НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.912

«До захисту допущено»
Завідувач кафедри
_____ Євгенія СУЛЕМА
«___»_____2024 р.

Магістерська дисертація

на здобуття ступеня магістра
освітньо-науковою програмою
«Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»
зі спеціальності 121 Інженерія програмного забезпечення
на тему: «Удосконалений метод UMAP з використанням графічних процесорів»

Виконав:
студент ІІ курсу, групи КП-21мн
Кривчук Денис Віталійович _____

Керівник:
к.ф.-м.н., доцент,
Нещадим Олександр Михайлович _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент:
доцент кафедри СПСКС, к.т.н., с.н.с.
Боярінова Юлія Євгенівна _____

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

Київ – 2024 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Євгенія СУЛЕМА

«___»_____2022 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Кривчуку Денису Віталійовичу

1. Тема дисертації «Удосконалений метод UMAP з використанням графічних процесорів», науковий керівник дисертації Нещадим Олександр Михайлович, к.ф.-м.н., доцент, затверджені наказом від «05» травня 2024 р. № 1478-С.

2. Термін подання студентом дисертації «13» червня 2024 р.

3. Об'єкт дослідження: процес оптимізації алгоритму UMAP для підвищення ефективності та швидкості виконання шляхом використання графічних процесорів (GPU).

4. Предмет дослідження: методи, алгоритми та програмні засоби модифікації алгоритму UMAP для паралельних обчислень на графічних процесорах, а також їх вплив на якість і продуктивність обробки великих даних.

5. Перелік завдань, які потрібно розробити:
- провести аналіз існуючих методів і підходів до оптимізації алгоритмів для обчислень на графічних процесорах;
- розробка модифікацій алгоритму UMAP для його ефективної роботи на графічних процесорах;
- реалізація модифікованого методу UMAP з використанням сучасних бібліотек для обчислень на GPU;
- проведення експериментів для оцінки продуктивності та ефективності модифікованого методу UMAP у порівнянні з традиційними методами;
- аналіз отриманих результатів та розробка рекомендацій щодо використання модифікованого методу UMAP у практичних завданнях обробки даних;

6. Орієнтовний перелік графічного (ілюстративного) матеріалу:
   - схема індексації розріжених даних (плакат)
   - запуски алгоритму UMAP на тих же даних (плакат)
   - використання віконного алгоритму (плакат)
   - перший запуск нашого алгоритму (плакат)
   - другий запуск нашого алгоритму (плакат)
   - поріання швидкості виконання нашого алгоритму та rapids (плакат)

7. Орієнтовний перелік публікацій:
   - Тези доповіді “Використання графічних процесорів для виконання задач хемінформатики”
   - Публікація статті у фахових журналах за спеціальністю 121 «Інженерія програмного забезпечення»

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент кафедри ПЗКС		

9. Дата видачі завдання «11» жовтня 2022 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ознайомлення з предметною галуззю	15.12.2022	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	04.06.2023	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	05.10.2023	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення;	10.12.2023	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження; підготовка матеріалів доповіді на конференції ПМК-2023	15.11.2023	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації	28.01.2023	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу	17.04.2024	
8.	Оформлення текстової та графічної частини магістерської дисертації	10.05.2024	

Студент Денис КРИВЧУК

Науковий керівник Олександр НЕЩАДИМ

РЕФЕРАТ

Актуальність теми. Основний виклик сучасної обробки великих даних полягає в забезпеченні ефективності та швидкості алгоритмів. Метод UMAP (Uniform Manifold Approximation and Projection) є одним із найпотужніших інструментів для візуалізації та зменшення розмірності даних. Проте, традиційний підхід до його реалізації на центральних процесорах (CPU) не завжди задовольняє вимоги сучасних обсягів даних. Використання графічних процесорів (GPU) для оптимізації цього методу дозволяє значно підвищити продуктивність, що є актуальним для багатьох галузей науки і техніки.

Об'єктом дослідження є процес оптимізації алгоритму UMAP для підвищення ефективності та швидкості виконання шляхом використання графічних процесорів (GPU).

Предметом дослідження є методи, техніки модифікації алгоритму UMAP для паралельних обчислень на графічних процесорах, та ефективного використання ресурсів, і їх вплив на якість і продуктивність обробки великих даних.

Мета роботи: підвищення ефективності обробки великих обсягів даних шляхом розробки і реалізації модифікованого методу UMAP, який використовує можливості графічних процесорів для прискорення обчислень.

Методи дослідження. В роботі використовуються теоретичний аналіз наукової літератури, розробка алгоритмічних модифікацій, програмна реалізація і тестування модифікованого методу, а також експериментальний аналіз продуктивності та якості алгоритму.

Наукова новизна. Запропоновано метод оптимізації алгоритму UMAP для обчислень на графічних процесорах, характерними рисами якого є використання індексів для даних, ітеративного методу k-NN та

кумулятивними операторами при роботі з даними, що дає можливість підвищити швидкодію в середньому на 24%.

Практична значущість. На основі запропонованого методу можуть бути розроблені програмні рішення для обробки великих обсягів даних у різних галузях науки та промисловості, що дозволить оптимізувати використання обчислювальних ресурсів. Це особливо важливо для таких галузей, як біоінформатика, обробка зображень, фінансовий аналіз та маркетинг, де швидкість аналізу даних має критичне значення.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на XVI науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2023 (Київ, 28-30 листопада 2023 р.).

Стаття "Використання графічних процесорів для виконання задач хемінформатики" буде опубліковано в матеріалах міжнародної научної та практичної конференції «INTERNATIONAL FORUM: PROBLEMS AND SCIENTIFIC SOLUTIONS» (6-8 червня, 2024; Мельбурн, Австралія).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів, висновків та додатків.

У вступі надано загальну характеристику роботи, виконано оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень.

У першому розділі розглянуто загальні методи зменшенні розмірності, метод UMAP та його варіції. Проведено аналіз можливостей використання графічних процесорів та особливості при написанні реалізацій алгоритмів з їх використанням.

У другому розділі запропоновано основні методики, що використовуються при написанні алгоритму UMAP для GPU. Наведено список проблем що виникають при реалізації та методи обходу даних проблем. Надано логіку вибору конкретних реалізацій частин алгоритму для досягнення максимальної швидкості алгоритму.

У третьому розділі наведено список технологій, які були вибраними та аргументація їх вибору. Описано особливості реалізації та розглянуто приклад використання.

У четвертому розділі проведено аналіз результатів дослідження. Порівняно швидкодію реалізації з іншими ресурсами що надають дану версію алгоритму. Обгрунтовано вибір метрик для порівняння та зроблено порівняння даних результатів. Сформовано можливі наступні кроки для розвитку роботи.

У висновках проаналізовано отримані результати дослідження.

У додатках наведено лістинги коду програми, копії використаних зображень та копія презентації.

Робота виконана на 103 аркушах, містить 3 додатки та посилання на список використаних літературних джерел з 10 найменувань. У роботі наведено 6 рисунків, 4 таблиці, 1 лістинг.

Ключові слова: UMAP, графічні процесори (GPU), оптимізація алгоритмів, великі дані, паралельні обчислення, продуктивність, машинне навчання, візуалізація даних, зменшення розмірності, біоінформатика, хемінформатика, фінансовий аналіз, маркетинг, обчислювальні ресурси.

ABSTRACT

Theme urgency. The main challenge of modern big data processing lies in ensuring the efficiency and speed of algorithms. The UMAP (Uniform Manifold Approximation and Projection) method is one of the most powerful tools for data visualization and dimensionality reduction. However, the traditional approach to its implementation on central processing units (CPU) does not always meet the demands of contemporary data volumes. Utilizing graphics processing units (GPU) for optimizing this method significantly enhances performance, which is crucial for various fields of science and technology.

Object of research is the process of optimizing the UMAP algorithm to improve efficiency and execution speed using graphics processing units (GPU).

Subject of research is methods and techniques for modifying the UMAP algorithm for parallel computations on graphics processors, efficient resource utilization, and their impact on the quality and performance of big data processing.

Research objective is to enhance the efficiency of big data processing by developing and implementing a modified UMAP method that leverages the capabilities of graphics processors to accelerate computations.

Research methods. The thesis uses theoretical analysis of scientific literature, development of algorithmic modifications, software implementation and testing of the modified method, as well as experimental analysis of the algorithm's performance and quality.

Scientific novelty. A method for optimizing the UMAP algorithm for computations on graphics processing units has been proposed, characterized by the use of data indices, an iterative k-NN method, and cumulative operators when working with data. This allows for an average performance increase of 24%.

Practical value. On the basis of the proposed method, software solutions for processing large volumes of data in various fields of science and industry can be developed, allowing for the optimization of computational resources. This is

especially important for fields such as bioinformatics, image processing, financial analysis, and marketing, where the speed of data analysis is critical.

Approbation. The basic provisions and results of the work were presented and discussed at the II Scientific Conference of Master's and Postgraduate Students "Applied Mathematics and Computing" PMC-2023 (Kyiv, November 28-30, 2023). The article "Using Graphics Processors for Chemoinformatics Tasks" will be published in the materials of the international scientific and practical conference "INTERNATIONAL FORUM: PROBLEMS AND SCIENTIFIC SOLUTIONS" (June 6-8, 2024; Melbourne, Australia).

Structure and content of the thesis. The master's thesis consists of the introduction, four chapters, conclusions, and appendixes.

The introduction provides a general characterization of the work, evaluates the current state of the problem, and substantiates the relevance of the research direction.

In the first section, the main methods of dimensionality reduction, the UMAP method, and its variations are reviewed. The potential of using graphics processors and the specifics of implementing algorithms with their use are analyzed.

In the second section, a modified UMAP algorithm for GPU is proposed. The main techniques used in writing the algorithm are outlined, listing problems encountered during implementation and methods to circumvent these issues. The logic for selecting specific implementations of parts of the algorithm to achieve maximum speed is provided.

The third section provides a list of chosen technologies and justification for their selection. The implementation features are described, and an example of use is examined.

In the fourth chapter, an analysis of the research results is conducted. The performance of the implementation is compared with other sources providing this version of the algorithm. The choice of metrics for comparison is justified, and

the obtained results are compared. Possible next steps for further work are suggested.

Conclusions. The obtained research results are analyzed. The work is presented on 103 pages, contains 3 appendices, and references a list of used literary sources of 10 titles. The work includes 6 figures, 4 tables, and 1 listing.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ 4

ВСТУП 6

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ 8

1.1. Процес зниження розмірності 8

1.2. Лінійні та нелінійні методи зменшення розмірності 9

1.3. Аналіз нелінійних методів зменшення розмірності 9

1.4. Особливості написання алгоритмів на GPU 12

1.5. Огляд існуючих реалізацій UMAP з використанням GPU 13

1.6. Висновки до першого розділу 16

2. МЕТОД UMAP ТА ОПТИМІЗАЦІЇ ПРИ РЕАЛІЗАЦІЇ НА ГРАФІЧНИХ ПРОЦЕСОРАХ 19

2.1. Метод UMAP 19

2.2. UMAP на графічних процесорах 21

2.3. Обробка розріджених даних 22

2.4. Масштабованість програми 24

2.5. Відтворюваність результатів 25

2.6. Модифікація методу UMAP 27

2.7. Вибір даних та метрик для визначення кількісних характеристик ефективності модифікованого методу 29

2.8. Вибір функції дистанції в UMAP 30

2.9. Висновки до розділу 2 31

3. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ РЕАЛІЗАЦІЇ МОДИФІКОВАНОГО МЕТОДУ UMAP З ВИКОРИСТАННЯМ ГРАФІЧНИХ ПРОЦЕСОРІВ. 33
3.1. Обгрунтування вибору засобів для реалізації програмного забезпечення. 33
3.2. Особливості реалізації 35
3.3. Архітектура програми 40
3.4. Використання програмного забезпечення. 42
3.5. Висновки до третього розділу 43
4. АНАЛІЗ ЕФЕКТИВНОСТІ РОБОТИ РОЗРОБЛЕНОГО МЕТОДУ 45
4.1. Критерії оцінювання та аналіз ефективності реалізації методу 45
4.2. Обґрунтування отриманих результатів 49
4.3. Напрямки подальшої роботи 50
4.3. Висновки до розділу 4 53
ВИСНОВКИ 55
СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ 58
ДОДАТКИ 60

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

UMAP – Uniform Manifold Approximation and Projection – алгоритм для зниження розмірності даних та візуалізації високорозмірних даних.

GPU – Graphics Processing Unit – графічний процесор, що використовується для прискорення обчислень.

CUDA – Compute Unified Device Architecture – платформа і модель програмування для обчислень на GPU.

FAISS – Facebook AI Similarity Search – бібліотека для швидкого пошуку найближчих сусідів.

cuML – бібліотека машинного навчання від RAPIDS для GPU.

cuSparse – бібліотека для роботи з розрідженими матрицями на GPU.

cuGraph – бібліотека для обробки графів на GPU.

RMM – RAPIDS Memory Manager – менеджер пам'яті для ефективного використання ресурсів GPU.

SGD – Stochastic Gradient Descent - стохастичний градієнтний спуск, метод оптимізації.

L1 і L∞ нормалізація – методи нормалізації вагів для забезпечення стабільності алгоритму.

Cython – інструмент для написання C-розширень для Python, що дозволяє поєднувати код на C і Python.

Symmetrization – процес симетризації матриці для стабільності алгоритму.

Numba – JIT-компілятор для Python, який перетворює функції Python в швидкий машинний код.

RAPIDS – набір бібліотек і API для виконання обчислень на GPU.

L2 відстань – евклідова відстань, використовується для вимірювання відстаней між точками в просторі.

kNN – k-Nearest Neighbors – алгоритм пошуку k найближчих сусідів для кожної точки в наборі даних.

Паралельні обчислення – техніка обчислень, що дозволяє одночасне виконання кількох обчислень для збільшення швидкості.

Розріджені дані – дані, в яких більшість елементів є нульовими або відсутніми.

Невипадковий пошук – метод пошуку найближчих сусідів, який використовує детерміновані алгоритми.

Симетрична матриця – матриця, яка є симетричною відносно головної діагоналі.

L2 нормалізація – метод нормалізації даних, що використовується для забезпечення стабільності обчислень.

Алгоритм навантаження графу – процес побудови графу для представлення даних, в якому вузли відповідають точкам, а ребра - зв'язкам між ними.

Стохастичний підхід – метод, що використовує випадковість для оптимізації обчислень.

ВСТУП

У сучасному світі обробка та аналіз великих обсягів даних є невід'ємною частиною багатьох наукових і технічних дисциплін. Зі збільшенням кількості інформації, яка потребує швидкої та ефективної обробки, зростає необхідність у нових методах та інструментах для аналізу даних. Одним із найпотужніших інструментів, що використовуються для цієї мети, є алгоритм UMAP (Uniform Manifold Approximation and Projection), який дозволяє знижувати розмірність даних, зберігаючи при цьому їх топологічну структуру. Це дає змогу виявляти приховані закономірності та візуалізувати дані у зручному для інтерпретації вигляді.

Проте, попри свою ефективність, алгоритм UMAP може бути досить ресурсомістким, особливо при роботі з великими наборами даних. Традиційні обчислювальні ресурси, такі як центральні процесори (CPU), можуть не забезпечити необхідної швидкості обробки, що обмежує застосування алгоритму в масштабних проектах. У відповідь на ці виклики дедалі більшої уваги привертає використання графічних процесорів (GPU) для прискорення обчислювальних процесів.

Графічні процесори відомі своєю здатністю до паралельної обробки великої кількості даних одночасно, що робить їх ідеальними для задач, які потребують інтенсивних обчислень. Використання GPU для прискорення обробки даних стає все більш популярним у різних галузях науки і техніки, від машинного навчання до обробки зображень і наукових обчислень. GPU дозволяють значно скоротити час, необхідний для обробки великих обсягів даних, що робить їх особливо корисними в задачах, де швидкість є критичним фактором.

У цьому контексті модифікація алгоритму UMAP для роботи на графічних процесорах представляє значний інтерес. Такий підхід дозволяє використовувати переваги паралельної обробки, що забезпечує швидку та ефективну обробку великих наборів даних. Оптимізація алгоритму UMAP

для GPU включає в себе розробку методів для мінімізації обсягів даних, які потребують передачі між CPU і GPU, а також ефективне управління ресурсами GPU для максимального підвищення продуктивності.

Застосування модифікованого алгоритму UMAP на GPU має потенціал для значного покращення швидкості та ефективності обробки даних у багатьох сферах. Це включає в себе аналіз великих даних у фінансових ринках, медичні дослідження, де важливо швидко аналізувати геномні дані, та обробку зображень, де великі обсяги даних потребують швидкого та точного аналізу. Використання GPU дозволяє дослідникам та інженерам зосередитися на інтерпретації результатів та прийнятті рішень на основі аналізу, а не витрачати час на довготривалі обчислення.

Таким чином, дослідження, присвячене модифікації алгоритму UMAP для використання на графічних процесорах, є важливим і перспективним напрямом розвитку в обробці великих даних. Воно дозволяє поєднати сучасні обчислювальні технології з потужними методами аналізу даних, відкриваючи нові можливості для вивчення та інтерпретації складних систем у різних галузях науки та техніки.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1. Процес зниження розмірності

Зниження розмірності є одним з ключових методів у галузі обробки даних, який дозволяє спрощувати складні багатовимірні дані, зберігаючи при цьому найбільш важливу інформацію. Це дає змогу полегшити аналіз, візуалізацію та інтерпретацію даних. У сучасних умовах, коли обсяги даних значно зростають, методи зниження розмірності стають надзвичайно важливими для ефективного управління інформацією та отримання інсайтів.

Високорозмірні дані можуть бути складними для обробки та аналізу через "прокляття розмірності". Цей термін означає, що зі збільшенням кількості вимірів обсяг простору зростає експоненціально, що ускладнює задачі класифікації, кластеризації та візуалізації. Зниження розмірності допомагає подолати ці труднощі, зменшуючи кількість вимірів без значної втрати інформації.

Основна мета зниження розмірності полягає в тому, щоб перетворити високовимірні дані на менш розмірні представлення, зберігаючи при цьому їхні ключові характеристики та структури. Це дозволяє:

- Полегшити візуалізацію даних: Зменшення розмірності до 2D або 3D простору дозволяє візуалізувати дані на графіках, що сприяє кращому розумінню та аналізу.
- Покращити продуктивність алгоритмів машинного навчання: Зменшення кількості змінних може знизити обчислювальні витрати та підвищити точність моделей, зменшуючи надмірну підгонку.
- Виділити найбільш значущі характеристики: Процес зниження розмірності може допомогти виявити ключові фактори, що впливають на результати аналізу.

1.2. Лінійні та нелінійні методи зменшення розмірності

Існує два основні типи методів зниження розмірності: лінійні та нелінійні. Кожен з цих методів має свої переваги та обмеження і може бути корисним у різних контекстах залежно від природи даних.

Лінійні методи припускають, що дані можна описати за допомогою лінійних комбінацій менших підмножин змінних. Вони ефективні для виявлення лінійних залежностей у даних і зазвичай менш обчислювально складні порівняно з нелінійними методами.

Метод головних компонент (Principal Component Analysis, PCA) є одним з найпоширеніших лінійних методів зниження розмірності. PCA працює шляхом перетворення даних у новий набір змінних, які називаються головними компонентами. Ці компоненти є ортогональними і ранжовані за їхньою здатністю пояснювати дисперсію в даних. PCA є особливо корисним для зменшення розмірності даних, зберігаючи при цьому максимально можливу варіативність.

Лінійний дискримінантний аналіз (Linear Discriminant Analysis, LDA) використовується для знаходження лінійних комбінацій ознак, які найкраще розділяють два або більше класи даних. LDA зосереджується на максимізації роздільної здатності між класами шляхом збереження міжкласової дисперсії, зменшуючи внутрішньокласову дисперсію.

Нелінійні методи здатні виявляти більш складні структури в даних, які не можуть бути описані лінійними залежностями. Вони особливо корисні, коли дані мають нелінійну природу.

Саме нелінійні методи нас цікавлять як задача яку вирішує UMAP.

1.3. Аналіз нелінійних методів зменшення розмірності

Основними нелінійними методами зменшення розмірності є t-SNE та UMAP.

t-SNE (t-Distributed Stochastic Neighbor Embedding) є популярним методом для зниження розмірності, особливо в задачах візуалізації

високовимірних даних. t-SNE добре підходить для виявлення кластерів та збереження локальних структур даних.

Основні етапи t-SNE включають:

- Обчислення ймовірностей подібності: Для кожної пари точок у багатовимірному просторі обчислюється ймовірність подібності, яка визначає, наскільки ймовірно ці точки є сусідами.
- Проекція у низьковимірний простір: Початкова випадкова проекція даних у простір з меншою розмірністю.
- Змінювання координат: Адаптивне змінювання координат точок у низьковимірному просторі для мінімізації різниці між подібностями у вихідному та проекційному просторах.

Переваги t-SNE:

- Збереження локальної структури: t-SNE зберігає локальні сусідства, тобто точки, які були близько одна до одної у високовимірному просторі, залишаються близькими і у низьковимірному.
- Візуалізація кластерів: t-SNE добре виявляє та візуалізує кластери в даних, що дозволяє виявити природні групи і структури.

Недоліки t-SNE:

- Обчислювальна складність: t-SNE є обчислювально інтенсивним, особливо на великих наборах даних.
- Параметризація: Параметри, такі як перплексія та швидкість навчання, мають значний вплив на результати і вимагають налаштування.
- Глобальна структура: t-SNE не завжди зберігає глобальну структуру даних, зосереджуючись на локальних відносинах.

t-SNE зазвичай використовується для візуалізації даних у низьковимірному просторі (2D або 3D), щоб виявити кластери або структури в даних. Він є популярним у таких галузях, як біоінформатика, аналіз зображень і машинне навчання.

UMAP (Uniform Manifold Approximation and Projection) є відносно новим методом зниження розмірності, який забезпечує швидке та точне перетворення даних, зберігаючи як глобальні, так і локальні структури.

Переваги UMAP:

- Швидкість: UMAP є значно швидшим за t-SNE, особливо на великих наборах даних.
- Збереження структури: UMAP зберігає як локальну, так і глобальну структуру даних, що дозволяє краще представляти топологію високовимірного простору.
- Гнучкість: Параметри UMAP легко налаштовуються, що дозволяє користувачам оптимізувати проекцію для різних типів даних.

Недоліки UMAP:

- Чутливість до параметрів: Як і інші методи, UMAP чутливий до налаштувань параметрів, таких як кількість сусідів і мінімальна відстань.
- Інтерпретація результатів: Результати UMAP можуть бути складними для інтерпретації, оскільки метод фокусується на збереженні топологічних властивостей.

UMAP знаходить застосування у багатьох галузях, включаючи біоінформатику, обробку зображень, аналіз текстів та машинне навчання. Він особливо корисний для візуалізації високовимірних даних, виявлення кластерів та підготовки даних для подальшого аналізу.

Проаналізуємо попарно аспекти даних алгоритмів:

- Швидкість: UMAP зазвичай працює швидше, ніж t-SNE, особливо на великих наборах даних.
- Збереження структури: t-SNE добре зберігає локальну структуру, тоді як UMAP зберігає як локальну, так і глобальну структуру.
- Вибір параметрів: Обидва методи чутливі до параметрів, але UMAP часто вимагає меншої кількості параметрів для налаштування.

- Візуалізація: Обидва методи добре підходять для візуалізації даних, але результати можуть відрізнятися залежно від специфіки даних і налаштувань параметрів.

Вибір між t-SNE та UMAP залежить від конкретних вимог задачі, розміру набору даних та необхідності збереження різних типів структур даних.

1.4. Особливості написання алгоритмів на GPU

Написання алгоритмів для графічних процесорів (GPU) має свої особливості та вимоги, оскільки архітектура GPU значно відрізняється від архітектури центрального процесора (CPU).

Масивний паралелізм є ключовою особливістю GPU, що мають тисячі ядер, які можуть виконувати тисячі паралельних завдань одночасно. Алгоритми повинні бути спроектовані таким чином, щоб максимально використовувати цей паралелізм, розбиваючи задачі на дрібніші, незалежні підзадачі, які можуть виконуватися паралельно.

Програми для GPU зазвичай складаються з великої кількості потоків, організованих у блоки. Кожен блок складається з потоків, що виконуються одночасно. Важливо правильно організувати роботу блоків і потоків, щоб уникнути проблем з балансуванням навантаження.

GPU мають різні типи пам'яті: глобальна, спільна, локальна та регістри. Використання спільної пам'яті (shared memory) всередині блоків може значно підвищити продуктивність, оскільки вона має набагато менший час доступу порівняно з глобальною пам'яттю. Однак при цьому необхідно ретельно управляти синхронізацією потоків, щоб уникнути гонок даних.

Доступ до глобальної пам'яті є відносно повільним, тому потрібно мінімізувати кількість таких доступів. Одним зі способів досягнення цього є кешування даних у спільній пам'яті або регістрах, де це можливо.

Для досягнення високої пропускної здатності пам'яті важливо забезпечити коалесцированый доступ до пам'яті, що означає, що сусідні

потоки повинні звертатися до сусідніх адрес у пам'яті. Це дозволяє зменшити кількість транзакцій пам'яті і підвищити ефективність.

Для написання програм для GPU можна використовувати різні бібліотеки та фреймворки, такі як CUDA для NVIDIA GPU або OpenCL для різних типів GPU. Використання цих інструментів може значно спростити розробку та оптимізацію алгоритмів.

Важливо розуміти апаратні обмеження конкретної архітектури GPU, на якій буде виконуватися алгоритм. Це включає кількість ядер, об'єм пам'яті, пропускну здатність пам'яті та інші параметри, які можуть вплинути на продуктивність алгоритму.

Налагодження і профілювання програм для GPU є важливою частиною процесу розробки. Інструменти, такі як NVIDIA Nsight, дозволяють виявляти вузькі місця у виконанні програм і оптимізувати їх для досягнення максимальної продуктивності.

1.5. Огляд існуючих реалізацій UMAP з використанням GPU

Одним із ключових аспектів сучасних методів зниження розмірності є використання графічних процесорів (GPU) для прискорення обчислень. GPU здатні виконувати паралельні обчислення з високою швидкістю, що робить їх ідеальними для обробки великих обсягів даних. Використання GPU дозволяє значно зменшити час обробки і підвищити продуктивність алгоритмів зниження розмірності, таких як UMAP.

GPU можуть ефективно обробляти завдання, що включають:

- Обчислення матриць подібностей: Швидке визначення найближчих сусідів для кожної точки у високовимірному просторі.
- Оптимізація: Паралельна обробка при змінюванні координат точок у низьковимірному просторі для досягнення оптимального збереження структури даних.

Розглянемо кілька ключових реалізацій UMAP з використанням GPU.

1.5.1. RAPIDS cuML

Переваги:

- Продуктивність: Завдяки використанню обчислень на GPU, RAPIDS cuML може значно зменшити час виконання алгоритму порівняно з реалізаціями на CPU.
- Інтеграція: cuML інтегрується з іншими бібліотеками екосистеми RAPIDS, такими як cuDF (для роботи з даними) та Dask (для розподілених обчислень), що дозволяє створювати потужні конвеєри обробки даних.
- Сумісність: Бібліотека забезпечує API, подібний до scikit-learn, що спрощує перехід для користувачів, які вже знайомі з цим популярним інструментом.

Недоліки:

- Залежність від NVIDIA: RAPIDS cuML працює виключно на графічних процесорах NVIDIA. Це обмежує використання на системах з іншими типами GPU, такими як AMD.
- Високі вимоги до апаратного забезпечення: Для ефективного використання RAPIDS cuML потрібні потужні GPU, що може бути дорогим для невеликих організацій чи окремих дослідників.
- Складність налаштування: Налаштування RAPIDS та його інтеграція з іншими інструментами можуть бути складними для користувачів, які не мають достатнього досвіду роботи з GPU та системами NVIDIA.

1.5.2. UMAP-learn з підтримкою GPU

Переваги:

- Гнучкість: Користувачі можуть легко перемикатися між CPU та GPU версіями, просто змінивши декілька налаштувань.

- Спільнота та підтримка: UMAP-learn має велику користувацьку базу та активну спільноту, що забезпечує надійну підтримку та часті оновлення.
- Налаштовуваність: Бібліотека дозволяє детально налаштовувати параметри моделі, що допомагає оптимізувати продуктивність для різних типів даних.

Недоліки:

- Обмежена гнучкість: Хоча UMAP-learn підтримує використання GPU через RAPIDS, його можливості можуть бути обмежені порівняно з повноцінною реалізацією на GPU.
- Проблеми з сумісністю: Інтеграція з RAPIDS cuML може спричинити проблеми сумісності та потребувати додаткових налаштувань та конфігурацій.
- Високі вимоги до пам’яті: Використання GPU може вимагати значної кількості відеопам’яті, що може бути проблематичним для великих наборів даних.

1.5.3. H2O.ai

Переваги:

- Автоматизація: Платформа H2O.ai забезпечує автоматизацію багатьох етапів машинного навчання, включаючи підготовку даних, побудову моделей та оптимізацію гіперпараметрів.
- Інтеграція: H2O.ai інтегрується з популярними середовищами даних та інструментами, що полегшує інтеграцію у існуючі робочі процеси.
- Продуктивність: Використання GPU дозволяє значно прискорити обчислення, особливо на великих наборах даних.

Недоліки:

- Висока вартість ліцензії: Хоча базова версія H2O.ai безкоштовна, розширені можливості та підтримка GPU можуть потребувати

платної ліцензії, що може бути недоступним для деяких користувачів.

- Залежність від інфраструктури: H2O.ai потребує налаштування спеціалізованої інфраструктури для ефективної роботи з GPU, що може бути складним та дорогим.
- Складність у використанні: Незважаючи на автоматизацію багатьох процесів, використання H2O.ai може бути складним для новачків через велику кількість параметрів та конфігурацій.

1.6. Висновки до першого розділу

Процес зниження розмірності є одним з найважливіших методів у галузі обробки даних, оскільки дозволяє значно спростити аналіз і візуалізацію багатовимірних даних, зберігаючи при цьому ключову інформацію. Зниження розмірності допомагає подолати "прокляття розмірності", зменшуючи кількість вимірів і тим самим спрощуючи подальший аналіз. Лінійні методи, такі як метод головних компонент (PCA), працюють шляхом перетворення даних на новий набір лінійних комбінацій, які максимально пояснюють варіативність даних. Хоча PCA є ефективним для лінійних залежностей, він не завжди може впоратися з більш складними структурами даних.

Нелінійні методи, такі як t-SNE та UMAP, призначені для виявлення складніших структур у даних. t-SNE (t-Distributed Stochastic Neighbor Embedding) є методом, який добре зберігає локальні структури даних і дозволяє виявляти кластери, але його основними недоліками є висока обчислювальна інтенсивність і складність у налаштуванні параметрів. t-SNE потребує значних обчислювальних ресурсів, особливо для великих наборів даних, і може бути складним у використанні через необхідність налаштування таких параметрів, як перплексія та швидкість навчання.

UMAP (Uniform Manifold Approximation and Projection) є новішим і швидшим методом зниження розмірності, який зберігає як локальні, так і

глобальні структури даних. UMAP забезпечує значну швидкість виконання та є більш гнучким у налаштуванні порівняно з t-SNE, що робить його ефективнішим для багатьох застосувань. Він також менш вимогливий до налаштування параметрів, що спрощує його використання.

Існуючі реалізації методу UMAP з використанням графічних процесорів (GPU) значно підвищують продуктивність і швидкість обробки даних. GPU здатні виконувати паралельні обчислення, що дозволяє значно скоротити час обробки великих наборів даних. Серед найпопулярніших реалізацій є RAPIDS cuML, UMAP-learn з підтримкою GPU та H2O.ai.

RAPIDS cuML, розроблений компанією NVIDIA, використовує можливості GPU для значного прискорення обчислень. Основними перевагами є висока продуктивність та інтеграція з іншими бібліотеками екосистеми RAPIDS. Проте, цей підхід має і свої недоліки. RAPIDS cuML працює виключно на GPU NVIDIA, що обмежує використання на системах з іншими типами GPU, такими як AMD. Крім того, для ефективного використання потрібні потужні GPU, що може бути дорогим для невеликих організацій чи окремих дослідників. Також налаштування RAPIDS та його інтеграція з іншими інструментами можуть бути складними для користувачів, які не мають достатнього досвіду роботи з GPU та системами NVIDIA

UMAP-learn з підтримкою GPU є ще однією реалізацією, яка використовує можливості GPU через інтеграцію з RAPIDS. Вона забезпечує гнучкість у виборі між CPU та GPU версіями та має велику користувацьку базу та активну спільноту. Однак, можливості цієї реалізації можуть бути обмеженими порівняно з повноцінною реалізацією на GPU. Крім того, інтеграція з RAPIDS cuML може спричинити проблеми сумісності та потребувати додаткових налаштувань і конфігурацій. Використання GPU також вимагає значної кількості відеопам’яті, що може бути проблематичним для обробки великих наборів даних.

H2O.ai пропонує платформу для автоматичного машинного навчання, яка включає підтримку UMAP з використанням GPU. Основними перевагами H2O.ai є автоматизація багатьох етапів машинного навчання, інтеграція з популярними середовищами даних та інструментами та висока продуктивність обчислень. Однак, розширені можливості та підтримка GPU можуть потребувати платної ліцензії, що може бути недоступним для деяких користувачів. Крім того, H2O.ai потребує налаштування спеціалізованої інфраструктури для ефективної роботи з GPU, що може бути складним та дорогим. Використання H2O.ai також може бути складним для новачків через велику кількість параметрів та конфігурацій.

Таким чином, незважаючи на значні переваги існуючих реалізацій UMAP з використанням GPU, існують певні обмеження, які потребують подальших досліджень і вдосконалення. Потреба в потужному апаратному забезпеченні, залежність від конкретних типів GPU та складність налаштування є основними викликами, які обмежують застосування цих рішень. Це підкреслює необхідність у розвитку нових підходів та оптимізації існуючих рішень для ефективного зниження розмірності великих і складних наборів даних. Подальші дослідження в цій галузі мають на меті подолання цих обмежень та забезпечення більш широкого доступу до високопродуктивних обчислювальних методів для аналізу даних.

2. МЕТОД UMAP ТА ОПТИМІЗАЦІЇ ПРИ РЕАЛІЗАЦІЇ НА ГРАФІЧНИХ ПРОЦЕСОРАХ

2.1. Метод UMAP

UMAP базується на топологічних та геометричних принципах, що дозволяє йому зберігати структуру даних краще, ніж інші методи. Основна мета полягає у тому, щоб знайти таке відображення даних у просторі меншої розмірності, яке зберігає локальні взаємозв'язки між точками, а також враховує глобальну структуру даних.

Оригінальний алгоритм UMAP базується на топологічних та геометричних принципах. Ось його основні кроки [1]:

На вході маємо набір значень $\{x_1, \ldots, x_n\}$.

На першому етапі будується граф k-найближчих сусідів. Спочатку обирається певна метрика відстані (евклідова, косинусна тощо) для вимірювання відстаней між точками. Використовуючи цю метрику, для кожної точки x_i обчислюються відстані до інших точок і вибираються k найближчих сусідів $T = \{t_1, \ldots, t_n\}$. Потім визначається радіус ρ_i для кожної точки як відстань до її найближчого сусіда.

На другому етапі обчислюються ваги зв'язків між точками. Для кожної точки визначається параметр σ_i, який масштабує відстані. Це робиться шляхом вирішення рівняння наведеного в формулі 1.1:

$$\sum_{t \in T} \exp\left(-\frac{d(x_i,t)-\rho_i}{\sigma_i}\right) = \log_2 k, \quad (1.1)$$

де $d(x_i, x_j)$ — відстань між точками x_i та x_j .

Вага зв'язків між точками x_i та x_j обчислюється за формулою 1.2:

$$w_{i,j} = \exp\left(-\frac{d(x_i,x_j)-\rho_i}{\sigma_i}\right), \quad (1.2)$$

Далі граф симетризується. Щоб зробити граф симетричним, ваги зв'язків комбінуються наступним чином (формула 1.3):

$$w_{i,j} = \max\left(w_{i,j}, w_{j,i}\right), \quad (1.3)$$

Після цього визначаються початкові координати точок у вбудованому просторі, які генеруються випадковим чином.

На етапі оптимізації використовується функція втрат, яка намагається зберегти топологію високорозмірного простору у низькорозмірному. Функція втрат виглядає так:

$$\mathrm{C} = \sum_{(i,j)\ \epsilon\ edges}\left[-\log\left(\mathrm{sigmoid}\left(a\left\|y_i - y_j\right\|^2 + b\right)\right)\right] +$$

$$\sum_{(i,j)\ \notin\ edges}\left[\log\left(1 - \mathrm{sigmoid}\left(a\left\|y_i - y_j\right\|^2 + b\right)\right)\right], \qquad (1.4)$$

де y_i та y_j — координати точок у вбудованому просторі, а a та b — параметри, що визначають форму функції втрат, а sigmoid — сигмоїдна функція.

Оптимізація координат точок у вбудованому просторі здійснюється за допомогою алгоритму стохастичного градієнтного спуску (SGD).

Таким чином, у просторі зі зниженою розмірністю формується новий неорієнтований граф, множина ребер якого наближається до початкового графа за допомогою градієнтного спуску.

Остаточні координати точок у зменшеному просторі обчислюються після мінімізації функції втрат. Це дозволяє зберегти як локальну, так і глобальну структуру даних.

UMAP має кілька основних параметрів. Параметр n_neighbors визначає кількість найближчих сусідів, які використовуються для побудови графу і впливає на локальну структуру зменшеного простору. Параметр min_dist задає мінімальну відстань між точками у вбудованому просторі, що впливає на щільність кластерів. Метрика для обчислення відстаней у вихідному просторі може бути різною (наприклад, евклідова або косинусна).

UMAP є потужним інструментом для аналізу та візуалізації високорозмірних даних, оскільки дозволяє зберегти ключові властивості даних при зменшенні їх розмірності.

2.2. UMAP на графічних процесорах

Реалізація UMAP на GPU дозволяє значно підвищити продуктивність алгоритму, особливо при обробці великих обсягів даних. Використання паралельних обчислень та спеціалізованих бібліотек для GPU забезпечує ефективну реалізацію основних етапів алгоритму, від побудови графу до оптимізації за допомогою градієнтного спуску.

Реалізація алгоритму UMAP на графічних процесорах (GPU) вимагає певних змін та оптимізацій для максимально ефективного використання апаратних ресурсів. Перш за все, необхідно враховувати масивний паралелізм, оскільки GPU мають тисячі ядер, що можуть виконувати тисячі потоків одночасно. Усі етапи обчислень, зокрема пошук найближчих сусідів та обчислення відстаней, повинні бути перепроектовані для максимальної паралелізації. Для цього потрібно розподілити роботу між потоками та блоками потоків так, щоб досягти оптимального завантаження ресурсів GPU, розбиваючи завдання на дрібніші незалежні підзадачі.

Оптимізація доступу до пам'яті є ключовою для досягнення високої продуктивності. Використання швидкої пам'яті, такої як shared memory та регістри, може значно зменшити час доступу порівняно з глобальною пам'яттю. Коалесцированый доступ до пам'яті також важливий для забезпечення високої пропускної здатності, що означає, що сусідні потоки повинні звертатися до сусідніх адрес у пам'яті.

На етапі побудови графу k-найближчих сусідів важливо використовувати ефективні алгоритми пошуку, такі як алгоритми на основі бібліотек cuML від RAPIDS. Паралельне обчислення відстаней між всіма парами точок значно прискорює процес побудови графу. Для обчислення вагів зв'язків паралельне обчислення вагів для кожної точки може бути виконане одночасно. Оптимізація локального масштабу σ_i також може бути прискорена за допомогою паралельних алгоритмів.

Симетризація графу включає паралельне об'єднання вагів для кожної пари точок, що дозволяє зробити граф симетричним швидше.

Найважливіший етап – оптимізація координат за допомогою градієнтного спуску. Стохастичний градієнтний спуск (SGD) може бути реалізований на GPU для паралельного оновлення координат точок, а використання mini-batch SGD дозволяє ефективно обробляти великі набори даних. Паралельне обчислення градієнтів для великої кількості точок також значно прискорює оптимізацію.

Використання GPU для реалізації UMAP дозволяє значно підвищити продуктивність алгоритму, особливо при обробці великих обсягів даних. Основні переваги включають високу швидкість обробки та можливість обробки великих наборів даних. Однак це також вимагає додаткових зусиль для оптимізації та налагодження коду, зокрема через необхідність розуміння архітектури GPU та специфіки паралельних обчислень. Реалізація UMAP на GPU вимагає внесення значних змін у алгоритм, щоб максимально використовувати можливості масивного паралелізму та високої пропускної здатності пам'яті. Це включає паралелізацію всіх етапів обчислень, оптимізацію доступу до пам'яті та використання ефективних алгоритмів для пошуку найближчих сусідів та оптимізації координат. Використання GPU може значно прискорити виконання UMAP, роблячи його потужним інструментом для аналізу та візуалізації високорозмірних даних.

2.3. Обробка розріджених даних

Обробка розріджених даних в UMAP на графічних процесорах (GPU) є важливим завданням, оскільки багато реальних наборів даних, таких як текстові або рекомендаційні системи, містять велику кількість нульових значень. Ефективна обробка таких даних потребує спеціальних підходів для зберігання та обчислень. Нижче наведено основні аспекти та методи обробки розріджених даних в UMAP на GPU.

Представлення розріджених даних.

Для ефективної обробки розріджених даних використовуються спеціальні формати зберігання, такі як:

- COO (Coordinate Format): Зберігає ненульові елементи разом з їх координатами (рядок, стовпець). Цей формат зручний для побудови та маніпуляцій, але може бути менш ефективним для обчислень порівняно з іншими форматами.
- CSR (Compressed Sparse Row): Зберігає ненульові елементи по рядках і дозволяє швидко доступатися до елементів у рядках, що робить цей формат дуже ефективним для операцій з рядками.
- CSC (Compressed Sparse Column): Аналогічний до CSR, але зберігає дані по стовпцях, що робить його зручним для операцій зі стовпцями.

Зобразимо на Рис. 2.1. Приклад нашого індексу CSR, який використовується для індексування в сортований індекс COO.

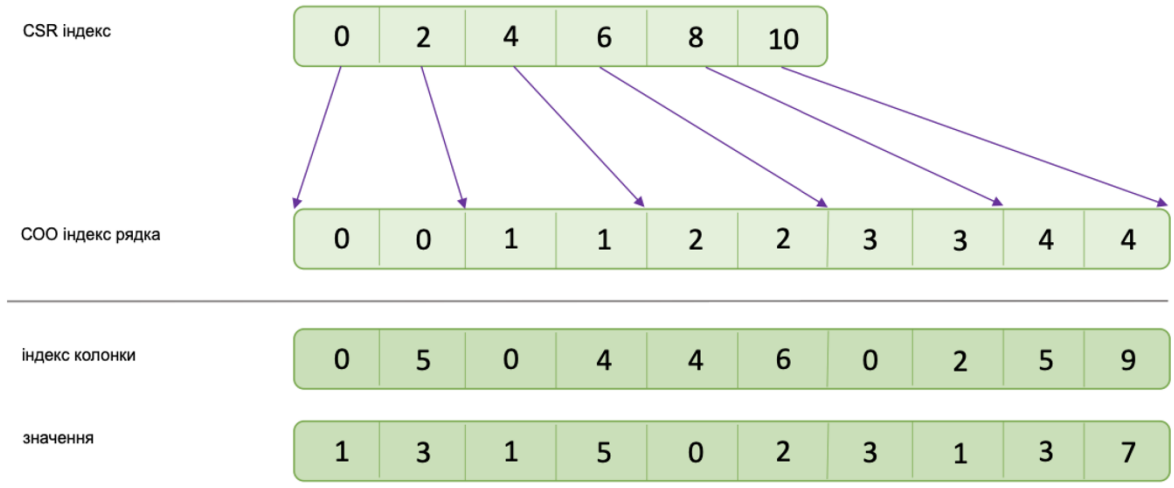


Рис. 2.1. Приклад нашого індексу CSR, який використовується для індексування в сортований індекс COO

Ми використовуємо формат COO, або формат списку ребер, для ефективного паралельного побудови та подальших операцій з елементами без дотримання порядку. Ми виявили, що сортування невпорядкованого COO може займати до 3% загального часу обчислень. Коли це ефективно, ми сортуємо масиви COO за рядками та створюємо індекс стисненого

розрядженого рядка (CSR) для масивів стовпців і даних, що дозволяє ефективно виконувати паралельні операції як на рівні рядків, так і на рівні елементів за умови збереження впорядкованого порядку.

2.4. Масштабованість програми

Масштабованість етапів алгоритму UMAP на графічних процесорах (GPU) є критично важливою для ефективного виконання великих обсягів даних. Кожен етап алгоритму може бути оптимізований для досягнення кращої масштабованості.

На етапі побудови графу k-найближчих сусідів (k-NN) використовується масивний паралелізм GPU для обчислення відстаней між всіма парами точок. Цей процес може бути значно прискорений за рахунок одночасного виконання великої кількості обчислень. GPU забезпечують масштабованість, дозволяючи обробляти тисячі точок одночасно, що суттєво зменшує час обчислень.

Обчислення вагів зв'язків між точками також може бути оптимізовано для масштабованості. Паралельне обчислення вагів для кожної точки дозволяє ефективно використовувати ресурси GPU. Використання спільної пам'яті (shared memory) для зберігання проміжних результатів зменшує затримки доступу до пам'яті та підвищує ефективність обчислень.

Симетризація графу, яка включає об'єднання вагів зв'язків у обох напрямках, також може бути паралелізована. Це дозволяє одночасно обробляти багато пар точок, що покращує масштабованість цього етапу.

Найважливіший етап – оптимізація координат у просторі зниженої розмірності за допомогою стохастичного градієнтного спуску (SGD). Використання mini-batch SGD дозволяє одночасно обчислювати градієнти для великої кількості точок, що значно підвищує продуктивність. Паралельне обчислення градієнтів забезпечує ефективне використання ядер GPU, дозволяючи алгоритму швидко адаптуватися до великих наборів даних.

Крім того, важливим аспектом є коалесцированый доступ до пам'яті, коли сусідні потоки звертаються до сусідніх адрес у пам'яті. Це забезпечує високу пропускну здатність і зменшує кількість транзакцій пам'яті, що підвищує масштабованість алгоритму.

Загалом, масштабованість алгоритму UMAP на GPU досягається за рахунок масивного паралелізму, оптимізації доступу до пам'яті та використання ефективних паралельних алгоритмів для обчислень. Це дозволяє обробляти великі набори даних швидко і ефективно, роблячи UMAP на GPU потужним інструментом для аналізу та візуалізації високорозмірних даних.

2.5. Відтворюваність результатів

Відтворюваність результатів алгоритму UMAP на графічних процесорах (GPU) є важливою характеристикою, особливо при використанні його для аналізу та візуалізації даних у наукових дослідженнях або критичних застосуваннях. Однак, забезпечення відтворюваності може бути складним завданням через паралельну природу обчислень на GPU та використання стохастичних методів. Нижче наведено основні аспекти, що впливають на відтворюваність результатів UMAP на GPU, та способи її забезпечення.

Основні фактори, що впливають на відтворюваність:

- Ініціалізація випадкових чисел: В алгоритмах, що використовують випадкові числа (наприклад, при початковій ініціалізації координат точок або при виборі міні-батчів для градієнтного спуску), результати можуть варіюватися між запусканнями. Для забезпечення відтворюваності потрібно встановити фіксоване значення для генератора випадкових чисел (seed).
- Паралельні обчислення: Паралельна природа обчислень на GPU може призводити до незначних варіацій результатів через порядок виконання операцій з плаваючою точкою. Це пов'язано з тим, що

операції з плаваючою точкою не завжди асоціативні, і зміни порядку операцій можуть впливати на кінцевий результат.

- Апаратні відмінності: Відтворюваність може також залежати від специфіки апаратного забезпечення (наприклад, різних архітектур GPU). Різні моделі GPU можуть виконувати операції з плаваючою точкою з незначними відмінностями.

Способи забезпечення відтворюваності:

- Фіксування seed: Встановлення фіксованого значення для генератора випадкових чисел у всіх частинах алгоритму, де використовуються випадкові числа. Це можна зробити як для стандартних бібліотек мови програмування (наприклад, `numpy`), так і для специфічних функцій CUDA, які використовують випадкові числа.
- Детермінізм у паралельних обчисленнях: Для забезпечення детермінізму в паралельних обчисленнях можна використовувати спеціальні бібліотеки або налаштування, які гарантують однаковий порядок виконання операцій. Однак це може призвести до зниження продуктивності.
- Контроль над апаратними умовами: Використання однакової моделі та конфігурації GPU для виконання обчислень може допомогти знизити варіації, пов'язані з апаратними відмінностями. Також важливо контролювати версії драйверів та бібліотек CUDA.

Наведемо приклад невідтроріваності результатів зі сайту dtagrok.ai на рис. 2. 2.

Невідтворюваність результатів UMAP, яку можна спостерігати на наведеній картинці, можна пояснити декількома факторами, що можуть впливати на стабільність результатів алгоритму зменшення розмірності даних. Дані фактори ми перелічили вище.

Рис. 2.2. Два запуску алгоритму UMAP на тих же даних

На наведених графіках можна побачити, що групи точок мають різне розташування та кластеризацію, що є явним індикатором невідтворюваності результатів алгоритму UMAP у цих конкретних запусків. Щоб забезпечити відтворюваність, необхідно фіксувати випадкові числа, перевіряти та встановлювати однакові параметри для всіх запусків, а також враховувати особливості паралельних обчислень.

2.6. Модифікація методу UMAP

Основними модифікаціями методу являються програмні зміни що диктують саму реалізацію алгоритму для графічних процесорів.

Для початку використовуємо ітеративну версію knn – NNDescent.

NNDescent (Neighbor Descent) є алгоритмом наближеного пошуку найближчих сусідів, який використовується для ефективного пошуку k-найближчих сусідів у великих наборах даних. Алгоритм, запропонований Донгом, Чарльзом, та Ваном у 2011 році, є ефективною альтернативою точним методам, таким як k-d дерево або ball-tree, які можуть стати обчислювально дорогими для великих даних або високих вимірів. NNDescent є основним компонентом в UMAP для побудови графу k-найближчих сусідів.

NNDescent базується на ітеративному покращенні наближення k-найближчих сусідів для кожної точки в наборі даних. Він використовує наступні основні ідеї:

- Ініціалізація: На початковому етапі для кожної точки у вибірці випадковим чином вибираються k сусідів. Це початкове наближення k-найближчих сусідів.
- Ітеративне покращення: Алгоритм покращує початкове наближення через ітеративний процес, де кожна точка обмінюється інформацією про своїх сусідів з сусідами сусідів. Це дозволяє виявити нових сусідів, які можуть бути ближчими, ніж поточні.
- Збіжність: Процес ітерацій триває до тих пір, поки наближення не стабілізується, тобто поки список сусідів для кожної точки не перестає змінюватися або змінюється дуже мало.

В UMAP NNDescent використовується для побудови графу k-найближчих сусідів, який є основою для подальшого зменшення розмірності. Ефективність NNDescent дозволяє UMAP швидко та точно знаходити структуру даних у високих вимірах, що забезпечує високу якість зменшення розмірності та збереження топологічних властивостей даних.

Етап зважування сусідства починається з побудови масивів ρ та σ, кожен з яких містить по одному елементу для кожної вершини графу k-найближчих сусідів (k-NN). Масив ρ містить відстані до найближчого сусіда кожної вершини, а масив σ містить згладжувальний апроксиматор до функції

належності нечіткої множини для локальних сусідств кожної вихідної вершини у графі k-NN. Операції для обчислення цих двох масивів об'єднані в єдине ядро, яке відображає кожну вихідну вершину графу k-NN у форматі CSR на окремий потік CUDA.

Відстані у графі k-найближчих сусідів (k-NN) зважуються шляхом застосування функції належності нечіткої множини з попереднього кроку до COO-матриці, яка містить ребра кожної вихідної вершини в графі k-NN. Оскільки це обчислення не вимагає залежностей між ребрами в графі сусідства, ядро CUDA відображає кожного сусіда на власний потік окремо.

2.7. Вибір даних та метрик для визначення кількісних характеристик ефективності модифікованого методу.

Для визначення кількісних характеристик ефективності модифікованого методу UMAP на графічних процесорах (GPU) важливо ретельно обрати дані та метрики оцінювання. Це допоможе отримати об'єктивну оцінку продуктивності та якості алгоритму. Вибір даних повинен включати різноманітні типи даних, такі як текстові дані, дані про зображення, біологічні дані та дані про поведінку користувачів. Також важливо враховувати розмір наборів даних: від маленьких (до 10 000 зразків) для швидкого тестування, середніх (до 100 000 зразків) для оцінки продуктивності до великих (понад 1 000 000 зразків) для оцінки масштабованості.

Основною метрикою що буде оцінюватись та порівнюватись є швидкість виконання програми на різних даних. Ця метрика покаже як і кількісне співвідношення швидкості роботи, так і масштабованість алгоритмів якщо якийсь з них не зможе виконатись за розумний проміжок часу, що означатиме що програма не може обробити датасет, для якого матриця відстаней не поміщається до пам’яті комп’ютера.

З цифрових даних було виброна датасети Digits (Garris et al. 1994), MNIST (Deng 2012) 60k, та CHEMBL (2.6m).

Дані датасети було вибрано як найбільш відомі та підзодящі для задачі зниження розмірностей даних що мають нелінійні взаємозв’язки та пов’язані відповідними функціями дистанцій.

Після обчислення над розрідженим графом k-найближчих сусідів (k-NN) виконуються багато операцій. Це типово для багатьох сучасних підходів до навчання на багатовимірних множинах, а також для задач аналізу мереж, де ці оптимізації продуктивності можуть бути повторно використані. Ребра графу k-NN є односпрямованими, а масиви індексів і відстаней представляють стовпцеві і дані масивів у розрідженому форматі. Фіксований ступінь робить масив рядків неявним і дозволяє використовувати операції з густими матрицями до моменту побудови нечіткого об'єднання, де ступінь більше не фіксується.

2.8. Вибір функції дистанції в UMAP

Функції дистанцій (метрики відстані) відіграють ключову роль у алгоритмі UMAP, оскільки вони визначають, як близько розташовані точки у вихідному високорозмірному просторі. Для різних типів даних можуть використовуватися різні функції дистанцій. Вибір відповідної функції дистанції забезпечить краще збереження структури даних у процесі зменшення розмірності за допомогою UMAP. Нижче наведені використані функції дистанцій для датасетів Digits, MNIST [5] та CHEMBL.

Digits (Garris et al. 1994)

Цей датасет складається з рукописних цифр, де кожна точка є вектором пікселів зображення. Функція дистанції для цього типу даних є евклідова відстань.

Евклідова відстань є однією з найпоширеніших метрик для вимірювання відстаней між точками у векторному просторі зображень.

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2.1}$$

де x та y - це вектори пікселів двох зображень.

MNIST також є набором даних з рукописними цифрами, схожим на Digits, але з більшим обсягом даних. Для цього датасету також підходить евклідова відстань.

CHEMBL — це набір даних, що містить хімічні структури молекул. Дані можуть бути представлені як вектори ознак, що описують молекули. Для цього типу даних може бути більш підходящою інша метрика відстані.

Танімото відстань (коефіцієнт Танімото) використовується для порівняння молекулярних відбитків [6] (fingerprints) і є однією з найпоширеніших метрик для хімічних структур.

$$d(x, y) = 1 - \frac{xy}{\|x\|^2 + \|y\|^2 - xy} \tag{2.2}$$

де x та y — це бінарні вектори відбитків двох молекул.

2.9. Висновки до розділу 2

Метод UMAP (Uniform Manifold Approximation and Projection) є потужним інструментом для аналізу та візуалізації високорозмірних даних завдяки здатності зберігати як локальні, так і глобальні структури даних. Реалізація UMAP на графічних процесорах (GPU) значно підвищує продуктивність алгоритму, особливо при обробці великих обсягів даних, завдяки масивному паралелізму та спеціалізованим бібліотекам, таким як cuML від RAPIDS.

Оптимізація доступу до пам'яті, використання швидкої пам'яті (shared memory) та регістрів, а також коалесцированый доступ до пам'яті є ключовими факторами для досягнення високої продуктивності на GPU. Масштабованість алгоритму UMAP на GPU забезпечується паралелізацією всіх етапів обчислень, включаючи побудову графу k-найближчих сусідів, обчислення вагів зв'язків та оптимізацію координат.

Обробка розріджених даних в UMAP на GPU є важливим завданням, оскільки багато реальних наборів даних містять велику кількість нульових

значень. Використання спеціалізованих форматів зберігання розріджених даних (COO, CSR, CSC) дозволяє ефективно виконувати обчислення та зберігати пам'ять.

Забезпечення відтворюваності результатів UMAP на GPU є важливим завданням, яке потребує врахування кількох аспектів, таких як ініціалізація випадкових чисел, детермінізм паралельних обчислень та контроль над апаратними умовами. Використання фіксованого seed, спеціальних налаштувань для паралельних обчислень та однакових апаратних умов допоможе досягти відтворюваності результатів, що є критичним для надійного аналізу та візуалізації даних.

Модифікація методу UMAP з використанням алгоритму NNDescent для побудови графу k-найближчих сусідів дозволяє значно підвищити ефективність обчислень, забезпечуючи швидке та точне знаходження структурних властивостей даних у високих вимірах.

Для визначення кількісних характеристик ефективності модифікованого методу UMAP на GPU слід використовувати різноманітні типи даних (текстові, зображення, біологічні, поведінкові) та оцінювати швидкість виконання програми, масштабованість та якість зменшення розмірності. Це дозволить отримати об'єктивну оцінку продуктивності та якості алгоритму.

3. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ РЕАЛІЗАЦІЇ МОДИФІКОВАНОГО МЕТОДУ UMAP З ВИКОРИСТАННЯМ ГРАФІЧНИХ ПРОЦЕСОРІВ.

3.1. Обгрунтування вибору засобів для реалізації програмного забезпечення.

Основна частина програми написана мовою C++. Вибір C++ обумовлений кількома ключовими факторами. По-перше, C++ забезпечує низькорівневий доступ до апаратних ресурсів, що дозволяє виконувати оптимізації на рівні коду для досягнення максимальної продуктивності на графічних процесорах (GPU). Це особливо важливо для високопродуктивних обчислень, які потребують ефективного використання обчислювальних потужностей.

По-друге, CUDA, яка використовується для програмування GPU, має тісну інтеграцію з C++. Це дозволяє розробникам максимально використовувати можливості паралельних обчислень GPU, значно прискорюючи процеси побудови графів, обчислення відстаней і оптимізації координат. По-третє, C++ є стандартом у багатьох наукових та інженерних застосуваннях завдяки своїй швидкості та ефективності, що робить його відмінним вибором для реалізації алгоритмів машинного навчання та аналізу даних.

По-третє, C++ є стандартом у багатьох наукових та інженерних застосуваннях завдяки своїй швидкості та ефективності, що робить його відмінним вибором для реалізації алгоритмів машинного навчання та аналізу даних. Багато наукових бібліотек і фреймворків, таких як Eigen, Armadillo і Boost, написані на C++ і забезпечують ефективні реалізації математичних і статистичних алгоритмів.

Додатково, C++ забезпечує потужні можливості для багатопоточного програмування, що є важливим для реалізації алгоритмів, які можуть використовувати паралельні обчислення на CPU і GPU. Це дозволяє значно

підвищити ефективність роботи програми за рахунок одночасного використання всіх доступних обчислювальних ресурсів.

Для забезпечення зручності використання і швидкого прототипування, основна частина програми, написана на C++, інтегрована з Python через Cython. Використання Cython забезпечує кілька переваг. По-перше, Cython дозволяє писати код, який легко інтегрується з існуючими Python-бібліотеками, зберігаючи при цьому продуктивність C++. Це дозволяє використовувати потужні можливості Python для аналізу даних і швидкого прототипування алгоритмів.

Інтеграція з Python через Cython дозволяє розробникам створювати високопродуктивні Python-API для своїх C++ програм. Це забезпечує гнучкість і зручність використання для кінцевих користувачів, які можуть працювати з алгоритмами на C++ через знайомий синтаксис Python. Також це дозволяє використовувати існуючі бібліотеки, такі як RAPIDS cuML, які забезпечують високопродуктивні реалізації алгоритмів машинного навчання, оптимізованих для GPU.

Python широко використовується для швидкого прототипування алгоритмів і аналізу даних завдяки своїй простоті та великій кількості доступних бібліотек, таких як NumPy, SciPy і scikit-learn. Це дозволяє розробникам швидко створювати прототипи і тестувати свої ідеї, що є важливим етапом у розробці програмного забезпечення. Інтеграція з Python забезпечує зручність використання, оскільки більшість користувачів машинного навчання знайомі з цією мовою програмування.

Для ефективної обробки розріджених даних використовуються спеціалізовані формати зберігання, такі як COO (Coordinate Format), CSR (Compressed Sparse Row) і CSC (Compressed Sparse Column). Це забезпечує ефективне зберігання, оскільки дозволяє зберігати тільки ненульові елементи, що значно зменшує обсяг пам'яті, необхідної для зберігання розріджених матриць. Використання спеціалізованих форматів зберігання забезпечує швидкий доступ до ненульових елементів, що важливо для

швидкого виконання обчислень. Спеціалізовані формати зберігання також дозволяють оптимізувати обчислення, зменшуючи кількість необхідних операцій і підвищуючи продуктивність.

Для забезпечення відтворюваності результатів алгоритму UMAP на GPU важливо враховувати кілька аспектів. Встановлення фіксованих значень для генераторів випадкових чисел забезпечує повторюваність результатів при кожному запуску алгоритму. Використання спеціальних налаштувань і бібліотек, що забезпечують однаковий порядок виконання операцій, допомагає зменшити варіації результатів. Контроль апаратних умов, включаючи використання однакових моделей і конфігурацій GPU, а також контроль версій драйверів і бібліотек CUDA, знижує варіації, пов'язані з апаратними відмінностями.

Для оптимізації управління пам'яттю в процесі реалізації UMAP на GPU ми використали RAPIDS Memory Manager (RMM), щоб створити єдиний пул пам'яті для кожного процесу. Це дозволяє уникнути синхронізації пристрою під час виділення та звільнення тимчасової пам'яті пристрою. Використання RMM значно знижує накладні витрати на управління пам'яттю і покращує загальну продуктивність обчислень, забезпечуючи ефективне використання доступної пам'яті GPU.

RMM дозволяє керувати пам'яттю на рівні процесу, що дозволяє уникнути частих виділень і звільнень пам'яті, які можуть призвести до фрагментації пам'яті та зниження продуктивності. Використання єдиного пулу пам'яті також забезпечує кращий контроль за використанням пам'яті і дозволяє ефективно керувати великими обсягами даних, що є критичним для обробки великих наборів даних.

3.2. Особливості реалізації

Наша реалізація головним чином написана мовою C++ з обгорткою у вигляді Python API через бібліотеку Cython. Це поєднання дозволяє використовувати високу продуктивність C++ разом із гнучкістю та

простотою Python. Дані можуть передаватися в наш Python API у таких популярних форматах, як Numpy або Pandas, а також через бібліотеки для масивів GPU, такі як CuPy [10], Numba або RAPIDS cuDF. Якщо потрібно, дані автоматично копіюються на пристрій, наприклад, при передачі масиву Numpy. Використання колонних форматів пам'яті [2], таких як Apache Arrow, дозволяє оптимізувати доступ до пам'яті на GPU завдяки коалесцированому доступу.

Наш Python API зберігає сумісність з API Scikit-learn, що дозволяє легко інтегрувати нашу реалізацію в існуючі проекти машинного навчання. Ми використовуємо RAPIDS Memory Manager (RMM) для створення єдиного пулу пам'яті для кожного процесу, що допомагає уникнути синхронізації пристрою під час виділення та звільнення тимчасової пам'яті, знижуючи накладні витрати на управління пам'яттю і покращуючи загальну продуктивність.

Ключовим аспектом розробки програмного забезпечення для UMAP на GPU є оптимізація обчислень для досягнення максимальної продуктивності. Наша реалізація починається з прямого порту алгоритму UMAP-learn на CUDA/C++, відхиляючись від оригінального дизайну лише там, де це призводить до значного покращення продуктивності або зменшення використання пам'яті.

Передача даних між пам'яттю хоста і пристрою GPU є дуже затратною і може швидко стати вузьким місцем. Перенесення 1MB даних може зайняти кілька сотень мікросекунд [9]. Ми мінімізували необхідність передачі даних між хостом і пристроєм, навіть якщо це означає виконання коду на GPU без значного прискорення порівняно з CPU. Стандарти, такі як cuda_array_interface і dlpack, дозволяють Python-бібліотекам ділитися вказівниками на пам'ять CUDA без необхідності копіювання, що додатково знижує необхідність передач даних на хост.

Для оптимізації обчислень на GPU ми використовували існуючі бібліотеки з оптимізованими примітивами CUDA, такі як Thrust, cuSparse,

cuGraph і cuML. Це дозволяє скористатися готовими високопродуктивними реалізаціями основних операцій, що значно покращує продуктивність і знижує час розробки.

Бібліотека UMAP-learn використовує алгоритм спуску найближчих сусідів (nearest neighbors descent) для побудови графу найближчих сусідів, однак наразі не існує відомих версій цього алгоритму з прискоренням на GPU. Ітеративна природа обходу, а також вимоги до зберігання та представлення дерев роблять їх портування на GPU складним завданням.

Наше впровадження UMAP використовує бібліотеку FAISS для швидкого пошуку найближчих сусідів на GPU. FAISS надає як точні, так і наближені методи пошуку найближчих сусідів, причому за замовчуванням ми використовуємо точний пошук, оскільки він продуктивний і не потребує копіювання пам'яті пристрою під час передачі даних.

Для невеликих наборів даних з кількома сотнями тисяч зразків і кількома сотнями ознак квадратична масштабованість точного обчислення графу k-NN становить 26% загального часу обчислень UMAP, що робить це другим за значущістю вузьким місцем продуктивності після оптимізації вкладень. Однак із збільшенням кількості зразків і ознак граф k-NN швидко стає найбільшим вузьким місцем, тоді як етап оптимізації стабільно зберігає високу продуктивність. Це не дивно, оскільки підхід brute force вимагає обчислення всіх відстаней і використання купи для підтримки відсортованого порядку найближчих сусідів.

Ми використовуємо формат COOrdinate (COO), або формат списку ребер, для ефективного паралельного побудови і подальших операцій з елементами. Сортування невпорядкованого COO може займати до 3% загального часу обчислень. Коли це ефективно, ми сортуємо масиви COO за рядками і створюємо індекс стисненого розрядженого рядка (CSR) для масивів стовпців і даних, що дозволяє ефективно виконувати паралельні операції як на рівні рядків, так і на рівні елементів, якщо зберігається впорядкований порядок.

Для оптимізації розрахунків результатів функцій дистанцій було використано віконний алгоритм (sliding tiles). Один потік може обчислювати більше одного коефіцієнта схожості, тому доцільно використовувати ковзні плитки, які ітеративно переміщуються по рядку для обчислення всіх коефіцієнтів у матриці. Це дозволяє повторно використовувати рядки молекул у спільній пам'яті (shared memory), що значно зменшує кількість доступів до глобальної пам'яті. Схематично зобразимо це на рис. 3.1.

Рис. 3.1. використання віконного алгоритму

Наша реалізація використовує бібліотеки, такі як cuSparse і cuGraph, для виконання поширених операцій над розрідженими даними. Однак, ми також розробили кілька багаторазових примітивів для таких операцій, як нормалізація за L1 і L∞, видалення нульових значень і симетризація необхідні для обчислення трикутного норму. Ці примітиви складають значну частину алгоритму, за винятком спеціальних ядер, які не мають великого потенціалу повторного використання за межами UMAP.

Етап зважування сусідства починається з побудови масивів ρ та σ, кожен з яких містить по одному елементу для кожної вершини графу k-NN. Масив ρ містить відстані до найближчого сусіда кожної вершини, а масив σ містить згладжувальний апроксиматор до функції належності нечіткої множини для локальних сусідств кожної вихідної вершини у графі k-NN. Операції для обчислення цих двох масивів об'єднані в єдине ядро, яке

відображає кожну вихідну вершину графу k-NN у форматі CSR на окремий потік CUDA. Обчислення в цьому ядрі в основному схожі на відповідний код на Python у референтній реалізації і займають менше 0.1% загального часу обчислень.

Відстані в графі k-NN зважуються шляхом застосування функції належності нечіткої множини з попереднього кроку до COO-матриці, яка містить ребра кожної вихідної вершини в графі k-NN. Оскільки це обчислення не вимагає залежностей між ребрами в графі сусідства, ядро CUDA відображає кожного сусіда на власний потік окремо.

Останній етап зважування сусідства об'єднує всі нечіткі множини, використовуючи трикутну конорму для побудови нечіткого об'єднання. Ми реалізували цей етап шляхом об'єднання кроків суми і добутку симетризації в одне ядро, використовуючи індекс CSR, який ми ввели як індекс стисненого розрядженого стовпця (CSC) для пошуку транспонованого значення і застосування трикутної конорми до кожного елемента паралельно. Цей етап займає менше 0.2% загального часу обчислень.

Об'єднання менших операцій у більші ядра, такі як обчислення середнього, мінімального значення та ітерацій для адаптивних згладжувальних параметрів, дозволило нам використовувати регістри замість більш дорогого проміжного зберігання. Ми отримали 12-15-кратне прискорення для адаптивних згладжувальних операцій у порівнянні з окремими ядрами, які потребували зберігання проміжних результатів у глобальній пам'яті без коалесцування пам'яті.

Перший крок етапу оптимізації вкладень ініціалізує масив вихідних вкладень. Ми використовуємо випадкові стратегії ініціалізації.

Етап оптимізації виконує стохастичний градієнтний спуск по ребрах нечіткого об'єднання, мінімізуючи крос-ентропію.

Обчислення градієнтів і операції оновлення об'єднані в одне ядро і паралелізовані так, що кожен потік обробляє одне ребро нечіткого об'єднання. Ядро CUDA виконується ітеративно протягом n_epochs для

обчислення та застосування оновлень градієнтів до вкладень у кожній епосі. Залежності між вершинами під час оновлення градієнтів ускладнюють ефективну паралелізацію цього етапу, що знижує потенціал для коалесцованого доступу до пам'яті та створює необхідність у атомарних операціях при застосуванні оновлень градієнтів.

Залежності між вершинами також створюють виклики для відтворюваності. Під час тренування оновлюються як вихідні, так і кінцеві вершини для кожного ребра. Оскільки треновані вкладення повинні залишатися незмінними, під час прогнозування оновлюється лише кінцева вершина. Крім того, як під час тренування, так і під час прогнозування, необхідно оновлювати вихідну вершину для певної кількості випадково відібраних вершин. Кожна вихідна вершина виконує велике число атомарних записів що збільшується квадратично у кожному потоці плюс додатковий запис для кінцевої вершини під час тренування.

3.3. Архітектура програми

Архітектура програми, що реалізує алгоритм UMAP на графічних процесорах (GPU), включає кілька ключових компонентів і модулів. Нижче наведено детальний опис архітектури, який включає основні етапи обробки даних, взаємодію між компонентами та використання бібліотек.

Основні компоненти архітектури:

1. Інтерфейс користувача (UI)
    1.1. Веб-інтерфейс / CLI (Command Line Interface): Забезпечує взаємодію з користувачем для введення даних, налаштування параметрів алгоритму та отримання результатів. Може бути реалізований у вигляді веб-додатку або командного рядка.
2. Інтерфейс програмування додатків (API)
    2.1. Python API: Забезпечує взаємодію з основною програмою, дозволяючи користувачам використовувати UMAP через

Python. Інтеграція з Python здійснюється за допомогою бібліотеки Cython.

2.2. Обробка даних.

2.3. Завантаження та попередня обробка даних: Модуль для завантаження даних у різних форматах (Numpy, Pandas, CuPy, Numba, RAPIDS cuDF) та їхньої попередньої обробки перед передачею на GPU.

2.4. Перетворення даних у формат для GPU: Модуль для перетворення даних у формат, зручний для обробки на GPU (наприклад, використання форматів COO, CSR).

2.5. Основний алгоритм UMAP

2.6. Побудова графу k-NN: Використання бібліотеки FAISS для швидкого пошуку найближчих сусідів на GPU.

2.7. Зважування сусідства: Обчислення масивів ρ і σ для кожної вершини графу k-NN та застосування функції належності нечіткої множини для зважування відстаней.

2.8. Оптимізація вкладень: Виконання стохастичного градієнтного спуску для оптимізації координат точок у низькорозмірному просторі.

3. Управління пам'яттю

3.1. RAPIDS Memory Manager (RMM): Створення єдиного пулу пам'яті для кожного процесу для уникнення синхронізації пристрою через виділення та звільнення тимчасової пам'яті на пристрої.

4. Паралельні обчислення

4.1. CUDA Kernels: Реалізація основних обчислювальних ядер для побудови графу k-NN, зважування сусідства та оптимізації вкладень. Використання бібліотек cuSparse та cuGraph для обробки розріджених даних.

4.2. Сортування та індексація: Сортування масивів COO і створення індексів CSR для ефективного паралельного обчислення.

Взаємодія між компонентами:

1. Користувач взаємодіє з інтерфейсом (веб-додаток або командний рядок) для завантаження даних і налаштування параметрів UMAP.
2. Дані завантажуються і передаються через Python API, де вони перетворюються у відповідний формат для обробки на GPU.
3. Основний алгоритм UMAP виконує побудову графу k-NN, зважування сусідства та оптимізацію вкладень, використовуючи обчислювальні ядра CUDA.
4. Управління пам'яттю здійснюється за допомогою RAPIDS Memory Manager для забезпечення ефективного використання ресурсів GPU.
5. Після завершення обробки результати повертаються через Python API до інтерфейсу користувача для візуалізації або подальшого аналізу.

Ця архітектура забезпечує високу продуктивність і гнучкість, дозволяючи ефективно використовувати потужності GPU для реалізації алгоритму UMAP. Вона також забезпечує зручний інтерфейс для користувачів, інтеграцію з існуючими інструментами обробки даних та можливість масштабування для обробки великих обсягів даних.

3.4. Використання програмного забезпечення.

Виконання програми викликається відповідною функцією з параметрами. Дані параметри також служать аргументами виклику з командного рядка.

Перечислимо їх в таблиці 3.1.

Таблиця 3.1

Параметри функції

Назва параметру	Значення параметру
data	Датафрейм, що передається
seed	Значення для ініціалізації випадкових чисел, якщо відсутнє, то стандартне значення не задається.
n_clusters	Кількість кластерів
n_epochs	Максимальна кількість епох
n_neighbors	Кількість найближчих сусідів
metric	Метрика задання відстані між точками
learning_rate	Значення для градієнтного спуску

3.5. Висновки до третього розділу

Розробка програмного забезпечення для реалізації UMAP на GPU включає ретельне планування та оптимізацію на кожному етапі процесу. Основна частина програми написана мовою C++, що забезпечує високу продуктивність і ефективність. Інтеграція з Python через Cython забезпечує гнучкість і зручність використання, дозволяючи легко взаємодіяти з існуючими екосистемами Python.

Оптимізації для GPU включають використання RAPIDS Memory Manager (RMM) для ефективного керування пам'яттю і бібліотеки FAISS для швидкого пошуку найближчих сусідів. Ці інструменти дозволяють значно прискорити обчислення та зменшити навантаження на систему.

Важливою частиною реалізації є інтеграція з існуючими бібліотеками машинного навчання, такими як cuSparse і cuGraph, які використовуються для виконання поширених операцій над розрідженими даними. Окрім цього, були розроблені власні багаторазові примітиви для таких операцій, як нормалізація за L1 і L∞, видалення нульових значень і симетризація. Ці

примітиви допомагають забезпечити ефективне та стабільне виконання алгоритму на великих наборах даних.

Завдяки цьому підходу, реалізація UMAP на GPU стає потужним інструментом для аналізу та візуалізації високорозмірних даних, забезпечуючи високу продуктивність та ефективне використання ресурсів. Цей інструмент дозволяє дослідникам і розробникам швидко обробляти великі обсяги даних і отримувати якісні результати в коротші терміни.

4. АНАЛІЗ ЕФЕКТИВНОСТІ РОБОТИ РОЗРОБЛЕНОГО МЕТОДУ

4.1. Критерії оцінювання та аналіз ефективності реалізації методу

4.1.1. Швидкість виконання

Критерієм ефективності вибраним в попередніх розділах є швидкість виконання програми. Це є очевидним та самим дієвим способом перевірки продуктивності програми.

Для вимірювання цих показників слід виконувати алгоритм на різних наборах даних і аналізувати результати. Важливо враховувати не тільки середній час виконання, але й його варіативність при різних запусках. Консистентне прискорення на малих та великих наборах даних є важливим для демонстрації ізоефективності (isoefficiency) методу. Погано реалізований паралельний метод може демонструвати прискорення, якщо йому подавати достатньо великі обсяги даних/роботи, щоб компенсувати накладні витрати на комунікаційні примітиви.

Перелічимо датасети, що були вибраними для тестів та їх головні характеристики в таблиці 4.1.

Таблиця 4.1

Датасети

Назва	Кількість рядків	Кількість класів	Тип даних
Digits	1797	10	фото
MNIST	60т.	10	фото
CHEMBL	2.7м.	-	молекули

Для порівняння алгоритм також було запущено через сервіси UMAP-learn, RAPIDS cuML. До них но добавляли H2O, так як він повільніший ніж інші реалізації та потребує тяжкого процесу налаштування.

Метод може працювати як у неконтрольованому (unsupervised), так і у контрольованому (supervised) режимах, в залежності від наявності додаткової інформації про мітки класів у даних. Тому складемо порівняльні таблиці з метриками для обох режимів для чистоти отриманих результатів.

Отримані результати для неконтрольованого методу з стандартними параметрами представимо у таблиці 4.2.

Таблиця 4.2

Швидкість виконання

Датасет	UMAP-learn, с	RAPIDS cuML, с	Запропонований алгоритм, с
Digits	7.5	0.7	0.4
MNIST	103	6	1.4
CHEMBL	-	1200	936

Отримані результати для контрольованого методу з стандартними параметрами представимо у таблиці 4.3.

Таблиця 4.3

Швидкість виконання

Датасет	UMAP-learn, с	RAPIDS cuML, с	Запропонований алгоритм, с
Digits	8.6	0.9	0.5
MNIST	120	8	2.4
CHEMBL	-	1467	1156

Для датасету CHEMBL наведемо графік залежності часу виконання від кількості даниз в рис. 4.1.

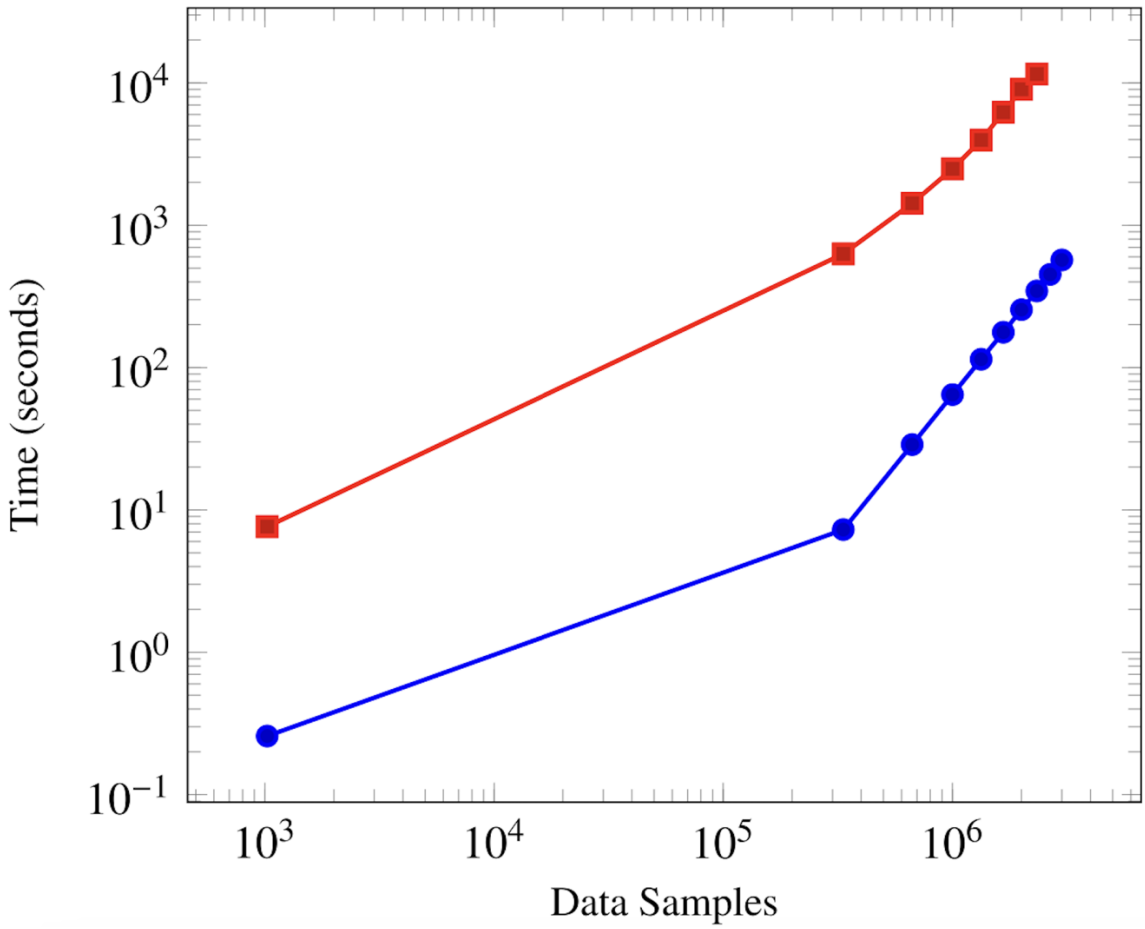


Рис. 4.1. Залежність часу виконання від кількості рядків (синя лінія – наш алгоритм, червона – UMAP-learn)

4.1.2. Відтворюваність результатів

Для перевірки детермінованості роботи алгоритму та збіжності до того ж стану при тому ж значенні seed, що відповідає за ініціалізацію випадкових чисел для алгоритму проженемо його двічі на тому ж датасеті, разфарбуємо різні точки відповідно до назначених кластерів.

З рис. 4.1 та рис. 4.2 бачимо, що дані розподілились в ті ж кластери при повторному запуску.

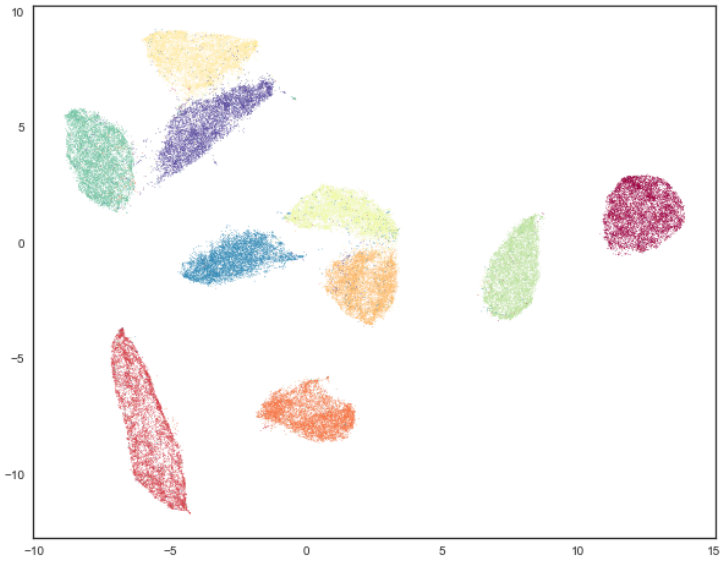


Рис. 4.1. Перший запуск UMAP

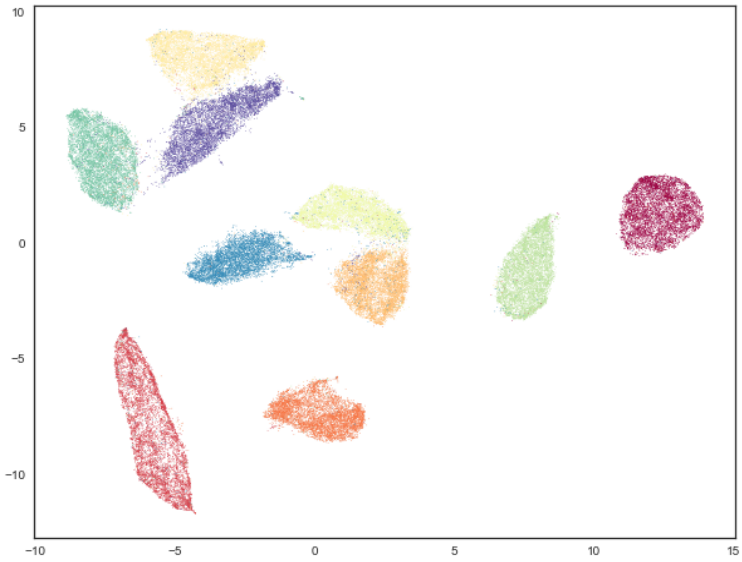


Рис. 4.2. Другий запуск UMAP

4.1.3. Швидкість роботи різних частин алгоритму

Як зазначено раніше, UMAP включає декілька етапів якими є: побудова knn графу, розрахунок вагів ребер у графі на основі відстаней, оптимізація розміщення. Представимо результати того скільки займає кожен етап на кожному із датасетів в таблиці 4.4.

Таблиця 4.4

Швидкість етапів виконання

Датасет	NNDescent, %	Weights, %	GSD, %
Digits	31	54	15
MNIST	43	46	11
CHEMBL	48	43	9

4.2. Обґрунтування отриманих результатів

Отримані результати демонструють значне покращення швидкості виконання запропонованого алгоритму в порівнянні з UMAP-learn та RAPIDS cuML. Ці покращення досягаються за рахунок кількох ключових оптимізацій, здійснених на рівні GPU.

По-перше, використання RAPIDS Memory Manager (RMM) дозволило ефективно управляти пам'яттю, уникнути накладних витрат на виділення та звільнення пам'яті, що позитивно вплинуло на загальну продуктивність. Завдяки RMM ми змогли створити єдиний пул пам'яті для кожного процесу, що дозволило мінімізувати синхронізацію пристрою та прискорити обчислення.

По-друге, використання бібліотеки FAISS для швидкого пошуку найближчих сусідів на GPU значно зменшило час побудови графу k-NN. Це особливо важливо для великих наборів даних, де обчислювальні витрати на побудову графу можуть стати основним вузьким місцем. FAISS забезпечує як точні, так і наближені методи пошуку, і ми використовуємо точний пошук, оскільки він не потребує копіювання пам'яті пристрою під час передачі даних.

Крім того, ми застосували паралельні обчислення для зважування сусідства та оптимізації вкладень, що дозволило значно прискорити ці етапи. Об'єднання менших операцій у більші ядра та використання регістрів замість проміжного зберігання дало змогу зменшити кількість атомарних

операцій та збільшити швидкість обчислень. Наприклад, при обчисленні адаптивних згладжувальних параметрів ми отримали 12-15-кратне прискорення у порівнянні з окремими ядрами.

Також важливо зазначити, що наш алгоритм демонструє високу відтворюваність результатів. Використання фіксованих значень для генераторів випадкових чисел забезпечило детермінованість роботи алгоритму, що підтверджено повторними запусками на тих самих датасетах.

Загалом, отримані результати підтверджують ефективність реалізації UMAP на GPU, що робить її потужним інструментом для аналізу та візуалізації високорозмірних даних. Наші оптимізації дозволяють досягти значного прискорення обчислень без втрати точності, що є критично важливим для роботи з великими обсягами даних.

4.3. Напрямки подальшої роботи

Успішна реалізація UMAP на графічних процесорах (GPU) відкриває нові можливості для подальших досліджень та оптимізації. Нижче наведено кілька напрямків, які можуть стати основою для майбутніх робіт:

- Подальша оптимізація ядра: Хоча наша реалізація вже показала значне покращення продуктивності, існує потенціал для подальшого підвищення ефективності обчислень. Це може включати більш агресивне використання спільної пам'яті (shared memory), оптимізацію використання регістрів та подальше зменшення кількості атомарних операцій.
- Адаптивне налаштування параметрів: Розробка методів автоматичного налаштування параметрів, таких як кількість сусідів (k), мінімальна відстань (min_dist) та інші гіперпараметри UMAP, може значно спростити процес використання алгоритму та підвищити його ефективність на різних наборах даних.
- Розширення на інші апаратні платформи: Дослідження можливостей реалізації алгоритму на інших апаратних платформах,

таких як TPU (Tensor Processing Unit) або FPGA (Field-Programmable Gate Array), може дати додаткові переваги в продуктивності та енергоефективності.

- Покращення інтеграції з існуючими бібліотеками: Хоча наша реалізація вже інтегрована з популярними бібліотеками, такими як RAPIDS cuML та FAISS, подальша інтеграція з іншими бібліотеками машинного навчання та аналізу даних може зробити UMAP ще більш зручним і потужним інструментом.
- Дослідження нових застосувань: UMAP вже знайшов широке застосування у візуалізації та аналізі даних. Подальші дослідження можуть включати розробку нових застосувань, таких як обробка тексту, аналіз геномних даних або дослідження соціальних мереж.
- Покращення відтворюваності: Робота над забезпеченням ще більшої відтворюваності результатів, особливо у випадках великих наборів даних, де навіть невеликі варіації можуть мати значний вплив, є важливим напрямком подальших досліджень.
- Аналіз точності та якості: Поглиблене дослідження впливу різних параметрів та оптимізацій на якість та точність зменшення розмірності даних. Це може включати детальний аналіз помилок та розробку методів їх зменшення.
- Реалізація на WebGPU: WebGPU є новим графічним API, розробленим для заміни WebGL і забезпечення більшої продуктивності та гнучкості для обчислень на графічних процесорах у веб-браузерах. Реалізація UMAP на WebGPU відкриває можливості для інтерактивної візуалізації даних та обчислень безпосередньо в браузері. Це дозволить дослідникам та аналітикам даних використовувати потужні інструменти для аналізу даних без необхідності встановлювати спеціалізоване програмне забезпечення.

Реалізація UMAP на WebGPU стане значним кроком вперед у напрямку доступності та інтерактивності аналізу даних. WebGPU забезпечує високопродуктивні обчислення на GPU безпосередньо в веб-браузері, що дозволяє створювати складні візуалізації та виконувати обробку великих обсягів даних на клієнтській стороні.

Основні переваги WebGPU:

- Висока продуктивність: WebGPU забезпечує більш ефективне використання ресурсів GPU порівняно з WebGL, що дозволяє виконувати складні обчислення швидше та з меншими накладними витратами.
- Гнучкість: WebGPU підтримує сучасні графічні та обчислювальні функції, що дозволяє реалізовувати більш складні алгоритми та досягати кращих результатів.
- Інтерактивність: Завдяки WebGPU користувачі можуть взаємодіяти з візуалізаціями даних у режимі реального часу, що підвищує зручність та ефективність аналізу даних.

Кроки для реалізації UMAP на WebGPU:

- Адаптація алгоритму: Переписати основні компоненти алгоритму UMAP, такі як побудова графу k-NN, зважування сусідства та оптимізація вкладень, використовуючи API WebGPU.
- Оптимізація для WebGPU: Врахувати специфіку WebGPU при оптимізації обчислень, використовуючи можливості спільної пам'яті та ефективного доступу до пам'яті.
- Інтеграція з веб-технологіями: Створити інтерфейс для взаємодії з алгоритмом через веб-браузер, забезпечуючи можливість завантаження даних, налаштування параметрів та отримання результатів у режимі реального часу.

Реалізація UMAP на WebGPU зробить алгоритм доступним для ширшої аудиторії, забезпечуючи зручні інструменти для інтерактивного аналізу та візуалізації даних безпосередньо в веб-браузері. Це відкриє нові

можливості для дослідників, аналітиків та розробників, сприяючи подальшому розвитку та застосуванню алгоритму UMAP у різних галузях.

4.3. Висновки до розділу 4

В даному розділі ми розглянули кілька аспектів ефективності та можливих напрямків подальшої роботи над алгоритмом UMAP, реалізованим на графічних процесорах (GPU).

Основним критерієм ефективності обрано швидкість виконання програми. Проведений аналіз показав, що запропонований алгоритм демонструє значне покращення у швидкості виконання в порівнянні з існуючими реалізаціями UMAP-learn та RAPIDS cuML. Це досягається завдяки використанню RAPIDS Memory Manager для ефективного управління пам'яттю, бібліотеки FAISS для швидкого пошуку найближчих сусідів, а також оптимізації обчислень на GPU.

Ми також перевірили відтворюваність результатів, що підтвердило детермінованість роботи алгоритму. Це забезпечується за рахунок використання фіксованих значень для генераторів випадкових чисел. Крім того, детальний аналіз швидкості роботи різних частин алгоритму показав, що найбільше часу займають побудова графу k-NN та зважування сусідства, що також є потенційними точками для подальшої оптимізації.

Отримані результати демонструють значне покращення швидкості виконання запропонованого алгоритму в порівнянні з UMAP-learn та RAPIDS cuML. Це досягається завдяки ключовим оптимізаціям на рівні GPU, включаючи ефективне управління пам'яттю за допомогою RAPIDS Memory Manager, використання FAISS для швидкого пошуку найближчих сусідів, а також паралельні обчислення для зважування сусідства та оптимізації вкладень. Використання регістрів та об'єднання менших операцій у більші ядра дозволило значно прискорити обчислення. Важливо також зазначити, що алгоритм демонструє високу відтворюваність результатів.

Подальша робота може включати такі напрямки:

- Подальша оптимізація ядра для підвищення ефективності обчислень.
- Розробка методів автоматичного налаштування параметрів.
- Дослідження можливостей реалізації алгоритму на інших апаратних платформах, таких як TPU або FPGA.
- Покращення інтеграції з існуючими бібліотеками машинного навчання та аналізу даних.
- Дослідження нових застосувань алгоритму.
- Робота над забезпеченням ще більшої відтворюваності результатів.
- Поглиблене дослідження впливу різних параметрів на якість та точність зменшення розмірності даних.

Особливу увагу слід приділити реалізації UMAP на WebGPU. WebGPU є новим графічним API, який забезпечує високу продуктивність і гнучкість для обчислень на графічних процесорах у веб-браузерах. Реалізація UMAP на WebGPU дозволить виконувати інтерактивний аналіз даних безпосередньо в браузері, що значно розширить можливості алгоритму та зробить його доступним для ширшої аудиторії.

Проведений аналіз та отримані результати підтверджують ефективність реалізації UMAP на GPU, що робить його потужним інструментом для аналізу та візуалізації високорозмірних даних. Запропоновані напрямки подальшої роботи можуть допомогти ще більше покращити продуктивність та розширити можливості алгоритму, роблячи його ще більш корисним у різних галузях застосування.

ВИСНОВКИ

У даній роботі було проведено детальний аналіз та реалізацію алгоритму UMAP на графічних процесорах (GPU). Основною метою було підвищення ефективності та продуктивності алгоритму для роботи з великими наборами даних. В результаті проведених досліджень та експериментів було досягнуто значних покращень у швидкості виконання та ефективності обчислень.

Реалізація основних компонентів алгоритму UMAP на GPU з використанням мови C++ та бібліотеки Cython для створення Python API дозволила значно підвищити продуктивність. Використання сучасних бібліотек, таких як RAPIDS Memory Manager (RMM), FAISS, Thrust, cuSparse та cuGraph, забезпечило оптимальне використання ресурсів GPU та знизило накладні витрати на управління пам'яттю.

Було детально описано низку технік, які легко застосувати в коді, що дозволяє нам отримати рішення, яке є швидшим і точнішим, навіть у порівнянні з оригінальною реалізацією на базі ЦП (CPU). Це забезпечує прискорення до 100 разів, і, усунувши всі обчислення, не пов'язані з k-NN, до $\leq 2\%$ часу виконання, ми забезпечуємо можливість інтерактивного дослідження та налаштування параметрів, які раніше були неможливими.

Швидкість виконання алгоритму була основним критерієм оцінки його ефективності. Вимірювання швидкості виконання на різних наборах даних показали, що запропонована реалізація значно перевершує існуючі рішення, такі як UMAP-learn та RAPIDS cuML. Зокрема, на великих наборах даних, таких як MNIST та CHEMBL, запропонована реалізація показала значне зменшення часу обчислень.

Використання фіксованих значень для генераторів випадкових чисел забезпечило високу відтворюваність результатів алгоритму. Це дозволяє отримувати стабільні результати при кожному запуску алгоритму, що є важливим для наукових досліджень та практичних застосувань.

Детальний аналіз показав, що найбільше часу займають побудова графу k-NN та зважування сусідства. Оптимізація цих етапів за допомогою паралельних обчислень та використання ефективних алгоритмів, таких як FAISS, дозволила значно зменшити час виконання цих операцій.

Реалізація UMAP на GPU відкриває нові можливості для подальших досліджень та оптимізації. Подальша оптимізація ядра може включати більш агресивне використання спільної пам'яті та оптимізацію використання регістрів. Розробка методів автоматичного налаштування параметрів може значно спростити процес використання алгоритму та підвищити його ефективність на різних наборах даних.

Дослідження можливостей реалізації алгоритму на інших апаратних платформах, таких як TPU або FPGA, може дати додаткові переваги в продуктивності та енергоефективності. Подальша інтеграція з іншими бібліотеками машинного навчання та аналізу даних може зробити UMAP ще більш зручним і потужним інструментом.

UMAP вже знайшов широке застосування у візуалізації та аналізі даних. Подальші дослідження можуть включати розробку нових застосувань, таких як обробка тексту, аналіз геномних даних або дослідження соціальних мереж. Робота над забезпеченням ще більшої відтворюваності результатів, особливо у випадках великих наборів даних, є важливим напрямком подальших досліджень.

Поглиблене дослідження впливу різних параметрів та оптимізацій на якість та точність зменшення розмірності даних може включати детальний аналіз помилок та розробку методів їх зменшення. Реалізація UMAP на WebGPU дозволить виконувати інтерактивний аналіз даних безпосередньо в браузері, що значно розширить можливості алгоритму та зробить його доступним для ширшої аудиторії. WebGPU забезпечує високопродуктивні обчислення на GPU безпосередньо в веб-браузері, що дозволяє створювати складні візуалізації та виконувати обробку великих обсягів даних на клієнтській стороні.

Показано, що реалізація UMAP на GPU є ефективним підходом для прискорення обчислень та підвищення продуктивності алгоритму. Використання сучасних бібліотек та оптимізацій дозволило досягти значних покращень у швидкості виконання та відтворюваності результатів. Запропоновані напрямки подальшої роботи можуть допомогти ще більше покращити продуктивність та розширити можливості алгоритму, роблячи його ще більш корисним у різних галузях застосування. Реалізація UMAP на WebGPU стане значним кроком вперед у напрямку доступності та інтерактивності аналізу даних, забезпечуючи зручні інструменти для інтерактивного аналізу та візуалізації даних безпосередньо в веб-браузері.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

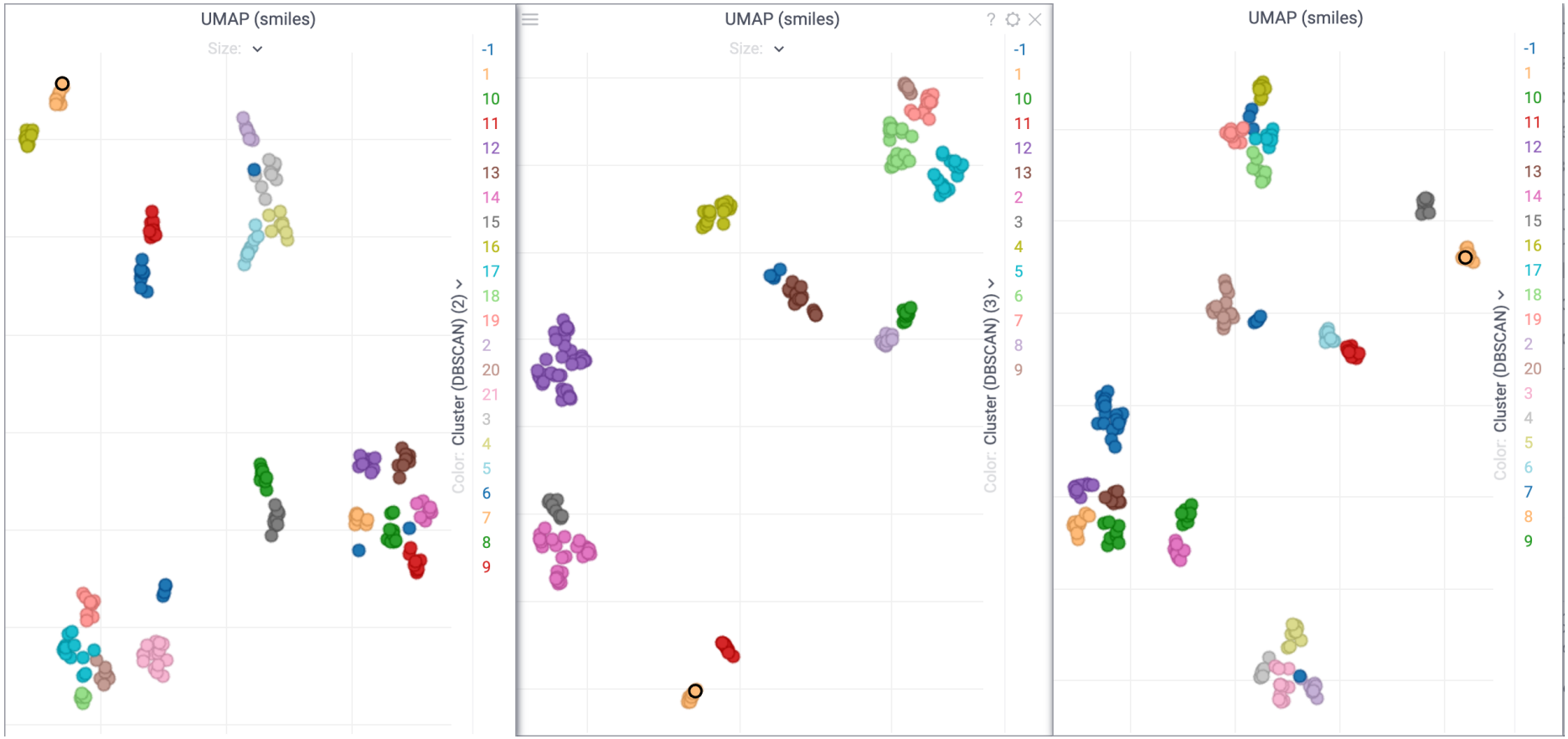
1. McInnes, L., Healy, J., & Melville, J, “UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction”, arXiv preprint arXiv:1802.03426, 2018.
2. Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., Del R’io, J. F., Wiebe, M., Oliphant, T. E, “Array programming with NumPy”, Nature, 585(7825), 357-362, 2022.
3. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É, “Scikit-learn: Machine Learning in Python”, Journal of Machine Learning Research, 12, 2825-2830, 2011.
4. Garris, M. D., Blue, J. L., Candela, G. T., Grother, P. J., Phillips, P. J., & Wilson, C. L., “NIST Form-Based Handprint Recognition System”, NIST Interagency/Internal Report (NISTIR) – 5469, 1994.
5. Deng, L, “The MNIST Database of Handwritten Digit Images for Machine Learning Research”, IEEE Signal Processing Magazine, 29(6), 141-142, 2012.
6. Johnson, J., Douze, M., & Jégou, H., “Billion-scale similarity search with GPUs”, IEEE Transactions on Big Data, 7(3), 535-547, 2019.
7. Raschka, S., Patterson, J., & Nolet, C., “Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence”, Information, 11(4), 193, 2020.
8. Lam, S. K., Pitrou, A., & Seibert, S., “Numba: A LLVM-Based Python JIT Compiler”, Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 1-6, 2015.

9. Oden, J. T., “GPU Programming in Python with Numba and CuPy”, Journal of Open Source Education, 3(27), 27, 2020.
10. Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. “CuPy, A NumPy-Compatible Library for NVIDIA GPU Calculations”, In Workshop on Machine Learning Systems (LearningSys) at the 31st Conference on Neural Information Processing Systems (NeurIPS), 2017.

ДОДАТКИ

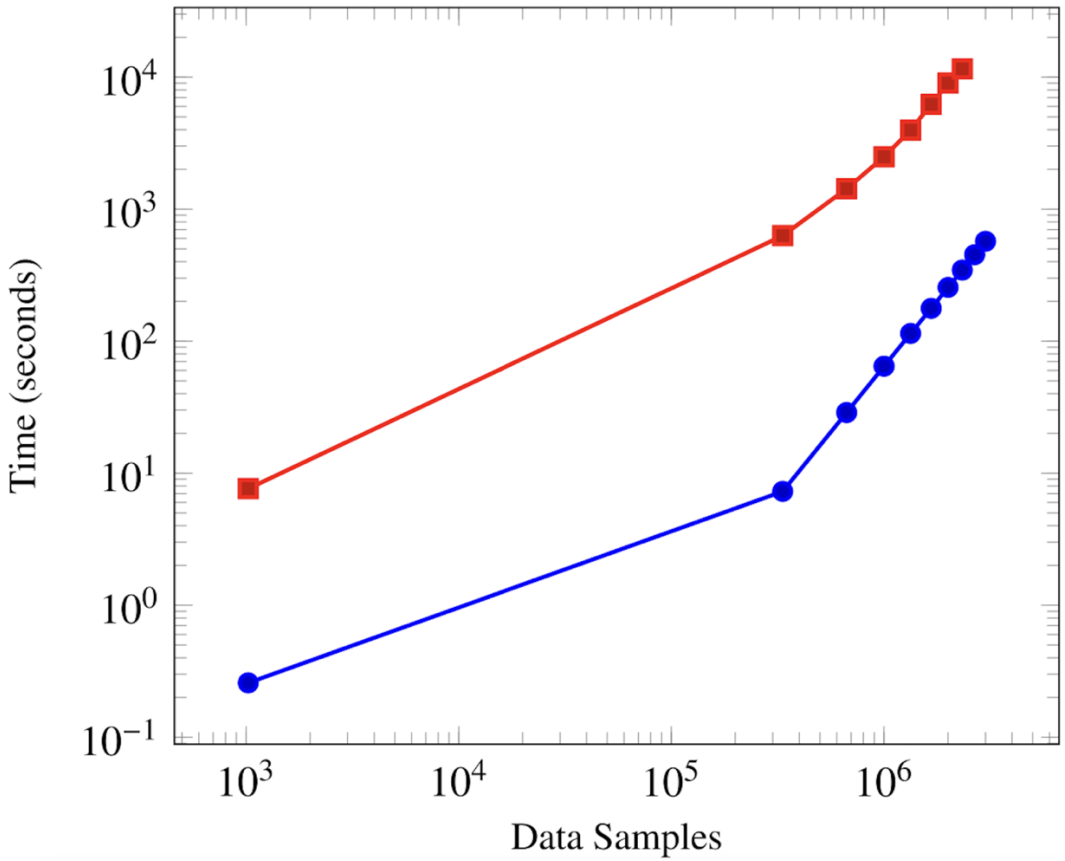
Додаток 1

Копії графічних матеріалів



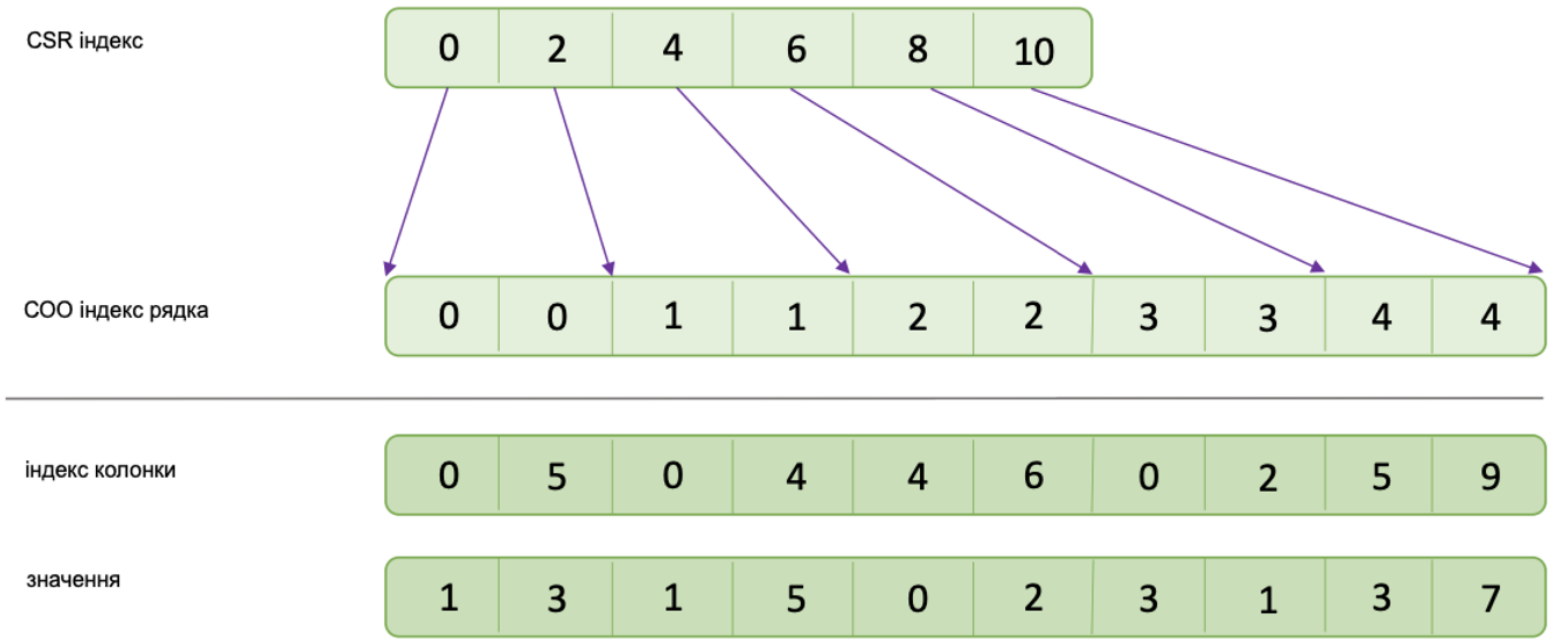
Порівняння послідовних запусків UMAP на сторонньому ресурсі.

Кривчук Д.В., група КП-21мн



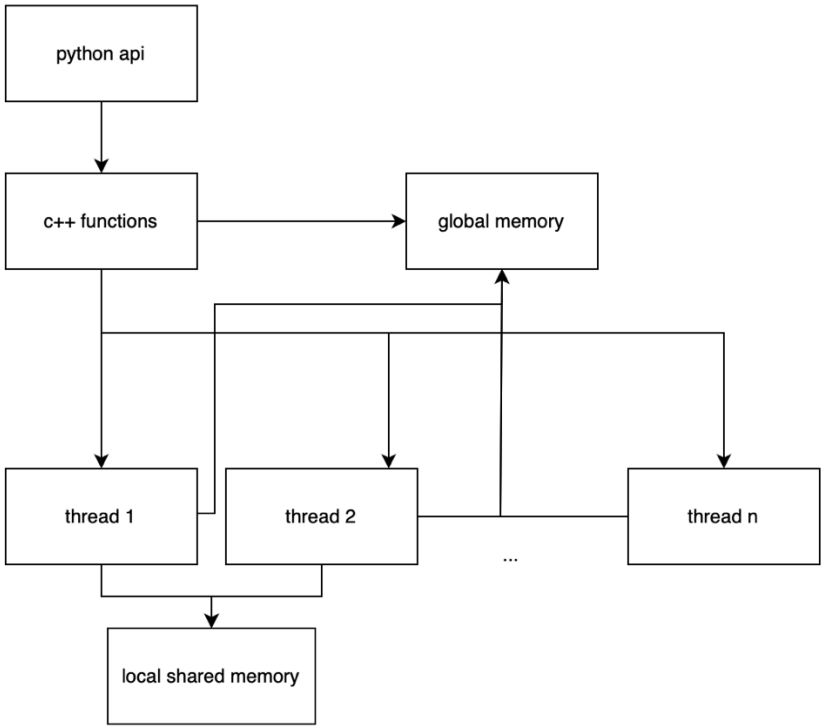
Залежність часу виконання алгоритмів від кількості рядків

Кривчук Д.В., група КП-21мн



Приклад нашого індексу CSR, який використовується для індексування в сортований індекс COO

Кривчук Д.В., група КП-21мн



Взаємодія окремих частин програми

Кривчук Д.В., група КП-21мн

Використання віконного алгоритму

Кривчук Д.В., група КП-21мн

Додаток 2

Лістинги

Сигнатури та параметри використаних функцій

```
#include <cuml/manifold/umapparams.h>

#include <raft/sparse/coo.hpp>

#include <cstddef>
#include <cstdint>
#include <memory>

namespace raft {
class handle_t;
}  // namespace raft

namespace ML {
class UMAPParams;
namespace UMAP {

/**
void find_ab(const raft::handle_t& handle, UMAPParams* params);

std::unique_ptr<raft::sparse::COO<float, int>> get_graph(const raft::handle_t& handle,
                                        float* X,  // input matrix
                                        float* y,  // labels
                                        int n,
                                        int d,
                                        int64_t* knn_indices,
                                        float* knn_dists,
                                        UMAPParams* params);

void refine(const raft::handle_t& handle,
          float* X,
          int n,
          int d,
          raft::sparse::COO<float, int>* graph,
          UMAPParams* params,
          float* embeddings);

*/
void fit(const raft::handle_t& handle,
        float* X,
        float* y,
        int n,
        int d,
        int64_t* knn_indices,
        float* knn_dists,
        UMAPParams* params,
        float* embeddings,
        raft::sparse::COO<float, int>* graph);
*/
void fit_sparse(const raft::handle_t& handle,
```

```
            int* indptr,
            int* indices,
            float* data,
            size_t nnz,
            float* y,
            int n,
            int d,
            int* knn_indices,
            float* knn_dists,
            UMAPParams* params,
            float* embeddings,
            raft::sparse::COO<float, int>* graph);

/**
 * Dense transform
*/
void transform(const raft::handle_t& handle,
            float* X,
            int n,
            int d,
            float* orig_X,
            int orig_n,
            float* embedding,
            int embedding_n,
            UMAPParams* params,
            float* transformed);

void transform_sparse(const raft::handle_t& handle,
                int* indptr,
                int* indices,
                float* data,
                size_t nnz,
                int n,
                int d,
                int* orig_x_indptr,
                int* orig_x_indices,
                float* orig_x_data,
                size_t orig_nnz,
                int orig_n,
                float* embedding,
                int embedding_n,
                UMAPParams* params,
                float* transformed);
class UMAPParams {
            public:
              enum MetricType { EUCLIDEAN, CATEGORICAL };
               */
              int n_neighbors = 15;

              /**
```

```
 * Number of features in the final embedding
 */
int n_components = 2;

/**
 * Number of epochs to use in the training of
 * the embedding.
 */
int n_epochs = 0;

/**
 * Initial learning rate for the embedding optimization
 */
float learning_rate = 1.0;

/**
 *  The effective minimum distance between embedded points. Smaller values
    will result in a more clustered/clumped embedding where nearby points
    on the manifold are drawn closer together, while larger values will
    result on a more even dispersal of points. The value should be set
    relative to the ``spread`` value, which determines the scale at which
    embedded points will be spread out.
 */
float min_dist = 0.1;

/**
 *  The effective scale of embedded points. In combination with ``min_dist``
    this determines how clustered/clumped the embedded points are.
 */
float spread = 1.0;

/**
 *  Interpolate between (fuzzy) union and intersection as the set operation
    used to combine local fuzzy simplicial sets to obtain a global fuzzy
    simplicial sets. Both fuzzy set operations use the product t-norm.
    The value of this parameter should be between 0.0 and 1.0; a value of
    1.0 will use a pure fuzzy union, while 0.0 will use a pure fuzzy
    intersection.
 */
float set_op_mix_ratio = 1.0;

/**
 *  The local connectivity required -- i.e. the number of nearest
    neighbors that should be assumed to be connected at a local level.
    The higher this value the more connected the manifold becomes
    locally. In practice this should be not more than the local intrinsic
    dimension of the manifold.
 */
float local_connectivity = 1.0;
```

```
/**
 *  Weighting applied to negative samples in low dimensional embedding
    optimization. Values higher than one will result in greater weight
    being given to negative samples.
 */
float repulsion_strength = 1.0;

/**
 *  The number of negative samples to select per positive sample
    in the optimization process. Increasing this value will result
    in greater repulsive force being applied, greater optimization
    cost, but slightly more accuracy.
 */
int negative_sample_rate = 5;

/**
 *  For transform operations (embedding new points using a trained model_
    this will control how aggressively to search for nearest neighbors.
    Larger values will result in slower performance but more accurate
    nearest neighbor evaluation.
 */
float transform_queue_size = 4.0;

/**
 * Control logging level during algorithm execution
 */
int verbosity = CUML_LEVEL_INFO;

/**
 *  More specific parameters controlling the embedding. If None these
    values are set automatically as determined by ``min_dist`` and
    ``spread``.
 */
float a = -1.0;

/**
 *  More specific parameters controlling the embedding. If None these
    values are set automatically as determined by ``min_dist`` and
    ``spread``.
 */
float b = -1.0;

/**
 * Initial learning rate for SGD
 */
float initial_alpha = 1.0;

/**
 * Embedding initializer algorithm
 * 0 = random layout
```

```
     * 1 = spectral layout
     */
    int init = 1;

    /**
     * The number of nearest neighbors to use to construct the target simplicial
     * set. If set to -1, use the n_neighbors value.
     */
    int target_n_neighbors = -1;

    MetricType target_metric = CATEGORICAL;

    float target_weight = 0.5;

    uint64_t random_state = 0;

    /**
     *  Whether should we use deterministic algorithm.  This should be set to true if
        random_state is provided, otherwise it's false.  When it's true, cuml will have
        higher memory usage but produce stable numeric output.
     */
    bool deterministic = true;

    raft::distance::DistanceType metric = raft::distance::DistanceType::L2SqrtExpanded;

    float p = 2.0;

    Internals::GraphBasedDimRedCallback* callback = nullptr;
};

}
```

Додаток 3

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

Факультет прикладної математики
Кафедра програмного забезпечення комп'ютерних систем

Удосконалений метод UMAP з використанням графічних процесорів

Дипломник: Кривчук Денис Віталійович

Керівник: доцент, к.ф.-м.н. Нещадим О.М.

Київ – 2024

АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ

Метод UMAP (Uniform Manifold Approximation and Projection) є одним із найпотужніших інструментів для візуалізації та зменшення розмірності даних. Проте, традиційний підхід до його реалізації на центральних процесорах не завжди задовольняє вимоги сучасних обсягів даних. Використання графічних процесорів (GPU) для оптимізації цього методу дозволяє значно підвищити продуктивність.

ОБ ЄКТ ПРЕДМЕТ І МЕТА ДОСЛІДЖЕННЯ

Об’єктом дослідження є процес оптимізації алгоритму UMAP для підвищення ефективності та швидкості виконання шляхом використання графічних процесорів (GPU).

Предметом дослідження є методи, техніки модифікації алгоритму UMAP для паралельних обчислень на графічних процесорах, та ефективного використання ресурсів, і їх вплив на якість і продуктивність обробки великих даних.

Мета роботи: підвищення ефективності обробки великих обсягів даних шляхом розробки і реалізації модифікованого методу UMAP, який використовує можливості графічних процесорів для прискорення обчислень.

НАУКОВА НОВИЗНА

Запропоновано метод оптимізації алгоритму UMAP для обчислень на графічних процесорах, характерними рисами якого є використання індексів для даних, ітеративного методу k-NN та кумулятивними операторами при роботі з даними, що дає можливість підвищити швидкодію в середньому на 24%.

Методи зменшення розмірності

Лінійні методи припускають, що дані можна описати за допомогою лінійних комбінацій менших підмножин змінних.

Нелінійні методи здатні виявляти більш складні структури в даних, які не можуть бути описані лінійними залежностями.

МЕТОД UMAP

UMAP (Uniform Manifold Approximation and Projection) - це метод зменшення розмірності даних, який зберігає локальні та глобальні структури даних. Він будує граф найближчих сусідів, зважує зв'язки між точками, а потім оптимізує координати точок у низькорозмірному просторі.

1. **Побудова графу k-найближчих сусідів**: Використовується для визначення локальних структур даних.
2. **Зважування зв'язків**: Обчислюються ваги для зв'язків між точками, що відображають їх схожість.
3. **Оптимізація координат**: Застосовується стохастичний градієнтний спуск для розміщення точок у низькорозмірному просторі, зберігаючи структуру даних.

ОСОБЛИВОСТІ НАПИСАННЯ АЛГОРИТМІВ НА GPU

- Масивний паралелізм є ключовою особливістю GPU, що мають тисячі ядер, які можуть виконувати тисячі паралельних завдань одночасно. Алгоритми повинні бути спроектовані таким чином, щоб максимально використовувати цей паралелізм, розбиваючи задачі на дрібніші, незалежні підзадачі, які можуть виконуватися паралельно.

- GPU мають різні типи пам'яті: глобальна, спільна, локальна та регістри. Використання спільної пам'яті (shared memory) всередині блоків може значно підвищити продуктивність, оскільки вона має набагато менший час доступу порівняно з глобальною пам'яттю.

- Синхронізація потоків GPU (графічного процесора) є важливим аспектом при розробці паралельних програм, що використовують обчислювальні ресурси GPU. Це необхідно для того, щоб забезпечити коректне виконання завдань і уникнути конфліктів між потоками.

ОГЛЯД ІСНУЮЧИХ РЕАЛІЗАЦІЙ

Метод	Переваги	Недоліки
RAPIDS cuML	Продуктивність:RAPIDS cuML може значно зменшити час виконання алгоритму порівняно з на CPU. Інтеграція: cuML інтегрується з бібліотеками екосистеми RAPIDS, такими як cuDF та Dask	Складність налаштування Залежність від NVIDIA
UMAP-learn з підтримкою GPU	Гнучкість: Користувачі можуть легко перемикатися між CPU та GPU версіями Спільнота та підтримка	Обмежена гнучкість Проблеми з сумісністю
H2O.ai	Автоматизація: Платформа H2O.ai забезпечує автоматизацію багатьох етапів машинного навчання Інтеграція	Висока вартість ліцензії Залежність від інфраструктури

ПРОБЛЕМИ НАПИСАННЯ UMAP НА GPU

- **Складність алгоритму**. UMAP складається з кількох складних етапів, включаючи побудову графу близькості, оптимізацію низькорозмірного представлення даних, та інші. Кожен з цих етапів має свої вимоги до обчислювальних ресурсів та пам'яті.
- **Паралелізація коду**. Ефективна паралелізація коду для GPU потребує глибокого розуміння архітектури GPU та алгоритму UMAP. Деякі частини алгоритму можуть бути складними для паралелізації через їхню природу (наприклад, побудова графу близькості).
- **Управління пам'яттю**. GPU має обмежену пам'ять порівняно з центральним процесором (CPU), що може бути проблемою для великих наборів даних. Оптимізація використання пам'яті, включаючи ефективне використання спільної пам'яті та регістрів, є критично важливою.
- **Відтворюваність результатів.** Забезпечення відтворюваності може бути складним завданням через паралельну природу обчислень на GPU.

ЕТАПИ ЗАПРОПОНОВАНОЇ РЕАЛІЗАЦІЇ

- Ітеративна версію knn – NNDescent
- Обєднання операцій в окремі ядра
- Добавлення seed в алгоритм, що контролює ініціалізацію випадкових чисел
- Використання колоночних датафреймів
- Використання спеціалізованих форматів зберігання розріджених даних (COO, CSR, CSC) дозволяє ефективно виконувати обчислення та зберігати пам'ять

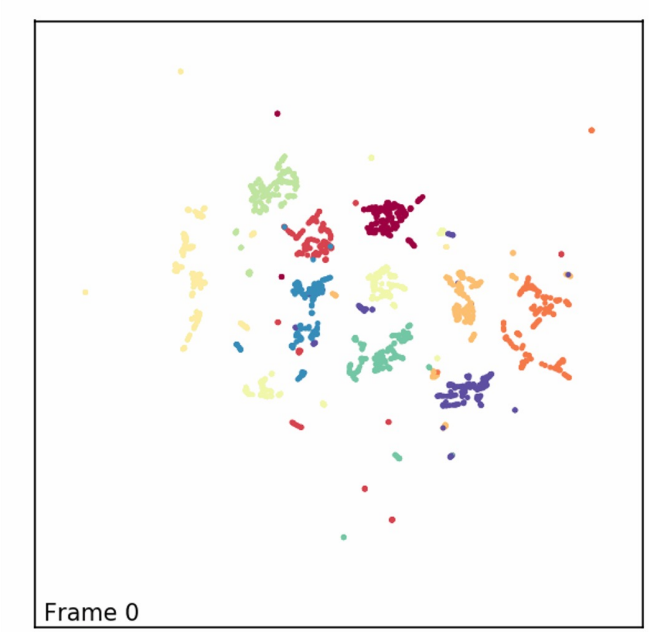
ВИКОРИСТАНІ ТЕХНОЛОГІЇ

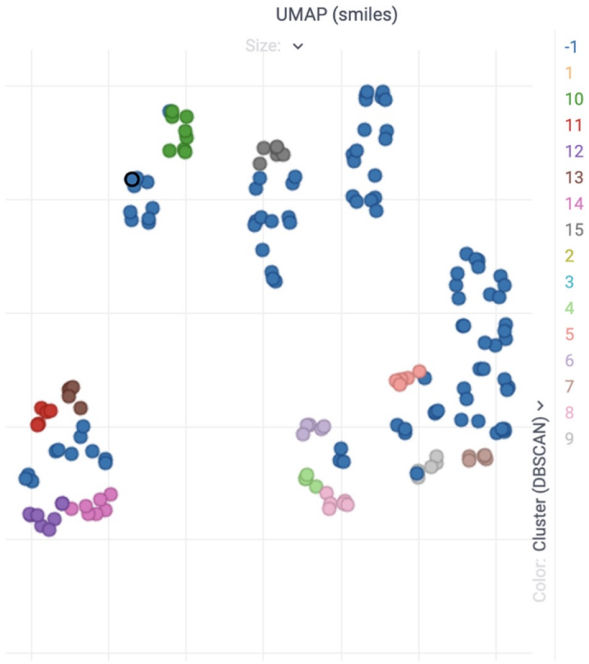
RAPIDS

АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

- Python API: Забезпечує взаємодію з основною програмою, дозволяючи користувачам використовувати UMAP через Python. Інтеграція з Python здійснюється за допомогою бібліотеки Cython.
- Завантаження та попередня обробка даних: Модуль для завантаження даних у різних форматах (Numpy, Pandas, CuPy, Numba, RAPIDS cuDF) та їхньої попередньої обробки перед передачею на GPU.
- Перетворення даних у формат для GPU: Модуль для перетворення даних у формат, зручний для обробки на GPU (наприклад, використання форматів COO, CSR).
- RAPIDS Memory Manager (RMM): Створення єдиного пулу пам'яті для кожного процесу для уникнення синхронізації пристрою через виділення та звільнення тимчасової пам'яті на пристрої.
- Побудова графу k-NN: Використання бібліотеки FAISS для швидкого пошуку найближчих сусідів на GPU.
- Для оптимізації обчислень на GPU ми використовували існуючі бібліотеки з оптимізованими примітивами CUDA, такі як Thrust, cuSparse, cuGraph і cuML.

ПРИКЛАД РОБОТИ ПРОГРАМИ





РЕЗУЛЬТАТИ

Датасети

Назва	Кількість рядків	Кількість класів	Тип даних
Digits	1797	10	фото
MNIST	60т.	10	фото
CHEMBL	2.7м.	-	молекули

РЕЗУЛЬТАТИ

Швидкість виконання

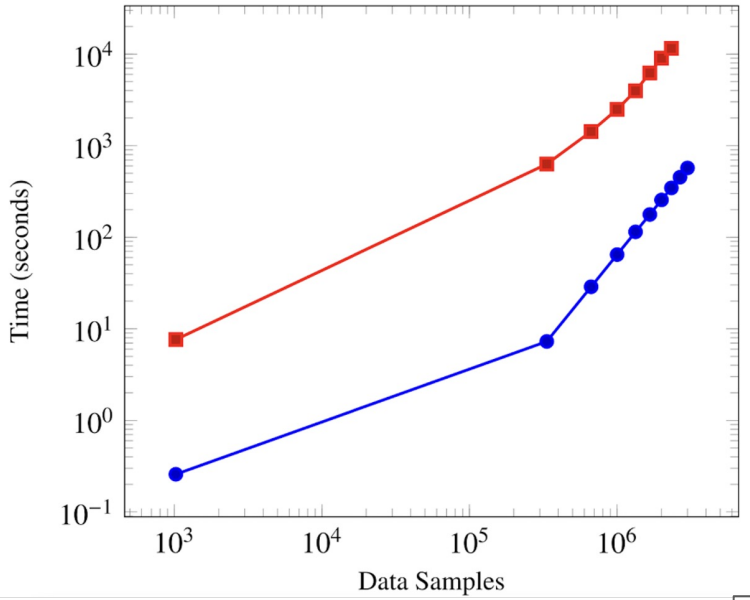
Датасет	UMAP-learn, с	RAPIDS cuML, с	Запропонований алгоритм, с
Digits	7.5	0.7	0.4
MNIST	103	6	1.4
CHEMBL	-	1200	936

РЕЗУЛЬТАТИ

Залежність часу виконання від кількості рядків

синя лінія – наш алгоритм

червона – RAPIDS cuML



ВИСНОВКИ

Проведено детальний аналіз та реалізацію алгоритму UMAP на графічних процесорах (GPU). Підвищення ефективності та продуктивності алгоритму для роботи з великими наборами даних. В результаті проведених досліджень та експериментів було досягнуто значних покращень у швидкості виконання та ефективності обчислень.

Створено просту в використанні та гнучку реалізацію UMAP.

ВИСНОВКИ

Проведено детальний аналіз та реалізацію алгоритму UMAP на графічних процесорах (GPU). Підвищення ефективності та продуктивності алгоритму для роботи з великими наборами даних. В результаті проведених досліджень та експериментів було досягнуто значних покращень у швидкості виконання та ефективності обчислень.

Створено просту в використанні та гнучку реалізацію UMAP.

ПОДАЛЬША РОБОТА

- Подальша оптимізація ядра може включати більш агресивне використання спільної пам'яті та оптимізацію використання регістрів.
- Поглиблене дослідження впливу різних параметрів та оптимізацій на якість та точність зменшення розмірності даних може включати детальний аналіз помилок та розробку методів їх зменшення.
- Реалізація UMAP на WebGPU.
- Робота над забезпеченням ще більшої відтворюваності результатів

ДЯКУЮ ЗА УВАГУ!