність і в 2,5 рази вищу енергоефективність, ніж інша нещодавня версія Архітектура PIM, реалізована DRAM на основі LUT, у базовому режимі роботи. Крім того, він пропонує в 12 разів більшу частоту кадрів і в 9 разів більшу ефективність на кадр для найнижчого налаштування точності операнда щодо власного базового режиму роботи.

У дослідженні [80] розроблено нову стратегію кооперативного водіння для автоматизованих транспортних засобів (CAV) на перехрестях без світлофорів, використовуючи розподілене пошукове дерево Монте-Карло (MCTS) для оптимізації порядку пропуску і мінімізації затримок. Стратегія включає обмін інформацією між CAV, визначення порядку пропуску та контроль траєкторії, використовуючи кореневий паралелізм MCTS для ефективного вибору майже оптимального шляху. Паралельна обробка даних тут критично важлива, оскільки вона дозволяє системі швидко аналізувати множину можливих сценаріїв проходження CAV, адаптуватися до динамічних умов дорожнього руху і забезпечувати безпечне і ефективне управління трафіком у реальному часі. Мета полягає в тому, щоб CAV здійснювали належне коригування траєкторії на основі отриманого порядку проходження, щоб мінімізувати затримки в русі, роблячи найменші коригування прискорення. Спільна платформа моделювання, що інтегрує SUMO та Python, розроблена для побудови сценаріїв перехресть без сигналів та створення запропонованої розподіленої кооперативної стратегії водіння.

1.4 Висновки до розділу

У розділі було проведено дослідження використання послідовних і паралельних методів обробки даних в різних сферах (IoT, Big Data, машинне навчання), використання обчислювальних можливостей GPU для підвищення ефективності обробки даних, проаналізовано існуючі методи оптимізації паралельної обробки даних з використанням CUDA та інших технологій. Проведене дослідження надало можливість не тільки поглибити знання в області різних методів обробки даних, в тому числі й з використанням CUDA, а й зрозуміти, що не дивлячись на велику кількість вже існуючих методів оптимізації паралельної обробки даних, ця тема залишається актуальною для нових розробок, оскільки:

- не всі з існуючих методів адаптовані під використання архітектури CUDA, що могло б ще більше їх пришвидшити завдяки обчислювальним потужностям GPU;
- багато з існуючих методів, що використовуються CUDA можуть бути поліпшені завдяки імплементації кластерної архітектури або адаптації їх під інші мови програмування.

2 МОДЕЛЬ ВИКОРИСТАННЯ РЕСУРСІВ І ЧАСУ ВИКОНАННЯ ЯДРА CUDA

2.1 Дослідження особливостей технології CUDA

Для оптимізації паралельної обробки даних з використанням програмно-апаратної архітектури CUDA, критично необхідно написати ядро, яке ефективно виконується на графічному процесорі. Зробити це не так просто, як написати функцію, наприклад, на C, оскільки невеликі зміни в ядрі, які можуть бути несуттєвими для послідовного коду ЦП, можуть суттєво вплинути на його продуктивність. Особливо поширеними вузькими місцями продуктивності, до яких слід підходити детально є варпи, необ’єднаний доступ до пам’яті та конфлікти спільного використання пам’яті.

Група потоків (часто 32 потоки, які називаються варпом) виконує ту саму інструкцію на різних даних. Функції C, однак, можуть виконувати довільне розгалуження, яке може спричинити розходження різних потоків варпів. CUDA здатна скомпілювати такий код і виконати його на графічному процесорі, але з досить високою вартістю продуктивності, оскільки дві гілки повинні виконуватися послідовно. Навіть якщо умовний вираз має лише одну гілку виконання, є нетривіальні накладні витрати, пов’язані з розбіжністю.

Існують вагомі причини віддавати перевагу надійному статичному аналізу над динамічним аналізом, особливо в додатках (наприклад, системах машинного навчання в реальному часі, таких як ті, що розгортаються в автономних транспортних засобах), де екземпляри вхідних даних, які спричиняють погіршення продуктивності системи поза очікуваними межами, можуть бути небезпечними. Тому можливість статично оцінити найгіршу межу використання ресурсів є не тільки корисною в області оптимізації паралельної обробки даних, а й необхідною, для деяких доменних областей.

Даний статичний аналіз використання ресурсів ядра CUDA аналіз визначає абстрактну вартість певних операцій або подій, залежно від показника ресурсу. Загалом, метрика ресурсу призначає операції невід’ємне раціональне число, яке є

верхньою межею фактичної вартості цієї операції незалежно від контексту. Призначена вартість може залежати від параметрів часу виконання, які мають бути приблизно оцінені під час статичного аналізу ресурсів. Наприклад, одна з метрик, що розглядатиметься, це «сектори», тобто оцінка кількоісті читань і записів пам’яті. У CUDA блоки пам’яті фіксованого розміру, які зчитуються та записуються за допомогою одного варпу, можуть оброблятися як одна апаратна операція, тому кількість таких операцій залежить від місця розташування ділянок пам’яті, до яких звертається варп, і обсягу пам’яті, доступ до якого здійснюється доступ за одну операцію (це параметр, що залежить від апаратного забезпечення).

Основна проблема статичного міркування про програми з використанням CUDA полягає в тому; у CUDA кожна програмна змінна потенційно має тисячі копій, по одній для кожного потоку. Тому міркування про вміст змінних вимагає незалежного міркування щодо кожного потоку, який важко масштабувати, або статичних міркувань про те, які потоки активні в кожній точці програми. Тому, з метою узагальнення, логіку, що використовуватиметься для оцінки використання ресурсів при оптимізації паралельної обробки даних, буде формалізовано в основному обчисленні, що моделює підмножину CUDA, достатню для виявлення трьох перерахованих вище вузьких місць продуктивності. Обчислення міститиме формалізацію оцінки вартості виконання ядра за заданою метрикою ресурсу.

Графічні процесори призначені для виконання однієї арифметичної або логічної інструкції в багатьох потоках одночасно, як вже відомо, такий підхід називається SIMT. Щоб відобразити це, потоки CUDA організовані в групи під назвою варпи. Виконання SIMT призводить до потенційного вузького місця продуктивності в коді CUDA. Якщо виконується операція розгалуження, наприклад, умова, і два потоки в варпі використовують різні шляхи виконання, тоді GPU має серіалізувати виконання цього варпу. Спочатку він деактивує потоки, які брали один шлях виконання, і виконує інший, а потім перемикається на виконання потоків, які брали другий шлях виконання. Це називається дивергенцією або дивергентним викривленням і може значно зменшити паралелізм ядра CUDA.

Наступним вузьким місцем продуктивності, на яке необхідно зважати для максимальної оптимізації паралельної обробки даних з використанням CUDA є доступ до глобальної пам’яті. CUDA варп може отримати доступ до 128 послідовних байтів такої пам’яті одночасно. Коли потоки в варпі отримують доступ до пам’яті, наприклад доступ до масивів, CUDA намагається об’єднати ці звернення в якомога менше окремих доступів. Якщо варп отримує доступ до чотирьох послідовних 32-розрядних елементів масиву, то пропускна здатність пам’яті цієї інструкції в чотири рази вища, ніж якщо вона виконує чотири непослідовних читання.

Зважаючи на те, що CUDA також надає окремий простір спільної пам’яті, який використовується лише потоками в межах блоку, слід приділяти певну увагу тому, щоб забезпечити продуктивність коду через її використання. Спільна пам'ять складається з певної кількості, зазвичай 32, окремих сховищ. Можливий одночасний доступ до окремих сховищ, але кілька одночасних доступів до окремих адрес в одному сховищі серіалізуються. Більшість графічних процесорів гарантують, що 32 послідовні 32-розрядні зчитування пам’яті не призведуть до жодних конфліктів. Однак, якщо блок звертається до спільного масиву з кроком, відмінним від 1, можуть накопичуватися конфлікти.

2.2 Модель використання ресурсів CUDA

Проаналізуємо функції CUDA, які представляють головний інтерес при оптимізації паралельної обробки даних, а саме керування потоком (з метою оцінки циклів та вивчення вартості розбіжностей у варпі) і доступ до пам’яті. Результатом аналізу буде модель оцінки використання ресурсів для ядер CUDA.

Під час аналітичних обчислень, було зроблено ряд спрощуючих припущень, які роблять аналіз чіткішим та зрозумілішим. Одне з помітних спрощення полягає в тому, що обчислення складаєтиметься з одного. Крім того, структуру індексів потоку та блоку буде позначено одним ідентифікатором - tid. Такі припущення на призводять до втрати загальності теорії, оскільки тривимірний індекс потоку

можна легко перетворити на одновимірний ідентифікатор потоку, враховуючи розмірність блоку. Аналіз ресурсів буде параметричним щодо індексу блоку та інших параметрів і оцінюватиме максимальне використання ресурсів будь-яким варпом у будь-якому блоці.

Синтаксис моделі представлено наступним чином:

- типи: $\beta := int \mid bool \mid B \mid arr(\beta)$,
- операнди: $o := x \mid p \mid c \mid tid$,
- масиви: $A := Gl \mid Sh$,
- вирази: $e := o \mid o\ op\ o \mid A[o]$,
- інструкції:

$s := skip \mid s; s \mid x \leftarrow e \mid A[o] \leftarrow e \mid if\ e\ then\ s\ else\ s \mid while\ (e)\ s.$

Розглянемо цей синтаксис більш детально. Два типи даних є особливо важливими для оцінки та аналізу вартості обчислення: цілі числа використовуються як для індексів масиву (що визначають вартість доступу до пам’яті), так і для меж циклу (що мають вирішальне значення для оцінки вартості циклів), а булеві значення використовуються в умовних виразах. Усі інші базові типи (наприклад, float, string) представлені абстрактним базовим типом B. Масиви також розглядаються незалежно від типу.

Терміни обчислення поділяються на оператори, які можуть впливати на потік керування або стан пам’яті, та вирази, які не мають ефекту. Крім того, розрізняються операнди, які складаються з локальних змінних потоку x, параметрів p (до них належать аргументи, передані функції ядра, а також параметри CUDA, такі як індекс блоку та розмір варпу, що, за припущенням, є призначеними лише для читання та потенційно мають різні вартості для доступу), константи c (типів int, $bool$ та B) і призначену змінну tid для доступу до ідентифікатора потоку. Для простоти всі змінні знаходяться в межах ядра (але кожен потік підтримує власну локальну копію кожної змінної). Додаткові вирази включають $o_1\ op\ o_2$, що означає довільну бінарну операцію, і доступ до масиву $A[o]$. Метазмінна A означає загальний масив. У відповідних випадках використовуються метазмінні, які вказують, чи зберігається масив у (Gl) (скорочено G) глобальній чи (Sh)

(скорочено S) спільній пам’яті. Варто звернути увагу, що підвирази виразів обмежені операндами; більш складні вирази повинні бути розбиті на бінарні шляхом прив'язки проміжних результатів до змінних. Крім того, результати читання масиву не можна використовувати безпосередньо як індекси в масиві. Вирази (і операнди) відрізняються від операторів тим, що вирази не змінюють пам’ять, а просто обчислюють значення

Інструкції включають два типи присвоєння: присвоєння локальній змінній і елементу масиву. Інструкції також включають умовні оператори та цикли $while$. Ключове слово $skip$ представляє «порожній» оператор. Інструкції можуть бути упорядковані крапкою з комою, наприклад, $s_1;\ s_2$.

Для розробки семантики обчислень, що покажуть узгодженість аналізу використання ресурсів CUDA, варто зазначити кілька правил. Першим є, безпека типів, що передбачає деякі перевірки, які будуть необхідні для подальших результатів, а саме, що індекси масиву є цілими числами. Наступним є визначення статичної семантики над сигнатурами Σ, які записують типи локальних змінних, параметрів, операторів, функцій і масивів. Формально, сигнатура — це відображення набору $Vars \cup Params \cup Operators \cup Functions \cup Arrays$ на типи. Типовим судженням для виразів (та операндів) таке $\Sigma \vdash e : \beta$, що вказує на те, що під сигнатурою Σ вираз e має тип β. В даних обчисоеннях, вирази не повертають значення, а отже, не мають типів, але судження $\Sigma \vdash s$ використовується для вказівки того, що під сигнатурою Σ вираз s є правильно сформованим, оскільки всі його підвирази мають очікуваний тип.

Приклади правил для операндів і виразів:

$$OR: Var \Longrightarrow \frac{\Sigma(x)=\beta}{\Sigma \vdash x: \beta}, \tag{2.1}$$

$$OR: Const \Longrightarrow \frac{Type(c)=\beta}{\Sigma \vdash c: \beta}, \tag{2.2}$$

$$ER: Op \Longrightarrow \frac{\Sigma(op)=\beta_1 \times \beta_2 \to \beta_3 \quad \Sigma \vdash o_1: \beta_1 \quad \Sigma \vdash o_2: \beta_2}{\Sigma \vdash o_1\ op\ o_2: \beta_3} \tag{2.3}$$

У правилі 2.2 функція $Type(\cdot)$ надає тип константи c, яка означає літерал (число з плаваючою комою, ціле число, рядок тощо). Загалом, правила 2.1 і 2.3 гарантують, що підвирази мають правильний тип для відповідних типів, наприклад, індекси масиву повинні мати тип int, а умовні вирази — тип $bool$.

Приклади правил для інструкцій:

$$SR{:}\,If \Longrightarrow \frac{\Sigma \vdash e{:}\,bool \quad \Sigma \vdash s_1 \quad \Sigma \vdash s_2}{\Sigma \vdash if\ e\ then\ s_1\ else\ s_2}, \tag{2.4}$$

$$SR{:}\,While \Longrightarrow \frac{\Sigma \vdash e{:}\,bool \quad \Sigma \vdash s}{\Sigma \vdash while(e)\ s}, \tag{2.5}$$

$$SR{:}\,Write \Longrightarrow \frac{\Sigma,x \vdash \beta{:}\, e \vdash \beta}{\Sigma,x{:}\beta \vdash x \leftarrow e} \tag{2.6}$$

Підоператори умовних операторів і циклів мають бути сформованими відповідно, за аналогією з правилами 2.4, 2.5 і 2.6. Логіка є параметричною щодо метрики, яка залежить від конкретного ресурсу, що розглядається. Метрика ресурсу MC — це функція, доменом якої є набір констант ресурсу, які визначають певні операції, що виконуються програмою CUDA. Метрика ресурсу відображає ці константи на раціональні числа, можливо, приймаючи додатковий аргумент залежно від наданої константи. Метрика ресурсу, застосована до константи rc, записується MC^{rc}, а її застосування до додаткового аргументу n, якщо потрібно, записується як $MC^{rc}(n)$. Єдиною константою ресурсу, яка не відповідає синтаксичній операції, є MC^{div}, яка є накладними витратами на розбіжності в варпі.

Вартість доступу до масиву залежить від параметра, який визначає кількість необхідних окремих доступів (для глобальної пам’яті), або кількість потоків, які намагаються отримати доступ до одного банку спільної пам'яті (для спільної пам'яті). Ці значення надаються двома додатковими параметрами для операційної семантики. Маючи набір індексів масиву R, функція $MemReads(R)$ повертає кількість окремих читань (або записів), необхідних для доступу до всіх індексів, а функція $Conflicts(R)$ повертає максимальну кількість індексів, які відображаються на одне сховище спільної пам'яті. Ці параметри відокремлені від

метрики ресурсу, оскільки вони залежать не від ресурсу, а від деталей апаратного забезпечення. Показники ресурсів, що застосовуються до відповідних констант, беруть вихідні дані цих функцій і повертають вартість (у будь-якому ресурсі) виконання такої кількості звернень до пам’яті. Вимагаємо лише, щоб ця вартість була монотонною, тобто якщо $i \leq j$, то $MC^{gread}(i) \leq MC^{gread}(j)$, і аналогічно для MC^{sread}, MC^{gwrite} та MC^{swrite}.

Перейдемо до визначення семантики для оцінки ядер CUDA, яка також відстежує вартість оцінки з урахуванням метрики ресурсу. Ця семантична модель буде використана, як основа для підтвердження обґрунтованості аналізу використання ресурсів. Операційна семантика оцінює вираз або оператор по всьому варпу одночасно, щоб отримати результат оцінки. Оскільки вона оцінює лише один варп і потоки варпу виконуються в покроковому режимі, семантику можна назвати семантикою «покрокового блокування». Однак, варто пам'ятати, що варпи можуть розгалужуватись за наявності умовних операторів, в результаті чого лише деяка підмножина потоків варпу буде активною в кожній гілці. Тому відстежується набір B активних потоків у варпі як параметр. Операційна семантика також вимагає сховища σ, що представляє значення, що зберігаються в пам'яті. Оскільки кожен потік може обчислювати різні значення, результатом є не одне значення, а радше сімейство значень, по одному на активний потік - сімейство R з індексом B. Результат обчислений потоком t записується як $R(t)$. Дотримуючись стандартних домовленостей, створюються сімейства результатів: $f_t \in B$, де f — математичний вираз, який може посилатися на t, який, як припускається, взято з області B. Як приклад, результат обчислення операнда tid записується $R_{tid} = \left(Tid(t)\right)_{t \in B}$, вказуючи, що для всіх $t \in B$ маємо $R_{tid}(t) = Tid(t)$. Сімейство результатів також може не посилатися на t, у цьому випадку результат є константою, наприклад, обчислення константи c обчислюється як сімейство результатів $(c)_{t \in B}$. І навпаки, оператори не повертають значення, а просто змінюють стан пам’яті; тому оцінка просто створює нове сховище. Приклад 2.7 семантичного формулювання для виразів:

$$\sigma; e \downarrow_R^B R; C \tag{2.7}$$

вказує, що в сховищі σ вираз e обчислюється в потоках від B до R із вартістю C. Для інструкцій аналогічне формулювання 2.8:

$$\sigma; s \Downarrow_M^B \sigma'; C \tag{2.8}$$

Приклади правил оцінки для операндів, виразів і інструкцій:

$$OCR: Const \Longrightarrow \frac{}{\sigma; c \downarrow_M^B (c)_{t \in B}; MC^{const}}, \tag{2.9}$$

$$OCR: Var \Longrightarrow \frac{}{\sigma; x \downarrow_M^B (\sigma(x,t))_{t \in B}; MC^{var}}, \tag{2.10}$$

$$ECR: Op \Longrightarrow \frac{\sigma; o_1 \downarrow_R^B R_1; C_1 \quad \sigma; o_2 \downarrow_R^B R_2; C_2}{\sigma; o_1 \ op \ o_2 \downarrow_R^B (R_1(t) \ op \ R_2(t))_{t \in B}; C_1 + C_2 + MC^{op}}), \tag{2.11}$$

$$SCR: ifT \Longrightarrow \frac{\sigma; e \downarrow_M^B True_{t \in B}; C_1 \quad \sigma; s_2 \Downarrow_M^B \sigma_1; C_2}{\sigma; if \ e \ then \ s_1 \ else \ s_2 \Downarrow_M^B \sigma_1; C_1 + MC^{if} + C_2}), \tag{2.12}$$

$$SCR: ifD \Longrightarrow B_T = \{t \in B | R(t)\} \neq \emptyset, \ B_F = \{t \in B | \neg R(t)\} \neq \emptyset \ \rightarrow$$

$$\frac{\sigma; e \downarrow_M^B R; C_1 \quad \sigma; s_1 \Downarrow_M^{B_T} \sigma_1; C_2 \quad \sigma_1; s_2 \Downarrow_M^{B_F} \sigma_2; C_3}{\sigma; if \ e \ then \ s_1 \ else \ s_2 \Downarrow_M^B \sigma_2; C_1 + MC^{if} + C_2 + C3 + MC^{div}} \tag{2.13}$$

$$SCR: WhileSome \Longrightarrow \sigma; e \downarrow_M^B R; C_1, \ \emptyset \neq B_T \neq B, \quad \sigma; s_1 \Downarrow_M^{B_T} \sigma_1; C_2 \rightarrow$$

$$\frac{B_T = \{t \in B | R(t)\} \quad \sigma_1; while(e) \ s \Downarrow_M^{B_T} \sigma_2; C_3}{\sigma; while(e) \ s \Downarrow_M^B \sigma_2; C_1 + C_2 + MC^{if} + MC^{div} + C_3} \tag{2.14}$$

$$ECR: VWrite \Longrightarrow \frac{\sigma; e \downarrow_R^B R; C}{\sigma; x \leftarrow e \Downarrow_M^B \sigma[(x,t) \mapsto R(t) | t \in B]; C + MC^{vwrite}}) \tag{2.15}$$

Як було запропоновано вище, елементи B є абстрактними ідентифікаторами потоку t. Функція $Tid(t)$ перетворює такий ідентифікатор на цілочисельний ідентифікатор потоку. Для масиву A (незалежно від того, чи зберігається він у глобальній чи спільній пам'яті), $\sigma(A, n)$ повертає $n-$ й елемент A. Для простоти представлення припускається, що індекси поза межами (включаючи від'ємні

індекси) конвертуються в певне значення за замовчуванням. Для локальної змінної x, $\sigma(x,t)$ повертає значення x для потоку t. $\Sigma \vdash_B \sigma$ означає, що сховище є правильно типізованим щодо сигнатури та набору потоків, якщо:

(1) Для всіх $A : \beta \in \Sigma$ і всіх $n \in N$ маємо $\Sigma \vdash \sigma(A,n): \beta$,

(2) Для всіх $x : \beta \in \Sigma$ і всіх $t \in B$ маємо $\Sigma \vdash \sigma(y,t): \beta$.

Типізація сховища зберігається шляхом взяття підмножин набору потоків: якщо $\Sigma \vdash_B \sigma$ і $T' \subset T$, тоді $\Sigma \vdash_{B'} \sigma$. При оцінці виразів підвирази оцінюються паралельно та відповідним чином поєднуєднуються, але з ретельним урахуванням витрат. Вартість виконання, як правило, визначається підсумовуванням витрат на оцінку підвиразів із вартістю головної операції, заданою метрикою ресурсу MC. Наприклад, правило $ECR\ Op$ оцінює два операнди та об'єднує результати за допомогою конкретної бінарної операції, представленої op. Вартість виразу — це вартість двох підвиразів, C_1 і C_2, плюс MC^{op}. Доступ до масиву оцінює операнд до встановлення індексів і зчитування значення з пам'яті за кожним індексом. Вартість цих операцій залежить від $MemReads(R)$ або $Conflicts(R)$, де R – набір індексів. Як обговорювалося раніше, ці функції залишаються як параметри, оскільки їх точні визначення можуть змінюватися в різних версіях CUDA та апаратних реалізаціях. Як приклади цих функцій надаються визначення 2.16 і 2.17, що відповідають загальним специфікаціям у сучасних реалізаціях:

$$MemReads(R) \triangleq |\{\lceil \frac{i}{32} \rceil \mid (i)\_t \in R\}|, \tag{2.16}$$

$$Conflicts(R) \triangleq \max_{j \in [0,31]} \{R(t) \equiv j \bmod 32 \mid t \in Dom(R)\} \tag{2.17}$$

Припускається, що вичитка даних з глобальної пам'яті має розмір 128 байт, а елементи масиву — 4 байти. Насправді та в даній реалізації MemReads(R) залежить від типу масиву.

Оцінка операторів є дещо складнішою, оскільки оператори можуть оновлювати стан пам'яті, а також впливати на потік керування: перше представлено оновленням сховища σ, а друге – зміною набору потоків B під час

оцінювання підвиразів і підвиразів. Для операторів присвоювання новий стан виникає в результаті оновлення за допомогою нового присвоєння. Щоб вказати стан σ, оновлений так, що для всіх $t \in B$ прив'язка (y, t) відповідає значенню локальної змінної x у потоці t і є елементом області визначення σ, тепер відображається на $R(t)$, тобто до значення сімейства результатів R, використовуєтьяс запис $\sigma[(y, t) \rightarrow R(t) \mid t \in B]$. Оновлення масиву записуються аналогічно. Однак, оновлення масиву $\sigma[(A, R(t)) \rightarrow R_v\ (t) \mid t \in B]$, де $R(t)$ індекс масиву, до якого записує потік t, не визначено, якщо $R(t) = R(t')$ для потоків $t \neq t'$, тобто якщо існує «гонка за даними» - не синхронізований доступ до одних і тих же даних з різних потоків. У CUDA гонка даних призводить до невизначеної поведінки, тому програми з таким випадком не оцінюються. Очікується, що при оптимізації паралельної обробки даних, доступ до них буде синхронізовано або забезпечено максимальну їх атомарність.

Умовні оператори та цикли while мають три правила. Якщо умова оцінюється як True для всіх потоків (правило 2.12), оцінюється гілка «if» з повним набором потоків B, і аналогічно, якщо всі потоки оцінюються як False. Проте, якщо існують непорожні набори потоків, де умова оцінюється як True і False (B_T і B_F відповідно), обидві гілки необхідно оцінити окремо. Гілку «if» оцінується за допомогою B_T, а гілка «else» — за допомогою B_F. Результуючий стан гілки «if» передається до оцінки гілки «else»; це відповідає тому, що CUDA послідовно виконує дві гілки. Правило 2.13, також додає вартість MC^{div} розгалуженого варпу. Три правила для циклів while подібним чином обробляють випадки, коли всі, деякі або жоден з потоків у B оцінюють умову як True. Перші два правила одночасно оцінюють тіло під набором потоків, для яких умова є істинною, а потім повторно оцінюють цикл. Правило 2.14 також вказує на те, що потрібно врахувати MC^{div} у вартості, оскільки варп розходиться.

Для випадку, якщо операнд (або вираз) має визначений тип з сигнатурою Σ і сховищем σ відповідає цій сигнатурі для набору потоків B, оцінювання операнду (або виразу) з цим набором потоків дає набір результатів, в якому кожен результат

має тип операнду (або виразу) якщо $\Sigma \vdash o : \beta$ і $\Sigma \vdash_\beta \sigma$, і $\sigma; o \downarrow_M^B R; C$, тоді для всіх $t \in B$, маємо $\Sigma \vdash R(t): \beta$; і якщо $\Sigma \vdash e : \beta$ і $\Sigma \vdash_\beta \sigma$, і $\sigma; e \downarrow_M^B R; C$, тоді для всіх $t \in B$, маємо $\Sigma \vdash R(t): \beta$.

Крім того, якщо оператор сигнатурою Σ і сховищем σ відповідає цій сигнатурі для набору потоків B, то оцінювання оператора дає в результаті правильно сформоване (за типом) сховище. Якщо $\Sigma \vdash s$ і $\Sigma \vdash_\beta \sigma$, і $\sigma; s \Downarrow_M^B \sigma'; C$, тоді $\Sigma \vdash_\beta \sigma'$. У даному випадку, "результат збереження для оцінки" означає, що якщо операнд або вираз правильно типізований перед оцінкою, то результат цієї оцінки також буде правильно типізованим. Це стосується роботи з багатьма потоками, де вираз або операнд оцінюються з використанням конкретного набору потоків, і результат цієї оцінки зберігає типи, задекларовані в початковому коді.

2.3 Модель використання ресурсів в межах одного варпу

Перейдемо до представлення правил, які можна використовувати для міркувань про використання ресурсів для виконання ядра. Для того, щоб оптимізація паралельної обробки даних з використанням програмно-апаратної архітектури CUDA була ефективною, необхідно правильно оцінювати ресурси, що можуть бути використані для покриття «вузьких» місць продуктивності. Ключова ідея цього аналізу полягає у присвоєнні станам обчислень невід'ємного числового потенціалу. Цей потенціал має бути достатнім, щоб покрити вартість наступного кроку і потенціал наступного стану. Для імперативних програм потенціал, як правило, є функцією значень локальних змінних. Правила кількісної логіки визначають, як ця функція потенціалу змінюється під час виконання оператора. Виведення в кількісній логіці потім будує набір обмежень на функції потенціалу в кожній точці програми.

Як приклад, розглянемо оператор для $(int\ i\ =\ N;\ i\ >=\ 0;\ i--f();\}$, і, припустимо, потрібно обмежити кількість викликів функції f. Це відповідає метриці ресурсів, в якій вартість виклику функції дорівнює 1, а всі інші операції є

безкоштовними. Функція потенціалу в кожній точці має бути функцією від значення i: виявиться, що правильним рішенням буде встановити потенціал на $i + 1$ в тілі циклу до виклику f і на i після виклику. Ця різниця "оплачує" вартість 1 для виклику функції. Вона також встановлює правильний потенціал для наступної ітерації циклу: коли i зменшується після циклу, потенціал знову стає рівним $i + 1$.

Особлива складність розробки такої логіки для CUDA полягає в тому, що кожен потік у варпі має окремий локальний стан. Для того, щоб зробити висновок масштабованим, розглядатиметься лише одна копія кожної змінної, але тоді варто бути обережним щодо того, що саме мається на увазі під будь-якою функцією стану. Щоб вирішити цю проблему, зробимо спостереження над CUDA програмами: часто існує поділ між локальними змінними програми, які несуть дані (наприклад, використовуються для зберігання даних, завантажених з пам'яті, або проміжних результатів обчислень), і тими, які несуть потенціал (наприклад, використовуються як індекси у циклах for). Щоб розробити обґрунтовану та корисну кількісну логіку, достатньо відстежувати потенціал для останнього набору змінних, які зазвичай мають однакове значення в усіх активних потоках.

Умови логіки мають вигляд $\{L; Q; Y\}$ і складаються з логічної умови L, функції потенціалу Q, а також множини Y змінних, значення яких є однорідними по всьому варпу і тому можуть бути використані як змінні, що несуть потенціал.

Вираз 2.18 означає, що для всіх $y \in Y$ і всіх $t_1,\ t_2 \in B$ маємо $\sigma(y, t_1) = \sigma(y, t_2)$.

$$\sigma, B \vdash Y \tag{2.18}$$

Прикладом логічної умови може бути $i = 2 * tid$, яка вказує на те, що програмна змінна i містить подвоєне значення ідентифікатора потоку для кожного потоку (якщо i використовується як доступ до масиву, цей факт дозволить в межах аналізу підрахувати кількість секторів, до яких здійснюється доступ, та кількість конфліктів).

Вираз 2.19 використовується для позначення того, що умова L виконується при сховищі σ і значеннях $t \in B$ для ідентифікатора потоку.

$$\sigma, B \vDash L \tag{2.19}$$

Якщо або сховище, або множина потоків не є релевантними у конкретному контексті, то може бути використане скорочення $\sigma \vDash L$. Для позначення того, що з L випливає L: тобто для всіх σ, B таких, що $\sigma, B \vDash L$, використовується вираз 2.20.

$$L \Rightarrow L' \tag{2.20}$$

Крім того, має місце випадок, коли $\sigma, B \vDash L'$. Сховище (яке не містить виділеної змінної tid) та множина потоків є незалежними, і тому якщо $B \vDash L$ і $\sigma \vDash L$, то $\sigma, B \vDash L$. Іноді буде вказано $\sigma \restriction_B, B \vDash L$, щоб показати, що L утримує тільки ту частину σ, яка відноситься до локального стану потоків у B. Припускається, що L поводиться з цими компонентами пам'яті незалежно, тобто, якщо $B_1 \uplus B_2 = B, \sigma \restriction_{B_1}, B_1 \vDash L$ і $\sigma \restriction_{B_2}, B_2 \vDash L$, тоді $\sigma, B \vDash L$.

Другий компонент умов – функція потенціалу Q, відображення від сховищ і наборів змінних Y, як описано вище, до невід'ємних раціональних потенціалів. Дана функція використовується для відстеження потенціалу через ядро для аналізу використання ресурсів. Якщо $\sigma, B \vdash Y$, то $Q_Y(\sigma)$ позначає потенціал σ за функцією Q, враховуючи лише змінні в Y. Формально, вимагається (як властивість функції потенціалу Q), що якщо для всіх $x \in Y$ і $t \in B$ маємо $\sigma_1(y,t) = \sigma_2(y,t)$, то $Q_Y(\sigma_1) = Q_Y(\sigma_2)$. Тобто, Q може враховувати лише змінні в Y.

Потенціал буде зменшуватися з часом; зменшення потенціалу буде використовуватися для "оплати" операцій програми, і таким чином початковий потенціал може бути використаним, щоб обмежити вартість операцій, які відбуваються під час виконання. Як простий приклад того, як функція потенціалу

може виглядати на практиці, розглянемо приклад циклу: $for\,(int\,i\,=\,N;\,i\,>=\,0;\,i—f\,();\,\}$, як зазначалося вище, функцією потенціалу на початку тіла циклу буде $i\,+\,1$, а в кінці тіла циклу - i. Функцією потенціалу на початку програми, яка задає потенціал для першої ітерації циклу, буде $N\,+\,1$. Зв'язок між цими різними функціями потенціалу обмежується правилами кількісної логіки.

Для невід'ємних витрат C використовується скорочення $Q\,+\,C$ для позначення функції потенціалу Q', такої, що для всіх σ і Y маємо $Q'_Y(\sigma)\,=\,Q_Y(\sigma)\,+\,C$. Використовується $Q\,\geq\,Q'$ для позначення того, що для всіх σ, B, X таких, що $\sigma,\,B\,\vDash\,P$, маємо $Q_Y(\sigma)\,\geq\,Q'_Y(\sigma)$.

Вираз 2.21 означає що $\sigma,\,B\,\vDash\,P$ та $\sigma,\,B\,\vdash Y$, тобто, умова L виконується при сховищі σ і значеннях $t\,\in\,B$ для ідентифікатора потоку і відповідні значення є однорідними по всьому варпу

$$\sigma;\,B\vDash\{L;Q;Y\} \tag{2.21}$$

Наразі конкретне представлення логічної умови та функції потенціалу є абстрактними, оскільки на меті маємо формалізацію оцінки використання ресурсів ядра незалежно від конкретного алгоритму паралельної обробки даних, що оптимізовується. Актуальними будуть припущення, викладені вище, а також припущення, що логічні умови підкоряються стандартним правилам булевої логіки. Крім того, припускається, що логічні умови та функції потенціалу забезпечені операцією "присвоєння", тобто $L'\Leftarrow L[y\leftarrow e]$ (відповідно, $Q'\Leftarrow Q[y\leftarrow e]$) таким чином, що якщо $\sigma,\,B\,\vDash\,L'$ і $\sigma;e\downarrow_M^B R;C$, тоді справедливими будуть наступні твердження:

– $\sigma[(y,t)\longmapsto R(t)\mid t\in B],\,B\vDash L$

– Якщо $y\,\in\,Y$ і існує v таке, що $R(t)=v$ для всіх $t\,\in\,B$, то $Q'_Y(\sigma)=Q_Y(\sigma[(x,t\,)\longmapsto R(t)|\,t\in B])$.

В умовах для логіки стандартною практикою є реалізація операції присвоєння шляхом простої підстановки e замість y в L для отримання L'.

Представлення функції потенціалу також підтримують таку операцію підстановки. Для простоти також припускається, що функція потенціалу залежить лише від значень локальних змінних у сховищі, а не від значень масивів. Цього достатньо для розгляду досліджуваних метрик.

Перед переходом до логіки висловлювань, буде введено більш просте судження, яке використовуватиметься для опису використання ресурсів операндів та виразів. Вираз 2.22 вказує на те, що за умови L оцінка e коштує не більше C. Ці правила подібні до правил що розглядались раніше, за винятком того, що тепер не відомо, яке саме сховище використовується для обчислення виразу, і потрібно консервативно оцінити вартість доступу до масиву, виходячи з можливого набору сховищ.

$$L \vdash_M e : C \tag{2.22}$$

Вираз 2.23 означає, що для всіх σ і всіх B таких, що $\sigma,\ B \vDash L$, якщо σ; $\sigma; o \downarrow_M^{\mathrm{B}} R; C$, тоді $MemReads(R) \leq n$. Значення $L \Rightarrow Conflicts(o) \leq n$ аналогічне.

$$L \Rightarrow MemReads(o) \leq n \tag{2.23}$$

Приклади правил логіки для оцінки використання ресурсів:

$$QR{:}\,Skip \Longrightarrow \frac{}{\vdash_M \{L;Q;Y\}\ skip\ \{L;Q;Y\}}, \tag{2.24}$$

$$QR{:}\,Seq \Longrightarrow \frac{\vdash_M \{L;Q;Y\}\ s_1\ \{L_1;Q_1;Y_1\} \quad \vdash_M \{L_1;Q_1;Y_1\}\ s_2\ \{L';Q';Y'\}}{\vdash_M \{L;Q;Y\}\ s_1;\ \vdash_M \{L';Q';Y'\} s_2}, \tag{2.25}$$

$$QR{:}\,If1 \Longrightarrow \frac{\begin{matrix} L \vdash_M e{:}C \quad \vdash_M \{L \wedge e;Q;Y\}\ s_1\ \{L';Q';Y'\} \\ L \mapsto e\ unif \quad \vdash_M \{L \wedge \neg e;Q;Y\}\ s_2\ \{L';Q';Y'\} \end{matrix}}{\vdash_M \{L;Q+MC^{if}+C;Y\}\ if\ e\ then\ s_1\ else\ s_2\ \{L';Q';Y'\}}, \tag{2.26}$$

$$QR{:}\,While1 \Longrightarrow \frac{\begin{matrix} L \mapsto e\ unif \qquad\qquad\qquad\qquad\qquad\qquad L \vdash_{MC} e{:}C \\ \vdash_M \{L \wedge e;Q;Y\}\ s\ \{L;Q+MC^{if}+C;Y\} \end{matrix}}{\vdash_M \{L;Q+MC^{if}+C;Y\}\ while\ (e)\ s\ \{L \wedge \neg e;Q;Y\}}, \tag{2.27}$$

$$Q{:}\,Weak \Longrightarrow \frac{\vdash_M \{L;Q_2;Y_2\}\ s\ \{L_2';Q_2';Y_2'\} \quad \begin{matrix} L_1 \mapsto L_2 & Q_1 \geq Q_2 & Y_1 \supset Y_2 \\ L_1' \mapsto L_2' & Q_1' \geq Q_2' & Y_1' \supset Y_2' \end{matrix}}{\vdash_M \{P_1;Q_1+C;X_1\}\ s\ \{P_1';Q_1'+C;X_1'\}} \tag{2.28}$$

Вирази для цих правил записується у вигляді форматі 2.29, за яким: що якщо виконується L, маємо Q ресурсів, змінні в Y є властивими потоку, то s завершується у стані, де L', залишилося Q'ресурсів, і всі змінні в Y' є властивими потоку.

$$\{L;Q;Y\}s\ \{L';Q';Y'\} \tag{2.29}$$

Найпростіші випадки (наприклад, 2.24 та 2.25) пропускають умови, не змінюючи їх (2.25 перетворює постумови s_1 у передумови s_2). Більшість інших правил вимагають додаткового потенціалу в передумові (наприклад, $Q\ +\ C$), який потім відкидається, оскільки використовується для «оплати» операції. Наприклад, якщо s_1 використовує ресурси C_1, а s_2 використовує ресурси C_2, то можна почати з $Q + C_1 + C_2$, залишити $Q + C_2$ в пост-умові s_1 і залишити Q в пост-умові s_2.

Важливими є правила, що стосуються умов $if\ e\ then\ s_1\ else\ s_2$, які враховують можливість розгалуження варпу. Існує чотири випадки, розглянемо один з них (для решти міркування відбуваються аналогічним чином). Для 2.26, можна статично визначити, що умовний вираз e не змінюється всьому варпі: це виражається за допомогою передумови 2.30, яка скорочено позначає $\forall \sigma, T.\, \sigma, T \vDash L\ \Rightarrow\ \exists c.\, \sigma; e\ \downarrow_M^{\mathrm{B}} (\mathrm{c})_{t\in \mathrm{B}}; C$.

$$L\ \longmapsto e\ unif \tag{2.30}$$

Тобто, для будь-якого сумісного сховища, e оцінюється як сімейство з постійним результатом. У цьому випадку тільки одна гілка виконується у варпі, а вартість виконання умови дорівнює максимальній вартості виконання двох гілок (плюс вартість MC^{if} умови і вартість C обчислення виразу, які додаються до передумови). Це виражається використанням Q' як функції потенціалу в постумові

для обох гілок. Якщо дві гілки не використовують однаковий потенціал, то та, яка має більший потенціалу, може використати правило $QR{:}\,Weak$, щоб відкинути свій додатковий потенціал і використати Q' як після умову. Таким чином, апроксимується потенціал, що залишився після виконання однієї гілки.

У двох випадках ($QR{:}\,If2$ і $QR{:}\,If3$) можна статично визначити, що умовний вираз є істинним або хибним у будь-якому сумісному сховищі (тобто, або $L \longmapsto e$, або $L \longmapsto \neg e$), і потрібна лише гілка "then" або "else", тому у цих правилах розглядається лише відповідна гілка.

В останньому випадку $QR{:}\,If4$ розглядається можливість того, що варп може розходитися. Крім того, що це правило враховує випадок, коли потрібно послідовно виконати s_1, а потім s_2, воно також повинно враховувати три попередні випадки, оскільки можлива ситуація, коли не вдалось визначити статично, що умова не розгалужується (тобто, не вдалось отримати передумови 2.26), але варп не розгалужується під час виконання програми. Щоб впоратися з обома випадками, вимагається, щоб передумова s_2 враховувала два випадки:

– $L \wedge \neg e$, передумовою умови разом з інформацією про те, що e є хибною, так що s_2 може виконуватися сам по собі, якщо умова не розгалужується;

– L_1, постумовою s_1, так що s_2 може виконуватися послідовно після s_2.

Аналогічно, вимагається, щоб постумова всієї умови випливала з окремих постумов обох гілок. Крім того, якщо варп розгалужується, то змінні, записані на s_1, більше не мають узгоджених значень по всьому варпу, тому множина змінних вилучається з Y_1, що може бути записано на s_1 (позначається $W(s_1)$), тому що. Аналогічно видалається $W(s_2)$ з Y_2.

Завжди правильно використовувати правило $QR{:}\,If4$ для перевірки умови. Однак використання цього правила у всіх випадках призведе до консервативної завищеної оцінки вартості, оскільки припускатиме, що деформація розходиться, навіть якщо можна показати, що це не так. За даним аналізом варто максимізує точність, обираючи найточніше правило, яке можна обґрунтоване.

Правила типу 2.27 додають початкову оцінку умови ($MC^{if} + C$) до передумови. Умова L повинна виконуватися як на початку, так і в кінці кожної ітерації тіла циклу (додатково відомо, що e виконується на початку тіла). Крім того, потенціал після тіла циклу повинен бути достатнім, щоб "заплатити" $MC^{if} + C$ за наступну перевірку умови, і ще повинен залишатися потенціал Q для виконання наступної ітерації, якщо це необхідно (як відомо - Q є функцією сховища). Отже, ця передумова вимагає, щоб значення елемента сховища (наприклад, лічильника циклу) змінювалося настільки, щоб відповідне зменшення потенціалу $Q_X(\sigma)$ могло окупити відповідні витрати. Різниця між цими двома правилами полягає в тому, що 2.27 припускає, що варп не розходиться, тому не потрібно платити MC^{div}, а також не потрібно видаляти змінні, присвоєні тілом циклу, з Y.

Правила локального присвоювання є розширенням стандартного правила присвоювання. Якщо $y \in Y$ і $L \longmapsto e\ unif$, то додається симетрична передумова для потенційної функції. В іншому випадку x може використовуватись лише як змінну, що оновлюємо логічну умову, а не несе потенціал. Правила присвоювання масивів подібні до правил доступу до масивів, але додатково включають вартість присвоєного виразу e. Правило 2.28 дозволяє посилювати передумови і послаблювати постумови виводу. Якщо s може виконуватися з передумовою $\{L_2; Q_2; Y\}$ і постумовою $\{L_2'; Q_2'; Y\}$, то вона також може виконуватися з передумовою L_1, яка передбачає L_2, і функцією потенціалу Q_1, яка завжди більша за Q_2. Крім того, правило може гарантувати будь-яку постумову, що випливає з L_2', і будь-яку потенційну функцію Q_1', яка завжди менша за Q_2'. Правило також дозволяє додавати постійний потенціал як до перед-, так і до після-умов, що можна формалізувати наступним чином:

– Якщо $\Sigma \vdash e : \beta,\ \Sigma \vdash_\beta \sigma,\ \Sigma \vdash_M e : C,\ \sigma,\ B \vDash L,\ e \downarrow_M^B R; C'$, тоді $C' \leq C$.

– Якщо $\{L; Q; Y\} s \{L'; Q'; Y'\}$, то $Y' \subset Y$ і $Q' \subset Q$.

Використовуючи ці вирази, можна стверджувати, що під час аналізу існує похідна, яка показує, що програма може виконуватися з передумовою $\{L; Q; Y\}$, то

для будь-якого сховища σ і будь-якої множини потоків B таких, що $\sigma;\ B \vDash \{L; Q; Y\}$, вартість виконання програми при σ і потоках B не перевищує $Q_X(\sigma)$. В результаті маємо твердження: якщо $\Sigma\ \vdash s,\ \Sigma\ \vdash_\beta \sigma,\ \{L; Q; Y\} s\ \{L'; Q'; Y'\},\ \sigma;\ B \vDash \{L; Q; Y'\}$ і $\sigma; s \Downarrow_M^B \sigma'; C_s$, тоді $\sigma';\ B \vDash \{L'; Q'; Y'\}$ і $Q_Y(\sigma) - C_s \geq Q'_{Y'}(\sigma') \geq 0$.

2.4 Модель оцінки часу виконання ядра

Попередньо розроблений аналіз корисний для прогнозування метрик витрат ядер CUDA, таких як розбіжності у варпах, глобальні доступи до пам'яті та конфлікти сховищ: можна вказати метрику ресурсів, яка підраховує відповідні операції, провести аналіз для визначення максимальної вартості варпу і помножити її на кількість варпів, які буде створено для виконання ядра (це вказується у головній програмі під час виклику ядра). Передбачити фактичний час виконання ядра дещо складніше.

Перший підхід полягає у визначенні вартості часу виконання кожної операції та проведенні аналізу за допомогою метрики вартості, яка призначає відповідні витрати. Такий підхід може приблизно визначити час виконання одного варпу, але не одразу зрозуміло, як скласти результати, щоб врахувати декілька варпів, на відміну від розгалужень або звернень до пам'яті, які просто підсумовуються разом. Дійсно, питання про те, як скласти час виконання варпів, є складним через те, як графічні процесори планують варпи. Кожен потоковий мультипроцесор, обчислювальна одиниця графічного процесора, може виконувати інструкції на декількох варпах одночасно, причому точна кількість залежить від апаратного забезпечення. Однак, коли запускається ядро, CUDA призначає кожному SM кількість потоків, яка, як правило, більша, ніж та, яку він може одночасно виконати. Це вигідно, тому що багато інструкцій мають певну затримку після виконання. Наприклад, якщо деформація виконує завантаження з пам'яті, яке займає 16 циклів, SM може використати ці 16 циклів для виконання інструкцій на інших деформаціях. На кожному такті SM вибирає якомога більше вершин, готових до виконання інструкцій, і видає інструкції на них.

Щоб передбачити час виконання ядра, потрібно враховувати як кількість виконуваних інструкцій, так і їх затримки. Тому перейдемо до аналізу, який може бути використано для отримання цих величин і, на їх основі, приблизних меж часу виконання блоку ядра CUDA (обирається рівень блоків для цього аналізу, оскільки саме на цьому рівні відбувається синхронізація, і тому складання часу виконання між блоками є більш простим). Щоб отримати таку межу часу виконання, використовуватиметься результат з області паралельного планування, який прогнозує час виконання програм на основі їхньої роботи, загальних обчислювальних витрат (без урахування затримок) операцій, які мають бути виконані, і пробігу - часу, необхідного для виконання лише операцій на критичному шляху (з урахуванням затримок). Можна вважати роботу часом, необхідним для виконання програми, яка одночасно виконує лише один потік, а проміжок часу - часом, необхідним для виконання програми, яка одночасно виконує всі потоки (припускаючи нескінченно паралельне апаратне забезпечення). Знаючи ці дві величини, можна обмежити час виконання програми за умови, що вона виконується жадібним планувальником, тобто таким, що не залишає процесори простоювати, коли для них є робота.

Припустимо, що затримка запису до глобальної пам'яті домінує над тривалістю обчислень. Кожен варп виконує лише один запис, і тому, аналізуючи час виконання програми для одного варпу (або потоків 0-31, або 32-63), враховується затримка лише одного запису. Однак, через синхронізацію в середині блоку, тривалість блоку повинна враховувати затримку двох записів: потоки 32 і вище повинні чекати, поки потоки з 0 по 31 виконають свої записи, перш ніж продовжити роботу. Без більш детального знання коду неможливо дізнатися, як знайти правильний інтервал для блоку, склавши інтервали двох варпів. Замість цього виконується аналіз з використанням кількісної логіки одразу для потоків 0-63; при цьому передбачається, що всі 64 потоки виконуються синхронно, що не є точним. Однак, у цьому випадку це припущення дає гарне наближення: обидві умови аналізуються як потенційно розбіжні умови, в яких неактивні потоки чекають на активні потоки.

Перейдемо до розробки семантики аналізу паралельної вартості, мотивованої вище, яка моделює цілі блоки CUDA і відстежує вартість у термінах роботи та проміжку. Семантика вартості відстежує роботу та проміжок для кожної гілки. При кожній синхронізації розглядається максимальний проміжок між усіма варпами, щоб врахувати той факт, що всі варпи в блоці повинні чекати один на одного в цей момент. Вартість тепер є парою (c^w, c^s) роботи та проміжку, відповідно. Використовуються функції fst та snd, щоб отримати ліву та праву компоненти пари відповідно. Метрика ресурсів MC відображає константи ресурсів на витрати, що відображають кількість інструкцій та затримок, необхідних для виконання операції. Проекції MC_w та MC_s такої метрики ресурсів проектують компоненти роботи та проміжку відповідно для кожної вартості. Для цілей розрахунку проміжку припускається, що проміжок операції (другий компонент вартості) відображає час, необхідний для обробки інструкції плюс затримка (іншими словами, затримка - це проміжок операції мінус робота). Вартість блоку представляється як індексоване за варпами сімейство витрат C. Використовується $\emptyset$ для позначення колекції $\left((0,0)_{i \in Warps}\right)$, де $Warps$ - це множина всіх варпів у блоці. Для того, щоб додати вартість (c^w, c^s) до набору вартості тільки для варпів з множини W, можна використовувати скорочений запис $C \oplus W\ (c^w, c^s)$, який за формулою 2.31, де $WarpOf(t)$ - варп, що містить потік t. Для цілей цього визначення, варпи можна вважати абстрактними об'єктами, тому $WarpOf(t)$ є відображенням від потоків до варпів, яке зберігає властивість, що кожен варп містить рівно 32 потоки.

$$\left(C \oplus W\ (c^w, c^s)\right)_i \triangleq f(x) = \begin{cases} (c_0^w + c^w, c_0^s + c^s), & C_i = (c^w, c^s) \wedge i \in W \\ C_i, & \text{у інших випадках} \end{cases} \tag{2.31}$$

У даній формалізації, варпи ідентифікуються цілими числами, тому визначається $WarpOf(t) \triangleq Tid(t) mod\ 32$, де $Tid(t)$ - цілочисельний

ідентифікатор потоку t. Роботу колекції позначається через $W(C)$, а проміжок через $S(C)$. Можна обчислити роботу та проміжок блоку шляхом підсумовування (2.32) та взяття максимуму (2.33), відповідно.

$$W(C) \triangleq \sum_{i \in Warps} fst\ C_i \tag{2.32}$$

$$S(C) \triangleq \max_{i \in Warps} snd\ C_i \tag{2.33}$$

Семантика вартості обчислює блок потоків паралельно (на відміну від одного варпу з кроком блокування, як операційна семантика). Однак, на відміну від стандартної семантики великого кроку, включаючи як семантику на рівні варпу, так і паралельну семантику на рівні блоків, ця семантика не обчислює програму повністю до завершення. Натомість, вона обчислює лише до появи нового оператора синхронізації, або до завершення програми, якщо синхронізація не зустрічається. Оскільки всі варпи синхронізуються при кожній синхронізації і виконуються паралельно, можна розділити виконання ядра на блоці потоків на декілька сегментів, розділених викликами синхронізації; всі потоки виконують кожен сегмент до завершення перед синхронізацією, а потім всі потоки одразу переходять до наступного сегменту. Таким чином, семантика моделює виконання одного з цих сегментів. Зупинка на границях сегментів, а не продовження до повернення ядра, полегшує доведення відповідності з обмеженою реалізацією, а також робить більш зрозумілим, складання проміжків кожного сегменту. Результатом оцінки є продовження програми, яке має виконуватися після точки синхронізації, тобто оператор, який має бути виконаний для запуску наступного сегмента.

Правила для паралельної семантики вартості (2.34 – 2.39), подібні до правил для операндів і виразів з семантики кроку блокування.

$$SCR: Sync \Longrightarrow \frac{}{\sigma; C; sync \Downarrow_M^B \sigma; C \oplus_B MC^{sync}; \mathrm{cont}}, \tag{2.34}$$

$$SCR: SeqFull \Longrightarrow \frac{\sigma;C;\ s_1 \Downarrow_M^B \sigma_1;C';skip \quad \sigma_1;C';\ s_2 \Downarrow_M^B \sigma_2;C'';s_2'}{\sigma_1;C;s_1;\ s_2 \Downarrow_M^B \sigma_2;C'';s_2'}, \tag{2.35}$$

$$SCR: SeqPar1 \Longrightarrow \frac{\sigma;C;\ s_1 \Downarrow_M^B \sigma';C';s_1' \quad s_1' \neq skip \quad s_1' \neq cont}{\sigma_1;C;s_1;\ s_2 \Downarrow_M^B \sigma';C';s_1';s_2}, \tag{2.36}$$

$$SCR: SeqPar2 \Longrightarrow \frac{\sigma;C;\ s_1 \Downarrow_M^B \sigma';C';cont \quad s_2 \neq skip}{\sigma;C;s_1;\ s_2 \Downarrow_M^B \sigma';C';s_2}, \tag{2.37}$$

$$SCR: IfParT \Longrightarrow \frac{\sigma;C;\ e \downarrow_M^B (T)_{t \in B};C' \quad \sigma;C';\ s_1 \Downarrow_M^B \sigma';C'';s_1'}{\sigma;C;if\ e\ then\ s_1\ else\ s_2 \Downarrow_M^B \sigma';C'' \oplus_B MC^{if};s_1'}, \tag{2.38}$$

$$SCR: IfParF \Longrightarrow \frac{\sigma;C;\ e \downarrow_M^B (F)_{t \in B};C' \quad \sigma;C';\ s_2 \Downarrow_M^B \sigma';C'';s_2'}{\sigma;C;if\ e\ then\ s_1\ else\ s_2 \Downarrow_M^B \sigma';C'' \oplus_B MC^{if};s_2'} \tag{2.39}$$

Для виразів тепер маємо формалізацію, яка означає, що на множині потоків B зі сховищем σ оператор s виконується до завершення або до точки синхронізації, що призводить до кінцевого сховища σ' і продовження s'. Якщо колекція вартості на початку дорівнює C, то вартість, збільшена на вартість виконання s, дорівнює C'.

$$\sigma; C;\ s\ \Downarrow_M^B\ \sigma'; C'; s' \tag{2.40}$$

Правило 2.34 повертає $cont$ при оцінці синхронізації. Оператор $cont$ є заповнювачем, який буде розповсюджено у зовнішні оператори для допомоги у побудові повного продовження; правила побудовано таким чином, що кінцевий оператор не міститиме $cont$, якщо весь оператор, який оцінюється, не є синхронізованим (що можна припустити без втрати загальності, оскільки синхронізація наприкінці ядра мало впливає на продуктивність). Умови, послідовності та цикли мають декілька правил, залежно від того, чи обчислюються під-вирази повністю, чи зупиняються у точці синхронізації.

Правило 2.35 обробляє випадок, коли s_1 обчислюється повністю до досягнення синхронізації; на це вказує той факт, що продовження пропускається. У цьому випадку, як і у правилах з блокуванням кроку, відбувається перехід до обчислення s_2. Якщо s_1 не обчислюється повністю (тобто його продовження не пропускається), то не відбувається перехід до обчислення s_2, а одразу повертається

продовження. Якщо продовження s_1 точно дорівнює $cont$, то продовженням буде просто s_2; обчислення s_2 почнеться одразу після відновлення обчислення (це обробляється правилом 2.37). В іншому випадку, $cont$ може бути вкладений в інші оператори всередині продовження s_1, які повинні бути виконані після відновлення обчислень (правило 2.36). У спеціальному випадку, коли s_2 є пропуском, воно ігнорується і відбувається перехід до продовження.

Правило 2.38 обробляє умову, де e є істинною для всіх потоків - воно виконує s_1, доки той не завершиться або не досягне синхронізації. Правило 2.39 аналогічне для випадків, коли умова має значення false на всіх потоках. Правило $SCR: IfFull$ (є поєднанням 2.38 і 2.39) обробляє випадки, коли умова не є ні істинною для всіх потоків, ні хибною для всіх потоків, такий випадок не обов'язково вказує на розбіжність у варпі, оскільки можливо, що всі потоки беруть ту саму гілку в межах кожного варпу. Це правило вимагає, щоб обидва потоки s_1 і s_2 були пропущені, тобто жоден з них не може синхронізуватися, оскільки синхронізація в гілці умови, яка не приймається всіма потоками в блоці, є помилкою в CUDA. Крім того, правило $SCR: IfFull$ додає вартість MC^{div} лише до тих варпів, для яких умова розходиться, що формалізується виразом 2.41

$$\{w | \exists t_1 \in B_T, t_2 \in B_F . w = WarpOf(t_1) = WarpOf(t_2)\}, \qquad (2.41)$$

тобто розглядаються варпи, що містять як потік t_1, де умова істинна, так і потік t_2, де умова хибна.

Правило $SCR: WhileFull$ подібне до $SCR: IfFull$, включаючи розрахунок того, на які варпи нараховувати вартість розгалуження (правило не не повинне визначати B_F, тому множина потоків, для яких умова є хибною, записується $B \backslash B_T$). Синхронізація всередині циклу, де не всі потоки активні, також заборонена.

Переконаємось, що даний аналіз, в результаті застосовування незалежно до роботи чи відрізку, добре апроксимує семантику паралельних витрат. По-перше, перевіримо, що витрати, отримані за допомогою паралельної семантики,

перенаближені витратами, отриманими за допомогою семантики витрат з блокуванням кроків (розширеної правилом, що розглядає синхронізацію). По-друге, перевіримо обґрунтованість цієї семантики витрат. Семантика вартості кроку блокування була розроблена для моделювання лише однократного виконання, і тому може здатися дивним моделювати цілий блок, використовуючи її. Однак, це призводить до надмірного наближення: наприклад, у ядрі, де семантика вартості кроку блокування ігнорує синхронізацію, але припускає, що дві гілки повинні виконуватися послідовно, оскільки не всі потоки беруть одну і ту ж гілку. Якщо обчислення виразу e з початковим набором витрат C призводить до набору витрат C', то e можна обчислити за допомогою MC_w та MC_s, згідно з семантикою блокувального кроку, з витратами C_w та C_s, відповідно. Різниця в проміжку між C і C' переоцінена на C_s, а різниця в роботі переоцінена на C_w, помножену на кількість варпів. В результаті маємо: якщо $\Sigma \vdash e : \beta$, $\Sigma \vdash_\beta \sigma$ і $\sigma; C;\ e \downarrow^B_M R; C'$, тоді існують такі C_w та C_s, що для них характерно:

1. $\sigma; e \downarrow^B_{M_w} R; C_w$
2. $\sigma; e \downarrow^B_{M_S} R; C_S$
3. $W(C') - W(C) \leq C_w\ |\{WarpOf(t) | t \in B\}|$
4. $S(C') - S(C) \leq C_s$

Аналогічні міркування можуть бути застосовані і для інструкцій.

Об’єднавши ці вирази та правила для використання ресурсів в межах одного варпу, отримаємо, що якщо $\Sigma \vdash s,\ \Sigma \vdash_\beta \sigma,\ \vdash_M \{L; Q_w; Y_w\}\ s\ \{L'; Q'_w; Y'_w\}$, $\vdash_M \{L; Q_s; Y_s\}\ s\ \{L'; Q'_s; Y'_s\}$, $\sigma;\ B \vDash \{L; Q_w; Y_w\}$, $\sigma;\ B \vDash \{L; Q_s; Y_s\}$ і $\sigma; \emptyset; s\ \Downarrow^B_M \sigma'; C$, тоді $W(C) \leq \left(Q_{w_{Y_w}}(\sigma) - Q'_{w_{Y'_w}}(\sigma')\right) |\{WarpOf(t) | t \in B\}|$ і $S(C) \leq Q_{s_{Y_s}}(\sigma) - Q'_{s_{Y'_s}}(\sigma')$. А отже семантика оцінки вартості ресурсу, що була розроблена в межах дослідження роботи одного варпу, може бути використана і для апроксимації часу роботи блоку.

2.5 Висновки

У розділі представлено моделі, що дозволяють репрезентувати використання ресурсів і час виконання ядра. Зокрема було визначено "семантику вартості кроку блокування", яка описує правила оцінки ядра при відстеженні вартості виконання. Вона передбачає набір потоків B , які виконуються з кроком блокування, тобто один варп графічного процесора. Визначено кількісну програмну логіку для обмеження вартості виконання ядра без його запуску або симуляції його виконання. Ключовим результатом є обґрунтованість моделі, тобто те, що вартість, передбачена логікою для даного ядра, є щонайменше вартістю фактичного виконання ядра, як це відслідковується семантикою вартості. Однак обмеження вартості виконання одного варпу, не пояснює як порівнювати вартість виконання ядра, коли активними є більше ніж один варп. Тому, враховуючи, що отримані результати в теорії паралельного планування пов'язували час виконання в таких випадках з кількістю процесорів (в даному випадку, кількістю варпів, які можуть виконуватися одночасно) та двома кількісними властивостями програми, відомими як робота і проміжок, було додатково продемонстровано, як модель може бути використана для обмеження роботи і проміжку блоку потоків. Таким чином, доведено, що навіть якщо логіка була розроблена лише для одного варпу шляхом використання "паралельної" семантики вартості, вона може бути використана для моделювання паралельне виконання блоку.

Отриманими таким чином моделями можна користуватися при використанні архітектури CUDA для оптимізації паралельної обробки даних. Зокрема, для оцінки доцільності залучення обчислень на графічному процесорі.

3 УДОСКОНАЛЕНИЙ МЕТОД ОПТИМІЗАЦІЇ ПАРАЛЕЛЬНОЇ ОБРОБКИ ДАНИХ ЗГІДНО З КЛАСТЕРНОЮ АРХІТЕКТУРОЮ CUDA

3.1 Основи удосконаленого методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA

З метою досягнення оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA, розглянемо основи синтезу методу, що дозволить ефективніше обробляти дані паралельно.

Ефективна паралелізація алгоритму дозволяє кратно зменшити час його роботи, за рахунок одночасного виконання взаємно-незалежних частин. Архітектура CUDA ж в свою чергу дозволяє отримати перевагу ще більшу перевагу в часі завдяки пришвидшенню паралельного виконання. Однак CUDA є складним і комплексним інструментом, тому важливим є розуміння як найкраще використовувати ресурси CUDA для підвищення продуктивності без непотрібного збільшення витрат або складності системи.

Почнемо з передачі даних між CPU та GPU. Оскільки переважна більшість операцій виконується саме на CPU, а CUDA є програмно-апаратною архітектурою, що використовує GPU, оптимізація передачі даних між цими процесорами є одним з фундаментальних аспектів оптимізації. Незалежно від алгоритму, ефективний обмін даними між основною пам'яттю та пам'яттю GPU визначає загальну продуктивність системи. Організація даних, їх пакування та розподіл перед передачею на GPU може значно покращити загальну продуктивність обчислень.

Ядро (kernel) CUDA є центральним елементом у виконанні паралельних обчислень і представляє собою набір операцій, що виконуватимуться паралельно у кожному потоці. Ядро є специфічним для кожного алгоритму і вимагає індивідуального підходу. Створення ефективного ядра вимагає глибокого розуміння як самого алгоритму, так і можливостей CUDA. При описі ядра необхідно гарантувати атомарність роботи з даними, таким чином, щоб кожен потік працював з фрагментом даних, який не може бути змінений іншим потоком і не чекав результатів іншого потоку.

Ефективне використання архітектури GPU залежить від грамотного визначення кількості блоків і потоків, що відіграє ключову роль в оптимізації обчислень. Для того, щоб максимально використати можливості GPU, потрібно уважно підходити до вибору кількості потоків у блоку, орієнтуючись на характеристики варпів, що в свою чергу, дозволяє зменшити простій та підвищити загальну ефективність виконання програми. Такий підхід допомагає забезпечити, щоб кожен SM був оптимально завантажений, але при цьому не перевантажений, що може спричинити зайву конкуренцію за обчислювальні ресурси.

Зважаючи, що виконання потоками різних інструкцій змушує GPU виконувати кожен шлях послідовно, відповідно, збільшуючи час виконання, варто уникати розгалужень у коді ядра. Код варто структурувати так, щоб уникнути умовних операторів, які ведуть до розгалуження в потоках одного варпа. У випадку, коли це не можливо, потрібно мінімізувати різницю у шляхах виконання.

Програмно-апаратна архітектура CUDA пропонує різні типи пам’яті: регістри, локальну, спільну, глобальну, текстурну і константу пам’яті. Вибір правильного типу пам'яті в CUDA і ефективне їх комбінування має критичне значення для забезпечення високої продуктивності, швидкого доступу до даних і зменшення затримок. Таким чином, при виборі типу пам’яті, варто розуміти, що регістри, це швидка, але невелика пам’ять кожного потоку, коли її не вистачає варто використати локальну пам’ять, що має хоч і вищі затримки, але й вищий об’єм. Спільну пам’ять корисно використовувати для доступу до даних потоків одного блоку, тоді як глобальна пам’ять доступна всім потокам і блокам. Константна і текстурна пам’яті є кешованими і оптимізованими, перша призначена для зберігання констант доступних усім потокам, а друга – для специфічних сценаріїв звернення, які не змінюють дані.

Важливість правильного вибору кількості блоків і потоків в CUDA не можна недооцінювати, оскільки від цього залежить ефективність використання ресурсів графічного процесора. Кожен блок складається з потоків, які виконуються на Streaming Multiprocessors (SM). Конфігурація гріду також має велике значення. Грід складається з декількох блоків і визначає загальну кількість блоків, які будуть

запущені на GPU. Правильний вибір розміру гріда впливає на здатність алгоритму масштабуватися і ефективно обробляти дані на великому обсязі. Якщо грід занадто великий, це може призвести до зайвих обчислень і збільшення часу на управління. З іншого боку, якщо грід занадто малий, деякі обчислювальні потужності GPU залишаться невикористаними, що знижує загальну продуктивність.

Ці інструменти й процеси, а головне – їх розуміння і вміння правильно використовувати, формують фундамент для оптимізації різних алгоритмів за допомогою CUDA і є критично важливими для розробки високопродуктивних паралельних обчислювальних систем.

3.1.1 Синтез методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA

Перед прийняттям рішення про необхідність використання архітектури CUDA для оптимізації паралельної обробки даних, необхідно провести аналіз алгоритму, щоб зрозуміти чи буде його паралельна версія ефективно виконуватись на GPU (важливими є загадані раніше кількість розгалужень, і атомарність даних). Крім того, переваги від паралелізму на GPU стають помітними при обробці великих об’ємів даних, де затрати на передачу даних компенсуватимуться зниженням часу обчислень. На прикладі множення матриць – не має сенсу переносити на графічний процесор обрахунки над матрицями розміром менше 1000x1000, бо загальна продуктивність системи, або не зросте взагалі, або знизиться.

Після проведеного аналізу алгоритму, потрібно обрати відповідну архітектуру GPU. Різні моделі GPU мають різну кількість ядер та об'єми пам'яті, що впливає на швидкість і ефективність обчислень. Важливо враховувати баланс між потужністю обчислень та вартістю GPU, а також забезпечити сумісність із наявною інфраструктурою та програмним забезпеченням. Належний вибір дозволяє досягти максимальної продуктивності при мінімальних витратах.

Враховуючи, що основні операції описані саме в ядрі, від його оптимальності і відповідності специфіці CUDA і залежить основний успіх оптимізації. Ефективне

ядро максимізує використання обчислювальних ресурсів GPU, забезпечуючи оптимальний розподіл навантаження між потоками та блоками, мінімізуючи затримки та зайве використання пам'яті.

Наступним етапом впровадження оптимізації є вибір серед потрібного типу (чи комбінації типів) пам’яті серед пропонованих програмно-апаратною архітектурою. Рішення про використання різних типів пам'яті має базуватися на обсягах обчислень і потребі в швидкодії доступу до даних. Не ефективне використання пам'яті може стати причиною зниження продуктивності системи.

Оскільки, на графічному процесорі відбуваються переважно обчислення, необхідно організувати передачу даних таким чином, щоб всі подготовчі процеси, як збір, фільтрація, нормалізація тощо, відбувались на стороні CPU, а на GPU передавалась мінімальна їх кількість у форматі готовому до використання у ядрі. Аналогічно, після обчислень, дані варто передати до центрального процесору у максимально ефективний спосіб, для зменшення потенційних затримок.

Визначення оптимального розміру блоку — це ключовий аспект для забезпечення максимальної ефективності. Розмір блоку повинен бути достатньо великим, щоб максимально завантажити кожний SM, але при цьому не настільки великим, щоб викликати конкуренцію за ресурси або перевищення ліміту реєстрів чи пам'яті на SM. Балансування між кількістю блоків і кількістю потоків в кожному блоку є критичним для забезпечення оптимального використання GPU. Оптимальна кількість потоків в блоку часто кратна 32, що відповідає розміру варпа в архітектурі NVIDIA. Це важливо для мінімізації кількості невикористаних потоків і збільшення швидкодії виконання алгоритмів. Кількість блоків у гріді повинна бути достатньою, щоб кожен SM мав достатньо роботи. Ідеально, коли всі SM зайняті, але без перевантаження, яке може спричинити зниження продуктивності через збільшення звернень до глобальної пам'яті та чекання завершення інструкцій. Визначення ідеальної кількості блоків і потоків залежить від багатьох факторів, включно з розміром даних, кількістю доступних ресурсів на GPU та специфікою самого алгоритму. Тому при оптимізації паралельної обробки даних з використанням CUDA, необхідно враховувати не тільки алгоритмічні

вимоги, але й архітектурні особливості GPU. Коректний підбір цих параметрів може значно підвищити ефективність виконання алгоритму, знизити час обчислень та забезпечити високу швидкість виконання паралельних задач.

Фінальним етапом можна вважати тестування та оцінку продуктивності системи. Під час тестування аналізуються різні аспекти системи, включаючи швидкодію обчислень, точність результатів та стабільність роботи та споживання ресурсів. Оцінка споживання ресурсів дає інформацію про використання пам'яті і обчислювальних потужностей. Це дозволяє зробити висновки про ефективність системи та ідентифікувати можливі місця для подальших оптимізацій, що може включати перерозподіл завдань між CPU і GPU або перегляд алгоритмів для кращого використання паралельних обчислень. Тестування дозволяє виявити та вирішити потенційні проблеми, що можуть виникнути під час виконання паралельних алгоритмів на GPU

Метод оптимізації паралельної обробки даних включає наступні кроки:

1. Аналіз потреб обчислень і оцінка доцільності використання CUDA.
2. Вибір відповідної архітектури GPU
3. Створення оптимізованого ядра CUDA.
4. Вибір використовуваних типів пам'яті CUDA.
5. Оптимізація передачі даних між CPU і GPU.
6. Конфігурація кількості блоків і потоків.
7. Тестування та оцінка продуктивності системи.

Ці кроки формують основу для створення ефективної, масштабованої та високопродуктивної обчислювальної системи, яка може адаптуватися до широкого спектру паралельних методів обробки даних.

3.2 Дослідження удосконаленого методу оптимізації паралельної обробки даних згідно з архітектурою CUDA на основі застосування швидкого перетворення Фур’є (FFT)

3.2.1 Базова паралелізація швидкого перетворення Фур’є (FFT) для пришвидшення обробки даних

З метою дослідження удосконаленого методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA, його було імплементовано на алгоритмах з різними математичними моделями і областями застосування. Першим таким алгоритмом є швидке перетворення Фур'є (FFT).

Паралелізація є причиною суттєвого прогресу в ефективності виконання дискретного перетворення Фур'є (DFT), яке є основним інструментом в аналізі сигналів. DFT перетворює послідовність чисел, що представляють сигнал у часовому домені, в компоненти частотного домену. Кожен компонент в цьому перетворенні вказує на амплітуду та фазу синусоїди, що відповідає конкретній частоті.

DFT трансформує числову послідовність, що представляє сигнал у часовому домені, у складові частотного домену, з'ясовуючи амплітуду та фазу синусоїд, пов'язаних з конкретними частотами. Оптимізація DFT через паралелізацію, зокрема за допомогою швидкого перетворення Фур'є (FFT), істотно зменшує час обчислень.

FFT, застосовуючи стратегії паралелізації, мінімізує обчислювальні витрати порівняно з прямим застосуванням DFT, розділяючи складну задачу на більш прості підзадачі, що ефективно вирішуються на паралельних обчислювальних системах. Таке розбиття дозволяє здійснювати обчислення паралельно, зменшуючи загальний час обробки і підвищуючи продуктивність системи.

Розуміння ролі DFT та FFT у аналізі сигналів демонструє важливість частотного аналізу у багатьох областях, включаючи цифрову обробку сигналів, радіокомунікації та обробку зображень. Ефективність, з якою FFT виконує цей аналіз, робить його критичним елементом у процесі оптимізації паралельної

обробки даних, дозволяючи досягти значних вигод у швидкості та точності обчислень.

Математичне формулювання алгоритму FFT можна розглядати як процес, що спрощує обчислення дискретного перетворення Фур'є (DFT). В основі DFT лежить перетворення послідовності комплексних чисел із часового домену в частотний домен. Загальна формула (3.1) DFT виглядає наступним чином:

$$X(k) = \sum_{n=0}^{N-1} x(n) * e^{-\frac{i2\pi kn}{N}}, \quad (3.1)$$

де $X(k)$ – k-та частотна компонента,

$x(n)$ – n-й-відлік в часовому домені,

N – загальна кількість відліків,

k – індекс конкретної частотної компоненти,

$e^{-\frac{i2\pi kn}{N}}$ – комплексний множник, що визначає фазовий зсув.

FFT оптимізує цей процес, розділяючи DFT на менші обчислення за допомогою методу "розділяй і володарюй". Це досягається шляхом розбиття первинної послідовності на дві частини: одна з парних індексів і одна з непарних. Таким чином, DFT для послідовності може бути виражений як сума двох DFT меншого розміру (формула 3.2):

$$DFT\big(x(n)\big) = DFT\big(x(2m)\big) + e^{-\frac{i2\pi kn}{N}} * DFT\big(x(2m+1)\big), \quad (3.2)$$

де m – індекс, що проходить через половину вихідних точок сигналу.

Цей підхід дозволяє виконувати обчислення ефективніше, адже кожну меншу частину послідовності можна обчислити паралельно. Циклічна структура експоненційних множників у формулі FFT використовується для зменшення загальної кількості обчислювальних операцій.

У результаті, FFT забезпечує значне зниження часу обчислень у порівнянні з безпосереднім виконанням DFT, переходячи з квадратичної складності $O(N^2)$ до $O(N \log N)$, де N – кількість відліків сигналу.

Алгоритм швидкого перетворення Фур'є (FFT) втілює принцип розпаралелення завдяки своїй здатності розділяти комплексне обчислення DFT на менші, більш керовані сегменти. Ця декомпозиція дозволяє ефективно використовувати сучасні багатоядерні процесори або спеціалізоване обчислювальне обладнання для одночасної обробки декількох частин сигналу.

Цей процес можна уявити як дерево, де кожен вузол представляє DFT меншого розміру. На кожному рівні дерева виконуються окремі DFT для парних і непарних елементів попереднього рівня. Рекурсивний поділ продовжується до того моменту, поки не буде досягнуто рівня, на якому кожен DFT має лише один або два елементи, що є базовим випадком обчислення.

Паралельне виконання цих менших DFT може значно прискорити загальний процес обчислення, оскільки кожна частина обробляється одночасно на різних процесорних ядрах або в різних обчислювальних вузлах. Отже, час виконання алгоритму значно скорочується, особливо для великих наборів даних.

Розпаралелення алгоритму FFT не тільки збільшує швидкість обчислень, але й підвищує загальну ефективність обчислювальних систем. Воно дозволяє оптимально використовувати доступні обчислювальні ресурси, знижуючи час простою та підвищуючи продуктивність. Таким чином, розпаралелення є ключовим аспектом у досягненні високопродуктивних обчислень, забезпечуючи ефективне виконання комплексних обчислювальних задач в режимі реального часу.

Ось як це відбувається:

- На першому рівні рекурсії, вихідний масив розбивається на дві половини, які можуть бути оброблені паралельно.
- На другому рівні, кожна з цих половин знову розбивається і так далі, до того моменту, коли досягається базовий випадок алгоритму (зазвичай, коли масив містить лише один або два елементи).

Цей процес може бути виконаний паралельно на кожному рівні рекурсії. Використання багатопоточності або багатоядерних обчислювальних систем дозволяє ефективно обробляти кожну частину сигналу одночасно, що значно знижує загальний час обчислення FFT.

3.2.2 Впровадження архітектури CUDA для оптимізації FFT

Процес обробки сигналу за допомогою FFT і CUDA починається зі зчитування сигналу. Це може бути аудіосигнал, відеопотік, радіочастотний сигнал або будь-який інший тип даних, що потребує аналізу частотних характеристик. На цьому етапі сигнал зазвичай перебуває у часовому домені, де кожен відлік представляє амплітуду сигналу в певний момент часу.

Після зчитування сигнал піддається попередній обробці на CPU. Ця обробка може включати нормалізацію, фільтрацію для видалення шуму або інші види підготовки, необхідні для підвищення якості результатів FFT. Під час цієї фази сигнал також може бути перетворений або переформатований для оптимальної сумісності з вимогами FFT.

Наступним кроком є передача обробленого сигналу з CPU на графічний процесор (GPU) через використання CUDA. Ця передача є ключовою, оскільки GPU забезпечує велику обчислювальну потужність, яка необхідна для ефективного виконання алгоритму FFT. Передавши дані на GPU, можна використовувати тисячі потоків для паралельної обробки сигналу, значно прискорюючи процес.

На GPU виконується сам алгоритм швидкого перетворення Фур'є. FFT перетворює сигнал із часового домену в частотний, розкладаючи його на складові з різними частотами. Це дозволяє аналізувати спектральний склад сигналу та ідентифікувати основні частотні компоненти, що мають важливе значення для подальшої обробки та аналізу.

Розглянемо більш детально, що потрібно зробити для оптимізації FFT з використанням архітектури CUDA. Почнемо з опису ядра (kernel), тобто основної виконуваної функції. Для алгоритму швидкого перетворення Фур'є (FFT), ядро

буде відповідальне за розрахунок елементарних DFT, бітову інверсію індексів, обчислення твідл-факторів та комбінування вихідних результатів. Перша за все, ядро виконуватиме обчислення DFT для менших сегментів сигналу на кожному етапі рекурсивного поділу FFT, сюди ж входить розрахунок складних експонент для отримання спектральних компонентів. Крім розрахунку самих DFT, для ефективного виконання FFT необхідно переставити елементи вхідного масиву відповідно до бітової інверсії їх індексів, що дозволить правильно вирівнювати дані для подальших етапів перетворення. Після обчислення DFT на різних рівнях рекурсії, ядро займається комбінуванням цих результатів для формування остаточного результату FFT. Це включає в себе сумування розрахованих DFT з відповідними ваговими коефіцієнтами. Для здійснення інверсії векторів на кожному етапі FFT, ядро обчислює твідл-фактори (складні експоненціальні множники) та застосовує їх до проміжних результатів.

Тепер, знаючи основні функції ядра, можна переходити до наступного важливого етапу - вибору пам'яті GPU, у якій зберігатимуться дані в ході розрахунків. Для оптимізації FFT може бути використана спільна пам'ять (shared memory), яка пропонується програмно-апаратною архітектурою CUDA і є швидкою та ефективною у контексті доступу потоків в одному блоку на GPU. У випадку FFT, спільна пам'ять використовується для зберігання проміжних результатів обчислень, зокрема твідл факторів і часткових сум, що дозволяє зменшити частоту і обсяг звернень до глобальної пам'яті, яка має вищу затримку.

Додатково, для зберігання вхідних та вихідних даних може використовуватися глобальна пам'ять GPU. Це включає первісний масив сигналів у часовому домені та кінцеві результати у частотному домені. Хоча доступ до глобальної пам'яті є повільнішим, вона забезпечує необхідний обсяг для зберігання великих масивів даних, які не вміщуються у спільну пам'ять. Така стратегія дозволяє збалансувати швидкість та обсяг пам'яті, оптимізуючи загальну продуктивність обчислень.

Для оптимізації використаних ресурсів графічного процесору потрібно правильно вибрати розміри блоку та форму гріда. Зазвичай для FFT вибір розміру

блоку впливає на кількість рівнів рекурсії, які можуть бути ефективно оброблені в межах одного блоку. Розмірність гріда, з іншого боку, дозволяє керувати загальною кількістю блоків, які виконуються паралельно. У контексті FFT, це дозволяє розподілити обробку сигналу між багатьма блоками, забезпечуючи, що кожен блок ефективно обробляє частину даних.

По завершенні обчислень FFT на GPU, результати передаються назад на CPU. На цій стадії оброблені дані можуть бути використані для виведення, подальшої обробки або аналізу залежно від конкретного застосування.

1. Аналіз математичної модель FFT і виділення обрахунків DFT як обчислень, що виконуватимуться паралельно.

2. Опис ядра FFT, в якому виконуватимуться розрахунок елементарних DFT, бітова інверсію індексів, обчислення твідл-факторів та комбінування вихідних результатів.

3. Вибір використовуваної пам’яті: спільна, для збереження проміжних обрахунків і глобальної для отримання й передачі даних.

4. Опис конфігурація блоків і потоків CUDA для ефективного використання ресурсів.

При фактичній імплементації швидкого перетворення Фур’є в конкретну систему з детермінованим набором даних, потрібно ще буде обрати архітектуру і конкретні значення для конфігурації блоків і потоків, основуючись на обсягу вхідних даних.

3.3 Дослідження удосконаленого методу оптимізації паралельної обробки даних згідно з архітектурою CUDA на основі імплементації алгоритму k-найближчих сусідів

3.3.1 Базова паралелізація швидкого перетворення Фур’є (FFT) для пришвидшення обробки даних

Наступною частиною дослідження удосконаленого методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA, було його застосування до алгоритму k-найближчих сусідів.

Вивчення алгоритму k-найближчих сусідів (KNN) як частини дослідження оптимізації паралельної обробки даних демонструє величезний потенціал паралельних обчислень. Алгоритм KNN застосовується в широкому спектрі застосувань, включаючи класифікацію зображень та системи рекомендацій, де він вимагає обчислення відстаней між множинами багатовимірних точок даних. Використання паралельної обробки даних може істотно прискорити ці обчислення, особливо при роботі з великими наборами даних.

Процес роботи KNN починається з вибору кількості сусідів k, яка є ключовим параметром алгоритму. Під час класифікації об'єкт призначається до класу, який домінує серед його k найближчих сусідів. У випадку регресії визначається середнє або інше агреговане значення відповідей сусідів. Ця операція вимагає вимірювання відстані між об'єктами, для чого зазвичай використовується евклідова відстань, хоча можливі й інші метрики залежно від специфіки даних та завдання.

Важливою особливістю KNN є його лінійність. Алгоритм використовує весь навчальний набір даних для визначення найближчих сусідів під час фази тестування. Це означає, що час виконання KNN залежить від розміру набору даних, що може призвести до значного збільшення обчислювальних витрат для великих наборів даних.

Паралелізація процесу обчислення в KNN включає розподіл завдань обчислення відстаней між окремими блоками і потоками процесора, що дозволяє кожному потоку обробляти свою частину даних незалежно. Завдяки паралелізації, KNN стає більш доступним для виконання на сучасних обчислювальних системах, що вимагають швидкої обробки великих об'ємів даних. Використання потужності паралельної обробки дозволяє реалізувати більш ефективні рішення і є прикладом того, як можна досягти високої продуктивності обчислень, зменшуючи час виконання завдань та забезпечуючи більш точні і оперативні аналітичні дані в різних областях застосування. Ця оптимізація демонструє силу паралельної

обробки даних як ключового елементу в сучасних обчислювальних системах, що забезпечує швидкість і масштабованість необхідних обчислювальних процесів.

Математичне формулювання методу k-найближчих сусідів (KNN) зосереджується на вимірюванні відстаней між точками даних у просторі ознак. Для визначення найближчих сусідів точки, алгоритм використовує метрику відстані, як-от евклідову, для визначення близькості між точками. Для двох точок у n-вимірному просторі $P = (p_1, p_2, \dots, p_n)$ та $Q = (q_1, q_2, \dots, q_n)$, евклідова відстань $d(P, Q)$ визначається за формулою 3.3.

$$d(P, Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}, \quad (3.3)$$

Після обчислення відстаней між тестовою точкою та всіма точками в навчальному наборі даних, вибирається k точок з найменшою відстанню до тестової точки. Ці точки вважаються найближчими сусідами.

У задачі класифікації KNN призначає тестовій точці клас, який найчастіше зустрічається серед її k найближчих сусідів. Математично це можна виразити як голосування більшістю (вираз 3.4).

$$\text{Клас}(P) = \text{мода}\big(\text{Класи}(k \text{ найближчих сусідів})\big), \quad (3.4)$$

У задачі регресії вихідним значенням для тестової точки буде середнє (або інше агреговане значення) (вираз 3.4) відповідей її k найближчих сусідів:

$$\text{Значення}(P) = \frac{1}{k}\sum_{i=1}^{k} y_i, \quad (3.5)$$

де y_i – значення і-го найближчого сусіда.

Таким чином, метод KNN використовує ідею "більшість вирішує" для класифікації або визначення середнього значення з найближчих сусідів для

регресії, враховуючи відстань між точками у просторі ознак для визначення "найближчих".

Розпаралелювання алгоритму k-найближчих сусідів може значно підвищити його ефективність, особливо при роботі з великими наборами даних. В основі алгоритму KNN лежить пошук k найближчих сусідів для кожної тестової точки, що здійснюється шляхом обчислення відстаней між тестовою точкою та всіма точками навчальної вибірки. Такий процес обчислення відстаней є ідеальним для паралельної обробки, оскільки відстані для різних тестових точок можуть бути обчислені незалежно одна від одної.

Паралельне виконання алгоритму KNN може бути реалізовано шляхом розділення навчальної вибірки на декілька сегментів, які можуть бути оброблені одночасно в різних потоках виконання. Кожен потік може відповідати за обчислення відстаней від тестових точок до точок у певному сегменті навчальної вибірки та визначення найближчих сусідів у цьому сегменті.

3.3.2 Впровадження архітектури CUDA для оптимізації алгоритму KNN

Перед тим як передати дані на GPU для виконання алгоритму KNN, необхідно виконати попередню обробку на CPU. Попередня обробка включає нормалізацію даних, щоб забезпечити єдину шкалу для всіх ознак, оскільки KNN чутливий до масштабу ознак. Нормалізовані дані допомагають покращити точність визначення найближчих сусідів, уникнути спотворень у вимірах відстані між точками. Також, на цьому етапі вибираються ознаки, які будуть використовуватися для класифікації, та відбувається їх перетворення у відповідний формат, який можна ефективно обробляти на GPU. Може включатися підготовка метаданих, таких як індекси та лейбли класів, які будуть використовуватися у подальших обчисленнях на GPU. В результаті, на CPU формується масив ознак навчального набору даних, який потім передається на GPU для виконання паралельних обчислень KNN. Відправлення оброблених даних з CPU на GPU для паралельного виконання обчислень KNN включає масив точок (ознак), відповідних лейблів

класів і набір тестових даних, тобто масив точок, для яких потрібно знайти найближчих сусідів. Ці дані використовуються для обчислення відстаней до навчальних точок на GPU.

Ядро CUDA для KNN виконуватиме обчислення відстаней між тестовими та навчальними точками та визначення k найближчих сусідів. У цьому ядрі кожен потік відповідатиме за обчислення відстані від однієї тестової точки до всіх навчальних точок, зберігання відстаней та індексів та вибір k найближчих сусідів для цієї тестової точки. Спочатку ядро виконує паралельне обчислення відстані між тестовою точкою і кожною навчальною точкою. Це може бути реалізовано за допомогою евклідової або іншої метрики відстані. Для кожної тестової точки ядро зберігає обчислені відстані та відповідні індекси навчальних точок в структурі даних, такій як масив або пріоритетна черга, для подальшого визначення найближчих сусідів. Після обчислення всіх відстаней дані використовуються для визначення k найближчих сусідів для тестової точки. Це може бути здійснено шляхом сортування відстаней або використанням алгоритмів пошуку k-ї найменшої відстані.

Для KNN найбільш критичною частиною є обчислення відстаней між тестовими та навчальними точками. Ця операція вимагає інтенсивного доступу до даних. Тому для тимчасового зберігання навчальних даних, які активно використовуються потоками в межах одного блоку слід використати спільну пам’ять. Потоки в межах одного блоку можуть швидко обмінюватися даними через спільну пам'ять, зменшуючи звернення до більш повільної глобальної пам'яті. У KNN часто використовуються одні й ті ж навчальні дані для обчислення відстаней до різних тестових точок, тому кешування цих даних у спільній пам'яті дозволяє зменшити загальну кількість обчислень. Для зберігання проміжних результатів обчислення відстаней і тимчасових значень, кожен потік може використовувати свої локальні регістри. Це забезпечує найшвидший доступ до даних і зменшує потребу в спільній пам'яті. Решта даних може зберігатися в глобальній пам'яті, з якої вони будуть за потреби переміщені в спільну пам'ять. Такий підхід зменшить затримки доступу до даних і збільшить продуктивність алгоритму.

Знаючи, які функції виконуватиме ядро і які типи пам’ті варто використовувати, можна переходити до підборів розмірів грідів і блоків. Кожен блок у CUDA може обробляти частину навчальних даних, виконуючи обчислення відстаней для певної підмножини тестових точок. Розмір блоку впливає на кількість одночасно оброблюваних тестових точок. Великі блоки можуть забезпечити більшу пропускну здатність, але також можуть призвести до невикористання обчислювальних ресурсів через обмеження пам'яті блоку. Грід визначає загальну кількість блоків, які будуть використані для обчислень. Розмір гріда повинен бути достатнім для покриття всіх тестових точок, які потрібно класифікувати. Надмірно великі гріди можуть призвести до зайвих обчислень і збільшення витрат на управління. Наприклад, якщо кожен блок обробляє 128 тестових точок і в навчальному наборі є 1024 точки, то для ефективного використання може бути потрібно 8 блоків. Якщо розмір гріда встановлено надто великим, наприклад 16 блоків, деякі блоки будуть простоювати без завдань, використовуючи обчислювальні ресурси неефективно. З іншого боку, якщо розмір гріда буде надто малим, деякі тестові точки не будуть оброблені в одному проході, що призведе до необхідності додаткових ітерацій і збільшення загального часу обчислень. Таким чином, важливо знайти баланс, вибираючи розміри грідів і блоків, щоб максимізувати використання обчислювальних ресурсів GPU та мінімізувати час виконання обчислень.

Після завершення обчислень на GPU, результати виконання алгоритму KNN потрібно передати назад на CPU для подальшого аналізу та використання у прикладних завданнях. Оптимізація процесу передачі даних включає в себе використання асинхронного переносу даних, коли це можливо, для зменшення часу простою CPU і GPU, дозволяючи їм виконувати інші задачі під час процесу передачі даних.

1. Аналіз математичної моделі k-найближчих сусідів і виділення обрахунків відстаней від тестових точок до точок у певному сегменті навчальної вибірки та визначення найближчих сусідів у цьому сегменті, як функцій, що виконуватимуться паралельно.

2. Опис ядра, в якому виконуватимуться обчислення відстані від однієї тестової точки до всіх навчальних точок, зберігання відстаней та індексів та вибір k найближчих сусідів для цієї тестової точки.

3. Вибір використовуваної пам’яті: спільна, для тимчасового зберігання навчальних даних і локальні регістри, для зберігання проміжних результатів обчислення відстаней.

4. Опис конфігурація блоків і потоків CUDA для ефективного використання ресурсів, з урахуванням кількості точок.

При фактичній імплементації методу k-найближчих сусідів в конкретну систему з детермінованим набором даних, як і для швидкого перетворення Фур’є, потрібно ще буде обрати архітектуру і конкретні значення для конфігурації блоків і потоків, основуючись на обсягу вхідних даних.

3.4 Дослідження удосконаленого методу оптимізації паралельної обробки даних згідно з архітектурою CUDA на основі імплементації алгоритму k-найближчих сусідів

3.4.1 Базова паралелізація швидкого перетворення Фур’є (FFT) для пришвидшення обробки даних

Наступною частиною дослідження удосконаленого методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA, було його застосування до алгоритму Дейкстри, який використовується для знаходження найкоротшого шляху від однієї вершини графу до всіх інших. Алгоритм Дейкстри є критично важливим у маршрутизації і геопросторовому аналізі, тому паралелізація обробки даних у ньому принесе багато користі.

Обробка даних починається з стартової вершини графу і поступово розширює найкоротші шляхи до всіх доступних вершин. Спочатку встановлюється дистанція до початкової вершини як 0, а до всіх інших вершин - як нескінченність. Це можна представити як $dist[v] = \infty$ для всіх вершин v, окрім початкової, для якої $dist[start] = 0$. На кожному кроці алгоритм вибирає вершину u з найменшою

оцінкою дистанції, яка ще не була оброблена, і розглядає всі її сусідні вершини v. Для кожної сусідньої вершини v, алгоритм оновлює $dist[v]$, якщо $dist[u] + w(u,v) < dist[v]$, де $w(u,v)$ - вага ребра між u і v. Якщо при релаксації ребра знайдено коротший шлях до вершини, то дистанція до цієї вершини оновлюється. Таким чином, $dist[v]$ стає рівним $dist[u] + w(u,v)$. Процес продовжується, поки усі вершини не будуть оброблені. Алгоритм закінчується, коли всі вершини мають остаточні оцінки найкоротших шляхів.

Результатом роботи алгоритму є мінімальна відстань до кожної вершини від стартової вершини і, як правило, шлях для кожної вершини, що дозволяє відновити найкоротший шлях.

Розпаралелення алгоритму Дейкстри передбачає розподіл роботи з обчислення найкоротших шляхів між багатьма процесорами або потоками, що значно збільшує його ефективність, особливо для великих графів.

Описати паралелізацію можна наступним чином: кожен або потік ініціалізує свою частину даних, встановлюючи дистанції до нескінченності, окрім стартової вершини, якщо вона належить до його частини графа. Вибір вершини з мінімальною відстанню можна виконувати паралельно для різних сегментів графа. Кожен потік обчислює мінімальну відстань для своєї частини графа. Далі кожен потік оновлює відстані до вершин, досяжних з обраної вершини з мінімальною відстанню.

Розпаралелення алгоритму може значно зменшити час виконання, особливо для графів з великою кількістю вершин і ребер. Великі графи, які важко обробити послідовно, стають більш керованими завдяки паралельній обробці.

3.4.2 Впровадження архітектури CUDA для оптимізації алгоритму Дейкстри

Для оптимізації паралельної обробки даних з використанням програмно-апаратної архітектури CUDA, під час реалізації алгоритму Дейкстри, необхідно розпочати з передачі даних на графічний процесор. Тому важливо здійснити попередню обробку даних на CPU, що включає в себе нормалізацію даних, тобто

забезпечення того, що всі ваги ребер мають однаковий масштаб, з метою уникнення надмірних числових розривів, які можуть вплинути на обчислення. Потрібно також конвертувати структуру даних графа (наприклад, списків суміжності або матриці суміжності) у формат, оптимальний для обробки на GPU. Це може включати стиснення даних або реорганізацію для забезпечення ефективного доступу в пам'яті. За потреби, варто також сформувати додаткові даних, як-от індекси стартових вершин, які потрібні для швидкого пошуку і визначення початкових точок обчислень на GPU. Ця попередня обробка важлива для забезпечення того, що дані оптимізовані для паралельної обробки на GPU, що дозволяє мінімізувати час передачі даних і забезпечити ефективність обчислень.

Ядро відповідатиме за виконання ключових обчислень для знаходження найкоротших шляхів у графі. Перш за все, воно міститиме обчислення відстаней від стартової вершини до інших вершин графа. Для кожної вершини, ядро визначає, чи може поточний шлях через обрану вершину зменшити загальну відстань до інших вершин. Якщо через поточну вершину знайдено коротший шлях до сусідньої вершини, ядро оновлює відстань до цієї вершини в масиві дистанцій. Ядро виконується паралельно для різних вершин, ефективно розподіляючи завдання обчислення найкоротших шляхів між багатьма потоками на GPU, що значно збільшує швидкість виконання алгоритму порівняно з послідовною обробкою.

На основі специфіки ядра потрібно обрати які типи пам’яті програмно-апаратної архітектури CUDA будуть задіяні для оптимізації паралельної обробки даних під час роботи алгоритму. Матриця суміжності графа або список суміжності, який представляє граф, варто зберігати в глобальній пам'яті, оскільки вона доступна всім потокам на GPU. Спільна пам'ять може бути використана для оптимізації доступу до часто використовуваних даних, таких як проміжні результати відстаней до вершин, що часто оновлюються. Локальні регістри, як найшвидший тип пам’яті, можуть бути використані для локальних мінімумів відстаней, які обчислюються кожним потоком. Статичні елементи графа, як от ваги ребер, ефективно було б зберігати в текстурній пам’яті, що надає кешування і таким чином покращує продуктивність читання. Вибір між цими типами пам'яті має

базуватися на специфіці графа і даних, що обробляються. Для графів з високою щільністю і великою кількістю вершин, оптимізація доступу до даних і зменшення кількості звернень до глобальної пам'яті може значно вплинути на швидкість виконання алгоритму.

Величина блоку в CUDA має прямий вплив на кількість вершин, які можна обробляти одночасно. Якщо блок занадто великий, це може призвести до неефективного використання пам'яті, оскільки блоки можуть перевантажуватися даними, що призводить до зниження продуктивності через необхідність частого доступу до глобальної пам'яті. Натомість, дрібні блоки можуть не повністю використовувати обчислювальний потенціал графічного процесора, що також призводить до збільшення часу обчислень через недостатнє використання ресурсів. Грід визначає загальну кількість блоків, які будуть задіяні у процесі обчислень. Для алгоритму Дейкстри, де кожен крок алгоритму потенційно може залучати обробку нової групи вершин, розмір гріда має бути достатньо великим, щоб забезпечити обробку всіх необхідних вершин без потреби в повторних ітераціях.

Після завершення обчислень алгоритму Дейкстри на GPU, результати необхідно підготувати до передачі назад на CPU для подальшої обробки. Результати, які зазвичай включають остаточні відстані до всіх вершин у графі, збираються у глобальній пам'яті GPU і передаються назад на CPU.

1. Аналіз математичної моделі алгоритму Дейкстри і виділення обрахунків мінімальної відстані в певній частині графа, як функції, що виконуватиметься паралельно.

2. Опис ядра, в якому виконуватимуться обчислення відстані від однієї тестової точки до всіх навчальних точок, зберігання відстаней та індексів та вибір k найближчих сусідів для цієї тестової точки.

3. Вибір використовуваної пам’яті: глобальна, для матриці суміжності, текстурна для вагів ребер, локальні регістри і спільна пам’ять для проміжних результатів обчислень.

4. Опис конфігурація блоків і потоків CUDA для ефективного використання ресурсів, з урахуванням кількості вершин графу.

При фактичній імплементації алгоритму Дейкстри в конкретну систему з детермінованим набором даних, як і для двох попередніх випадках, потрібно ще буде обрати архітектуру графічного процесору і конкретні значення для конфігурації блоків і потоків, основуючись на обсягу вхідних даних.

3.5 Висновки

У розділі синтезовано метод оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA. Завдяки виявленню загальних принципів розпаралелювання алгоритмів і інтеграції програмно-апаратної архітектури CUDA а також аналізу особливостей кластерної архітектури CUDA, було синтезовано метод який підходить для оптимізації алгоритмів з широкого спектру задач.

Метод оптимізації паралельної обробки даних з використанням кластерної архітектури CUDA включає наступні кроки:

1. Аналіз потреб обчислень і оцінка доцільності використання CUDA.
2. Вибір відповідної архітектури GPU
3. Створення оптимізованого ядра CUDA.
4. Вибір використовуваних типів пам'яті CUDA.
5. Оптимізація передачі даних між CPU і GPU.
6. Конфігурація кількості блоків і потоків.
7. Тестування та оцінка продуктивності системи.

Синтезований метод було застосовано на основі алгоритмів FFT, kNN, і Дейкстри. Для кожного алгоритму було проведено базове дослідження його моделі в контексті паралелізації та описано процес оптимізації з CUDA. Таким чино було продемонстровано, що хоч ці алгоритми і мають різну математичну структуру, сферу застосування та демонструють важливість індивідуалізованого підходу при їх паралелізації, можуть бути оптимізовані з залученням CUDA, відповідно до синтезованого методу.

4 СИСТЕМА ОПТИМІЗАЦІЇ ПАРАЛЕЛЬНОЇ ОБРОКИ ДАНИХ З ВИКОРИСТАННЯМ АРХІТЕКТУРИ CUDA

4.1 Опис вхідних даних експерименту

З метою оцінки ефективності оптимізації паралельної обробки даних з використанням архітектури CUDA, було реалізовано метод на прикладі оптимізації алгоритму к-найближчих сусідів (kNN). Для оцінки продуктивності алгоритму kNN, який застосовується на різних обчислювальних платформах, було обрано два основні типи апаратного забезпечення: звичайний центральний процесор (CPU) та графічний процесор (GPU). Це дозволяє провести детальне порівняння ефективності виконання задач на різних системах. В рамках експерименту було використано випадкові набори даних (датасети). Всього було використано 3 датасети, що відрізняються кількістю даних та розмірністю векторів, забезпечуючи, таким чином, можливість адаптації до різноманітних вимог дослідження.

Для кожної реалізації алгоритму kNN на CPU та GPU обчислюється час виконання необхідний для пошуку найближчих сусідів для кожного набору даних. Це здійснюється шляхом вимірювання часу, потрібного для обчислення евклідової відстані від кожного вектору до всіх інших векторів у наборі. Евклідова відстань є стандартним методом вимірювання відстані між двома точками у багатовимірному просторі, що робить її ідеальною для визначення близькості між елементами даних.

Спочатку реалізується послідовна програма, яка аналізує датасет і знаходить k найближчих сусідів для кожної вибірки даних. Цей підхід дозволяє встановити базовий показник часу виконання для подальшого порівняння з паралельними версіями програми. Після цього реалізується паралельна версія алгоритму на центральному процесорі, яка використовує декілька потоків для одночасного виконання обчислень. Цей підхід значно зменшує загальний час виконання завдання за рахунок ефективного розподілу роботи між потоками. Наступним підходом є паралельна програма на GPU за допомогою технології CUDA, яка дозволяє виконувати обчислення з використанням тисяч потоків, що працюють

одночасно. Час виконання для цієї версії також реєструється, і його варіації вивчаються в залежності від зміни параметрів n (тобто кількості векторів у наборі), d (розмірності вектору) та k (кількості найближчих сусідів), які задаються перед початком обчислень. Завдяки цьому можна оцінити масштабованість алгоритму і ефективність використання CUDA для його оптимізації.

Для дослідження ефективності оптимізації алгоритму kNN, використовувався комп'ютер з наступними характеристиками:

- CPU: чотириядерний IntelCore i5-7300HQ (3.5 ГГц)
- GPU: GeForce GTX 1050

В якості датасетів було використано 9 наборів (розмірами 500, 1500 і 3000 векторів та розмірностями 16, 32 та 64) в комбінації з 3-ма різними параметрами кількості найближчих сусідів (1, 7, 11). Таким чином, всього було досліджено швидкість роботи kNN на 27 різних комбінаціях вхідних даних.

4.2 Дослідження ефективності послідовної реалізації алгоритму kNN

Спочатку програму (реалізацію алгоритму kNN) було виконано на CPU в однопотоковому режимі, що дозволяє оцінити ефективність алгоритмів, оптимізованих для простих обчислювальних систем без паралельної обробки. Послідовний код програми написано мовою C, яка відома своєю ефективністю та контролем над системними ресурсами, зокрема, управлінням пам'яттю. Для зберігання вхідних даних алгоритму kNN виділено необхідний обсяг пам'яті, що забезпечує швидкий доступ до даних під час виконання обчислень.

Структуру програми (псевдокод) представлено нижче, для базового розуміння логіки виконання та основних компонентів, повний лістинг представлено в Додатку Б:

```
#include <header_files.h> // Підключення заголовочних файлів
#define INPUT_FILE "<input_file>"
#define ROWS <rows>
#define COLS <cols>
#define K < Neighbours >
int main() {
   // Плейсхолдер для читання та ініціалізації даних з файлу
   int mat[ROWS][COLS];
```

```
    // Плейсхолдер для ініціалізації даних та перевірки на помилки
    clock_t start = clock();
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < ROWS; j++) {
            // Плейсхолдер для обчислення відстаней
        }
    }
    // Сортування обчислених відстаней для кожного рядка
    // Визначення K найближчих сусідів
    for (int i = 0; i < ROWS; i++) {
        // Плейсхолдер для сортування відстаней для кожного рядка
        // Плейсхолдер для вибору K найближчих сусідів
    }
    // Вимірювання часу закінчення виконання програми
    clock_t end = clock();
}
```

Час виконання програми включає в себе критичні обчислення евклідових відстаней між точками вибірок, а також їх подальше сортування для визначення найближчих сусідів за критерієм найменшої відстані. Ці обчислення виконуються для різних наборів вхідних даних та для різної кількості найближчих сусідів k, що забезпечує комплексне тестування програми під різні задачі.

Для отримання точних результатів та оцінювання стабільності виконання програми на одноядерному процесорі, кожен набір даних обробляється в 20 ітераціях. Це дозволяє зменшити випадкові відхилення в часі виконання, що можуть виникати через зовнішні процеси або коливання в навантаженні системи. У таблиці 4.1 наведено середній час виконання програми, що дає змогу аналізувати ефективність реалізації алгоритму kNN в однопотоковому режимі при різних комбінаціях вхідних даних, а на Рисунку 4.1 дані візуалізовано.

Таблиця 4.1 - Час виконання послідовної реалізації алгоритму kNN

	d=16			d=32			d=64		
	k=1	k=7	k=11	k=1	k=7	k=11	k=1	k=7	k=11
n=500	0,255	0,263	0,275	0,547	0,550	0,562	1,087	1,096	1,110
n=1500	1,089	1,111	1,122	2,238	2,243	2,237	4,187	4,240	4,253
n=3000	4,195	4,201	4,214	8,783	8,790	8,786	16,410	16,478	16,501

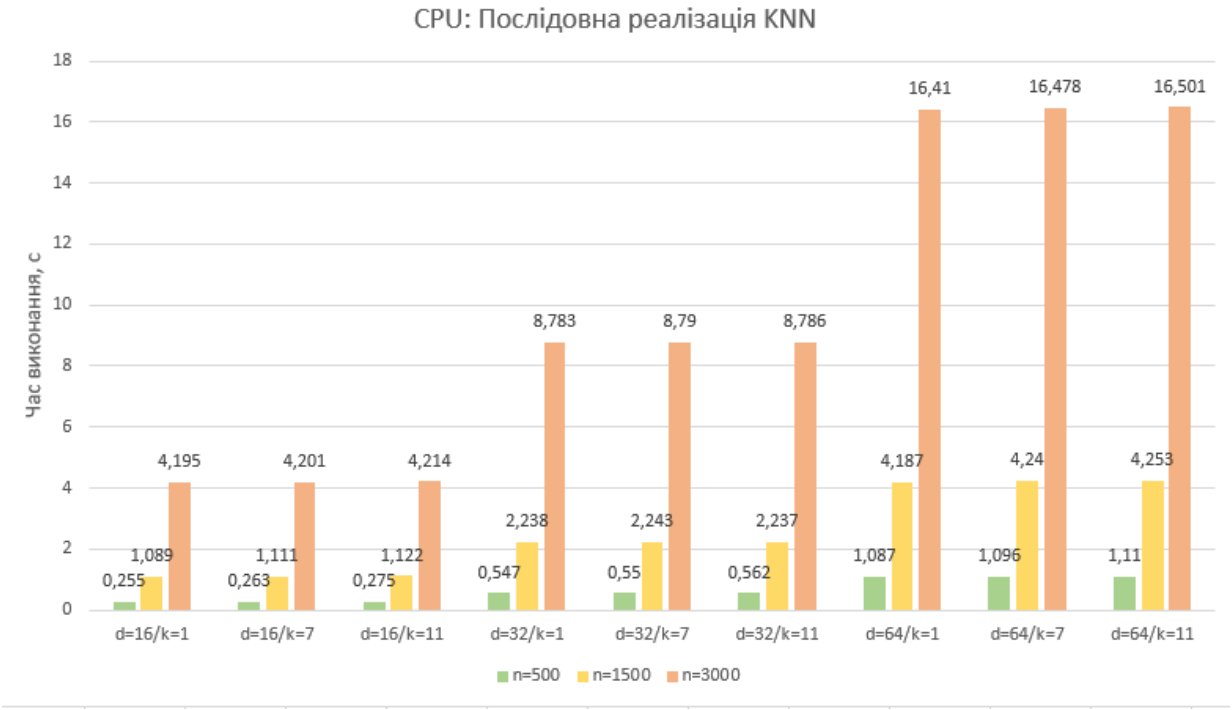


Рисунок 4.1 – Візуалізація часу виконання послідовної реалізації алгоритму kNN

Ці дані критично важливі для розуміння обмежень та потенціалу послідовної обробки даних у сучасних обчислювальних середовищах, де існує потреба в балансуванні між ресурсами апаратного забезпечення та вимогами до швидкості та точності обробки даних. Аналіз результатів часу виконання алгоритму kNN, представлених у таблиці, дозволяє зробити кілька важливих висновків. По перше, часу виконання алгоритму характерне лінійне зростання: час виконання значно збільшується зі збільшенням кількості векторів (n). Наприклад, зростання від n=500 до n=3000 при d=16 та k=1 збільшує час виконання з 0.255 секунд до 4.195 секунд. Це свідчить про лінійну або близьку до лінійної залежність часу виконання від кількості елементів у датасеті. По-друге, Розмірність даних (d) також має вплив на час виконання:: зі збільшенням розмірності час виконання також зростає. Це пояснюється збільшенням обчислювального навантаження при розрахунку відстаней між більш великими векторами. Наприклад, для n=1500 і k=11 час виконання збільшується від 1.122 секунд при d=16 до 4.240 секунд при d=64. Однак, час виконання не демонструє значних відмінностей при зміні кількості

найближчих сусідів (k). Це може свідчити про те, що основне обчислювальне навантаження припадає на розрахунок відстаней, а не на процес сортування чи вибору k елементів.

4.3 Дослідження ефективності паралельної реалізації алгоритму kNN на CPU

Для випадку паралельної реалізації програми на CPU також було обрано мову C, використовуючи широко відому бібліотеку Pthreads, яка є стандартом для паралельного програмування в середовищі POSIX. Ця бібліотека дозволяє ефективно використовувати можливості багатоядерних процесорів шляхом розподілу задач між окремими потоками. В Pthreads ідентифікатори функцій та змінних зазвичай починаються з префікса pthread_, що допомагає забезпечити їхню однозначність та легке розпізнавання.

Вихідним пунктом виконання будь-якої програми є один основний потік, з якого можуть бути запущені додаткові потоки. Ці потоки можуть бути створені за допомогою відповідних API-викликів, які дозволяють контролювати їх життєвий цикл, у тому числі створення, завершення, та синхронізацію. В контексті виконання на CPU, програма використовує потужності кількох ядер процесора, зокрема, процесор, використаний для даного дослідження, має 4 фізичних та 2 віртуальних ядра, що в сумі дозволяє використовувати 8 потоків. Це дозволяє максимізувати продуктивність обчислень, ефективно розподіляючи обробку даних між всіма доступними ядрами. При такому підході вся програма налаштована на роботу так, що кожен потік обробляє порівну частину вхідних даних, забезпечуючи рівномірний розподіл навантаження.

Псевдокод програми, що дозволяє не тільки візуально оцінити структуру програми, але й спрощує розуміння процесів взаємодії та синхронізації між потоками, що є ключовими для досягнення оптимальної продуктивності в багатопотоковому середовищі, зображено нижче, а повний лістинг наведено в Додатку В:

```
#include <header_files.h> // Підключення заголовочних файлів
#define INPUT_FILE "<input_file>"
#define ROWS <rows>
#define COLS <cols>
#define K < Neighbours >
#define THREADS 8
void *KNN(void *arg) {
    int id = *(int*)arg;
    for (int i = id; i < ROWS; i += THREADS) {
        for (int j = 0; j < ROWS; j++) {
            // Плейсхолдер для обчислення відстані між рядками i та j
        }
    }
    // Плейсхолдер для сортування
    pthread_exit(NULL);
}
int main() {
    // Плейсхолдер для читання та ініціалізації даних з файлу
    int mat[ROWS][COLS];
    // Плейсхолдер для ініціалізації даних та перевірки на помилки
    pthread_t threads[THREADS]; int threadID[THREADS];
    clock_t start = clock();
    for (int i = 0; i < THREADS; i++) {
        threadID[i] = i;
        if (pthread_create(&threads[i], NULL, KNN, (void *)&threadID[i]) != 0) {
            fprintf(stderr, "Помилка створення потоку\n");
            return 1;
        }
    }
    for (int i = 0; i < THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    // Вимірювання часу закінчення виконання програми
    clock_t end = clock();
}
```

У таблиці 4.2 наведено середній час виконання програми, а на Рисунку 4.2 час виконання візуалізовано.

Таблиця 4.2 - Час виконання паралельної реалізації алгоритму kNN на CPU

	d=16			d=32			d=64		
	k=1	k=7	k=11	k=1	k=7	k=11	k=1	k=7	k=11
n=500	0,045	0,049	0,076	0,127	0,133	0,141	0,232	0,254	0,265
n=1500	0,213	0,235	0,278	0,431	0,446	0,466	0,771	0,782	0,798
n=3000	0,789	0,792	0,999	1,492	1,518	1,540	2,800	2,875	2,891

Загальні тенденції співпадають з варіантом для однопотокової реалізації: можемо спостерігати зростання часу виконання з ростом розміру датасету і розмірності вектора, і незалежність від параметру кількості найближчих сусідів.

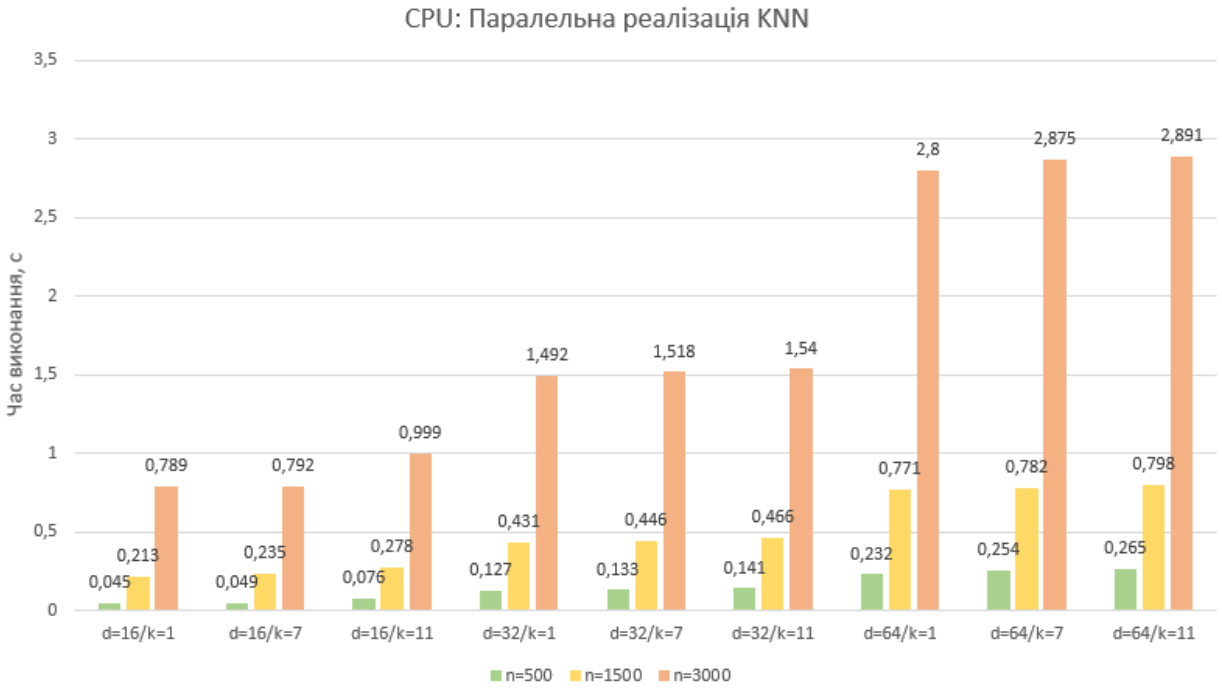


Рисунок 4.2 – Час виконання паралельної реалізації алгоритму kNN на CPU

4.4 Дослідження ефективності паралельної реалізації алгоритму kNN на GPU

Питання паралелізації алгоритму kNN досліджувалось у розділі 3. Основні аспекти алгоритму, що включають обчислення відстаней між всіма парами точок та подальше сортування цих відстаней для визначення k найближчих сусідів для кожної точки. ідеально підходять для розподілу по багатьох обчислювальних ядрах або потоках на GPU

Для проведення імплементації алгоритму kNN з використанням GPU, була обрана відеокарта NVIDIA GeForce GTX 1050. Цей вибір виявився особливо вдалим, оскільки GTX 1050 забезпечує відмінне співвідношення ціни до продуктивності, а також підтримку сучасних технологій, що є важливим для розробки та тестування обчислювальних алгоритмів.

По-перше, GTX 1050 використовує архітектуру Pascal, яка забезпечує значні поліпшення у продуктивності і енергоефективності порівняно зі старішими архітектурами, такими як Kepler або Maxwell, що використовуються в серіях GTX

700 і GTX 900 відповідно. Архітектура Passcal дозволяє виконувати більшу кількість операцій з плаваючою комою за цикл, що особливо корисно для задач обробки великих даних, де потрібні інтенсивні обчислення.

По-друге, GTX 1050 споживає значно менше енергії порівняно з новішими картами, такими як серії RTX, що робить її більш доступною для експериментів у неспеціалізованих лабораторіях без необхідності складної системи охолодження або дорогого енергозабезпечення. В порівнянні з новішими моделями, GTX 1050 надає достатню продуктивність для більшості додатків машинного навчання та обробки даних без необхідності інвестувати в значно дорожчі рішення. Це робить GTX 1050 ідеальним варіантом для розробників, які шукають баланс між вартістю і потужністю обладнання.

В алгоритмі kNN, що реалізований для обчислення на GPU, найбільше часу витрачається на обчислення відстані та визначення k найближчих сусідів. Ці два процеси можуть бути ефективно реалізовані за допомогою двох методів: обчислення відстані та сортування. Структура (псевдокод) програми, повний лістинг наведено в Додатку Г:

```
#include <header_files.h> // Підключення заголовочних файлів
#define INPUT_FILE "<input_file>"
#define ROWS <rows>
#define COLS <cols>
#define K < Neighbours >
int main() {
    // Плейсхолдер для читання та ініціалізації даних з файлу
    int mat[ROWS][COLS];
    // Плейсхолдер для ініціалізації даних та перевірки на помилки
(myfile);
    double *d_a, *d_c;
    int *d_k;
    // Виділення пам'яті на пристрої
    cudaMalloc((void **) &d_a, ROWS * COLS * sizeof(int));
    cudaMalloc((void **) &d_c, ROWS * ROWS * sizeof(double));
    cudaMalloc((void **) &d_k, sizeof(int));
    // Копіювання даних з хоста на пристрій
    cudaMemcpy(d_a, mat, ROWS * COLS * sizeof(int), cudaMemcpyHostToDevice);
    clock_t start = clock();
    // Виклик ядра CUDA для обчислення відстаней
    dim3 blocks(ROWS), threads(COLS);
    distance<<<blocks, threads>>>(d_a, d_k, d_c);
    // Виклик ядра CUDA для сортування
    sorting<<<blocks, threads>>>(d_c, d_k);
    // Копіювання результатів з пристрою на хост
    cudaMemcpy(mat, d_c, ROWS * ROWS * sizeof(double), cudaMemcpyDeviceToHost);
    // Вимірювання часу закінчення виконання програми
    clock_t end = clock();
    // Звільнення виділеної пам'яті
```

```
    cudaFree(d_a);
    cudaFree(d_c);
    cudaFree(d_k);
    return 0;
}
```

Метод обчислення відстані виконується таким чином, що кожен потік незалежно вираховує відстань між всіма точками, що дозволяє повністю паралелізувати цей процес. Дані передаються з центрального процесора (CPU) на графічний процесор (GPU), і кожен потік бере участь у вирахуванні відстані. При великій кількості точок активується значна кількість потоків та блоків. Метод distance<<<blocks,threads>>>(d_a,d_k,d_c) відповідає за обчислення відстаней і збереження їх у спільній пам'яті, де кожен потік обробляє одну відстань.

Після завершення обчислення відстаней, задача сортування цих відстаней та визначення k найближчих сусідів стає наступним кроком. Відстані, збережені у спільній пам'яті, сортуються за допомогою методу sorting<<<blocks,threads>>>(d_c,d_k), а потім копіюються у глобальну пам'ять для вирахування k найближчих сусідів, після чого результати передаються назад на процесор.

У таблиці 4.3 представлено середній час виконання алгоритму на GPU. Результати візуалізовано на Рисунку 4.3. Кількість запущених блоків залежить від обсягу вхідних даних та кількості потоків, що активуються в кожному блоку.. Ці параметри передаються як аргументи до функцій ядра, що дозволяє точно налаштовувати виконання програми відповідно до поточних потреб обчислення.

Таблиця 4.3 - Час виконання паралельної реалізації алгоритму kNN на GPU

	d=16			d=32			d=64		
	k=1	k=7	k=11	k=1	k=7	k=11	k=1	k=7	k=11
n=500	0,019	0,033	0,044	0,073	0,065	0,097	0,164	0,176	0,198
n=1500	0,121	0,181	0,212	0,197	0,204	0,236	0,339	0,387	0,412
n=3000	0,330	0,358	0,401	0,586	0,595	0,601	1,013	1,027	1,039

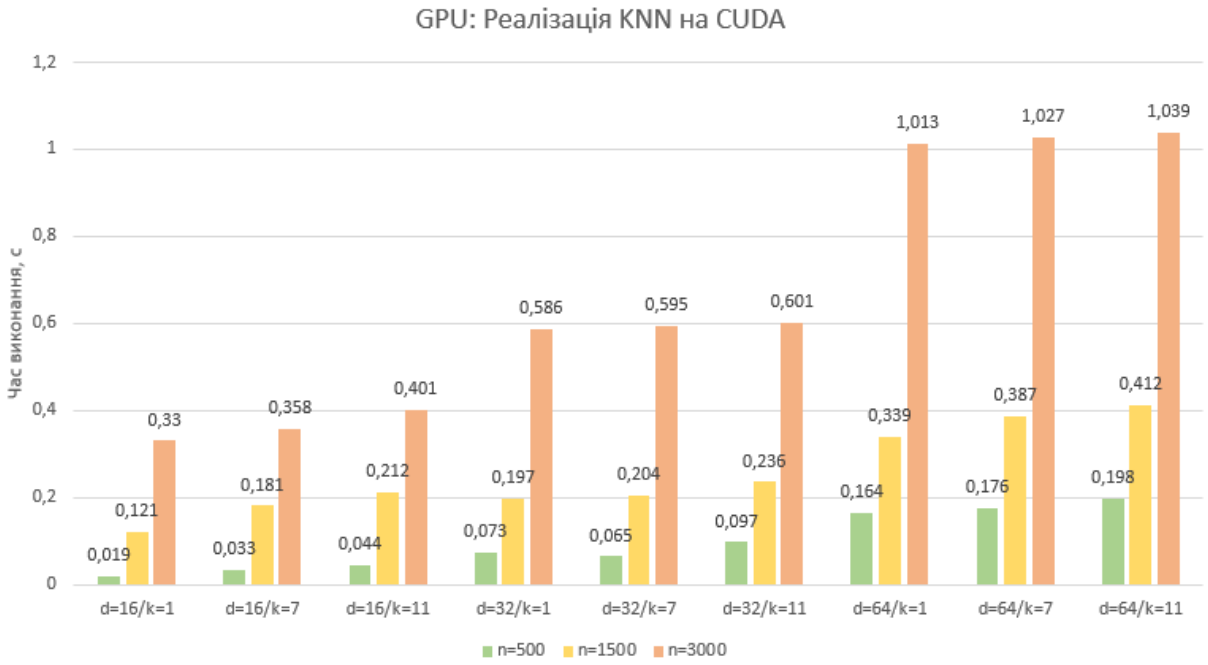


Рисунок 4.3 – Час виконання паралельної реалізації алгоритму kNN на GPU

Загальна ситуація аналогічна двом попереднім реалізаціям (послідовній імплементації алгоритму kNN і паралельній з використанням CPU): час виконання не залежить від параметру кількості найближчих сусідів, однак зростає зі збільшенням датасету і розмірності вектора.

4.4 Дослідження ефективності оптимізації алгоритму kNN з використанням CUDA

Порівняння таблиць результатів часу виконання алгоритму kNN в різних імплементаціях — однопотоковому, багатопотоковому на CPU та на GPU з використанням CUDA, демонструє значні переваги використання графічних процесорів і програмно-апаратної архітектури CUDA зокрема. Перехід від однопотокового обчислення до багатопотокового на CPU значно знижує час виконання, що підтверджує ефективність паралелізму. Проте, ще більш помітні переваги досягаються при використанні обчислень на графічних процесорах. Використання GPU з CUDA для виконання алгоритму kNN згідно з таблицею 4.3 показує не тільки низькі часи виконання, але й демонструє більш ніж десятикратне

зменшення часу виконання порівняно з однопотоковими і мультипоточними CPU варіантами. Це особливо важливо для обробки великих обсягів даних і складних обчислень, які вимагають високої продуктивності та ефективності.

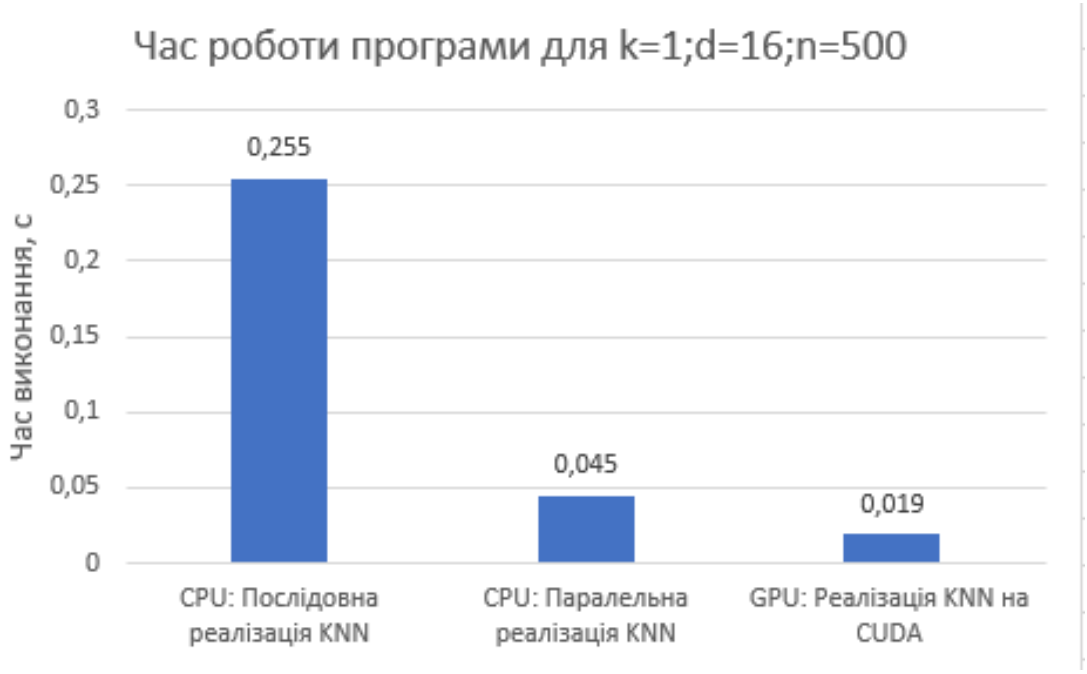


Рисунок 4.4 – Порівняння часу виконання kNN для найменшого датасету

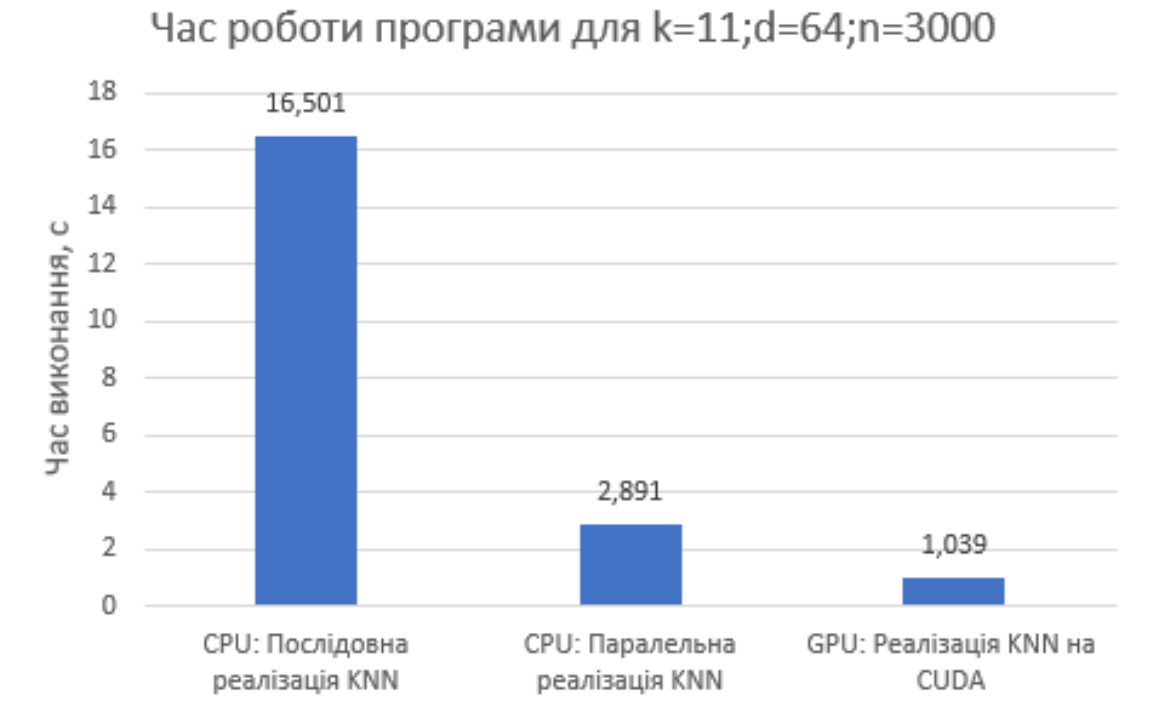


Рисунок 4.5 – Порівняння часу виконання kNN для найбільшого датасету

На прикладі з даними, збільшення обсягу даних з n=500 при d=16 (Рисунок 4.4) до n=3000 при d=64 (Рисунок 4.5) показує значне абсолютне прискорення від використання GPU, оскільки різниця у часі виконання між CPU і GPU складає близько 0.25 секунди для n=500 та d=16, але зростає до приблизно 16 секунд для n=3000 та d=64, тобто з ростом обсягу даних абсолютне прискорення, яке надає використання GPU, стає ще більш значущим.

4.5 Висновки

У розділі подано дослідження застосування синтезованого методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA, шляхом порівняння реалізації алгоритму kNN в трьох різних варіаціях: послідовний, паралельній на CPU та паралельній на GPU. Програмний код для всіх трьох імплементацій було написано на мові програмування C.

Перехід від однопотокового обчислення до багатопотокового на CPU значно знижує час виконання, проте, ще більш помітні переваги досягаються при використанні обчислень на GPU. Аналіз результатів показав, що часу виконання алгоритму характерне зростання залежно від кількості та розмірності векторів. Однак, час виконання не демонструє значних відмінностей при зміні кількості найближчих сусідів. Це свідчить про те, що основне обчислювальне навантаження припадає на розрахунок відстаней, а не на процес сортування. При малих обсяг даних, оптимізація, яку пропонує CUDA не є настільки ж значущою Однак, переваги включають не тільки зменшення часу обчислень, але й можливість більш ефективно обробляти великі обсяги даних, забезпечуючи швидку обробку інформації.

Таким чином можемо зробити висновок, що застосування синтезованого методу оптимізації, відповідно до описаних кроків, дозволило пришвидшити час виконання алгоритму в 16 разів.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень синтезовано метод оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

У першому розділі було проведено дослідження використання існуючих методів послідовної і паралельної обробки даних в різних доменних областях. Зокрема, досліджувались основи використання обчислювальних можливостей GPU з залученням кластерної архітектури CUDA. Дослідження показало, що розробка методу для оптимізації паралельної обробки даних з використанням CUDA є доцільною, оскільки обчислення на GPU надає прискорення обробці даних, а наявні методи, що використовують CUDA ґрунтуються на конкретних алгоритмах.

У другому розділі представлено моделі, що дозволяють репрезентувати використання ресурсів і час виконання ядра. Зокрема було визначено модель, яка описує правила оцінки ядра при відстеженні вартості виконання одного варпу. Зокрема наведено обґрунтованість моделі, тобто те, що вартість, передбачена логікою для даного ядра, є щонайменше вартістю фактичного виконання ядра, як це відслідковується семантикою вартості. Крім того, доведено, що навіть якщо логіка була розроблена лише для одного варпу шляхом використання "паралельної" семантики вартості, вона може бути використана для моделювання паралельне виконання блоку. Отримані таким чином моделі можна використовувати, для оцінки доцільності залучення обчислень на графічному процесорі.

У третьому розділі було синтезовано метод оптимізації паралельної обробки даних з використанням кластерної архітектури CUDA та описано його застосування на основі алгоритмів FFT, kNN, і Дейкстри. Метод включає наступні кроки:

1. Аналіз потреб обчислень і оцінка доцільності використання CUDA.
2. Вибір відповідної архітектури GPU
3. Створення оптимізованого ядра CUDA.
4. Вибір використовуваних типів пам'яті CUDA.

5. Оптимізація передачі даних між CPU і GPU.
6. Конфігурація кількості блоків і потоків.
7. Тестування та оцінка продуктивності системи.

У четвертому розділі подано дослідження застосування синтезованого методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA, відповідно до описаних кроків. На основі імплементації алгоритму kNN трьома способами обробки даних: послідовним, паралельним на CPU та паралельним на GPU, було продемонстровано пришвидшення часу виконання алгоритму в 16 разів.

Набув подальшого розвитку метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA, який на відміну від відомих методів пропонує генералізований підхід без прив’язки до конкретного алгоритму, що дозволяє оптимізувати паралельної обробки даних. Набули подальшого розвитку програмно-технічні засоби оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

За темою кваліфікаційної роботи магістра опубліковані тези у матеріалах конференції XXIV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів «Стан, досягнення та перспективи інформаційних систем і технологій» 18-19 квітня 2024 р., Одеса, Україна (Додаток 1).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Khan Z.A. Identifying Hot Topic Trends in Streaming Text Data Using News Sequential Evolution Model Based on Distributed Representations. *IEEE Access*, 2023. Vol. 11. URL: https://doi.org/10.1109/ACCESS.2023.3312764.

2. Fazla A., Aydin M.E., Kozat S.S. Joint Optimization of Linear and Nonlinear Models for Sequential Regression. *Digital Signal Processing*. 2023. Vol. 132. URL: https://doi.org/10.1016/j.dsp.2022.103802.

3. Lin J.C.-W., Shao Y., Djenouri Y., Yun, U. ASRNN: A Recurrent Neural Network with an Attention Model for Sequence Labeling. *Knowledge-Based Systems*, 2021. Vol. 212. URL: https://doi.org/10.1016/j.knosys.2020.106548.

4. Challa S.K., Kumar A., Semwal V.B. Multibranch CNN-BiLSTM Model for Human Activity Recognition Using Wearable Sensor Data. *The Visual Computer*. 2022. Vol. 38. URL: https://doi.org/10.1007/s00371-021-02283-3.

5. Robeson, M.S. Reproducible Sequence Taxonomy Reference Database Management. *PLoS Computational Biology*. 2021. Vol. 17. №11. URL: https://doi.org/10.1371/journal.pcbi.1009581.

6. Arita M., Karsch-Mizrachi I., Cochrane G. The International Nucleotide Sequence Database Collaboration. *Nucleic Acids Research*. 2021. Vol. 49. URL: https://doi.org/10.1093/nar/gkaa967.

7. Chen T. The Genome Sequence Archive Family: Toward Explosive Data Growth and Diverse Data Types. *Genomics, Proteomics & Bioinformatics*. 2021. Vol. 19, №4, URL: https://doi.org/10.1016/j.gpb.2021.08.001.

8. Chen H., Gao J., Gao Z., Chen D., Yang T. A Sequential Iterative Deep Learning Seismic Blind High-Resolution Inversion. *Journal of Selected Topics in Applied Earth Observations and Remote Sensing*. 2021, Vol. 14. URL: https://doi.org/10.1109/JSTARS.2021.3100502.

9. Demir S. Turkish Data-to-Text Generation Using Sequence-to-Sequence Neural Networks. *ACM Transactions on Asian and Low-Resource Language Information Processing*. 2023. Vol. 22. №2. URL: https://doi.org/10.1145/3543826.

10. Phkhovelishvili M., Archvadze N., Gasitashvili Z. Peculiarities of the Hybrid Model Built Using Parallel Data. *7th International Conference on Mathematics and Computers in Sciences and Industry*. 2022. URL: https://doi.org/10.1109/MCSI55933.2022.00018.

11. Dias L.A., Damasceno A.M.P., Gaura E., Fernandes M.A.C. "A Full-parallel Implementation of Self-Organizing Maps on Hardware." *Neural Networks*. 2021. Vol. 143. URL: https://doi.org/10.1016/j.neunet.2021.05.021.

12. Feldmann J., Youngblood N., Karpov M. Parallel Convolutional Processing Using an Integrated Photonic Tensor Core. *Nature*, 2021. Vol. 589. URL: https://doi.org/10.1038/s41586-020-03070-1.

13. Yazdeen A.A., Zeebaree S.R.M., Sadeeq M., Kak S.F., Ahmed O.M., Zebari R.R. FPGA Implementations for Data Encryption and Decryption via Concurrent and Parallel Computation. *Qubahan Academic Journal*. 2021. Vol. 1. №2. URL: https://doi.org/10.48161/qaj.v1n2a38.

14. Steingrímsson S., Lohar P., Loftsson, H., Way A. Do Not Discard – Extracting Useful Fragments from Low-Quality Parallel Data to Improve Machine Translation. *Proceedings of the Second Workshop on Corpus Generation and Corpus Augmentation for Machine Translation*. 2023.

15. Guo Y., Wang Z., Hong Q., Luo H., Qiu X., Liang L. A 60-Mode High-Throughput Parallel-Processing FFT Processor for 5G/4G Applications. *Transactions on Very Large Scale Integration Systems*. 2023. Vol. 31. №2. URL: https://doi.org/10.1109/TVLSI.2022.3227346.

16. Mencagli G., Torquati M., Cardaci A., Fais A., Rinaldi L., Danelutto, M. WindFlow: High-Speed Continuous Stream Processing With Parallel Building Blocks. *Transactions on Parallel and Distributed Systems*. 2021. Vol. 32. №11. URL: https://doi.org/10.1109/TPDS.2021.3073970.

17. Liu T., Young E.F.Y. Rethinking AIG Resynthesis in Parallel. *Design Automation Conference* . 2023. URL: https://doi.org/10.1109/DAC56929.2023.10247961.

18. Huang Y., Xiang X., Zhou H., Tang D., Sun Y. Online Identification-Verification-Prediction Method for Parallel System Control of UAVs. *Aerospace*. 2021. Vol. 8. №4. URL: https://doi.org/10.3390/aerospace8040099.

19. Cai, Y., Yang, Z., Ni, L., Liu, J., Xie, B., Li, X. Parallel AIG Refactoring via Conflict Breaking. *arXiv*. 2024. URL: https://arxiv.org/abs/2404.13617.

20. Kuderov P., Dzhivelikian E., Panov A.I. Stabilize Sequential Data Representation via Attraction Module. *Brain Informatics. Lecture Notes in Computer Science*. 2023. Vol. 13974, URL: https://doi.org/10.1007/978-3-031-43075-6_8.

21. Chen X. Moving Object Segmentation in 3D LiDAR Data: A Learning-Based Approach Exploiting Sequential Data. *IEEE Robotics and Automation Letters*. 2021. Vol. 6. №4. URL: https://doi.org/10.1109/LRA.2021.3093567.

22. Yoon J.H., Jang B. Evolution of Deep Learning-Based Sequential Recommender Systems: From Current Trends to New Perspectives. *IEEE Access*. 2023. Vol. 11. URL: https://doi.org/10.1109/ACCESS.2023.3281981.

23. Xu X., Zhang S., Guo J., Xin T. Biclustering of Log Data: Insights from a Computer-Based Complex Problem Solving Assessment. *Journal of Intelligence*. 2024. Vol. 12. №1. URL: https://doi.org/10.3390/jintelligence12010010.

24. Zhang J., Wang C., Lu J. Modeling Item Revisiting Behavior in Computer-Based Testing: Exploring the Effect of Item Revisitations as Collateral Information. *Behavior Research Methods*. 2023. URL: https://doi.org/10.3758/s13428-023-02209-y.

25. Weerakody P.B., Wong K.W., Wang G., Ela W. A Review of Irregular Time Series Data Handling with Gated Recurrent Neural Networks. *Neurocomputing*. 2021. Vol. 441. URL: https://doi.org/10.1016/j.neucom.2021.02.046.

26. Adnan M., Kalra S., Cresswell J.C. Federated Learning and Differential Privacy for Medical Image Analysis. *Scientific Reports*, 2022. Vol. 12. URL: https://doi.org/10.1038/s41598-022-05539-7.

27. He K., Gan C., Li Z., Rekik I., Yin Z., Ji W., Gao Y., Wang Q., Zhang J., Shen D. Transformers in Medical Image Analysis. *Intelligent Medicine*. 2023. Vol. 3. №1. URL: https://doi.org/10.1016/j.imed.2022.07.002.

28. Krishnan R., Rajpurkar P., Topol E.J. Self-supervised Learning in Medicine and Healthcare." *Nature Biomedical Engineering*. 2022. Vol. 6. URL: https://doi.org/10.1038/s41551-022-00914-1.

29. Shehab M., Abualigah L., Shambour Q., Abu-Hashem M.A., Shambour M.K.Y., Alsalibi A.I., Gandomi A.H. Machine Learning in Medical Applications: A Review of State-of-the-art Methods. *Computers in Biology and Medicine*. 2022. Vol. 145, URL: https://doi.org/10.1016/j.compbiomed.2022.105458.

30. Mustafa D., Alkhasawneh R., Obeidat F., Shatnawi A.S. MIMD Programs Execution Support on SIMD Machines: A Holistic Survey. *IEEE Access*. 2024. Vol. 12. URL: https://doi.org/10.1109/ACCESS.2024.3372990.

31. Thapa C., Mahawaga Arachchige P.C., Camtepe S., Sun L. SplitFed: When Federated Learning Meets Split Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*. 2022. Vol. 36. №8. URL: https://doi.org/10.1609/aaai.v36i8.20825.

32. Mammen P.M. Federated Learning: Opportunities and Challenges. *Distributed, Parallel, and Cluster Computing*. 2021. URL: https://doi.org/10.48550/arXiv.2101.05428.

33. Xiao D., Yang C., Wu W. Mixing Activations and Labels in Distributed Training for Split Learning. *Transactions on Parallel and Distributed Systems*. 2022. Vol. 33. №11. URL: https://doi.org/10.1109/TPDS.2021.3139191.

34. Dong S., Wang P., Abbas K. A Survey on Deep Learning and Its Applications. *Computer Science Review*. 2021. Vol. 40. URL: https://doi.org/10.1016/j.cosrev.2021.100379.

35. Yu F., Wang D., Shangguan L., Zhang, M., Tang X., Liu C., Chen X. A Survey of Large-Scale Deep Learning Serving System Optimization. *Challenges and Opportunities*. 2021.

36. Zhao C., Gao W., Nie F., Zhou H. A Survey of GPU Multitasking Methods Supported by Hardware Architecture. *Transactions on Parallel and Distributed Systems*. 2022. Vol. 33. №6. URL: https://doi.org/10.1109/TPDS.2021.3115630.

37. Chi H. Safety Management of Internet of Things Engineering Construction Based on GPU Parallel Computing. *Optik*. 2023. Vol. 273. URL: https://doi.org/10.1016/j.ijleo.2022.170447.

38. Shen S., Yang H., Liu Y., Liu Z., Zhao Y. CUDA-Accelerated RNS Multiplication in Word-Wise Homomorphic Encryption Schemes for Internet of Things. *Transactions on Computers*. 2023. Vol. 72. №7. URL: https://doi.org/10.1109/TC.2022.3227874.

39. Chandrashekhar B.N., Sanjay H.A., Srinivas T. Performance Analysis of Parallel Programming Paradigms on CPU-GPU Clusters. *International Conference on Artificial Intelligence and Smart Systems*. 2021. URL: https://doi.org/10.1109/ICAIS50930.2021.9395977.

40. Cecilia J.M., Morales-García J., Imbernón B., Prades J., Cano J.-C., Silla, F. Using Remote GPU Virtualization Techniques to Enhance Edge Computing Devices. *Future Generation Computer Systems*. 2023. Vol. 142. URL: https://doi.org/10.1016/j.future.2022.12.038.

41. Peñaranda Cebrián C., Reaño C., Silla, F. Pipelined Compression in Remote GPU Virtualization Systems using rCUDA: Early Experiences. *Workshop Proceedings of the 51st International Conference on Parallel*. 2023. URL: https://doi.org/10.1145/3547276.3548628.

42. Lapegna M., Balzano W., Meyer N., Romano D. Clustering Algorithms on Low-Power and High-Performance Devices for Edge Computing Environments. *Sensors*. 2021. Vol. 21. №16. URL: https://doi.org/10.3390/s21165395.

43. Han K., Lee W.K., Hwang S.O. Optimized Implementation of the Gimli Authenticated Encryption and Hash Function on GPU for IoT Applications. *Cluster Computing*. 2022. Vol. 25. URL: https://doi.org/10.1007/s10586-021-03415-z.

44. Chakkour T. Parallel Computation to Bidimensional Heat Equation Using MPI/CUDA and FFTW Package. *Frontiers in Computer Science*. 2024. Vol. 5. URL: https://doi.org/10.3389/fcomp.2023.1305800.

45. Al-Shafei A., Zareipour H., Cao Y. High-Performance and Parallel Computing Techniques Review: Applications, Challenges and Potentials to Support Net-

Zero Transition of Future Grids. *Energies*. 2022, Vol. 15. №22. URL: https://doi.org/10.3390/en15228668.

46. Fernandes D.F., Santos M.C., Silva A.C., Lima A.M.M. Comparative Study of CUDA-Based Parallel Programming in C and Python for GPU Acceleration of the 4th Order Runge-Kutta Method. *Nuclear Engineering and Design*. 2024. Vol. 421. URL: https://doi.org/10.1016/j.nucengdes.2024.113050.

47. Bozorgmehr B., Willemsen P., Gibbs J.A., Stoll R., Kim J.-J., Pardyjak E.R. Utilizing Dynamic Parallelism in CUDA to Accelerate a 3D Red-Black Successive Over Relaxation Wind-Field Solver. *Environmental Modelling & Software*. 2021. Vol. 137. URL: https://doi.org/10.1016/j.envsoft.2021.104958.

48. Kang P. Programming for High-Performance Computing on Edge Accelerators. *Mathematics*. 2023. Vol. 11. №4. URL: https://doi.org/10.3390/math11041055.

49. Araujo G., Griebler D., Rockenbach D.A., Danelutto M., Fernandes L.G. NAS Parallel Benchmarks with CUDA and Beyond. *Software Practice and Experience*. 2023, Vol. 53, №1, URL: https://doi.org/10.1002/spe.3056.

50. Imankulov T., Daribayev B., Mukhambetzhanov S. Comparative Analysis of Parallel Algorithms for Solving Oil Recovery Problem Using CUDA and OpenCL. *International Journal of Nonlinear Analysis and Applications*. 2021. Vol. 12. №1. URL: https://doi.org/10.22075/ijnaa.2021.4809.

51. Fichte J.K., Hecher M., Roland V. Benchmark Data and Source Code. *Zenodo*. 2021. URL: https://doi.org/10.5281/zenodo.5159903.

52. Afzal A., Saleel C.A., Prashantha K. Parallel Finite Volume Method-Based Fluid Flow Computations Using OpenMP and CUDA Applying Different Schemes. *Journal of Thermal Analysis and Calorimetry*. 2021. Vol. 145. URL: https://doi.org/10.1007/s10973-021-10637-1.

53. Temirbekov A., Baigereyev D., Temirbekov N., Urmashev B., Amantayeva A. Parallel CUDA Implementation of a Numerical Algorithm for Solving the Navier-Stokes Equations Using the Pressure Uniqueness Condition. *AIP Conference Proceedings*. 2021. Vol. 2325. №1. URL: https://doi.org/10.1063/5.0041039.

54. Liu Y., Yue Y., Bo M., Qi S. Mathematical Verification and Analysis of CUDA Based Parallel Matrix Multiplication. *International Conference on Information Science, Parallel and Distributed Systems*. 2021. URL: https://doi.org/10.1109/ISPDS54097.2021.00046.

55. Xiao H., Guo B., Zhang H., Li C. A Parallel Algorithm of Image Mean Filtering Based on OpenCL. *IEEE Access*. 2021. Vol. 9. URL: https://doi.org/10.1109/ACCESS.2021.3068772.

56. Asaduzzaman A., Trent A., Osborne S., Aldershof C., Sibai F.N. Impact of CUDA and OpenCL on Parallel and Distributed Computing. *8th International Conference on Electrical and Electronics Engineering*. 2021. URL: https://doi.org/10.1109/ICEEE52452.2021.9415927.

57. Eddelbuettel D. Parallel Computing with R: A Brief Review. *WIREs Computational Statistics*. 2021. Vol. 13. URL: https://doi.org/10.1002/wics.1515.

58. Yin L., Zhang Y., Zhang Z., Peng Y., Zhao P. ParaX: Boosting Deep Learning for Big Data Analytics on Many-core CPUs. *Proceedings of the VLDB Endowment*. 2021, Vol. 14. №6. URL: https://doi.org/10.14778/3447689.3447692.

59. Rockenbach D.A., Löff J., Araujo G., Griebler D., Fernandes L.G. "High-Level Stream and Data Parallelism in C++ for GPUs. *Proceedings of the XXVI Brazilian Symposium on Programming Languages*. 2023. URL: https://dl.acm.org/doi/10.1145/3561320.3561327.

60. Di Domenico D., Lima J.V.F., Cavalheiro G.G.H. Parallel Benchmarks with Python: A Performance and Programming Effort Analysis Focusing on GPUs. *Journal of Supercomputing*. 2023. Vol. 79. URL: https://doi.org/10.1007/s11227-022-04932-3.

61. Windisch D., Kaever C., Juckeland G., Bieberle A. Parallel Algorithm for Connected-Component Analysis Using CUDA. *Algorithms*. 2023. Vol. 16. №2. URL: https://doi.org/10.3390/a16020080.

62. Pekkilä J., Väisälä M.S., Käpylä M.J., Rheinhardt M., Lappi O. Scalable Communication for High-Order Stencil Computations Using CUDA-aware MPI. *Parallel Computing*. 2022. Vol. 111, URL: https://doi.org/10.1016/j.parco.2022.102904.

63. Yi X., Stokes, D., Yan Y., Liao C. CUDAMicroBench: Microbenchmarks to Assist CUDA Performance Programming. *International Parallel and Distributed Processing Symposium Workshops*. 2021. URL: https://doi.org/10.1109/IPDPSW52791.2021.00068.

64. Sharma A. Guided Parallelized Stochastic Gradient Descent for Delay Compensation. *Applied Soft Computing*. 2021. Vol. 102. URL: https://doi.org/10.1016/j.asoc.2021.107084.

65. Ma Y., Rusu F., Wu K., Sim A. Adaptive Stochastic Gradient Descent for Deep Learning on Heterogeneous CPU+GPU Architectures. *International Parallel and Distributed Processing Symposium Workshops*. 2021. URL: https://doi.org/10.1109/IPDPSW52791.2021.00012.

66. Elahi F., Fazlali M., Malazi H.T., Elahi M. Parallel Fractional Stochastic Gradient Descent With Adaptive Learning for Recommender Systems. *Transactions on Parallel and Distributed Systems*. 2024. Vol. 35. №3. URL: https://doi.org/10.1109/TPDS.2022.3185212.

67. Wu D., Guan Q., Fan Z., Deng H., Wu T. AutoML With Parallel Genetic Algorithm for Fast Hyperparameters Optimization in Efficient IoT Time Series Prediction. *Transactions on Industrial Informatics*. 2023. Vol. 19. №9. URL: https://doi.org/10.1109/TII.2022.3231419.

68. Harada T., Alba E., Luque G. Fresh Approach to Evaluate Performance in Distributed Parallel Genetic Algorithms. *Applied Soft Computing*. 2022. Vol. 119. URL: https://doi.org/10.1016/j.asoc.2022.108540.

69. Wang K., Li X., Gao L., Li P., Gupta S.M. Genetic Simulated Annealing Algorithm for Parallel Partial Disassembly Line Balancing Problem. *Applied Soft Computing*, 2021. Vol. 107. URL: https://doi.org/10.1016/j.asoc.2021.107404.

70. Ivanov B.D., Kropaczek D.J. Assessment of Parallel Simulated Annealing Performance with the Nexus/ANC9 Core Design Code System. *EPJ Web of Conferences*, 2021. Vol. 247. URL: https://doi.org/10.1051/epjconf/202124702019.

71. Li Q. Parallel and Distributed Optimization Method With Constraint Decomposition for Energy Management of Microgrids. *Transactions on Smart Grid*. 2021. Vol. 12. №6. URL: https://doi.org/10.1109/TSG.2021.3097047.

72. Maher S.J., Ralphs T.K., Shinano Y. Assessing the Effectiveness of (Parallel) Branch-and-bound Algorithms. *Laboratory for Computational Optimization Research*. 2021. URL: https://doi.org/10.48550/arXiv.2104.10025.

73. Gusmeroli N., Hrga T., Lužar B., Povh J., Siebenhofer M., Wiegele A. BiqBin: A Parallel Branch-and-bound Solver for Binary Quadratic Problems with Linear Constraints. *ACM Transactions on Mathematical Software*. 2022. Vol. 48. №2. URL: https://doi.org/10.1145/3514039.

74. Singh P. Scheduling Tasks Based on Branch and Bound Algorithm in Cloud Computing Environment. *8th International Conference on Signal Processing and Integrated Networks*. 2021. URL: https://doi.org/10.1109/SPIN52536.2021.9565972.

75. Yang C., Ye W., Li Q. Review of the Performance Optimization of Parallel Manipulators. *Mechanism and Machine Theory*. 2022. Vol. 170. URL: https://doi.org/10.1016/j.mechmachtheory.2022.104725.

76. Abualigah L., Masri B.A. Advances in MapReduce Big Data Processing: Platform, Tools, and Algorithms. *Artificial Intelligence and IoT. Studies in Big Data*, Mol 85. 2021. URL: https://doi.org/10.1007/978-981-33-6400-4_6.

77. Hassan A.O., Hasan A.A. Simplified Data Processing for Large Cluster: A MapReduce and Hadoop Based Study. *Advances in Applied Sciences*, 2021. Vol. 6. №3. URL: https://doi.org/10.11648/j.aas.20210603.11.

78. Chang D., Li L., Chang Y. Implementation of MapReduce Parallel Computing Framework Based on Multi-Data Fusion Sensors and GPU Cluster. *EURASIP Journal on Advances in Signal Processing*. 2021. Vol. 2021. №77. URL: https://doi.org/10.1186/s13634-021-00787-7.

79. Sutradhar P.R. Look-up-Table Based Processing-in-Memory Architecture With Programmable Precision-Scaling for Deep Learning Applications. *Transactions on Parallel and Distributed Systems*. 2022. Vol. 33. №2. URL: https://doi.org/10.1109/TPDS.2021.3066909.

80. Li H., Dong W., Lu L., Wang Y., Wang X. Distributed Cooperative Driving Strategy for Connected Automated Vehicles at Unsignalized Intersections Based on Monte Carlo Method. *Journal of Advanced Transportation*. 2024. URL: https://doi.org/10.1155/2024/6586774.

ДОДАТОК А
(ОБОВ'ЯЗКОВИЙ)

ТЕЗИ

УДК 004.424

МЕТОД ОПТИМІЗАЦІЇ ПАРАЛЕЛЬНОЇ ОБРОБКИ ДАНИХ ЗГІДНО З КЛАСТЕРНОЮ АРХІТЕКТУРОЮ CUDA

ДЗЮБЧИК О. Л. (sasha32083@gmail.com)
Хмельницький національний університет

Розглядаються основи методу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA. В основі методу лежить специфіка використання програмно-апаратної архітектури CUDA.

З метою вирішення задачі підвищення продуктивності паралельної обробки даних, було розроблено метод оптимізації з використанням кластерної архітектури CUDA. Для підвищення ефективності паралельної обробки даних було впроваджено підходи, що базуються на інструментарії та структурі CUDA.

Метод заснований на оптимізації паралельних обчислень шляхом передачі їх виконання з CPU на GPU. Запропонований підхід використовує передові техніки програмування на CUDA, зокрема розподіл обчислень між блоками та потоками у GPU з комбінуванням різних типів пам'яті, що надає CUDA.

Паралельні обчислення на GPU за допомогою архітектури CUDA мають ряд особливостей, які роблять їх ефективними для обробки великих обсягів даних та складних обчислювальних задач. Перш за все це архітектура графічних процесорів, GPU має сотні або тисячі ядер, які можуть виконувати обчислення паралельно, що значно збільшує швидкість обробки даних порівняно з традиційними CPU. Виконання коду за допомогою векторизації дозволяє обробляти дані великими блоками замість окремих елементів. Це знижує кількість інструкцій, необхідних для виконання програми, і підвищує загальну продуктивність. Крім цього, CUDA надає гнучкі інструменти для керування пам'яттю на GPU, включаючи різні типи пам'яті, що відрізняються за обсягом і швидкістю доступу. Розуміння структури пам'яті CUDA дозволяє виконувати обчислення з високою інтенсивністю доступу до даних швидше та ефективніше.

Таким, чином, метод включає наступні кроки.

Аналіз потреб обчислень і оцінка доцільності використання CUDA. Перед прийняттям рішення про необхідність використання архітектури CUDA для оптимізації паралельної обробки даних, необхідно провести аналіз алгоритму, щоб зрозуміти чи буде його паралельна версія ефективно виконуватись на GPU (необхідно забезпечити атомарність даних, обчислення над якими виконуватимуться паралельно і мінімізувати кількість розгалужень у коді). Крім того, переваги від паралелізму на GPU стають помітними при обробці великих об'ємів даних, де затрати на передачу даних компенсуватимуться зниженням часу обчислень.

Вибір відповідної архітектури GPU: Після проведеного аналізу алгоритму, потрібно обрати відповідну архітектуру GPU. Різні моделі GPU мають різну кількість ядер та об'єми пам'яті, що впливає на швидкість і ефективність обчислень. Важливо враховувати баланс між потужністю обчислень та вартістю GPU, а також забезпечити сумісність із наявною інфраструктурою.

Створення оптимізованого ядра CUDA. Всі основні операції, які будуть виконуватись в кожному потоці, описуються саме в ядрі, тому від його оптимальності й залежить основний успіх оптимізації. Ефективне ядро максимізує використання обчислювальних ресурсів GPU, мінімізуючи затримки та зайве використання пам'яті.

Вибір використовуваних типів пам'яті CUDA. Наступним етапом впровадження оптимізації є

вибір серед потрібного типу (чи комбінації типів) пам'яті серед пропонованих програмно-апаратною архітектурою. Рішення про використання різних типів пам'яті має базуватися на обсягах обчислень і потребі в швидкодії доступу до даних. Не ефективне використання пам'яті може стати причиною зниження продуктивності системи. Таким чином, при виборі типу пам'яті, варто розуміти, що регістри, це швидка, але невелика пам'ять кожного потоку, коли її не вистачає варто використати локальну пам'ять, що має хоч і вищі затримки, але й вищий об'єм. Спільну пам'ять корисно використовувати для доступу до даних потоків одного блоку, тоді як глобальна пам'ять доступна всім потокам і блокам. Константна і текстурна пам'яті є кешованими і оптимізованими, перша призначена для зберігання констант доступних усім потокам, а друга – для специфічних сценаріїв звернення, які не змінюють дані.

Оптимізація передачі даних між CPU і GPU. Оскільки, на графічному процесорі відбуваються переважно обчислення, необхідно організувати передачу даних таким чином, щоб всі подготовчі процеси, як збір, фільтрація, нормалізація тощо, відбувались на стороні CPU, а на GPU передавалась мінімальна їх кількість у форматі готовому до використання у ядрі.

Конфігурація кількості блоків і потоків. Визначення оптимального розміру блоку — це ще один вкрай важливий етап забезпечення максимальної ефективності. Оптимальна кількість потоків в блоку часто кратна 32, що відповідає розміру варпа в архітектурі NVIDIA. Це важливо для мінімізації кількості невикористаних потоків і збільшення швидкодії виконання алгоритмів. Кількість блоків у гріді повинна бути достатньою, щоб кожен SM мав достатньо роботи. Ідеально, коли всі SM зайняті, але без перевантаження, яке може спричинити зниження продуктивності через збільшення звернень до глобальної пам'яті та очікування завершення інструкцій.

Тестування та оцінка продуктивності системи. Фінальним етапом можна вважати тестування та оцінку продуктивності системи. Під час тестування аналізуються різні аспекти системи, включаючи швидкодію обчислень, точність результатів, стабільність роботи та споживання ресурсів. Оцінка споживання ресурсів дає інформацію про використання пам'яті і обчислювальних потужностей. Тестування дозволяє виявити та вирішити потенційні проблеми, що можуть виникнути під час виконання паралельних алгоритмів на GPU

Висновки. Розроблено метод оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA. Новий метод базується на використанні специфічних можливостей CUDA для збільшення швидкості і ефективності обробки великих масивів даних у паралельних обчислювальних середовищах. Застосовано підходи паралельної обробки і специфікації пам'яті, спеціально адаптовані для архітектури CUDA, що дозволяє значно оптимізувати процеси обчислення та аналізу даних у різних областях, від обробки сигналів до аналітики великих даних.

ДОДАТОК Б
ЛІСТИНГ ПОСЛІДОВНОЇ РЕАЛІЗАЦІЇ kNN

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define ROWS 3000
#define COLS 64

void readMatrixFromFile(const char* filename, int matrix[ROWS][COLS]) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        printf("Помилка читання файлу\n");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            if (!fscanf(file, "%d,", &matrix[i][j])) {
                printf("Помилка при читанні даних\n");
                exit(EXIT_FAILURE);
            }
        }
    }
    fclose(file);
}

void computeDistances(double distances[ROWS][ROWS], int matrix[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = i + 1; j < ROWS; j++) {
            double sum = 0.0;
            for (int k = 0; k < COLS; k++) {
                double diff = matrix[i][k] - matrix[j][k];
                sum += diff * diff;
            }
            distances[i][j] = sqrt(sum);
            distances[j][i] = distances[i][j]; // Відстань симетрична
        }
    }
}

void sortDistances(double distances[ROWS][ROWS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < ROWS - 1; j++) {
            for (int k = j + 1; k < ROWS; k++) {
                if (distances[i][j] > distances[i][k]) {
                    double temp = distances[i][j];
                    distances[i][j] = distances[i][k];
                    distances[i][k] = temp;
                }
            }
        }
    }
}

int main() {
    int matrix[ROWS][COLS];
    double distances[ROWS][ROWS];
```

```
    readMatrixFromFile("test.csv", matrix);

    clock_t start = clock();
    computeDistances(distances, matrix);
    sortDistances(distances);
    clock_t end = clock();

    double total_time = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Час виконання: %f секунд\n", total_time);

    return EXIT_SUCCESS;
}
```

ДОДАТОК В
ЛІСТИНГ ПАРАЛЕЛЬНОЇ РЕАЛІЗАЦІЇ kNN на CPU

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define NUM_THREADS 8 // Кількість потоків
#define NUM_ROWS 3000 // Кількість рядків у матриці
#define NUM_COLS 64 // Кількість стовпців у матриці

int dataMatrix[NUM_ROWS][NUM_COLS]; // Матриця даних
double distanceMatrix[NUM_ROWS][NUM_ROWS] = {{0}}; // Матриця відстаней

void *calculateDistances(void *threadId) {
    int threadNum = *(int*)threadId;

    // Обчислення відстані між рядками матриці
    for (int currentRow = threadNum; currentRow < NUM_ROWS; currentRow += NUM_THREADS) {
        for (int otherRow = currentRow + 1; otherRow < NUM_ROWS; otherRow++) {
            double sum = 0.0;
            for (int col = 0; col < NUM_COLS; col++) {
                double diff = dataMatrix[currentRow][col] - dataMatrix[otherRow][col];
                sum += diff * diff;
            }
            distanceMatrix[currentRow][otherRow] = sqrt(sum);
            distanceMatrix[otherRow][currentRow] = distanceMatrix[currentRow][otherRow]; // Симетрія матриці
відстаней
        }
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int threadIDs[NUM_THREADS];
    clock_t start, end;
    double totalTime;

    // Завантаження даних з файлу
    FILE *file = fopen("test.csv", "r");
    if (!file) {
        printf("Помилка при відкритті файлу\n");
        exit(1);
    }

    for (int i = 0; i < NUM_ROWS; i++) {
        for (int j = 0; j < NUM_COLS; j++) {
            fscanf(file, "%d", &dataMatrix[i][j]);
        }
    }
    fclose(file);

    start = clock();
    // Створення потоків для паралельних обчислень
    for (int i = 0; i < NUM_THREADS; i++) {
        threadIDs[i] = i;
        if (pthread_create(&threads[i], NULL, calculateDistances, &threadIDs[i])) {
```

```
            fprintf(stderr, "Помилка створення потоку\n");
            return EXIT_FAILURE;
        }
    }

    // Очікування завершення всіх потоків
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    end = clock();
    totalTime = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Час виконання програми: %f секунд\n", totalTime);

    return EXIT_SUCCESS;
}
```

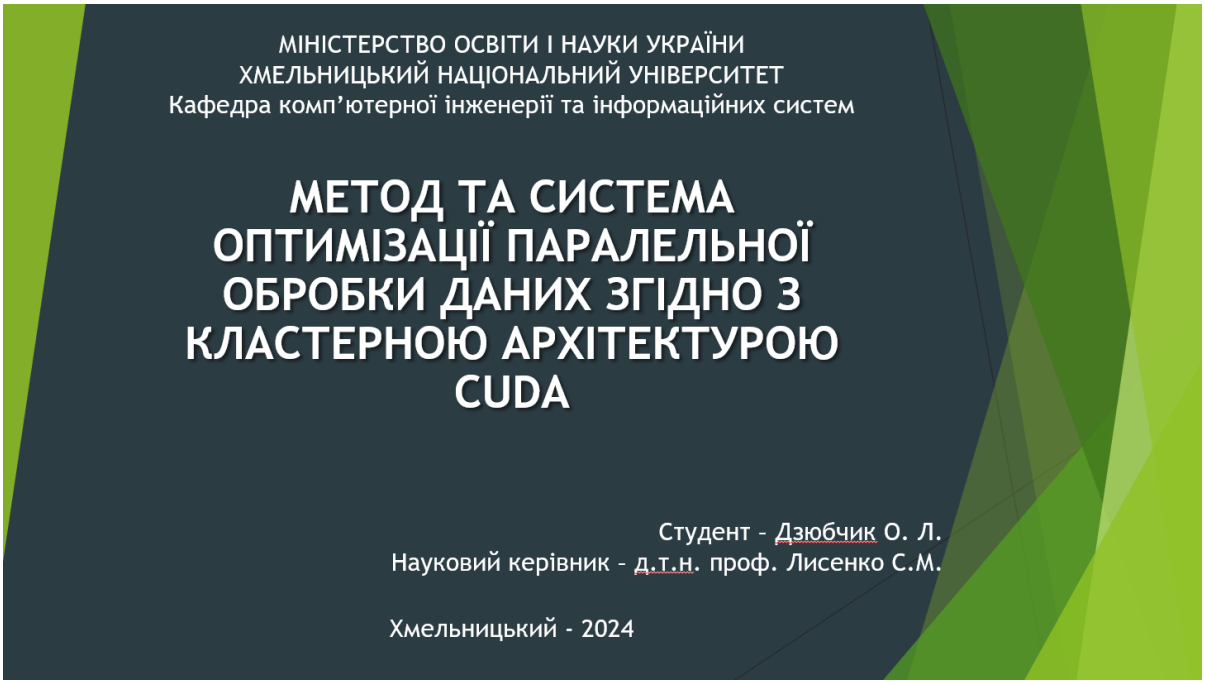
ДОДАТОК Г
ЛІСТИНГ ПАРАЛЕЛЬНОЇ РЕАЛІЗАЦІЇ kNN на GPU

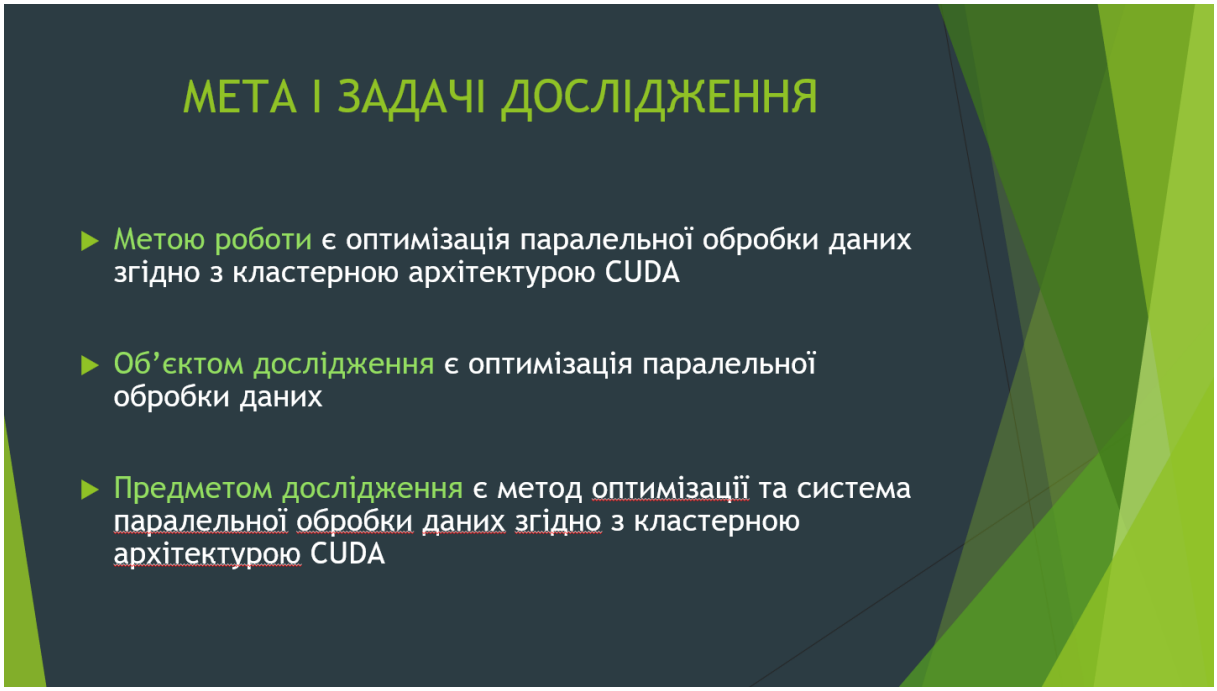
```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

#define NUM_ROWS 3000
#define NUM_COLS 64

int inputMatrix[NUM_ROWS][NUM_COLS];
double distanceMatrix[NUM_ROWS][NUM_ROWS];

__global__ void calculateDistances(const int* input, double* distances) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    double sum = 0;

    for (int i = idx; i < NUM_ROWS; i += stride) {
        for (int j = i + 1; j < NUM_ROWS; j++) {
            sum = 0;
            for (int k = 0; k < NUM_COLS; k++) {
                double diff = input[i * NUM_COLS + k] - input[j * NUM_COLS + k];
                sum += diff * diff;
            }
            distances[i * NUM_ROWS + j] = sqrt(sum);
            distances[j * NUM_ROWS + i] = distances[i * NUM_ROWS + j]; // Симетрія матриці відстаней
        }
    }
}

__global__ void sortDistances(double* distances, int numNearest) {
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < NUM_ROWS; i += blockDim.x * gridDim.x) {
        for (int j = 0; j < numNearest; j++) {
            for (int k = j + 1; k < NUM_ROWS; k++) {
                if (distances[i * NUM_ROWS + j] > distances[i * NUM_ROWS + k]) {
                    double temp = distances[i * NUM_ROWS + j];
                    distances[i * NUM_ROWS + j] = distances[i * NUM_ROWS + k];
                    distances[i * NUM_ROWS + k] = temp;
                }
            }
        }
    }
}

int main() {
    printf("Введіть кількість найближчих сусідів: ");
    int numNearest;
    scanf("%d", &numNearest);

    FILE* file = fopen("test.csv", "r");
    if (file == NULL) {
        printf("Не вдалося відкрити файл\n");
        exit(1);
    }
```

```
    for (int i = 0; i < NUM_ROWS; i++) {
        for (int j = 0; j < NUM_COLS; j++) {
            fscanf(file, "%d,", &inputMatrix[i][j]);
        }
    }
    fclose(file);

    int *dev_input;
    double *dev_distances;
    cudaMalloc((void**)&dev_input, NUM_ROWS * NUM_COLS * sizeof(int));
    cudaMalloc((void**)&dev_distances, NUM_ROWS * NUM_ROWS * sizeof(double));
    cudaMemcpy(dev_input, inputMatrix, NUM_ROWS * NUM_COLS * sizeof(int), cudaMemcpyHostToDevice);

    dim3 blockSize(256);
    dim3 gridSize((NUM_ROWS + blockSize.x - 1) / blockSize.x);

    clock_t start = clock();
    calculateDistances<<<gridSize, blockSize>>>(dev_input, dev_distances);
    sortDistances<<<gridSize, blockSize>>>(dev_distances, numNearest);
    cudaDeviceSynchronize();

    cudaMemcpy(distanceMatrix, dev_distances, NUM_ROWS * NUM_ROWS * sizeof(double),
cudaMemcpyDeviceToHost);
    clock_t end = clock();

    cudaFree(dev_input);
    cudaFree(dev_distances);

    double totalTime = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Обробка даних завершена. Час виконання: %f секунд.\n", totalTime);
    return 0;
}
```

ДОДАТОК Д
(ОБОВ'ЯЗКОВИЙ)

ПРЕЗЕНТАЦІЯ





НАУКОВА НОВИЗНА ТА ПРАКТИЧНА ЦІННІСТЬ ОТРИМАНИХ РЕЗУЛЬТАТІВ

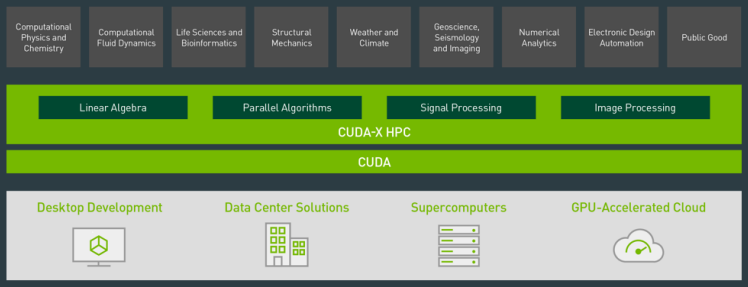
- Наукова новизна отриманих результатів:

1. Набув подальшого розвитку метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA, який на відміну від відомих методів пропонує генералізований підхід без прив'язки до конкретного алгоритму, що дозволяє оптимізувати паралельної обробки даних.

2. Набули подальшого розвитку програмно-технічні засоби оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

- Практична цінність отриманих результатів полягає у розробці апаратно-програмного засобу оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

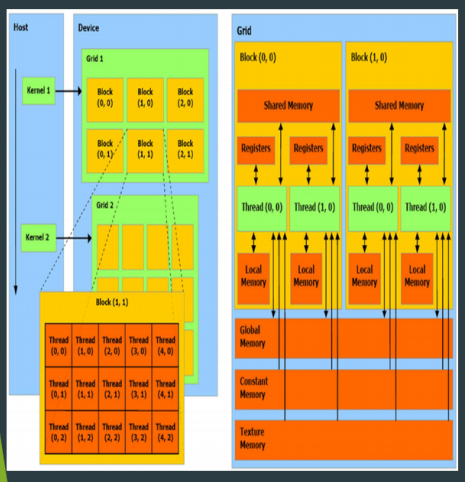
АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ



- У сучасному світі обчислювальні технології розвиваються стрімкими темпами, а потреба в ефективній обробці великих обсягів даних стає дедалі актуальнішою. Інноваційні підходи в паралельній обробці даних, як-от кластерна архітектура CUDA від NVIDIA, відіграють ключову роль у досягненні нових висот в обчислювальній ефективності.

- Архітектура CUDA перетворила графічні процесори на масивно-паралельні обчислювальні машини, які відкривають нові можливості для обробки великих даних та вирішення складних обчислювальних задач. Розвиток паралельної обробки даних за допомогою кластерної архітектури CUDA стає ключем до прискорення наукових досліджень та інженерних розробок, оскільки вона дозволяє ефективно використовувати обчислювальні потужності GPU.

ОСОБЛИВОСТІ АРХІТЕКТУРИ CUDA



- Ядро - функція, написана на мові програмування, яка виконується паралельно на GPU, кожен екземпляр якої обробляє різні дані за допомогою різних потоків.
- Потік - найменша одиниця виконання у CUDA, яка виконує певну частину коду ядра незалежно від інших потоків.
- Варп - група 32 потоків, що виконуються одночасно на одному SM GPU і є основною одиницею планування виконання інструкцій.
- Блок - група потоків, яка може містити кілька варпів, і розділяє загальні ресурси виконуючись на одному SM.
- Грід - збір блоків, які виконують одне і те саме ядро і розподіляються між SM GPU для паралельного виконання.

МОДЕЛЬ ВИКОРИСТАННЯ РЕСУРСІВ

Базовий синтаксис моделі

- типи: $\beta := int \mid bool \mid B \mid arr(\beta)$,
- операнди: $o := x \mid p \mid c \mid tid$,
- масиви: $A := Gl \mid Sh$,
- вирази: $e := o \mid o\ op\ o \mid A[o]$,
- інструкції: s := $while\ (e)\ s$

Приклади правил оцінки використання ресурсів

$$OCR: Const \Rightarrow \frac{}{\sigma; c \downarrow_M^B (c)_{t \in B}; MC^{const}},$$

$$ECR: Op \Rightarrow \frac{\sigma; o_1 \downarrow_R^B R_1; C_1 \quad \sigma; o_2 \downarrow_R^B R_2; C_2}{\sigma; o_1\ op\ o_2 \downarrow_R^B \left(R_1(t)\ op\ R_2(t)\right)_{t \in B}; C_1 + C_2 + MC^{op}})$$

МЕТОД ОПТИМІЗАЦІЇ ПАРАЛЕЛЬНОЇ ОБРОБКИ ДАНИХ ЗГІДНО З КЛАСТЕРНОЮ АРХІТЕКТУРОЮ CUDA

Метод включає наступні кроки:

1. Аналіз потреб обчислень і оцінка доцільності використання CUDA.
2. Вибір відповідної архітектури GPU
3. Створення оптимізованого ядра CUDA.
4. Вибір використовуваних типів пам'яті CUDA.
5. Оптимізація передачі даних між CPU і GPU.
6. Конфігурація кількості блоків і потоків.
7. Тестування та оцінка продуктивності системи.

ЗАСТОСУВАННЯ МЕТОДУ ОПТИМІЗАЦІЇ ДО АЛГОРИТМУ kNN

- ✓ Паралельне виконання алгоритму kNN може бути реалізовано шляхом розділення навчальної вибірки на декілька сегментів, які будуть оброблені одночасно в різних потоках виконання. Кожен потік відповідатиме за обчислення відстаней від тестових точок до точок у певному сегменті навчальної вибірки та визначення найближчих сусідів у ньому.
- ✓ Вибір моделі GPU залежить від обсягів вхідних даних.
- ✓ Ядро CUDA для KNN виконуватиме обчислення відстаней між тестовими та навчальними точками та визначення k найближчих сусідів.
- ✓ Для зберігання даних, які активно використовуються потоками в межах одного блоку слід використати спільну пам'ять. Для зберігання проміжних результатів обчислення відстаней і тимчасових значень, кожен потік може використовувати свої локальні регістри

ЗАСТОСУВАННЯ МЕТОДУ ОПТИМІЗАЦІЇ ДО АЛГОРИТМУ kNN

- ✓ Після завершення обчислень на GPU, результати виконання алгоритму KNN потрібно передати назад на CPU для подальшого аналізу та використання у прикладних завданнях. Оптимізація процесу передачі даних включає в себе використання асинхронного переносу даних
- ✓ Кожен блок у CUDA може обробляти частину навчальних даних, виконуючи обчислення відстаней для певної підмножини тестових точок. Розмір блоку впливає на кількість одночасно оброблюваних тестових точок, відповідно великі блоки можуть забезпечити більшу пропускну здатність, але також можуть призвести до невикористання обчислювальних ресурсів через обмеження пам'яті блоку.
  - ✓ Грід визначає загальну кількість блоків, які будуть використані для обчислень. Розмір гріда повинен бути достатнім для покриття всіх тестових точок, які потрібно класифікувати. Надмірно великі гріди можуть призвести до збільшення витрат на управління.

ЗАСТОСУВАННЯ МЕТОДУ ОПТИМІЗАЦІЇ ДО АЛГОРИТМІВ FFT і ДЕЙКСТРИ

Алгоритм FFT

1. **Аналіз математичної моделі**: виділення обрахунків DFT як обчислень, що виконуватимуться паралельно.
2. **Опис ядра**: розрахунок елементарних DFT, бітова інверсію індексів, обчислення твідл-факторів та комбінування вихідних результатів.
3. **Вибір пам'яті**: спільна, для збереження проміжних обрахунків і глобальної для отримання й передачі даних.
4. **Конфігурація блоків і потоків**: з урахуванням розміру вхідного масиву.

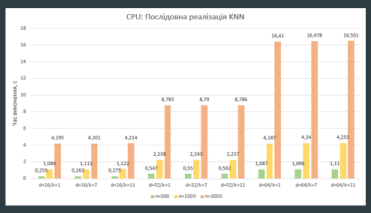
Алгоритм Дейкстри

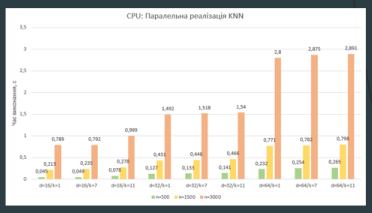
1. **Аналіз математичної моделі**: виділення обрахунків мінімальної відстані в певній частині графа, як функції, що виконуватиметься паралельно.
2. **Опис ядра**: обчислення відстані від однієї тестової точки до всіх навчальних точок, зберігання відстаней та індексів та вибір k найближчих сусідів для цієї тестової точки.
3. **Вибір пам'яті**: глобальна, для матриці суміжності, текстурна для вагів ребер, локальні регістри і спільна пам'ять для проміжних результатів обчислень.
4. **Конфігурація блоків і потоків**: з урахуванням кількості вершин графу.

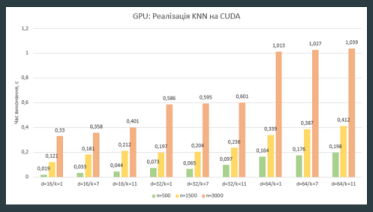
ВХІДНІ ДАНІ ДОСЛІДЖЕННЯ ЗАСТОСУВАННЯ МЕТОДУ ОПТИМІЗАЦІЇ

- Алгоритм: kNN (k-nearest neighbor/k-найближчих сусідів)
- Тестові Дані:
  - вектори розмірностей: 16, 32, 64;
  - датасети розмірів: 500, 1500, 3000;
  - параметр k-найближчих сусідів: 1, 7, 11.
- CPU: чотириядерний IntelCore i5-7300HQ (3.5 ГГц)
- GPU: GeForce GTX 1050
- RAM: 8 Gb

РЕЗУЛЬТАТИ ЗАСТОСУВАННЯ МЕТОДЩУ ОПТИМІЗАЦІЇ ДО АЛГОРИТМУ kNN







ПОРІВНЯННЯ РЕЗУЛЬТАТІВ ЗАСТОСУВАННЯ МЕТОДУ ОПТИМІЗАЦІЇ ДО АЛГОРИТМУ KNN

ПУБЛІКАЦІЇ

За темою кваліфікаційної роботи магістра опубліковані тези у матеріалах конференції XXIV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів «Стан, досягнення та перспективи інформаційних систем і технологій» 18-19 квітня 2024 р., Одеса, Україна.

ВИСНОВКИ

В результаті досліджень було:

- досліджено методи оптимізації паралельної обробки даних;
- досліджено існуючі методи оптимізації паралельної обробки даних з використанням CUDA;
- описано модель використання ресурсів та оцінки часу виконання варпом CUDA;
- удосконалено метод оптимізації паралельної обробки даних згідно з кластерною архітектури CUDA;
- реалізовано удосконалений метод оптимізації на прикладі алгоритму kNN.

ДЯКУЮ ЗА УВАГУ!

UNICHECK
by Turnitin

Ім'я користувача:
Кафедра КІ

ID перевірки:
1016247634

Дата перевірки:
14.05.2024 05:40:04 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
14.05.2024 06:44:18 EEST

ID користувача:
100005591

Назва документа: **Дзюбчик_Метод оптимізації та система паралельної обробки даних згідно з кластерною ар...**

Кількість сторінок: **100** Кількість слів: **24165** Кількість символів: **180241** Розмір файлу: **271.30 KB** ID файлу: **1016032800**

1.57%
Схожість

Найбільша схожість: **0.86%** з джерелом з Бібліотеки (ID файлу: **1016004946**)

1% Джерела з Інтернету	84	Сторінка 102
1.07% Джерела з Бібліотеки	29	Сторінка 102

0% Цитат

Цитати	1	Сторінка 103
Посилання	1	Сторінка 103

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	137

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 1.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 10%**

	Документ		Сумарний збіг по Базі Даних	
ID: 126087 Назва: МКР Метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA Додано в БД: 2024-05-13 Автора: Дзюбчик О.Л. Керівники: Лисенко С.М. Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	153815	1152	1566 (1%)	19 (2%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувач: Дзюбчик Олександр Леонідович

Тема: Метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень – ; кількість сторінок записки 82

1. Короткий зміст роботи та прийнятих рішень У роботі запропоновано метод оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA.

2. Висновок про відповідність роботи дипломному завданню Кваліфікаційна робота магістра відповідає виданому завданню

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі проведено огляд методів оптимізації паралельної обробки даних, зокрема з використанням кластерної архітектури CUDA. Досліджено відомі рішення та засоби в цій сфері. У другому розділі запропоновано модель використання ресурсів і оцінки часу виконання ядра в межах варпу CUDA. У третьому розділі запропоновано метод оптимізації паралельної обробки даних згідно з кластерно архітектурою CUDA, розглянуто його застосування до алгоритмів kNN, FFT і Дейкстри. У четвертому розділі запропоновано програмно апаратну систему для оптимізації паралельної обробки даних на прикладі реалізації алгоритму kNN.

4. Позитивні сторони роботи: Запропонований апаратно-програмного засіб оптимізації паралельної обробки даних згідно з кластерною архітектурою CUDA, дозволяє пришвидшити час виконання алгоритмів паралельної обробки даних.

5. Негативні сторони роботи: В першому розділі роботи приділено недостатньо увагу відомим засобам використання GPU для паралельної обробки даних.

6. Оцінка графічного оформлення та пояснювальної записки роботи: -

7. Відгук про роботу в цілому: В загальному робота виконана на достатньому рівні.

8. Інші зауваження: -

9. Оцінка кваліфікаційної роботи:

Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи вважаю, що робота заслуговує оцінки «задовільно» 3.50 (D)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) д.т.н.. професор, Бармак О.В., завідувач кафедри комп'ютерних наук

"13 травня" 2024р.

Завідувачу кафедри КІІС
д-р.техн.наук, проф. Говорущенко Т. О.

Дзюбчика Олександра Леонідовича
ПІБ здобувача вищої освіти

ФІТ, 2 курсу, групи КІ2м-22-2

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 01.07.2022, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

22 квітня 2024 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОМАЦІЙНИХ СИСТЕМ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Метод оптимізації та система паралельної обробки даних згідно з кластерною архітектурою CUDA

Автор: Дзюбчика Олександра Леонідовича

Спеціальність: 123 – Комп'ютерна інженерія

Освітня програма: освітньо-наукова

Науковий керівник: Лисенко Сергій Миколайович, д.т.н, професор

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;
2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;
3) в якості запозичень в окремих місцях системою зафіксовано послідовності рядків коду на мові програмування С, які є вхідними даними до великої кількості задач і не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;
4) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості Unicheck, складає 1.57% і адресується до 113 першоджерела; та системою Anti-Plagiarism складає 1%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи _____ С. М. Лисенко

Гарант ОП _____ О. С. Савенко

Завідувач кафедри КІІС _____ Т. О. Говорущенко