Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

Алгоритм обчислення тридіагональної матриці ортогональним розкладанням на графічному процесорі

Текстова частина до курсової роботи
за спеціальністю „Комп'ютерні науки"

Керівник курсової роботи
проф., д. ф.-м. н.
Малашонок Г. І.

_____
(підпис)
"____" _____ 2021 р.

Виконав студент 1 курсу
магістерської програми
Комп'ютерні науки
Сухарський С.С.
"____" _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри мережних
технологій,
проф., д. ф.-м. н.
Малашонок Г. І.

_____
(підпис)
“____” _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Сухарському Сергію факультету інформатики 5 курсу
ТЕМА “ Алгоритм обчислення тридіагональної матриці ортогональним розкладанням на графічному процесорі”.

Зміст ТЧ до курсової роботи:

Зміст
Анотація
Вступ
1. Опис алгоритму ортогонального розкладання Хаусхолдера.
2. Опис платформи та середовища CUDA.
3. Опис програмної реалізації.
4. Опис проведених експериментів та отриманих результатів.
Висновки
Список використаних джерел

Дата видачі „13” жовтня 2020 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Алгоритм обчислення тридіагональної матриці ортогональним розкладанням на графічному процесорі.

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	13.10.2020	
2.	Огляд літератури за темою роботи.	13.11.2020	
3.	Аналіз алгоритмів обчислення тридіагональної матриці.	13.12.2020	
3.	Аналіз роботи з графічними процесорами.	13.01.2021	
4.	Розробка алгоритму для виконання на графічному процесорі.	01.02.2021	
5.	Реалізація алгоритму в середовищі JCUDA.	01.03.2021	
6.	Тестування розробленої програми.	08.03.2021	
7.	Аналіз тестування та виправлення помилок.	21.03.2021	
8.	Аналіз отриманих результатів з керівником.	01.04.2021	
9.	Корегування роботи.	13.04.2021	
10.	Оформлення текстової частини.	01.05.2021	
11.	Підготовка презентації та доповіді.	11.05.2021	
12.	Захист курсової роботи.	17.05.2021	

Студент Сухарський С. С.

Керівник Малашонок Г.І.

“_____” _____

Зміст

Анотація

У роботі розглянуто та реалізовано алгоритм ортогонального розкладання матриці, який є частиною алгоритму SVD. Також описано платформу та середовище для роботи з графічними процесорами NVIDIA CUDA. Наведено реалізацію бідіагоналізації матриці та обчислення ортогональних множників методом Хаусхолдера в середовищі jCUDA. Проведено експерименти та дослідження отриманих результатів пришвидшення обчислень з використанням графічного процесора порівняно з реалізацією на центральному процесорі.

Ключові слова: алгоритм, процесор, GPU, CPU, матриця, бідіагоналізація, ортогональність, симетричність, SVD, Хаусхолдер, метод, програма, CUDA, вказівник, відеокарта, об'єкт, тип, клас, число.

Вступ

Сучасний світ дуже сильно зав’язаний на обчислення. З розвитком Big Data, Штучного інтелекту та інших популярних сьогодні сфер, потреба в швидкісних та ефективних обчисленнях стала одним з найважливіших завдань сьогодення. Саме з такою метою протягом останніх більш як десяти років активно розвивається галузь обрахунків на графічних процесорах, які містять тисячі ядер. Однією з передових платформ для цього є технологія розроблена NVIDIA під назвою CUDA. Саме з цією технологією пов’язане це дослідження, а точніше, з реалізацією частини SVD алгоритму з використанням потужності відеокарт, який є одним з головних методів роботи прогресивних рекомендаційних систем; який вирішує багато завдань лінійної алгебри та може бути застосований для розвитку штучного інтелекту.

Метою дослідження є реалізація алгоритму використовуючи метод Хаусхолдера та проект jCUDA, для роботи з відеокартами в мові Java та порівнянь ефективності цього підходу зі вже існуючими. Наприклад в роботі Савченка [2] досліджувався алгоритм SVD з використанням прямого QR алгоритму, який показав пришвидшення відносно ітеративних версій алгоритму, проте підхід з використанням методу Хаусхолдера виглядає перспективнішим по ефективності обчислень. Схожий підхід з використанням серії трансформацій Хаусхолдера використовували в своїй роботі Лагабар та Нараянан [1], проте в їхній роботі все ще не було запропоновано хорошого способу для роботи з великими матрицями, а це є невід’ємною частиною сучасних обчислень, тому актуальність поточного дослідження значна. Середовище jCUDA було обрано для подальшої інтеграції з великим комплексом бібліотек для паралельного програмування Mathpartner.

Розділ 1. Опис математичної частини

1.1 Загальні відомості

Означення. Нехай A – матриця над полем комплексних (або дійсних) чисел розміру $m \times n$. Її розклад в добуток трьох матриць

$$A = U\Sigma V^T,$$

де A – будь яка матриця розміру $m \times n$, U – матриця $m \times m$ та ортогональна, V – матриця $n \times n$ та ортогональна і Σ – діагональна $m \times n$ матриця з впорядкованими за спаданням невід'ємними елементами на діагоналі, називається сингулярним розкладом (Singular Value Decomposition).

Цей розклад який надає багато інформації про вхідну матрицю A, наприклад її ранг та нульовий простір. Також розклад володіє багатьма властивостями, наприклад є можливість отримати наближений розклад в якому в матриці Σ можна залишити лише n елементів (λ) на діагоналі, цим самим зменшити кількість стовпчиків та рядків в матрицях U та V відповідно і отримати наближену матрицю до A, проте з використанням меншої кількості ресурсів та все ще високою точністю.

Стандартним способом обрахунку SVD раніше було ітеративне виконання QR алгоритму, поки не буде отримано потрібний результат. [4] Проте, цей алгоритм досить затратний по часу та ресурсах. Покращений SVD алгоритм складається з трьох етапів [2]:

1. Для вхідної матриці A виконується обрахунок QR розкладу.
2. Розкладання матриці R^T в добуток $L_1^T D_2 R_1^T$, де D_2 – дводіагональна матриця.
3. Розкладання матриці D_2 в добуток $L_2^T D_1 R_2^T$, де D_1 – діагональна матриця.

Проте, ми використовуємо інший, сучасний двоетапний підхід з використанням методу Хаусхолдера, який краще підходить для обрахунків на графічних процесорах [1]. На першому етапі ми отримуємо такий розклад вхідної матриці A:

$$U_1 D_2 W_1^T,$$

де U та W ортогональні, а D_2 верхня бідіагональна. В випадку, коли матриця A симетрична, D_2 буде тридіагональною. Після цього, на другому етапі ми ітеративно виконуємо ще один розклад:

$$U_2 D W_2^T,$$

де D діагональна матриця з впорядкованими за спаданням невід'ємними елементами λ, а матриці U та W ортогональні.

1.2 Метод Хаусхолдера

1.2.1 Визначення

Розглянемо метод Хаусхолдера детальніше. Нехай задано нормований n-вимірний лінійний простір V над полем дійсних чисел з евклідовою нормою

$$\forall v \in V: \left|\left|v\right|\right|^2 = v^T v = \sum_{i=1}^{n} v_i^2$$

Перетворення Хаусхолдера (відображення Хаусхолдера) – це лінійне перетворення векторного простору, яке використовується в лінійній алгебрі для обрахунків ортогональних розкладів:

$$A = QR, A = UDV^T,$$

де Q, V, U – ортогональні (унітарні для поля комплексних чисел) матриці, R – права трикутна, а D дводіагональна матриці [5, 6].

1.2.2 Теорема Хаусхолдера

Для ненульових векторів $x, y \in V$ з однаковою нормою ($\left||x|\right| = \left||y|\right|$) існує ортогональна симетрична матриця P, така, що

$$y = Px.$$

Доведення

Позначимо *v*-нормовану різницю цих векторів і складемо матрицю *P:*

$$P = I - 2vv^T, v = \frac{(x-y)}{\left||x-y|\right|}, \left||v|\right| = 1.$$

Обрахуємо квадрат цієї матриці:

$$P^2 = I - 4vv^T I + 4v(v^T v)v^T = I$$

Таким чином,

$$P = P^T = P^{-1},$$

звідси випливає, що P є ортогональною симетричною матрицею. Вектори $(x-y)$та$(x+y)$, ортогональні:

$$(x-y)^T(x+y) = x^T x + x^T y - y^T x - y^T Y = \left||x|\right| - \left||y|\right| = 0,$$

тому:

$$P(x+y) = (I - 2vv^T)(x+y) = (x+y).$$

Обчислимо:

$$P(x-y) = (I - 2vv^T)(x-y) = -(x-y).$$

Звідси отримуємо:

$$P(x) = \frac{1}{2}\big(P(x+y) + P(x-y)\big) = y.$$

Теорему доведено.

1.2.3 Означення та приклади

Означення

Для ненульового вектора *u* матриця

$$P = I - \frac{2}{u^T u} u u^T$$

називається **матрицею Хаусхолдера**, а вектор *u* – **вектором Хаусхолдер** цієї матриці.

Доповнення

Можна легко побудувати матрицю Хаусхолдера, яка при множенні на заданий ненульовий вектор *x* перетворює його в $y = Px$, так, щоб змінилось значення не більше однієї компоненти і при цьому, обнулились будь які інші компоненти даного вектора. Наприклад, нехай потрібно обнулити всі компоненти окрім першої. Тоді, потрібно взяти:

$$y_1 = \left(||x||, 0, .., 0\right) \text{ або } y_2 = \left(-||x||, 0, .., 0\right).$$

Для зменшення похибки обчислень, беруть такий варіант, при якому різниця ||*y* - *x*|| буде найбільшою. Тобто:

$$y = \left(-sign(x_1)||x||, 0, .., 0\right), u = x - y = \left(x_1 + sign(x_1)||x||, x_2, .., x_n\right).$$

Приклад

Нехай дано матрицю:

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 2 & 1 & 1 \end{pmatrix}$$

Визначимо вектор *x* як перший стовпчик, та обчислимо наступні компоненти:

$$\text{x} = \begin{pmatrix}1\\1\\1\\1\end{pmatrix};\ norm_x = 2;\ u = \begin{pmatrix}1+2\\1\\1\\1\end{pmatrix};\ norm_u^2 = 12;$$

$$P_1 = I_{4,4} - (2/12)uu^T;$$

Позначимо через dT вектор-рядок, отриманий в результаті множення:

$$d_T = (u^T * A);$$

Тоді, добуток $P_1 * A$ дорівнюватиме:

$$A_1 = P_1 A = \left(I_{4,4} - (2/12)uu^T\right)A = A - (1/6)u(u^T A) = A - (1/6)ud_T;$$

$$P_1 = \begin{pmatrix}-0.5 & -0.5 & -0.5 & -0.5\\ -0.5 & 0.83 & -0.17 & -0.17\\ -0.5 & -0.17 & 0.83 & -0.17\\ -0.5 & -0.17 & -0.17 & 0.83\end{pmatrix}$$

$$A_1 = \begin{pmatrix}-2 & -2.5 & -2.5 & -2.5\\ 0 & -0.17 & 0.83 & -0.17\\ 0 & -0.17 & -0.17 & 0.83\\ 0 & 0.83 & -0.17 & -0.17\end{pmatrix}$$

Зазначимо, що множення $dT = (u^T * A)$ дуже ефективне. Рядок будемо множити на матрицю.

$$dT = (u^T * A) = \sum_{i=1}^{n} u_i A_i$$

Тобто, елемент u^T з номером i множиться н всі елементи рядка i та додається до результату. Всього n^2 операцій множення і приблизно стільки ж

операцій додавання. Множення $u * dT$ це множення вектор-стовпчика на вектор-рядок і його теж можна виконувати порядково.

Наша ціль, звести матрицю A до дводіагонального виду, тому, тепер потрібно обнулити перший рядок. При цьому, не можна змінювати перший стовпчик. Отримаємо:

$$\text{x} = (0 \quad -2.5 \quad -2.5 \quad -2.5);$$

$$norm1_x2 = 2.5^2;$$

$$norm2_x2 = 2 \cdot 2.5^2; norm_x = \sqrt{norm1_x2 + norm2_x2};$$

$$u = (0 \quad -2.5 - norm_x \quad -2.5 \quad -2.5);$$

$$normSq_u = 2((norm_x)^2 + 2.5 \cdot norm_x) = 2 \cdot norm_x(norm_x + 2.5);$$

$$Q_1 = I_{4,4} - (2/normSq_u) \cdot u^T \cdot u;$$

$$d = (A_1 \cdot u^T);$$

$$A_2 = A_1 \cdot Q_1 = A_1 \cdot \left(I_{4,4} - (2/normSq_u)u^T u\right) =$$

$$= A_1 - A_1 \cdot (2/normSq_u) \cdot u^T \cdot u = A_1 - (2/normSq_u) \cdot d \cdot u;$$

$$Q_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -0.58 & -0.58 & -0.58 \\ 0 & -0.58 & 0.79 & -0.21 \\ 0 & -0.58 & -0.21 & 0.79 \end{pmatrix};$$

$$A_1 = \begin{pmatrix} -2 & 4.33 & 0 & 0 \\ 0 & -0.29 & 0.79 & -0.21 \\ 0 & -0.29 & -0.21 & 0.79 \\ 0 & -0.29 & -0.58 & -0.58 \end{pmatrix};$$

Тепер потрібно обнулимо другий стовпчик. При цьому, не можна змінювати перший рядок. Отримаємо:

$$x = (0 \quad -0.29 \quad -0.29 \quad -0.29); \; norm_x = \sqrt{3} \cdot 0.29^2 = \sqrt{3} \cdot 0.29;$$

$$u = (0 \quad -0.29 - norm_x \quad -0.29 \quad -0.29);$$

$$normSq_u = 2 \cdot norm_x(norm_x + 0.29);$$

$$P_2 = I_{4,4} - (2/normSq_u)u^T u;$$

$$A_3 = P_2 \cdot A_2 = A_2 - (2/normSq_u)ud_T;$$

$$P_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -0.58 & -0.58 & -0.58 \\ 0 & -0.58 & 0.79 & -0.21 \\ 0 & -0.58 & -0.21 & 0.79 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} -2 & 4.33 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & -0.5 & 0.87 \\ 0 & 0 & -0.87 & -0.5 \end{pmatrix}$$

Другий рядок вже має дводіагональний вигляд, тому потрібно обнуляти лише останній елемент в третьому стовпчику. За таких умов не можна змінювати перший та другий рядок:

$$x = (0 \quad 0 \quad -0.5 \quad -0.87);\ norm_x = \sqrt{0.5^2 \cdot 0.29^2};$$

$$u = (0 \quad 0 \quad -0.5 - norm_x \quad -0.87);$$

$$normSq_u = 2 \cdot norm_x(norm_x + 0.5);$$

$$P_3 = I_{4,4} - (2/normSq_u)u^T u;$$

$$A_4 = P_3 \cdot A_3 = A_3 - (2/normSq_u)u^T u;$$

$$P_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.5 & -0.87 \\ 0 & 0 & -0.87 & 0.5 \end{pmatrix}$$

$$A_4 = \begin{pmatrix} -2 & 4.33 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Перевірка

Зробимо перевірку на правильність обчислень. Перевіримо, що

$$A = U \cdot A_4 \cdot W^T, \text{ де}$$

$$P = (P_3 \cdot P_2 \cdot P_1);\ U = P^T;\ W = Q_1;\ B = U \cdot A_4 \cdot W;$$

$$Check = B - A;$$

$$Ch_U = (U^{-1} - U^T);$$

$$Ch_W = (W^{-1} - W^T);$$

$$U = \begin{pmatrix} -0.5 & 0.87 & 0 & 0 \\ -0.5 & -0.29 & 0.79 & 0.21 \\ -0.5 & -0.29 & -0.21 & -0.79 \\ -0.5 & -0.29 & -0.58 & 0.58 \end{pmatrix}$$

$$W = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -0.58 & -0.58 & -0.58 \\ 0 & -0.58 & 0.79 & -0.21 \\ 0 & -0.58 & -0.21 & 0.79 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 2 & 1 & 1 \end{pmatrix}$$

$$Check = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, Ch_U = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, Ch_w = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Отримали, що $B = A$ та всі матриці перевірок нульові (насправді, там елементи наближені до нуля, але в ідеальному випадку вони дуже малі – наприклад 1 * 10^-17).

Перевірку пройдено.

1.2.4 Зведення симетричної матриці A до симетричного тридіагонального вигляду

Для того, щоб звести симетричну матрицю до симетричного тридіагонального вигляду, на першому кроці матриця Хаусхолдера $P_1 = Q_1$ будується так, щоб перший рядок не змінювався під час множення на P_1, а перший стовпчик не змінювався під час множення на Q_1. Після цього, аналогічно, $P_2 = Q_2$, не змінюють перших два рядки і перших два стовпчики і так далі.

Приклад

Нехай дано матрицю:

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\text{x} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix};\ norm_x = \sqrt{3};$$

$$u = \begin{pmatrix} 0 \\ 1 + norm_x \\ 1 \\ 1 \end{pmatrix}$$

$$normSq_u = 2 \cdot norm_x(norm_x + 1);$$

$$P_1 = I_{4,4} - (2/normSq_u)uu^T;$$

$$A_1 = P_1\, A\, ((P_1)^T);$$

$$P_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -0.58 & -0.58 & -0.58 \\ 0 & -0.58 & 0.79 & -0.21 \\ 0 & -0.58 & -0.21 & 0.79 \end{pmatrix}$$

$$A_1 = \begin{pmatrix} 1 & -1.73 & 0 & 0 \\ -1.73 & 4.33 & -0.24 & 0.91 \\ 0 & -0.24 & 0.04 & -0.17 \\ 0 & 0.91 & -0.17 & 0.62 \end{pmatrix}$$

Другий крок:

$$\mathrm{x} = \begin{pmatrix} 0 \\ 0 \\ -0.24 \\ 0.91 \end{pmatrix};$$

$$norm_x = \sqrt{0.24^2 \cdot 0.91^2};$$

$$u = \begin{pmatrix} 0 \\ 0 \\ -0.24 - norm_x \\ 0.91 \end{pmatrix}$$

$$normSq_u = 2 \cdot norm_x(norm_x + 0.24);$$

$$P_2 = I_{4,4} - (2/normSq_u)uu^T;$$

$$A_2 = P_1\, A_2\, ((P_2)^T);$$

В результаті отримали тридіагональну симетричну матрицю A_2:

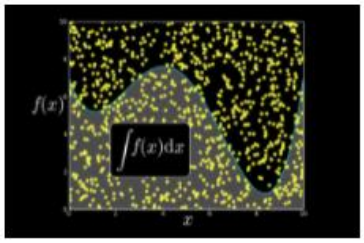
$$P_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.26 & 0.97 \\ 0 & 0 & 0.97 & 0.26 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} 1 & -1.73 & 0 & 0 \\ -1.73 & 4.33 & 0.94 & 0 \\ 0 & 0.94 & 0.67 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Розділ 2. Опис CUDA

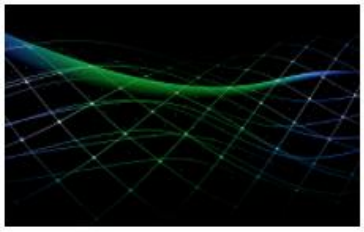
2.1 Загальні відомості

CUDA – це платформа для паралельних обрахунків та програмна модель, розроблена NVIDIA для обчислень на графічних процесорах (GPUs). Завдяки цій платформі, розробники можуть істотно пришвидшити розрахунки, використовуючи можливості GPU. CUDA постачається з програмним середовищем, яке дозволяє використовувати написані на C++ бібліотеки. Проте, вже є багато бібліотек розроблених під інші мови, про яку детальніше йтиметься в наступній частині цього розділу. На рисунку 1 можна побачити популярні доступні бібліотеки:



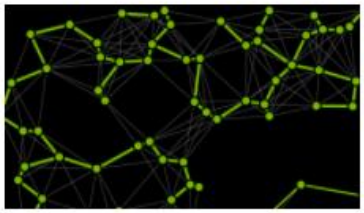
cuRAND

NPP

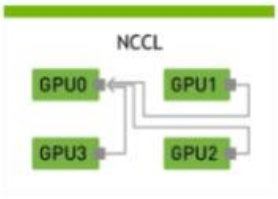


Math Library

cuFFT



nvGRAPH



NCCL

Рис 1. Бібліотеки CUDA [3].

В цій роботі використовувались математичні бібліотеки. Ось загальний перелік доступних бібліотек для математичних обрахунків в CUDA:

1. **cuBLAS**. Бібліотека базової лінійної алгебри.
2. **cuFFT**. Функції для обрахунків швидкого перетворення Фур'є. Працює до 10 разів швидше, ніж реалізації на CPU.
3. **CUDA Math Library.** Стандартні математичні функції на GPU.
4. **cuRAND.** Генерування випадкових чисел на GPU.
5. **cuSOLVER.** Алгоритми для обрахунків на щільних та розріджених марицях.
6. **cuSPARSE.** Алгоритми базової лінійної алгебри оптимізовані під розріджені матриці.
7. **cuTENSOR.** Тензорні обрахунки.
8. **AmgX.** Алгоритми для симуляцій та неструктурованих методів.

2.2 Переваги GPU для паралельних обчислень

Графічний процесор (GPU) забезпечує набагато більшу пропускну здатність інструкцій та пропускну здатність пам'яті, ніж центральний процесор, за аналогічного рівня ціни та потужності. Багато програм використовують ці вищі можливості для швидшої роботи на графічному процесорі, ніж на центральному процесорі. Інші обчислювальні пристрої, такі як FPGA, також дуже енергоефективні, але пропонують значно меншу гнучкість програмування, ніж графічні процесори. Ця різниця у можливостях між GPU та CPU існує, оскільки вони розроблені з урахуванням різних цілей. Незважаючи на те, що центральний процесор призначений для досягнення найвищої швидкості в виконанні послідовності операцій, яка називається потоком, і може виконувати кілька десятків цих потоків паралельно,

графічний процесор призначений для досягнення успіху при паралельному виконанні тисяч з них (повільніша однопотокова продуктивність для досягнення більшої пропускної здатності). GPU спеціалізується на паралельних обчисленнях і тому розроблений таким чином, що більше транзисторів приділяється обробці даних, а не кешуванню даних та контролю потоку. Схема на рисунку 2 показує приклад розподілу ресурсів мікросхеми центрального та графічного процесорів [3].

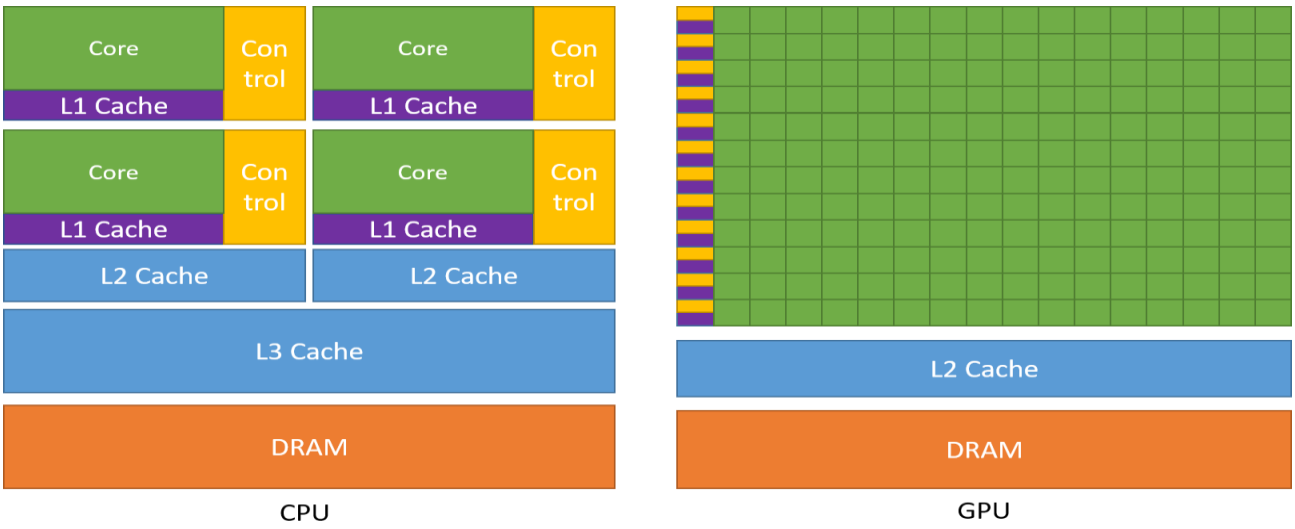


Рис 2. Порівняння мікросхем CPU та GPU [3].

2.3 Архітектура

Архітектура обчислень на платформі CUDA оперує поняттями ядра, потоку, блоку та сітки.

Сітка (grid) містить в собі всі дані, які необхідно обробити. Сітка знаходиться на вершині ієрархії потоків. Вона може містити в собі декілька блоків та потоків. Також сітка підтримує до трьох можливий розмірностей, тобто вона може бути одновимірною, двовимірною або тривимірною. Зазвичай розмір сітки відповідає розмірності даних або складності роботи, яку необхідну виконати. Наприклад, для матриць розмірність буде рівна двом.

Кожна сітка може містити в собі декілька блоків. Максимальна кількість блоків обмежена для кожної розмірності і залежить від конкретного графічного процесора, який використовується для обчислень. Розмірність блоків відповідає розмірності сітки.

Кожен блок може містити в собі декілька потоків. Максимальна кількість потоків обмежена характеристиками обладнання, зазвичай це 1024 потоки. Розмірність залежить від розмірності блока. Важливо зазначити, що кожен блок має бути незалежним один від одного. Це пов’язано з тим, що програмний код в блоці може виконуватись на різних пристроях, що не можуть спілкуватись між собою або ж спілкування займає велику частину часу. Також блоки можуть виконуватись як послідовно, так і паралельно.

Ядром (kernel) називається функція, що виконується одним потоком CUDA. Ядро виконується окремо на різних потоках паралельно. Кожен такий виконавчий потік має свій ідентифікатор, доступ до якого можна отримати всередині ядра.

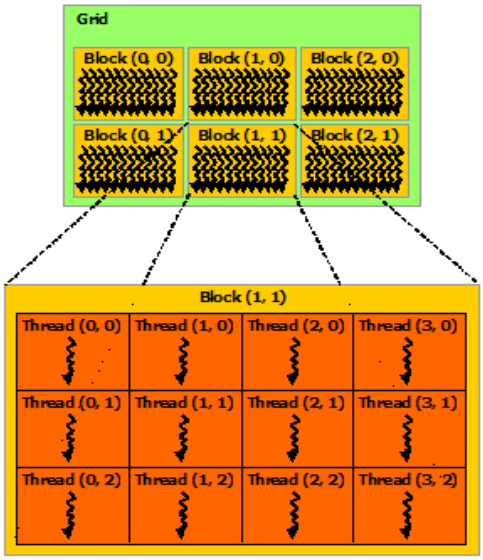


Рис 3. Архітектура CUDA [3].

Варто зазначити, що платформа CUDA не надає інструментів для синхронізації на рівні сітки для більшості графічних процесорів. Такі інструменти доступні лише на деяких нових графічних процесорах. Всі обчислення необхідно планувати, виходячи з відсутності комунікації між блоками.

Причина відсутності інструментів міжблочної синхронізації полягає у тому, що не всі блоки виконуються одночасно. Графічний процесор самостійно формує чергу з блоків і може виконувати обчислення у випадковому порядку, відповідно до наявних ресурсів – завантаженості процесора, вільної пам’яті тощо. У випадку, якщо один блок намагатиметься очікувати результат виконання іншого блоку, програма може зависнути (так званий deadlock) - оскільки немає гарантії, що очікуваний блок буде виконано до завершення поточного блоку.

Водночас, комунікація між потоками всередині блоку допускається. Функція __syncthreads() дозволяє всім потокам всередині блоку синхронізуватися у заданій точці під час виконання. Фактично, це єдиний надійний спосіб синхронізації обчислень без необхідності очікувати на їх завершення.

2.4 Програмна модель

Технологія CUDA (компілятор nvcc.exe) вводить ряд додаткових розширень для мови C, які необхідні для написання коду для GPU:

1. Специфікатори функцій, які показують, як і звідки буду виконуватися функції.
2. Специфікатори змінних, які служать для вказівки типу використовуваної пам'яті GPU.
3. Специфікатори запуску ядра GPU.

4. Вбудовані змінні для ідентифікації потоків, блоків інших параметрів при виконанні коду в ядрі GPU.
5. Додаткові типи змінних.

Як було сказано, специфікатори функцій визначають, як і звідки будуть викликатися функції.

- *__host__* - виконується на CPU, викликається з CPU.
- *__global__* - виконується на GPU, викликається з CPU.
- *__device__* - виконується на GPU, викликається з GPU.

Специфікатори запуску ядра служать для опису кількості блоків, потоків і пам'яті, яку ви хочете виділити при розрахунку на GPU. Синтаксис запуску ядра має наступний вигляд:

*myKernelFunc <<< gridSize, blockSize, sharedMemSize, cudaStream >>> (float * param1, float * param2),*

де *gridSize* - розмірність сітки блоків (dim3), виділеної для розрахунків,

blockSize - розмір блоку (dim3), виділеного для розрахунків,

sharedMemSize - розмір додаткової пам'яті, що виділяється при запуску ядра,

cudaStream - змінна cudaStream_t, що задає потік, в якому буде проведений виклик.

Ну і звичайно сама *myKernelFunc* - функція ядра (специфікатор *__global__*). Деякі змінні при виклику ядра можна опускати, наприклад *sharedMemSize* і *cudaStream*.

Так само варто згадати про вбудовані змінні:

gridDim - розмірність гріда, має тип dim3. Дозволяє дізнатися розмір гріда, виділеного при поточному виклику ядра.

blockDim - розмірність блоку, так само має тип dim3. Дозволяє дізнатися розмір блоку, виділеного при поточному виклик ядра.

blockIdx - індекс поточного блоку в обчисленні на GPU, має тип uint3.

threadIdx - індекс поточного потоку в обчисленні на GPU, має тип uint3.

warpSize - розмір warp'a, має тип int.

2.5 Приклад програми

Наступний зразок коду, використовуючи вбудовану змінну *threadIdx*, додає два вектори *A* і *B* розміром *N* і зберігає результат у векторі *C*. Тут кожен з *N* потоків, що виконують *VecAdd ()*, виконує одне парне додавання.

```
// визначення ядра
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    ...
    // виклик ядра з N потоками
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

2.6 Приклади застосування

Графічні процесори NVIDIA CUDA використовують у багатьох областях, що потребують високих обчислювальних характеристик з плаваючою крапкою. Перелік включає:

- Моделювання клімату, погоди та океану

- Data science та аналітика
- Deep learning та machine learning
- Оборона та розвідка
- Виробництво AEC (архітектура, інжиніринг та будівництво): CAD та CAE (включаючи обчислювальну динаміку рідини, обчислювальну механіку конструкцій, проектування та візуалізацію, а також електронну автоматизацію проектування)
- Медіа та розваги (включаючи анімацію, моделювання та візуалізацію; корекція кольорів та управління зернистістю; обробка та ефекти; редагування; кодування та цифровий розподіл; ефірна графіка)
- Медицина
- Фінанси
- Кібербезпека
- Дослідження: обчислювальна хімія та біологія, чисельна аналітика та фізика

2.7 JCUDA

В цій роботі для реалізації програми я використовував jCUDA – Java прив’язки для CUDA. Проект спрямований на підтримку роботи з середовищем CUDA для Java програм. Це проект з відкритим вихідним кодом, в якому реалізовані найпопулярніші бібліотеки: jCuBLAS – бібліотека з алгоритмами базової лінійної алгебри, яка використовувалась в цій роботі найбільше; jCuFFT, jCuSPARSE, jCuSolver та інші [12].

Розділ 3. Опис практичної частини

3.1 Послідовний алгоритм на CPU

Для подальших порівнянь та досліджень описаний в першому розділі алгоритм було реалізовано мовою Java в двох варіантах:

- Послідовний алгоритм на центральному процесорі
- За допомогою jCUDA на графічному процесорі

В бібліотеці mathpar було додано новий клас SVD, в якому реалізовано метод *bidiagonalize(MatrixD A, Ring ring),* що приймає на вхід матрицю *A* та змінну типу Ring, яка визначає алгебраїчний простір поточних змінних. Клас *MatrixD* має готові функції для роботи з матрицями (транспонування, обернення тощо). На виході функція повертає масив з трьома об'єктами MatrixD: U – лівий множник, D – бідіагональна матриця, W – правий множник. Матриця A, що передається в функцію заповнюється згенерованими числами типу Double в діапазоні від 1 до 2. Ring виставляється R64 – для роботи з дійсними числами.

На початку методу визначається матриця ***I*** – одинична матриця. Після цього, визначається потрібна кількість операцій для приведення матриці до бідіагонального вигляду. Формула *N – 1*, де *N* – кількість рядків або стовпчиків квадратної матриці *N * N*. Далі визначаємо три об'єкти типу *ArrayList<MatrixD>* - для того, щоб зберігати проміжні результати для матриць *A, P* та *Q*, де *A* – матриця яку приводимо до бідіагонального вигляду, *P* – матриця множник зліва, *Q* – матриця множник справа. Потім, запускаємо ітеративний процес, який під час однієї операції виконує "удари" (множення) і справа і зліва, обнуляючи спочатку стовпчик, а потім рядок матриці *A*. Для роботи з векторами, використано клас *VectorS*. Цей клас надає можливість обраховувати норму вектора, множити його на скаляр тощо. Також, в програмі

використовується клас Element, який також використовується і в MatrixD і в VectorS, що надає можливість без додаткового програмування виконувати додавання, множення та інші операції між об’єктами цих класів. Для того, щоб не виконувати зайвих операцій, виконується перевірка чи *norm2* вектора рівна нулю. Якщо так, тоді нам не потрібно виконувати усіх множень та інших операцій, а лише взяти одиничну матрицю як поточний множник. Ця логіка виконується для обох частин (обнулення стовпчика та рядка). Сама процедура бідіагоналізації оптимізована таким чином, що нам не потрібно множити матриці, а лише вектори між собою, вектори та матриці між собою, матрицю на скаляр. Множення матриць потрібне лише для пошуку множників, а також, в кінці всієї процедури для перевірки правильності обчислень. Якщо обчислення правильні, то перемноживши отримані в результаті матриці і віднявши під початкової, ми повинні отримати нульову матрицю (значення можуть бути наближені до нуля, залежно від похибки обчислень).

3.2 Реалізація алгоритму на jCUDA (GPU-based).

3.2.1 Опис програми

Програма під відеокарту була розроблена за схожим алгоритмом, проте, усі функції підлаштовані та оптимізовані під виконання на графічному процесорі. Перш за все, програма виконується таким чином, що контроль виконання (виклик функцій та розгалуження логіки) виконується з центрального процесора, який оптимізований під виконання потоку операцій. А усі інтенсивні обчислення виконуються на відеокарті. При цьому, кожна функція може виконуватись з різною розмірністю сітки та блоку. Це і є перевагою виклику функцій з хоста. В випадку, коли вся логіка відбувається в функції на GPU, ми прив’язані до однієї кількості блоків та потоків, яку важко контролювати. Проте, є алгоритми де такий підхід навпаки буде оптимальним,

наприклад другий етап SVD за Хаусхолдером, де на другому етапі виконується ітеративний процес приведення матриці до діагонального вигляду.

Програма складається з класу SVD, в якому реалізовано метод *biDiagonalize(double[] m),* який приймає на вхід представлену в вигляді стрічки (вектора) матрицю з числами типу Double. На виході, метод повертає масив об'єктів типу Pointer – вказівники на об'єкти, розташовані на відеокарті. Більшість методів використано з бібліотеки JCublas, яка надає готові алгоритми та функції для базової лінійної алгебри. Перед початком обчислень, клас ініціалізується з об'єктом типу *cublasHandle*, відповідальним за виконання певного типу функцій та доданих самостійно кернелів. Процедура ініціалізації виглядає ось так:

```
JCublas.cublasInit();
cublasHandle cublasHandle = new cublasHandle();
cublasCreate(cublasHandle);
SVD svd = new SVD(cublasHandle);
```

Сама ж матриця генерується та заповнюється випадковими числами типу Double в діапазоні від 1 до 2. Далі, на початку методу бідіагоналізації, ініціалізуються написані самостійно кернели (функції, що виконуються на відеокарті). Ось перелік кернелів: *getMatrixColumn, getMatrixRow, applyNorm, applyNormDiv, calculateNormDiv.* Детальніше, ці функції будуть описані далі в цьому розділі. Після ініціалізації кастомних кернелів, йде визначення кількості операцій необхідних для приведення матриці до дводіагонального вигляду, що в даному випадку, обраховується за формулою *Math.sqrt(m.length) – 1*. Потім, ініціалізуються об'єкти класу Pointer, які слугуватимуть доступом до об'єктів на відеокарті. Перелік необхідних для обрахунків вказівників:

```
Pointer pAlpha = Pointer.to(new double[] { 1 });
Pointer pBeta = Pointer.to(new double[] { 1 });
Pointer d_v = new Pointer(),
        d_u = new Pointer(),
```

```
        d_normDiv = new Pointer(),
        d_d = new Pointer(),
        d_uut = new Pointer(),
        d_utu = new Pointer(),
        d_Pi = new Pointer(),
        d_U = new Pointer(),
        d_Qi = new Pointer(),
        d_W = new Pointer(),
        d_I = new Pointer(),
        d_dT = new Pointer(),
        d_udT = new Pointer(),
        d_du = new Pointer(),
        d_Ai = new Pointer();
Pointer[] matrixPointers = new Pointer[] {d_udT, d_du, d_uut, d_utu, d_Pi,
d_Qi, d_I};
Pointer[] vectorPointers = new Pointer[] {d_dT, d_d, d_v, d_u};
```

Вказівники об’єднано в масиви для матриць та векторів, оскільки для цих типів вказівників виділяється різна кількість пам’яті. За допомогою цих масивів виділяти та очистити пам’ять стає набагато простіше. Проте, не всі вказівники додано до масивів, оскільки нам не потрібно очищати пам’ять для вказівників *d_A*, *d_U* та *d_W*, оскільки це вказівники на три матриці, що є результатом методу бідіагоналізації. Префікс *d_* вказує, що це вказівник на об’єкт, розташований на GPU.

Після ініціалізації усіх вказівників, виділяємо пам’ять під об’єкти на які вони будуть вказувати на відеокарті, викликавши метод *cublasAlloc*.

```
JCublas.cublasAlloc(N * N, Sizeof.DOUBLE, d_U);
JCublas.cublasAlloc(N * N, Sizeof.DOUBLE, d_Ai);
JCublas.cublasAlloc(N * N, Sizeof.DOUBLE, d_W);
JCublas.cublasAlloc(1, Sizeof.DOUBLE, d_normDiv);

for (Pointer matrixPointer : matrixPointers) {
    JCublas.cublasAlloc(N * N, Sizeof.DOUBLE, matrixPointer);
}

for (Pointer vectorPointer : vectorPointers) {
    JCublas.cublasAlloc(N, Sizeof.DOUBLE, vectorPointer);
}
```

Метод *cublasAlloc* приймає кількість елементів під яку треба виділити пам’ять, тип та вказівник на перший елемент масиву.

Далі, запускається ітеративний процес, який під час однієї ітерації обнуляє стовпчик та рядок матриці. На кожному етапі ми обнуляємо *N – (i+ 1)* елементів в стовпці *i* та *N – (i + 2)* елементів в рядку *i*. Якщо всі елементи вже і так нулі, тоді, щоб не виконувати зайвих операцій, виконується перевірка норми поточного вектору (стовпчика або рядка) і якщо вона дорівнює 0, тоді операції виконувати вже не потрібно. Норму обраховуємо ось таким чином:

```
double nrm2 = JCublas.cublasDnrm2(N, d_v, 1);
```

Тут викликається функція *cublasDnrm2* яка приймає на вхід розмірність вектору, вказівник на вектор та кількість місця між елементами вектору. Самі ж стовпчики та рядки (вектори) зчитуються за допомогою кернелів *getMatrixColumn* та *getMatrixRow*. Для кожного рядка та стовпчика нам також потрібно рахувати вектор з застосованою нормою. Для цього, спочатку копіюємо вектор на відеокарті з одного місця в інше використовуючи функцію cublasDcopy.

```
JCublas.cublasDcopy(N, d_v, 1, d_u, 1);
```

Символ *D* перед *copy* означає, що операція виконується для чисел типу Double. А потім, викликаємо кернел applyNorm, який до відповідного елементу додає або віднімає норму. Після цього, відбувається множення вектору рядка, на вектор стовпчик, щоб отримати матрицю, за допомогою функції *cublasDgemm:*

```
JCublas.cublasDgemm('N', 'N', N, N, 1, 1, d_u, N, d_u, 1, 0, d_uut, N);
```

Тут ми передаємо тип матриці ‘N’ або ‘T’ (transposed), кількість рядків першої матриці, кількість стовпчиків другої, а третім параметром йде однакова кількість стовпчиків першої та кількість рядків другої матриць. Потім вказівники на самі матриці, інформація про лідируючу розмірність (в *N x 1* буде N, а в *1 x N* буде 1), параметри альфа та бета і вказівник, куди записувати

результат. Після того, як ми отримаємо необхідну матрицю, викликаються кернели calculateNormDiv та applyNormDiv які обраховують розширену норму за представленою в описі алгоритму в першому розділі формулою, та застосовують її до отриманої перед цим матриці. Потім виконується функція *cublasDgeam* яка додає до одиничної матриці помножену на -1 під час функції застосування норми матрицю uut (UU^T). Результатом буде матриця *Pi* яка є лівим множником. Для того, щоб не рахувати все в кінці та не зберігати усі проміжні множники, одразу ж виконується їх перемноження і результат зберігається в матриці *U*:

```
JCublas.cublasDgemm('N', 'N', N, N, N, 1, d_Pi, N, d_U, N, 0, d_U, N);
```

Після обрахунку лівого множника, відбувається обнулення стовпчика. Проте, в оптимізованому вигляді нам не потрібно буде виконувати операцію множення матриць. Для цього, спочатку виконується множення матриці на вектор з застосованою нормою:

```
JCublas.cublasDgemm('N', 'T', 1, N, N, 1, d_u, 1, d_Ai, N, 0, d_dT, 1);
```

Далі, застосовуємо розширену норму до вектору *u* і множимо його на вектор результат попереднього множення, отримавши в результаті матрицю, які віднімаємо від останнього стану матриці *A*:

```
applyNormDiv(applyNormDivFunction, d_u, d_normDiv, N);
JCublas.cublasDgemm('N', 'N', N, N, 1, 1, d_u, N, d_dT, 1, 0, d_udT, N);
JCublas2.cublasDgeam(_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_T, N, N, pAlpha,
d_Ai, N, pBeta, d_udT, N, d_Ai, N);
```

На цьому, частина пов’язана зі стовпчиками закінчується і починається частина з рядками. Там все відбувається за таким же сценарієм. Операції ті ж, тільки застосовуються до інших елементів і порядок множення відрізняється. Детальніше можна побачити в третьому підрозділі першого розділу.

В кінці алгоритму, ми очищаємо виділену пам’ять на відеокарті за допомогою методу *cublasFree,* яикй приймає вказівник:

```
for (Pointer matrixPointer : matrixPointers) {
    JCublas.cublasFree(matrixPointer);
}

for (Pointer vectorPointer : vectorPointers) {
    JCublas.cublasFree(vectorPointer);
}

JCublas.cublasFree(d_normDiv);
```

Програма завершується функцією перевірки, яка перемножує отримані матриці між собою, та віднімає результат від початкової матриці. Якщо обчислення проведено правильно, елементи будуть нульові. Проте, на великих рохмірах матриці елементи лише наближені до нуля, через накопичення похибки. Перевірка елементів відбувається за допомогою реалізованого самостійно кернела ensureAllElementsAreZeros.

Після усіх перевірок, очищається пам’ять виділена під три основні матриці, та знищується cublasHandle:

```
for (Pointer p: res) {
    JCublas.cublasFree(p);
}

cublasDestroy(svd._cublasHandle);
JCublas.cublasShutdown();
```

На цьому програма закінчується.

3.2.2 Опис кернелів

3.2.2.1 initIdentityMatrix

Функція ініціалізації одиничної матриці. Виконується на багатьох потоках та блоках паралельно, тому на початку ми визначаємо індекс, враховуючи розмірність та індекс блоку, а також поточного потоку.

```
extern "C"
__global__ void initIdentity(double *m, int n) {
       int i = blockIdx.x * blockDim.x + threadIdx.x;
       if (i < n * n)
       {
           m[i] = i % (n + 1) == 0 ? 1 : 0;
       }
   }
```

3.2.2.2 getMatrixRow

Функція зчитування рядка матриці. Виконується на сітці великої розмірності з фіксованим розміром блоку – 256. Розмірність сітки визначається за формулою: `(int) Math.ceil((double) (N * N) / blockSizeX).`

```
extern "C"
__global__ void getMatrixRow(int n, int row, int fromIndex, double *m, double *re
s) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n * row && i < n * (row + 1))
    {
        int index = i - (n * row);
        res[index] = index >= fromIndex ? m[i] : 0;
    }
}
```

3.2.2.3 getMatrixColumn

Функція зчитування стовпчика матриці. Виконується на сітці великої розмірності з фіксованим розміром блоку – 256.

```
extern "C"
__global__ void getMatrixColumn(int n, int col, int fromIndex, double *m, double*
res) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if ((i - col) % n == 0)
    {
        int index = (i - col) / n;
        res[index] = index >= fromIndex ? m[i] : 0;
    }
}
```

3.2.2.4 applyNorm

Функція застосування норми до вектору. Застосовується до конкретного елементу вектору за індексом. Виконується на одному блоці з одним потоком.

```
extern "C"
__global__ void applyNorm(double *v, double norm, int index)
{
    v[index] = v[index] < 0 ? v[index] - norm : v[index] + norm;
}
```

3.2.2.5 applyNormDiv

Функція застосування розширеної норми, обрахованої на GPU. Застосовується до кожного елементу вектору (або матриці). Виконується на сітці великого розміру з фіксованою кількістю потоків в одному блоці – 256.

```
extern "C"
__global__ void applyNormDiv(double *v, double* normDiv, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n)
    {
        v[i] = -1 * normDiv[0] * v[i];
    }
}
```

3.2.2.6 calculateNormDiv

Функція обчислення розширеної норми на відеокарті. Виконується на одному блоці з одним потоком.

```
extern "C"
__global__ void calculateNormDiv(double *v, double nrm2, double* res, int index)
{
    res[0] = 2 / (2 * nrm2 * (v[index] < 0 ? nrm2 - v[index] : nrm2 + v[index]));
}
```

3.2.2.7 ensureAllElementsAreZeros

Функція, яка виконує перевірку елементів на наближеність до нуля. Виконується на сітці великого розміру з фіксованою кількістю потоків для блоку – 256. В випадку, якщо хоча б один елемент не проходить перевірку, в відповідну змінну *status* ставиться код про помилку -1.

```
extern "C"
__global__ void ensureAllElementsAreZeros(double *v, double tolerance, int n, int
* status)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n && status[0] >= 0)
    {
        if (v[i] < 0 && -1 * v[i] > tolerance)
        {
            status[0] = -1;
        }
        else
        {
            if (v[i] > tolerance)
            {
                status[0] = -1;
            }
        }
    }
}
```

Розділ 4. Експерименти та результати

4.1 Загальні відомості

Експерименти проводились на двох різних графічних процесорах та на центральному процесорі. Основну увагу приділено дослідженню швидкості виконання на різних пристроях, за різних розмірів матриці, для визначення часової складності та прискорення відносно інших алгоритмів.

4.2 Параметри пристроїв

4.2.1 NVIDIA GeForce MX150

Графічний процесор: GP108

Кількість CUDA ядер: 384

Пам’ять: 2048MB GDDR5

Архітектура: Pascal

Bus width: 64 bit

Пропускна здатність пам’яті: 48 GB/s

Тактова частота пам’яті: 1502MHz

Спільна пам’ять на блок: 48 KB

Глобальна пам’ять: 2GB

Максимальна кількість потоків в блоці: 1024

4.2.2 NVIDIA GeForce GTX 1660 Ti

Графічний процесор: TU116

Кількість CUDA ядер: 1536

Пам’ять: 6144 MB GDDR6

Архітектура: Turing

Bus width: 192 bit

Пропускна здатність пам’яті: 288 GB/s

Тактова частота пам’яті: 1500MHz

Спільна пам’ять на блок: 48 KB

Глобальна пам’ять: 6GB

Максимальна кількість потоків в блоці: 1024

4.2.3 Dell Latitude 5401
Процесор: Intel(R) Core(TM) i7-9850H CPU @ 2.60GHz 2.59 GHz

ОЗУ: 16.0 GB

Тип системи: x64

Операційна система: Windows 10 Pro

4.3 Результати

Перш за все було перевірено залежність часу виконання від розміру матриці для послідовного алгоритму, та алгоритму на відеокарті. Розміри матриці брались від 4 до 1024 (зі збільшенням степеню двійки). Послідовний алгоритм вдалось протестувати до розміру матриці 256 (включно). 512 вже завершився з помилкою про нестачу пам’яті. На відеокарті ж, усі розміри спрацювали добре, навіть вдалось перевірити на розмірі 2048, де час виконання алгоритму бідіагоналізації склав 31 хвилину. Але цей тест проводився на першому описаному пристрої (MX 150), який є дуже простим графічним процесором. На більш потужних машинах час повинен бути меншим. Цей час не було додано на графік, оскільки він є аутлаєром в даному випадку. Нижче можна побачити таблицю та графік з порівнянням часу виконання.

Розмір матриці	4	8	16	32	64	128	256	512	1024	2048
Послідовний алгоритм	2 мс	5 мс	16 мс	90 мс	745 мс	9.4 с	183.635 с	N/A	N/A	N/A

GPU	2 мс	6 мс	8 мс	16 мс	31 мс	107 мс	711 мс	9.109 с	119.769 с	31 хв
Покращення (в *n* разів)	0	-0.17	2	5.625	24.03	87.85	258.27	N/A	N/A	N/A

Таблиця 1. Порівняння послідовного алгоритму та GPU алгоритму

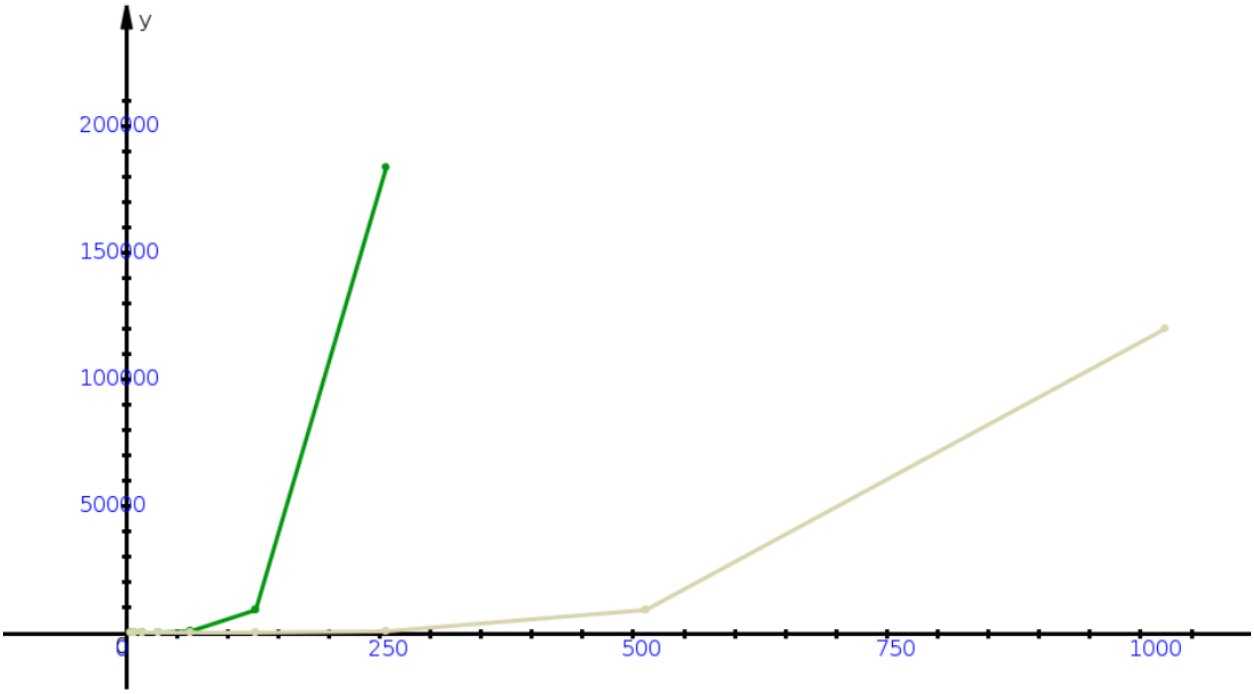


Рис 4. Час виконання послідовного алгоритму та алгоритму на GPU

Також, було проведено порівняння алгоритму на різних графічних процесорах. Тут вже було додано обчислення на розмірі 2048, і як і очікувалось, на потужнішому графічному процесорі результат значно кращий – 7 хвилин. Нижче наведено таблицю та графік порівняння:

Розмір матриці	4	8	16	32	64	128	256	512	1024	2048
MX 150	2 мс	6 мс	8 мс	16 мс	31 мс	107 мс	711 мс	9.109 с	119.769 с	31 хв
GTX 1660 Ti	2 мс	8 мс	7 мс	14 мс	33 мс	89 мс	287 мс	2.045 с	28.688 с	7.05 хв
Покращення (в *n* разів)	0	-0.25	1.14	1.14	-0.07	1.2	2.47	4.45	4.17	4.42

Таблиця 2. Порівняння часу виконання алгоритму на різних графічних процесорах

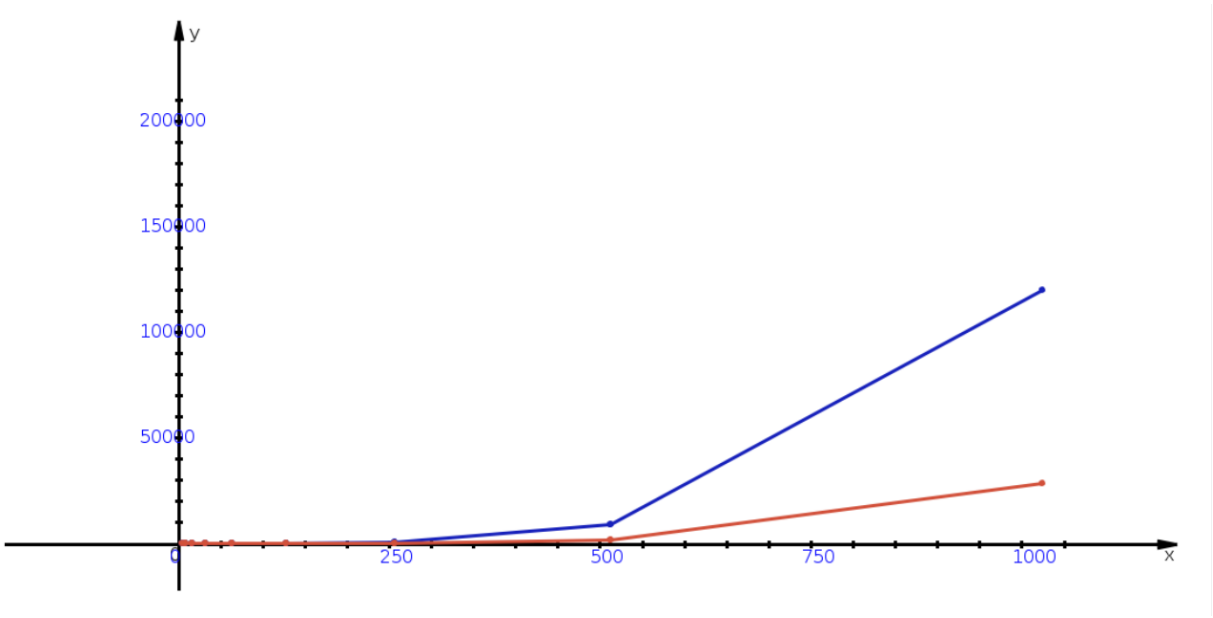


Рис 5. Порівняння часу виконання на MX 150 та GTX 1660 Ti

Як бачимо, GTX 1660 Ti (червона крива) за рахунок більшої кількості CUDA ядер, більшої пропускної здатності пам'яті, та більшого розміру пам'яті виграє. При чому, чим більший розмір матриці, тим помітніша є різниця.

Досить цікавим та незвичним є порівняння накопичення похибки на графічному та центральному процесорах.

Розмір матриці	4	8	16	32	64	128	256	512	1024
CPU	0.4	0.5	0.2	0.19	0.14	0.12	0.03	N/A	N/A
GTX 1660 Ti	$1.3 \cdot 10^{-15}$	$1.99 \cdot 10^{-15}$	$4.4 \cdot 10^{-15}$	$3.1 \cdot 10^{-15}$	$4.8 \cdot 10^{-15}$	1.99	1.95	5.6	6.8

Таблиця 3. Порівняння похибок обчислень на CPU та GPU

Похибка обраховувалась знаходженням максимального абсолютного значення серед елементів матриці перевірки. Дуже цікавим є той факт, що похибка на CPU зі збільшенням розміру матриці ставала меншою. Графік виглядає відповідно незвично:

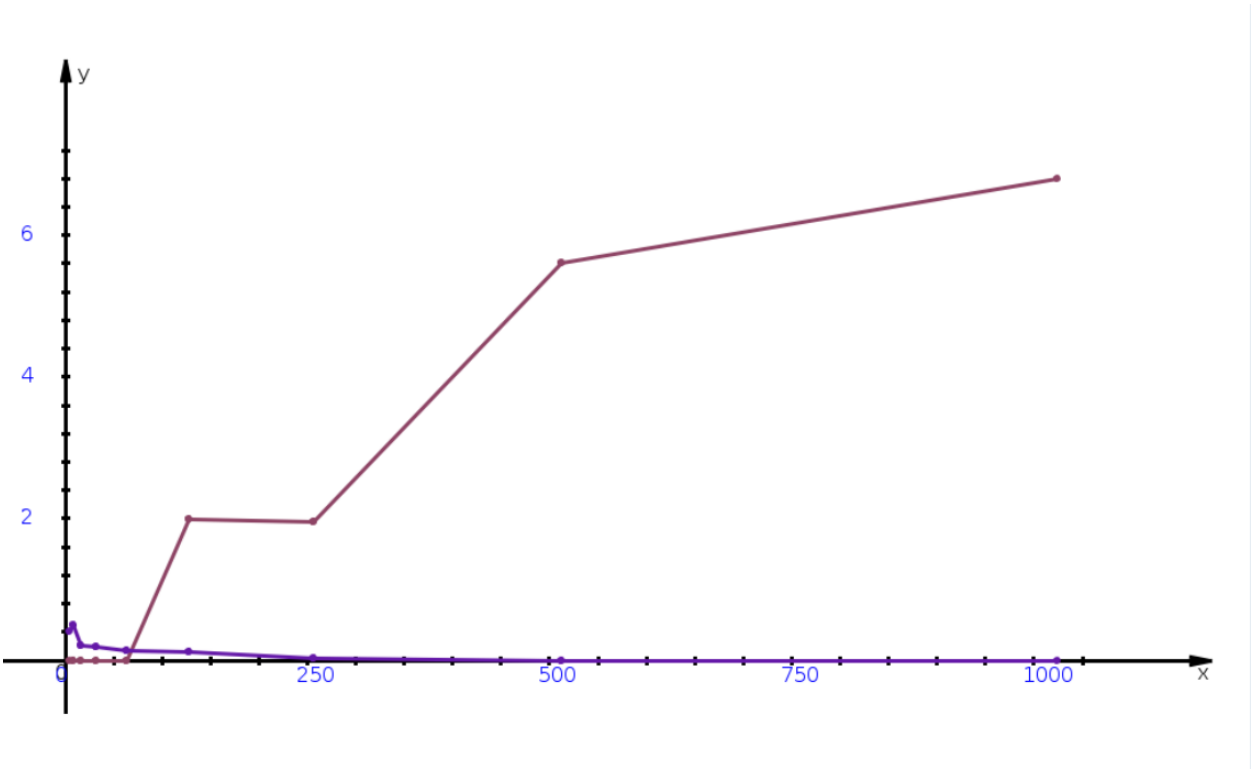


Рис 6. Порівняння похибки обчислень на GPU та CPU

4.4 Підсумки

Як видно з результатів, прискорення обчислень відносно CPU збільшується зі збільшенням розміру матриці. Таким чином, на розмірі матриці 256 прискорення складає 258.27! Якщо порівнювати з потужнішим процесором, то воно відповідно буде ще більшим. Але звісно цікавішим буде визначення прискорення відносно інших алгоритмів. Ці обчислення можна буде провести після реалізації другого етапу SVD за Хаусхолдером, який продемонстрував дуже хороші результати на першому етапі.

Висновки

Було досліджено алгоритм Хаусхолдера для бідіагоналізації матриці та тридіагоналізації симетричної матриці. Після цього цей алгоритм було реалізовано на графічному процесорі за допомогою платформи CUDA та середивища JCuda мовою Java. Крім цього, було реалізовано послідовний алгоритм мовою Java для проведення порівняльного аналізу.

Проведено дослідження швидкості виконання залежно від різних розмірів вхідної матриці на відеокарті та на центральному процесорі. Максимальний розмір матриці був 2048 для відеокарти, та 256 для центрального процесора. Також досліджувалась похибка обчислень. Результати показали значну перевагу в швидкості обчислень на графічному процесорі, проте, накопичення похибки також було більшим саме на відеокарті. Також між собою було порівняно дві відеокарти різних потужностей, і на кращій з них ефективність обчислень була ще вищою. Метод Хаусхолдера показав гарні результати і весь агоритм SVD з використанням цього методу виглядає багатообіцяючим. Дослідження будуть проведені після реалізації другого етапу – діагоналізації отриманої дво- або тридіагональної матриці.

Список використаних джерел

1. S. Lahabar and P. J. Narayanan, "Singular value decomposition on GPU using CUDA," 2009 IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1-10, doi: 10.1109/IPDPS.2009.5161058. [Електронний ресурс]. – режим доступу: https://ieeexplore.ieee.org/document/5161058/citations?tabFilter=papers#citations. Дата звертання: 2020.10.29
2. Малашонок Г. І., Савченко С. О., “Матричні алгоритми розбиття множин для рекомендаційних систем”. НаУКМА, 2019.
3. Документація NVIDIA CUDA. [Електронний ресурс]. – режим доступу: https://developer.nvidia.com/cuda-zone. Дата звертання: 2020.11.18
4. J. Lambers, “The SVD Algorithm”, CME 335, Lecture 6. [Електронний ресурс]. – режим доступу: https://web.stanford.edu/class/cme335/lecture6.pdf. Дата звертання: 2021.02.03
5. Persson, “Householder Reflectors and Givens Rotations”, MIT 18.335J / 6.337J *Introduction to Numerical Methods.* [Електронний ресурс]. – режим доступу: https://math.dartmouth.edu/~m116w17/Householder.pdf. Дата звертання: 2020.12.05
6. Cornell University, “Numerical linear algebra and matrix factorizations”, [Електронний ресурс]. – режим доступу: http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/Householder.html. Дата звертання: 2020.12.05
7. Malaschonok G., Gevondov G. Quick triangular orthogonal decomposition of matrices // International Conference Polynomial Computer Algebra. 2019. pp. 89 - 93.
8. Demmel J., Kahan W. Accurate Singular Values of Bidiagonal Matrices // SIAM J. Sci. Stat. Comput., v. 11, n. 5, 1990. pp. 873-912.
9. Малашонок Г. І. Хмарна математика MathPartner у Києво-Могилянській академії // Наукові записки НаУКМА. 2017. Том 198. с. 27 – 35.
10. Документація бібліотеки CUBLAS. [Електронний ресурс]. – режим доступу: https://docs.nvidia.com/cuda/cublas/index.html . Дата звертання: 2021.04.15
11. M. Heller, “What is CUDA? Parallel programming for GPUs”, 2018. . [Електронний ресурс]. – режим доступу:

https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html. Дата звертання: 2020.11.02
12. Опис проекту JCuda. [Електронний ресурс]. – режим доступу: http://jcuda.org/. Дата звертання: 2021.03.10
13. Специфікації відеокарт. [Електронний ресурс]. – режим доступу: https://www.techpowerup.com/. Дата звертання: 2021.05.01