Національний університет
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра мультимедійних систем

Аналіз та обробка даних з використанням графічних процесорів

Курсова робота
Студентки III року навчання
Спеціальності 122 Комп’ютерні науки

Стешенко Катерини Сергіївни

Науковий керівник –
кандидат фізико-математичних наук, доцент
Жежерун О. П.

Київ – 2023

ЗМІСТ

ВСТУП

Актуальність. Щодня програмним забезпеченням оброблюється та генерується величезна кількість інформації. Беззаперечним фактом є те, що більшість процесів будуть швидше виконуватись паралельно, якщо розподілити задачу між множиною обчислювальних одиниць. Розвиток науки призвів до появи апаратної та програмної архітектури, що уможливлює подібну обробку. На початку ери розподілених обчислень, графічні процесори використовувались переважно для обробки матриць (графіки), проте з розвитком ефективності апаратного забезпечення було виявлено ряд інших задач, де доцільно застосувати дані технології. [1]

Мета дослідження – проаналізувати ідею підвищення ефективності обчислення за допомогою графічних процесорів, дослідити особливості їх архітектури та технологію Nvidia CUDA.

Завдання дослідження – розглянути різноманітні методи підвищення ефективності обробки та добування даних за допомогою графічних процесорів, зосередившись на особливостях їх архітектури та використанні технології Nvidia CUDA. Практичний результат дослідження полягатиме у розробці методів використання графічного процесора для покращення ефективності обробки даних та виявлення потенційних напрямів подальшого дослідження у цій галузі.

Об’єкт дослідження – сучасні технології для паралельної обробки даних з використанням графічних процесорів

Предмет дослідження – є методи впровадження швидкої обробки інформації з використанням графічного процесора.

Джерела дослідження – електронні версії наукових видань, програмна документація, записи виступів з конференцій тощо.

РОЗДІЛ 1. ПРИНЦИПИ РОБОТИ ГРАФІЧНИХ ПРОЦЕСОРІВ

1.1 Обмеження багатоядерних обчислень

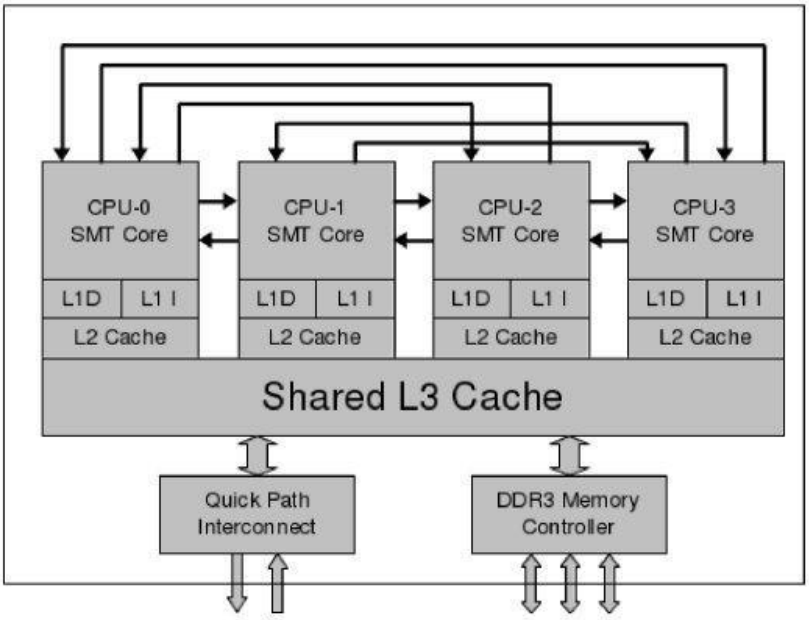
В другій половині XX століття Джин Амдал описав власний закон для особливого випадку паралельного використання n процесорів (ядер), коли він доводив достовірність однопроцесорного підходу для досягнення великомасштабних обчислювальних можливостей. Науковець зазначив, що частина програми, що не піддається виконанню в багатопоточній середі, вагомо впливатиме на кінцевий результат, тож збільшення кількості ядер значно не вплине на прискорення.

$$Speedup = \frac{1}{(1-p) + p/N}$$

Аргументи Амдала на користь одноядерних процесорів були виправдані, і одно або декілька поточний підходи домінували в обчислювальному просторі. Тому, входячи в багатоядерну еру, важливо застосовувати добре розпаралелювані методи й алгоритми. [2]

1.2. Архітектура багатоядерних центральних процесорів

В сучасному світі існує багато видів багатоядерних систем. Для базового розуміння відмінностей між центральними та графічними модулями, які часто використовуються в сучасних обчислювальних машинах, розглянемо схему перших процесорів Intel Core i7.



Процесор Core i7 містить чотири ядра (останні моделі вже є 16-ти ядерними, а Xeon w9 – 56)

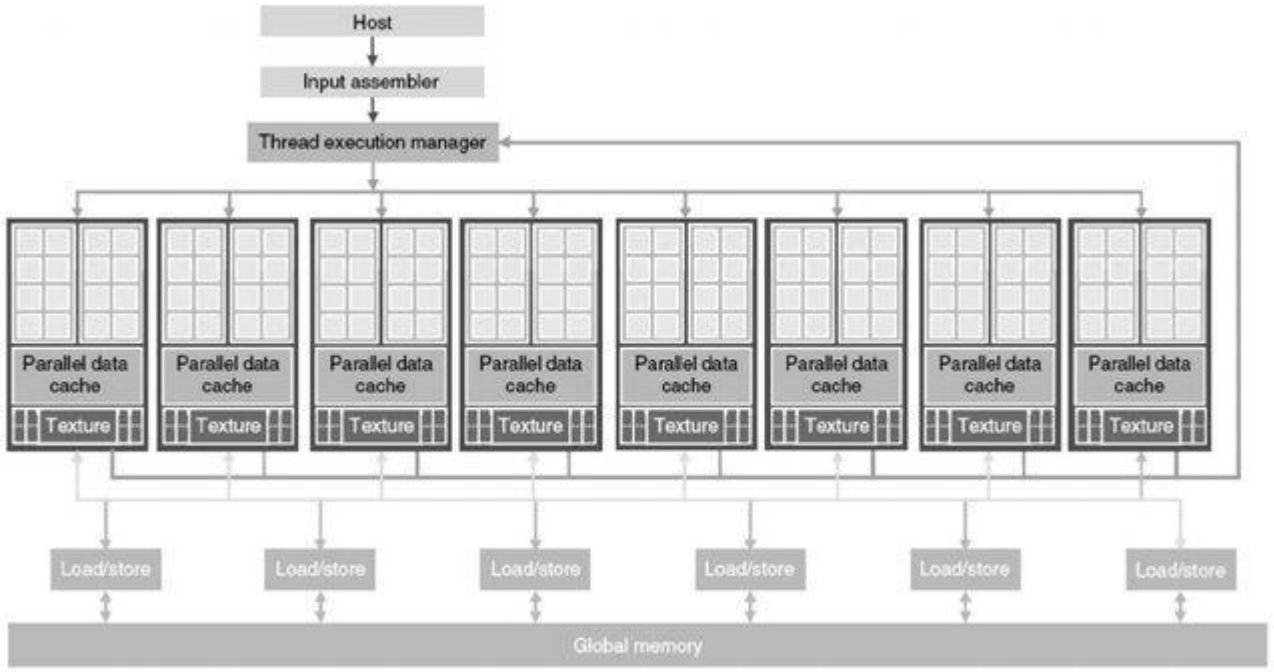
Кожне з цих ядер має свої L1-кеш для даних і L1-кеш для команд і L2-кеш. Також, є спільний для всіх ядер L3-кеш.

*Рис. SEQ Рис._ * ARABIC 1.1 Архітектура 4-ядерного ЦП*

Архітектуру CPU (central processing unit) можна охарактеризувати таким чином: декілька процесорів, що вміють виконувати арифметико-логічні операції, спільний кеш та модуль керування пам’яттю.

1.3. Архітектура графічних процесорів

Розглянемо будову GPU (graphical processing unit) на прикладі архітектури Nvidia Tesla 8. Такий апарат містить набагато більше обчислювальних одиниць, кеш та модулі управління для блоків обчислювальних пристроїв.



Варто додати, що комп’ютерна шина в GPU є в декілька разів більшою ніж у CPU. Це дозволяє контролеру направляти дані в обчислювальні пристрої, з тією швидкістю, з якою вони можуть їх обробляти.

1.3. Особливості взаємодії між GPU та CPU

Враховуючи архітектурні особливості пристроїв, треба виділити важливі моменти які впливають на програмування за допомогою них:

1. Графічний процесор – це периферійний пристрій, який вміє швидко виконувати прості арифметико-логічні операції в багато потоків.

2. Для програмування на GPU потрібні спеціальні технології, які «запускаються» з центрального процесора (Обчислення загального призначення на графічних процесорах або GPGPU – далі)

3. CPU та GPU з’єднані між собою комп’ютерною шиною (bus), що передає інформацію з значно меншою швидкістю, ніж шини в самих процесорах. Тому, потрібно обмежити обмін даними між процесорами, щоб забезпечити базову ефективність програм. [3]

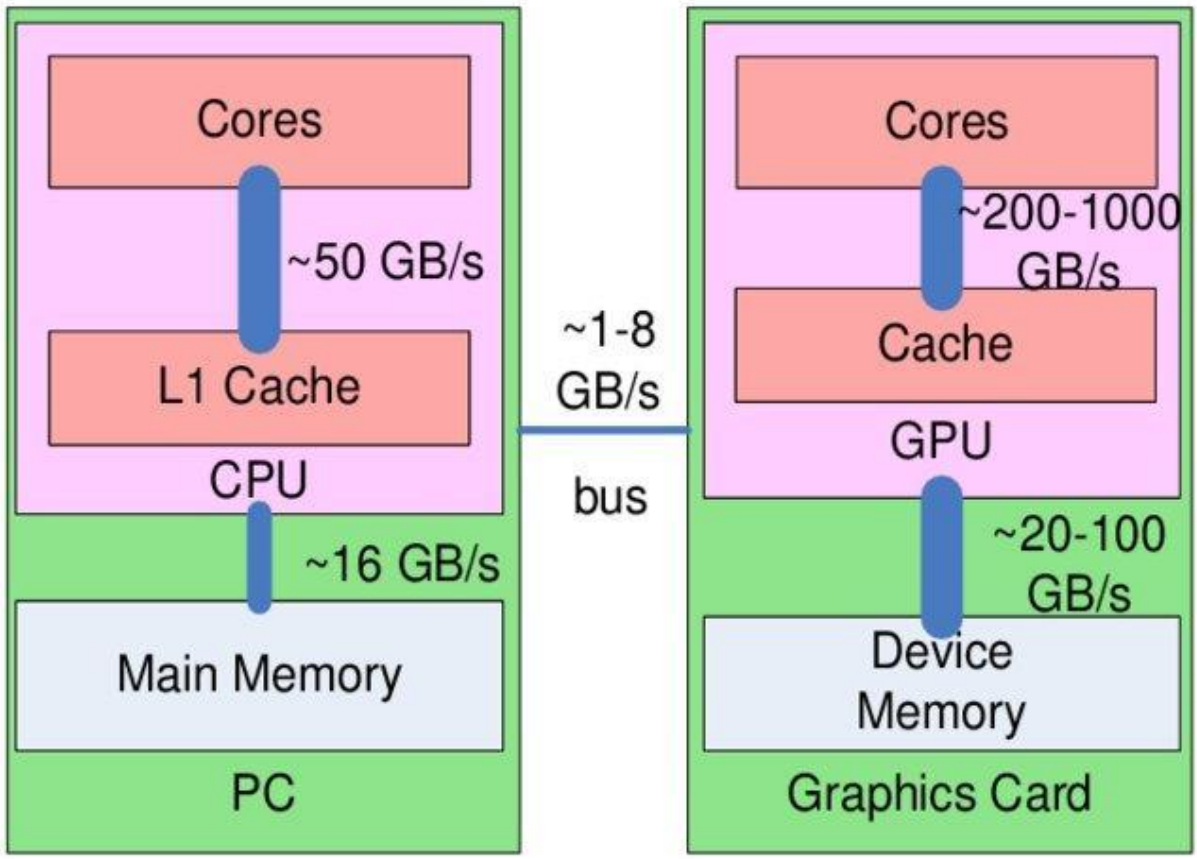


Рис. 1.3. Ілюстрація швидкостей передачі даних в шинах

РОЗДІЛ 2. ТЕХНОЛОГІЇ ОБЧИСЛЕННЯ ЗАГАЛЬНОГО ПРИЗНАЧЕННЯ НА ГРАФІЧНИХ ПРОЦЕСОРАХ

2.1. Реалізації GPGPU

OpenCL — це фреймворк для створення програм з паралельною обробкою на GPU та CPU, розроблений спільно багатьма компаніями та корпораціями: Apple, AMD, Intel, Nvidia та ін. Платформа підтримується більшістю операційних систем та апаратним забезпеченням різних виробників, проте має вужчий функціонал в порівнянні з CUDA [4]

DirectCompute — прикладний програмний інтерфейс, який дозволяє робити обчислення на відеокартах на операційних системах сімейства Microsoft Windows. Даний інтерфейс є частиною DirectX, і підтримується починаючи з 10 версії DirectX. Використовується переважно у розробці ігор для поліпшення рендеринга, штучного інтелекту, освітлення і фізики.

C++ AMP — бібліотека на основі DirectX 11 розроблена компанією Microsoft для паралельного програмування для гетерогенних систем на сучасному C++. Бібліотека має на меті дозволити розробникам розподілити виконання програм між CPU та GPU.

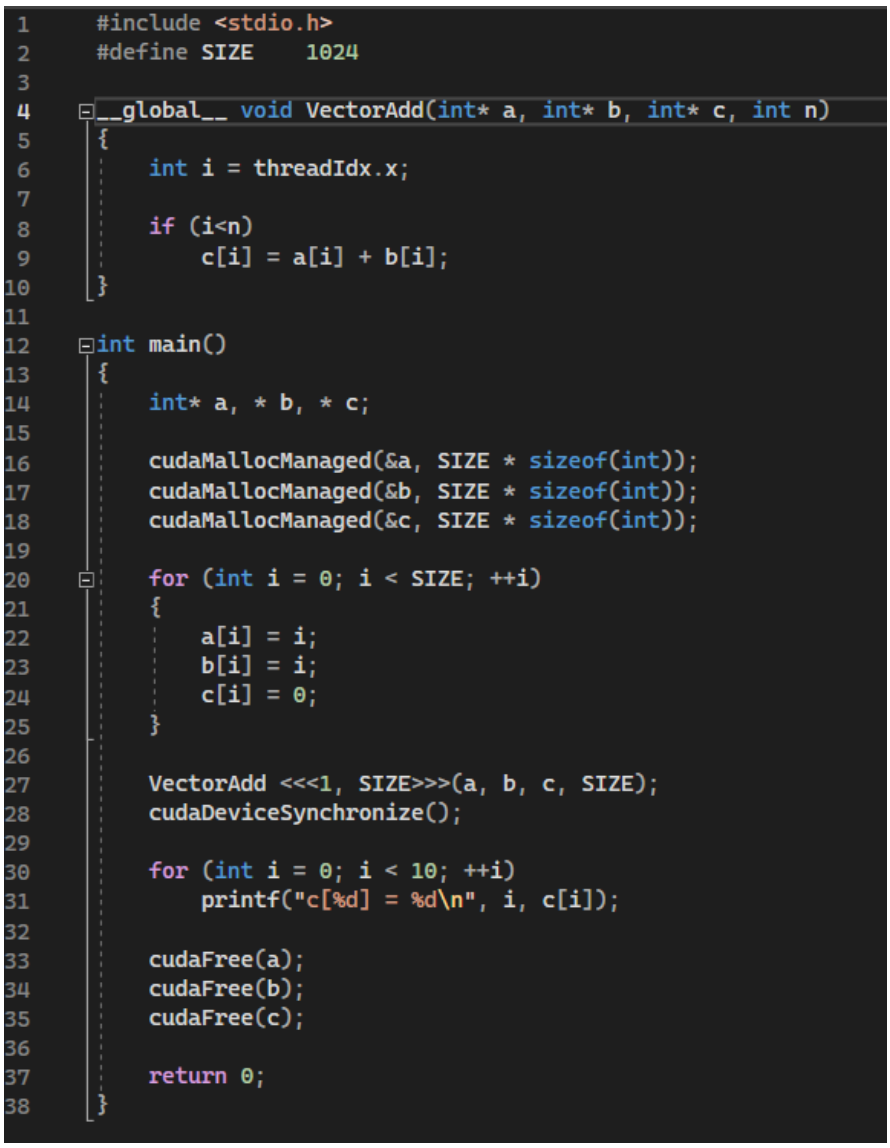
CUDA — це паралельна обчислювальна платформа, розроблений компанією Nvidia для обчислень загального призначення на власних GPU. CUDA надає прямий доступ до віртуального набору інструкцій GPU. Для роботи з інтерфейсом програміст має бути знайомим з мовами високого рівня: C, C++, Fortran, або Python.

AMD FireStream — програмна архітектура для обчислень за допомогою графічних процесорів від компанії AMD. На рівні інтерфейсу дає схожі можливості, що і CUDA, проте зовсім іншу реалізацію і філософію. Хоча в деяких тестах дана технологія показує себе краще ніж CUDA [5], через довге панування на ринку відеокарт від компанії Nvidia вона рідше застосовується в розробці програмного забезпечення.

2.2. Бібліотеки та надбудови CUDA

Графічні процесори компанії Nvidia є найпопулярнішими в світі, тож їх технології наразі є провідними і знайшли застосування в багатьох сферах, зокрема науці про данні, аналітиці, машинному та глибокому навчанні. Саме тому ця технологія найкраще підходить для реалізації теми даного дослідження.

Оригінально програмування за допомогою CUDA вимагав написання коду на мові C з певними розширеннями. Розглянемо простий приклад коду, який частково виконується на GPU:



```cpp
#include <stdio.h>
#define SIZE    1024

__global__ void VectorAdd(int* a, int* b, int* c, int n)
{
    int i = threadIdx.x;

    if (i<n)
        c[i] = a[i] + b[i];
}

int main()
{
    int* a, * b, * c;

    cudaMallocManaged(&a, SIZE * sizeof(int));
    cudaMallocManaged(&b, SIZE * sizeof(int));
    cudaMallocManaged(&c, SIZE * sizeof(int));

    for (int i = 0; i < SIZE; ++i)
    {
        a[i] = i;
        b[i] = i;
        c[i] = 0;
    }

    VectorAdd <<<1, SIZE>>>(a, b, c, SIZE);
    cudaDeviceSynchronize();

    for (int i = 0; i < 10; ++i)
        printf("c[%d] = %d\n", i, c[i]);

    cudaFree(a);
    cudaFree(b);
    cudaFree(c);

    return 0;
}
```

Рис. 2.1. Додавання векторів на CUDA

Як видно з прикладу, код не є чистим C. Розширенням мови є специфікатори (наприклад __global__), що вказують на якому процесорі має виконуватись функція, та накрадають певні обмеження, нові типи даних, спеціальні змінні (threadIdx), виклик ядра (функції зі специфікатором __global__, яка виконується на GPU: VectorAdd<<<1, SIZE>>>), та інші [6].

Приклад коду на C, дає змогу програмісту частково зрозуміти, що відбувається «під капотом» бібліотек Rapids, глибокого навчання тощо, які уможливлюють виконання коду з синтаксисом Python. Програмування на GPU навіть на Python часто вимагає знання особливостей CUDA, тому варто ознайомитись з літературою,

присвячену їй. Розглянемо структуру яка міститься між CUDA та прикладним інтерфейсом.

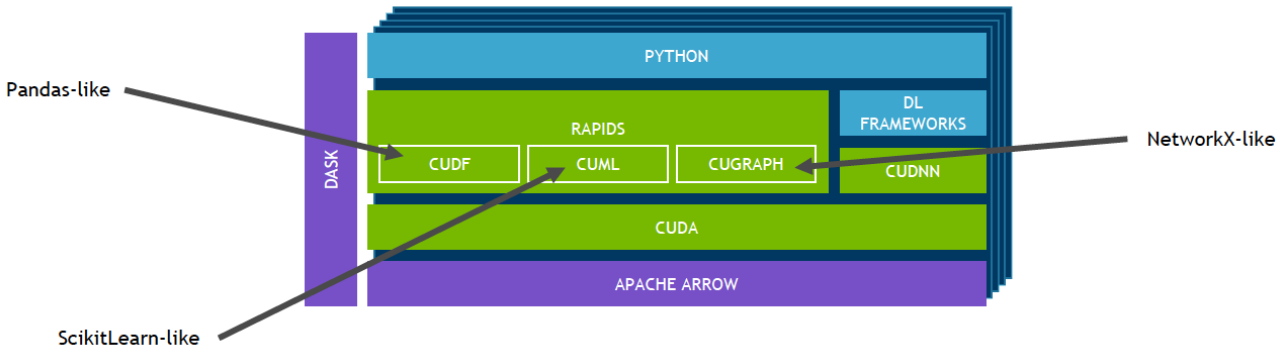


Рис. 2.2. Архітектура функціоналу для програмування на GPU, доступного через інтерфейс Python

Існує дуже широкий спектр бібліотек написаних на архітектурі CUDA, які мають застосування в різних галузях. Деякі з них представляють інтерфейс мовою Python і можуть бути корисними у роботі з великими і складними обчисленнями.

Підтримка формату Apache Arrow уможливлює швидкий доступ до даних і їх обмін між різними процесами, а Dask забезпечує можливість виконання паралельних обчислень на декількох підключених графічних картах. Dask частково вирішує проблему відсутності автоматичного керування пам'яттю: якщо датасет завеликий для обробки на одній GPU, можна підключити більше процесорів і додати пам'яті.

Для конвертації коду з Python в CUDA застосовується компілятор Numba, а саме модуль CUDA. На основі оригінального коду компілятор генерує C++ код, який вже виконується в архітектурі CUDA. Це дає програмісту можливості написання функцій на Python, маючи ті ж інструменти, які є в C++. Безперечно, такий підхід буде менш продуктивним ніж запуск ядер, написаних на C++, проте він є більш приємним для роботи тим, хто не дуже знайомий з синтаксисом C або C++, та також показує хороші результати.

Із причин описаних вище програмування графічних процесорів на Python дуже схоже за синтаксисом на оригінальну CUDA. Щоб отримати доступ до набору інструментів потрібно лише імпортувати бібліотеку:

```
from numba import cuda
```

Розглянемо простий приклад розв'язку задачі суми векторів (Рис.2.3). Порівнюючи його з попереднім прикладом, легко помітити, що програміст отримує подібний набір інструментів: спеціальні змінні, специфікатор функції, запуск ядра тощо.

```python
@cuda.jit
def add_array(a, b, c):
    i = cuda.threadIdx.x + cuda.blockDim.x * cuda.blockIdx.x
    if i < a.size:
        c[i] = a[i] + b[i]

N = 20
a = np.arange(N, dtype=np.float32)
b = np.arange(N, dtype=np.float32)
dev_c = cuda.device_array_like(a)

add_array[4, 8](a, b, dev_c)

c = dev_c.copy_to_host()
print(c)

#  [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18. 20. 22. 24. 26. 28. 30. 32. 34. 36. 38.]
```

Рис. 2.3 Додавання векторів на Python (GPU)

Задачі, які виникатимуть при роботі з наступними бібліотеками часто будуть вимагати написання функцій для виконання на GPU, а отже і знання програмування на CUDA.

2.1 Rapids

Rapids – це сімейство бібліотек для аналізу інформації (cudf), графів (cugraph), машинного навчання (cuml) тощо. В ідеалі вони мають надавати аналогічний інтерфейс як у відомих Pandas, NetworkX та Scikit-learn. Тобто якщо програміст замінив *import pandas as pd* на *import cudf as pd,* код би відпрацював так само тільки з більшою швидкістю. Проте в реальності для коректної роботи цих бібліотек та покращення ефективності коду існує багато обмежень, з якими приходиться зіштовхуватися при переведенні частин програм на GPU.

Перше правило ефективного програмування на GPU: дані не мають покидати його і починати оброблюватись на CPU, а потім знову на графічному процесорі: це тільки уповільнить виконання коду. Це правило випливає з архітектурних особливостей комп’ютера, які були описані в попередніх розділах. Тому для реальної ефективності, одного Rapids може бути недостатньо. Для подолання проблеми існує ряд бібліотек, які прийшли на заміну звичним для дата-сайєнтистів утилітам. Вони є сумісними між собою через підтримку технології Apache Arrow.

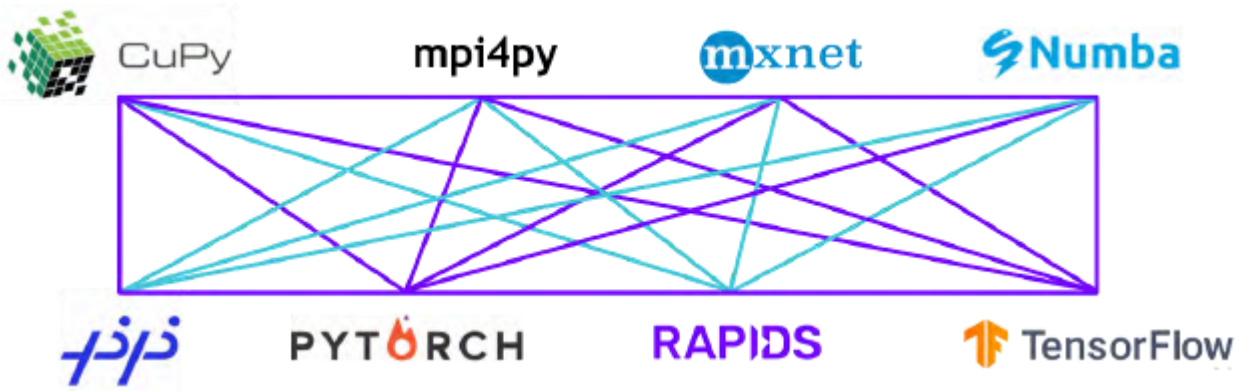


Рис. 2.4. Бібліотеки для Data Science, що підтримують GPU

2.2 Бібліотеки для глибокого навчання

Nvidia підтримує розробку бібліотек для глибокого навчання та тренування моделей на GPU. Це в десятки разів пришвидшує тренування моделей з великою кількістю параметрів. Для розробки бібліотеки знадобиться cuDNN, що входить в CUDA toolkit. Проте кінцеві споживачі можуть скористатися вже існуючими, які підтримують архітектуру CUDA, наприклад Tensorflow, Pytorch або Mxnet.

2.3 Інші бібліотеки для Data Science

На сайті Nvidia представлений широкий список бібліотек, що можуть знадобитись при розробці застосувань для обчислень на GPU. Коротко опишемо деякі з них:

- cuBLAS зі стандартними процедурами лінійної алгебри
- cuSPARSE надає основні функції для операцій з розрідженими матрицями
- cuSOLVER для лінійної оптимізації
- Бібліотеки паралельних алгоритмів: NCCL, Thrust
- CuPy використовує попередні бібліотеки та має схожий до Numpy інтерфейс на Python, проте повністю використовує архітектуру GPU для обчислень. Також розробнику надається можливість писати власні ядра на C++ прямо в коді.

РОЗДІЛ 3. ПРАКТИЧНЕ ВТІЛЕННЯ ТА АНАЛІЗ ШВИДКОДІЇ ОБРОБКИ ДАНИХ

3.1. Вибір графічної карти

Безкоштовна версія Google Colab надає доступ до середовища виконання з прискоренням на Nvidia Tesla T4. В той час як мій ноутбук має не надто потужну Nvidia GeForce MX230. На основі порівняння [7] цих графічних процесорів було обрано більш потужний – віддалене середовище виконання у Colab. Основними перевагами Tesla T4 є більший у 8 разів розмір пам’яті та швидкодія. Це дасть можливість оброблювати більші об’єми пам’яті, і покаже кращий результат.

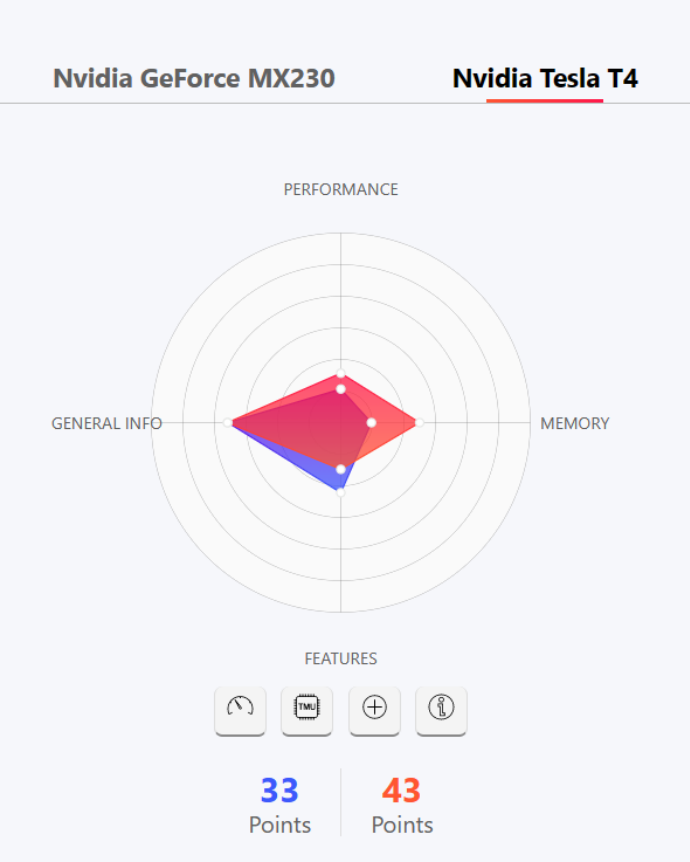


Рис. 3.1. Результат порівняння відеокарт Nvidia Tesla T4 та GeForce MX230

Також протягом виконання обчислень, було виявлено, що використовувалось до 9 GB пам’яті GPU. А отже, експеримент було б неможливо провести на GeForce MX230.

3.2. CuDF

В інтернеті існує багато задач порівняння швидкодії обчислень на CPU з допомогою бібліотеки Pandas з на GPU та бібліотеки CuDF. Дуже часто вони дають очевидний результат. Обчислення на GPU значно швидші, проте крім перетворень даних часто вони мають передаватися в інший процес (наприклад тренування моделі або візуалізація). Тому для вимірювання реального виграшу в часі потрібно враховувати перенесення даних з GPU на CPU та аналізувати задачу в цілому.

Ще однією причиною потреби в комплексному оцінюванні є той факт, що дані бібліотеки іноді мають різний функціонал, тому виникають ситуації коли підходи до розв'язку задачі, що легко програмуються та швидко виконуються на CPU, є не дуже тривіальними для аналогічного виконання на GPU (наприклад лямбда-функції для рядкового типу даних або об'єктів)

Для аналізу виграшу в швидкодії я взяла частину проекту з аналізу дата сету мого Telegram, та перевела найцікавіші задачі для обробки на GPU. Протягом виконання даного проекту, однією із найбільших проблем у всіх студентів були повільні обчислення на CPU, тому cudf має шанс стати хорошим рішенням проблеми. Слід зауважити, що для вирішення деяких задач було повністю змінено логіку, для отримання оптимального відношення:

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

Обчислення задач в середньому займало до 30с на CPU та до 10с на GPU. Тому сенс прискорення обчислень втрачається, якщо для пошуку швидшого методу на пару секунд програмісту потрібно витрати в рази більше часу.

Порівняємо результати отримані з 5 прикладів:

- Concat та Merge

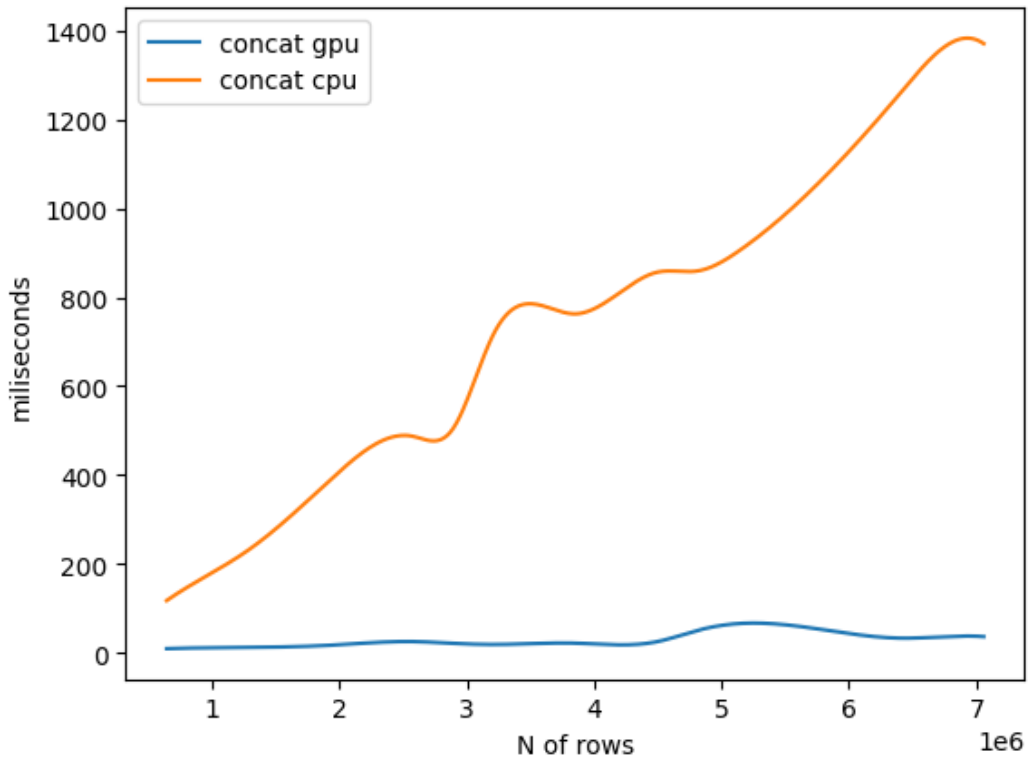


Рис. 3.2. Залежність часу виконання операції concat від об'єму даних

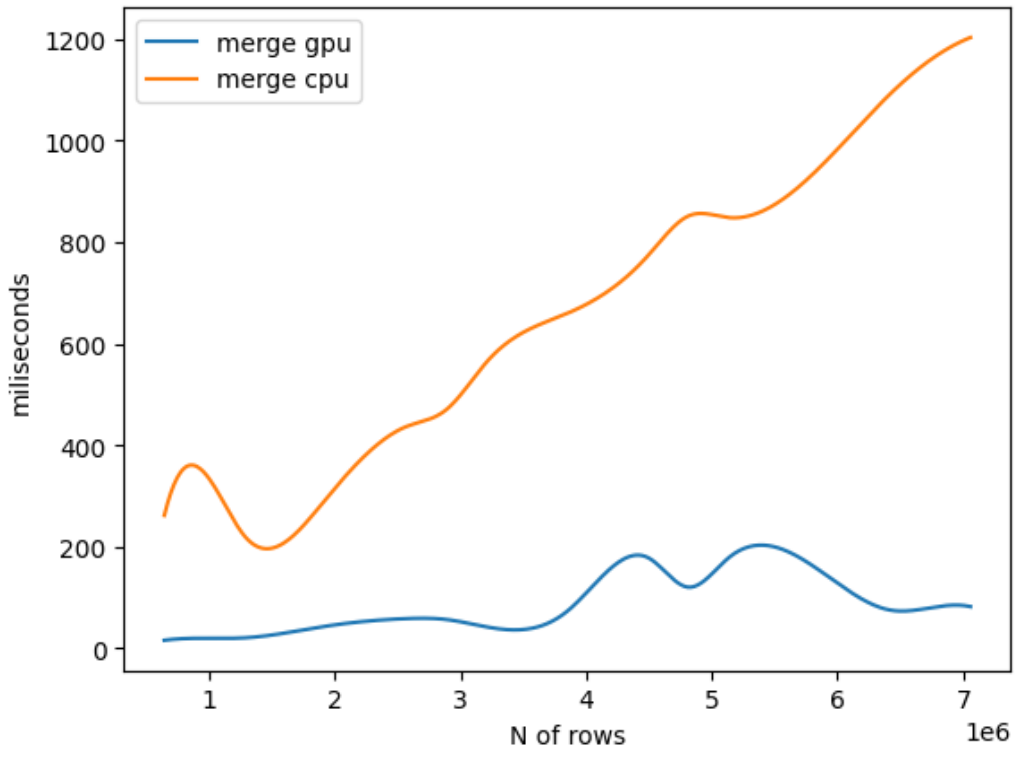


Рис. 3.3. Залежність часу виконання операції merge від об'єму даних

- Задача 1

Першою задачею, був обраний обрахунок інтенсивності листування з визначеним контактом. Використано операції вибірки, групування, а також перенаправлення результатів на CPU для створення графіку через matplotlib.

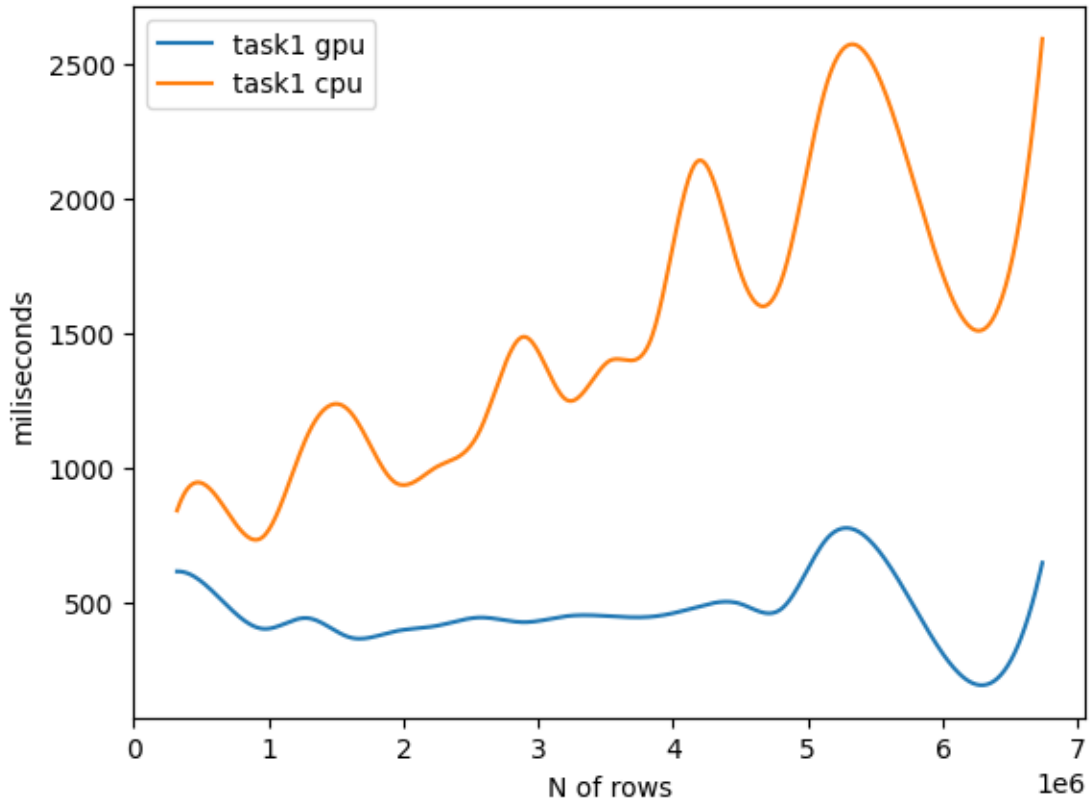


Рис. 3.4 Залежність часу обчислення тестової задачі 1 від об'єму даних

Аналізуючи результат часу виконання обчислень, при збільшенні кількості рядків в дата сеті, отримуємо прискорення на декілька секунд. Проте при дата сеті, що містить менше ніж 1 млн рядків, різниця майже непомітна.

- Задача 2

Наступна задача полягає в знаходженні середньої довжини повідомлень за місяць. До дата фрейму pandas, застосовувалась функція apply. А до cudf аналогічні схожі за результатом операції для рядкових та списків series. Основна причина такого вибору: методи для рядків у pandas працюють повільніше ніж такі ж функції з інших бібліотек.

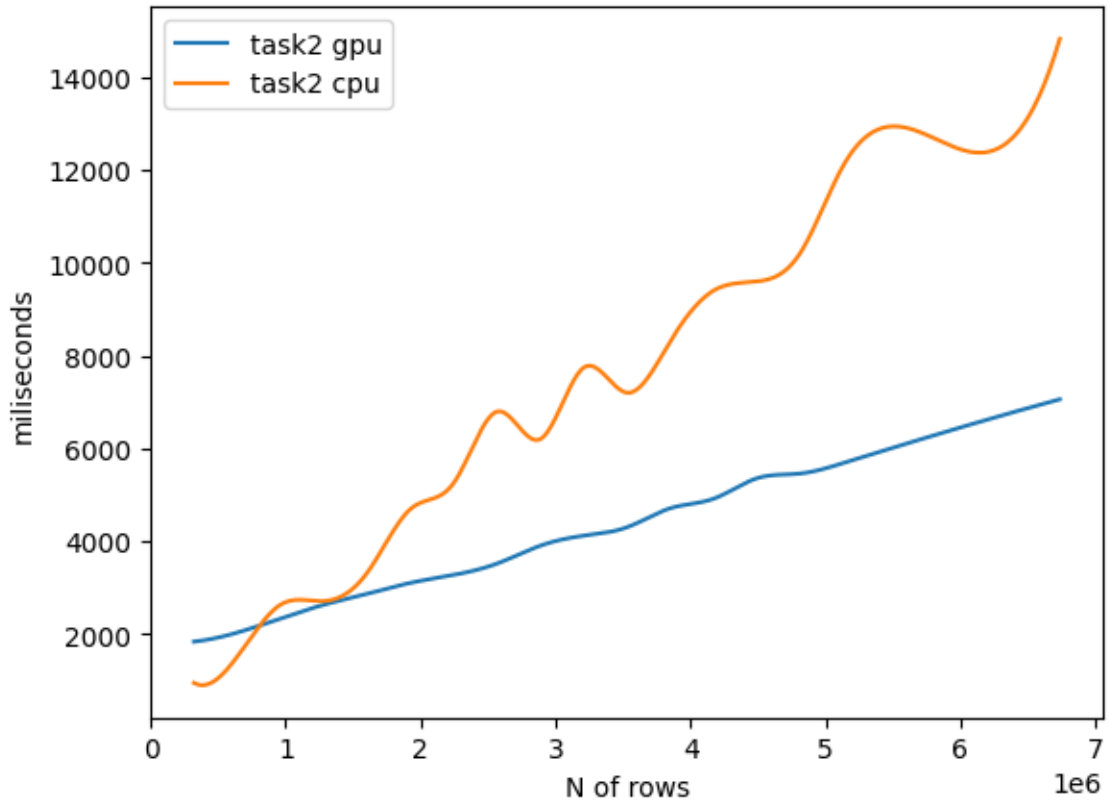


Рис. 3.5. Залежність часу обчислення тестової задачі 2 від об'єму даних

Аналізуючи результат, визначаємо, що метод розв’язання задачі на Pandas при дата сеті до 1,5 млн рядків є більш ефективним або майже еквівалентний способу, який використовує прискорення на графічному процесорі. Проте картина стрімко змінюється із зростанням об’єму даних.

- Задача 3

Метою цієї задачі було розробити алгоритм знаходження тональності повідомлень. Особливістю прикладу є ідентичність наборів викликів методів, за винятком моменту розбиття всього дата фрейму на за ознакою відправлених та отриманих повідомлень. Були використані методи для роботи із рядковими series.

Цей приклад нативно доводить, що застосування методів Pandas для рядкового типу даних є дуже неефективним. Тому потрібно шукати інші бібліотеки. В той час як на графічному процесорі, методи показують вражаючий результат.

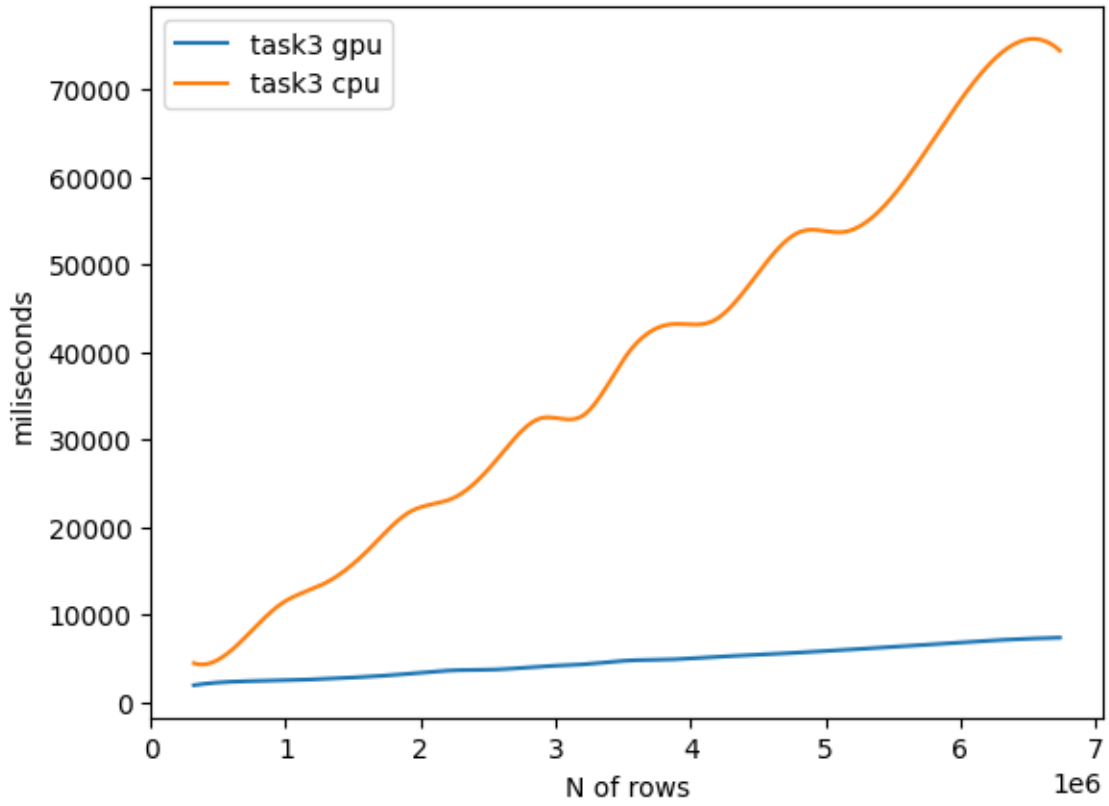


Рис. 3.6. Залежність часу обчислення тестової задачі 3 від об'єму даних

3.3. Deep Neural Networks

Вже декілька років графічні процесори стали важливим правилом у тренуванні моделей нейронних мереж. Розробники Google Translate проводили багато тижневих запусків Translate на GPU, які могли б зайняти місяці при обчисленнях на CPU.
Для переконанні в цьому результаті було протестовано архітектуру Autoencoder та CNN модель для класифікації, з такою структурою:

```
Model: "denoise_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_5 (Sequential)   (None, 7, 7, 8)           1320

 sequential_6 (Sequential)   (None, 28, 28, 1)         1897

=================================================================
Total params: 3,217
Trainable params: 3,217
Non-trainable params: 0
_________________________________________________________________
None
```

Рис. 3.7. Структура моделі Autoencoder

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 28, 28, 32)        320

 max_pooling2d_1 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 flatten_1 (Flatten)         (None, 6272)              0

 dense_1 (Dense)             (None, 10)                62730

=================================================================
Total params: 63,050
Trainable params: 63,050
Non-trainable params: 0
_________________________________________________________________
None
```

Рис. 3.7. Структура моделі CNN

При запуску тренування з використанням GPU прискорення та без, було виявлено скорочення часу виконання в 4-5 разів на моделях з 3 та 63 тис параметрів. Швидкодія покращилась за рахунок наявності згорткових шарів, в обчисленнях яких застосовується множення матриць. На моделях без згорткових шарів приріст швидкості незначний, тому при запуску тренування їх з GPU прискоренням потрібно враховувати повільний процес обміну даними між процесорами. Використання звичайних шарів призводить до збільшення кількості параметрів: відповідно об’єму даних, який потрібно передами. Цей процес може призвести до значного сповільнення, або відсутності користі від використання GPU.

Ці результати доводять доцільність використання GPU навіть на неглубоких CNN моделях та обов’язковість для таких, що містять сотні тисяч параметрів.

ВИСНОВКИ

На підставі експерименту можна зробити висновок, що використання графічних процесорів для прискорення великих обчислень є ефективним і перспективним рішенням. У той же час корисно пам’ятати про деякі обмеження та деталі програмування GPU.

По-перше, дослідження показують, що CUDA, Rapids та інші технології на основі графічного процесора є значно швидшими, ніж обчислення ЦП на великих наборах даних. Однак для невеликих обчислень використання GPU може бути неефективним, оскільки збільшення швидкості може бути незначним порівняно з вартістю передачі даних між центральним та графічним процесорами.

Також важливо враховувати розмір даних і технічні характеристики відео карти, оскільки не всі комп’ютери (а особливо ноутбуки) підходять для програмування на GPU. Графічні процесори мають відмінну від CPU архітектуру, що може ускладнювати процес програмування та вимагати від розробників додаткових знань та навичок. Великі обчислення вимагають багато пам’яті, при переповненні змінні графічного процесора не записуються на диск автоматично. Це може зробити неможливим запуск програми на певній архітектурі та неочікуваним небажаним результатам.

Незважаючи на ці проблеми та обмеження, технологія GPU продовжує зростати в популярності та розвиватися. У зв’язку зі стрімким розвитком технологій очікується, що в майбутньому кількість продуктів, що використовують прискорення в графічних процесорах, зросте. Особливо у сфері машинного навчання та інших важких обчислень використання GPU може стати нормою.

Таким чином, використання прискорення GPU є потужним рішенням для великомасштабних обчислень, але вимагає певного рівня знань і досвіду програмування GPU. Оскільки технологія прискорення GPU продовжує розвиватися і може стати стандартом у різних галузях промисловості та науки, важливо досліджувати та використовувати її потенціал для підвищення продуктивності та ефективності обчислень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] E. K. Jason Sanders, CUDA by Example: An Introduction to General-Purpose GPU Programming, 2010: Addison-Wesley Professional.

[2] M. R. M. Mark D. Hill, «Amdahl's Law in the Multicore Era,» *Computer,* № 41, pp. 33-38, 15 July 2008.

[3] «Wikipedia: Bus (computing)».

https://en.wikipedia.org/wiki/Bus_(computing)

[4] D. Exterman, «CUDA vs OpenCL: Which to Use for GPU Programming,» *Incredibuild blog,* 2021.

https://www.incredibuild.com/blog/cuda-vs-opencl-which-to-use-for-gpu-programming

[5] Г. С. Стребков, «Комплексне дослідження методів програмної спеціалізації для графічних процесорів,» 2020.

https://openarchive.nure.ua/server/api/core/bitstreams/012e65e6-12a4-429f-b000-3ac3ab822bb1/content

[6] Х. А. А. Боресков А. В., «Основы работы с технологией CUDA,» ДМК, Москва, 2010.

[7] «Nvidia GeForce MX230 vs Nvidia Tesla T4».

https://versus.com/en/nvidia-geforce-mx230-vs-nvidia-tesla-t4