Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Факультет інформатики

Кафедра мережних технологій

Магістерська робота

Освітній ступінь: магістр

На тему: «ОБЧИСЛЕННЯ СИНГУЛЯРНОГО РОЗКЛАДУ МАТРИЦЬ З ВИКОРИСТАННЯМ ГРАФІЧНОГО ПРОЦЕСОРА»

Виконав: студент 2 року навчання

Спеціальності

122 Комп’ютерні науки

Сухарський Сергій Сергійович

Керівник

Малашонок Г. І.

проф., д. ф.-м. н.

Рецензент С. С. Гороховський

Магістерська робота захищена

з оцінкою_____

Секретар ЕК С.А. Мелещенко

«___» _____2022

Київ 2022

Тема: Обчислення сингулярного розкладу матриць з використанням графічного процесора

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на магістерську роботу.	14.10.2021	
2.	Огляд технічної літератури за темою роботи.	14.11.2021	
3.	Аналіз алгоритму Хаусходера для приведення матриці до бідіагонального вигляду	25.11.2021	
3.	Розробка алгоритму діагоналізації матриці	30.12.2021	
4.	Реалізація алгоритмів на графічному процесорі.	20.04.2020	
5.	Тестування розробленої програми.	01.05.2020	
6.	Аналіз, тестування та виправлення помилок.	15.05.2020	
8.	Написання пояснювальної роботи.	05.05.2020	
9.	Аналіз отриманих результатів з керівником.	20.06.2022	
10.	Коригування роботи за результатами попереднього захисту.	30.06.2022	
11.	Підготовка презентації та доповіді.	01.07.2022	
12.	Захист магістерської роботи.	06.07.2022	

Студент Сухарський С. С.

Керівник Малашонок Г.І.

“_____”_____

Зміст

Анотація

У роботі розглянуто та реалізовано алгоритм сингулярного розкладу матриці, який складається з двох частин: ортогонального розкладання матриці, та приведення матриці до діагонального вигляду. Проведено огляд та опис програмного середовища та платформи CUDA розробленої компанією NVIDIA. Наведено реалізацію до дводіагонального вигляду матриці з обчисленням ортогональних множників за методом Хаусхолдера і діагоналізації, з використанням матриці повороту Ґівенса, в середовищі jCUDA. Проведено експерименти, результати яких ретельно досліджені на предмет пришвидшення обчислень з використанням графічного процесора порівняно з реалізацією на центральному процесорі, а також проведено порівняння з альтернативними способами реалізації SVD алгоритму для виконання на графічних процесорах.

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

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

ЗАТВЕРДЖУЮ
Зав. кафедри мережних технологій,
проф., д. ф.-м. н.
_____ Г. І. Малашонок
(підпис)
„____”_____2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту факультету інформатики 6 курсу Сухарському Сергію
ТЕМА “ Обчислення сингулярного розкладу матриць з використанням графічного процесора”

Зміст ТЧ до магістерської роботи:

Зміст
Анотація
Індивідуальне завдання
Вступ
1 Опис математичної частини
2 Опис технології CUDA
3 Опис практичної частини
4 Опис проведених експериментів
Висновки
Список публікацій за темою магістреської роботи
Список використаних джерел

Дата видачі „14” жовтня 2021 р.
Керівник _____
(підпис)
Завдання отримав _____
(підпис)

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 обчислень 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 SVD алгоритму з використанням потужності відеокарт, який є одним з головних методів роботи прогресивних рекомендаційних систем; вирішує багато завдань лінійної алгебри та може бути застосований для розвитку штучного інтелекту. Ще одним свідченням актуальності обчислень на відеокартах є нещодавній прорив в сфері суперкомп'ютерів, де вперше вдалось досягти ефективності в один екзафлопс якраз за рахунок поєднання центрального та чотирьох графічних процесорів в вузол, яких суперкомп'ютер Frontier налічує близько 9.5 тисяч [14].

Метою дослідження є розробка те реалізація алгоритму для виконання SVD на графічному процесорі. Для цього, на базі попередніх напрацювань [15] потрібно провести оптимізацію поточної версії програми та покращити її таким чином, щоб подолати проблему з похибкою обчислень. Крім того, реалізувати запропонований Малашонком та Семилітко [16] підхід до другої частини алгоритму сингулярного розкладу в середовищі jCUDA та покращити підхід за рахунок ефективнішого використання наявної на графічному процесорі великої кількості ядер. Для ефективного вирішення поставленої задачі в першій частині алгоритму використовуватиметься метод Хаусхолдера, оскільки він виглядає значно перспективніше ніж QR розклад, який досліджували Савченко та Малашонок [2]. Лагабар та Нараянан в своєму дослідженні [1] також використовують серію трансформації Хаусхолдера в першій частині алгоритму, проте, їхня реалізація все ще не вирішує проблем з похибкою обчислень на великих матрицях. В другій частині алгоритму вони використовують ітеративний QR алгоритм, що по факту є

таким же підходом, як обрано в цьому дослідженні, проте тут найбільшу роль відіграють тонкощі реалізації та ефективність використання ресурсів графічного процесора, де ми розраховуємо досягти покращення. Крім того, для того, щоб в майбутньому можна було інтегруватись з великим комплексом бібліотек для паралельного програмування Mathpartner [13] та системою децентралізованого управління розподіленими обчисленнями Drop Amine Pine [17] було обрано середовище JCUDA.

Щодо наукової новизни, то вона полягає в тому, що запропоновані спеціалізовані підходи та реалізована програма повинні ефективно та вдало виконуватись на графічних процесорах, що дасть поштовх в пришвидшенні обчислень з великими даними.

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

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

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 ($\left|\left|x\right|\right| = \left|\left|y\right|\right|$) існує ортогональна симетрична матриця P, така, що

$$y = Px.$$

Доведення

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

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

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:

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

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

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

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:

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

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

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 на 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:

$$\mathrm{x} = \begin{pmatrix} 0 & -2.5 & -2.5 & -2.5 \end{pmatrix};$$

$$norm1_x2 = 2.5^2;$$

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

$$u = \begin{pmatrix} 0 & -2.5 - norm_x & -2.5 & -2.5 \end{pmatrix};$$

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

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

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

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

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

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

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

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 обнулити 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:

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

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

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

$$normSq_u = 2 \cdot norm_x(norm_x + 1);$$
$$P_1 = I_{4,4} - (2/normSq_u)uu^T;$$

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

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

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

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:

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

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

1.3 Огляд алгоритму діагоналізації тридіагональної матриці

Другою частиною SVD алгоритму є приведення отриманої після першого етапу тридіагональної матриці до діагонального виду, а також отримання матриць множників L та R. Розглянемо приклад вхідної матриці на рисунку 1.

$$A = \begin{pmatrix} a & b & 0 & 0 \\ 0 & c & d & 0 \\ 0 & 0 & e & f \\ 0 & 0 & 0 & g \end{pmatrix}$$

Алгоритм полягає в тому, що ми ітеративно проходимось вздовж діагоналі, та застосовуємо множення матрицею обертання Ґівенса спочатку до вхідної матриці, а потім, до лівої (L) або правої (R) матриці, які на початку роботи алгоритму є просто одиничними, залежно від того, верхній чи нижній елемент відносно діагоналі потрібно обнулити. Якщо верхній, тоді множимо на R, якщо нижній, тоді на L. Проте, обнуливши один елемент, інший (розташований навпроти) може засмітитись певним значенням, але воно вже стане меншим, ніж було раніше. Відповідно, тут і з'являється можливість застосування ітеративного множення матрицею повороту, що рано чи пізно приведе до того, що значення буде якщо не нульове, то дуже мале, близьке до нуля. По суті, ми можемо виділити певні квадрати (блоки) з якими ми працюємо.

$$A = \begin{pmatrix} a & b & 0 & 0 \\ 0 & c & d & 0 \\ 0 & 0 & e & f \\ 0 & 0 & 0 & g \end{pmatrix}$$

Рис 1. Квадрати, які обробляються.

Для побудови матриці Ґівенса ми обчислюємо синус та косинус за формулою:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \times \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix},$$

$$r = \sqrt{a^2 + b^2},$$

$$c = a/r,$$

$$s = b/r,$$

Сама ж матриця виглядатиме так:

$$G = \begin{pmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Помноживши її на нашу вхідну матрицю, ми зможемо обнулити верхній елемент над діагоналлю, а сама матриці виглядатиме відповідно так:

$$A' = \begin{pmatrix} a' & 0 & 0 & 0 \\ b' & c' & d & 0 \\ 0 & 0 & e & f \\ 0 & 0 & 0 & g \end{pmatrix}$$

Якщо ж елементи над та під діагоналлю одразу нульові, то обробка цього квадрату пропускається.

Разом з множенням матриці G на нашу базову матрицю, ми також множимо її на матриці L та R відповідно. Розглянемо на прикладі, як обчислюються додаткові множники. Для наочності впливу, розглянемо вже проміжний етап алгоритму, а не його початок:

$$G = \begin{pmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$L = \begin{pmatrix} 1 & 2 & 3 & 1 \\ 2 & 3 & 1 & 2 \\ 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 \end{pmatrix};$$

$$L1 = G \cdot L;$$

$$out:$$

$$\begin{pmatrix} (2\cdot\ s + c) & (3\cdot\ s + 2\cdot\ c) & (s + 3\cdot\ c) & (2\cdot\ s + c) \\ ((-1)\cdot\ s + 2\cdot\ c) & ((-2)\cdot\ s + 3\cdot\ c) & ((-3)\cdot\ s + c) & ((-1)\cdot\ s + 2\cdot\ c) \\ 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 \end{pmatrix}$$

Рис 2. Обчислення матриці L.

Як бачимо, зоною впливу в цьому випадку є відповідно перші два рядки матриці L. Для матриці ж R ситуація відрізнятиметься тим, що там під впливом множення будуть змінюватись стовпчики:

$$G = \begin{pmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$R = \begin{pmatrix} 1 & 2 & 3 & 1 \\ 2 & 3 & 1 & 2 \\ 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 \end{pmatrix};$$

$$R1 = R \cdot G;$$

$$out:$$

$$\begin{pmatrix} ((-2)\cdot\ s + c) & (s + 2\cdot\ c) & 3 & 1 \\ ((-3)\cdot\ s + 2\cdot\ c) & (2\cdot\ s + 3\cdot\ c) & 1 & 2 \\ ((-1)\cdot\ s + 3\cdot\ c) & (3\cdot\ s + c) & 2 & 3 \\ ((-2)\cdot\ s + c) & (s + 2\cdot\ c) & 3 & 1 \end{pmatrix}$$

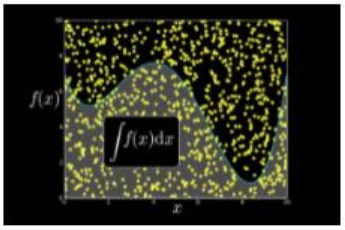
Рис 3. Обчислення матриці R.

Наведені приклади були зроблені в системі MathPartner [13]. Просуваючись по блоках, які ми обробляємо, матриця G відповідно також буде змінюватись, зсуваючи заповнений блок вздовж діагоналі.

Розділ 2: Опис технології CUDA

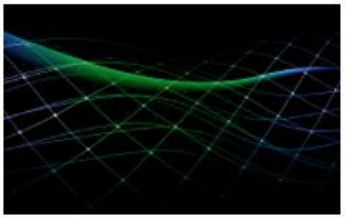
2.1 Загальні відомості

CUDA – це платформа для паралельних обчислень 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:



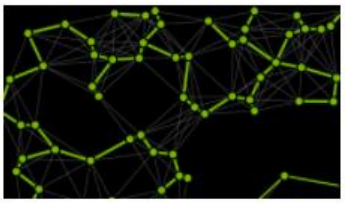
cuRAND

NPP

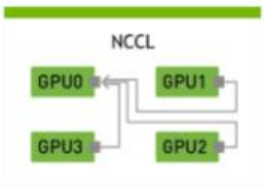


Math Library

cuFFT



nvGRAPH



NCCL

Рис 4. 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].

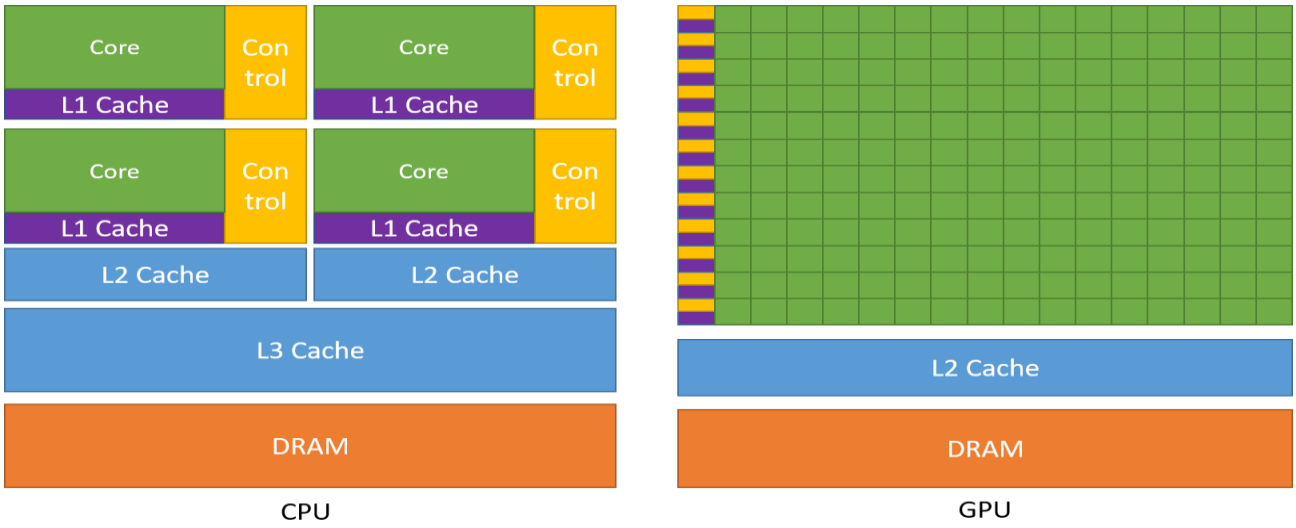


Рис 5. Порівняння мікросхем CPU та GPU [3].

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.

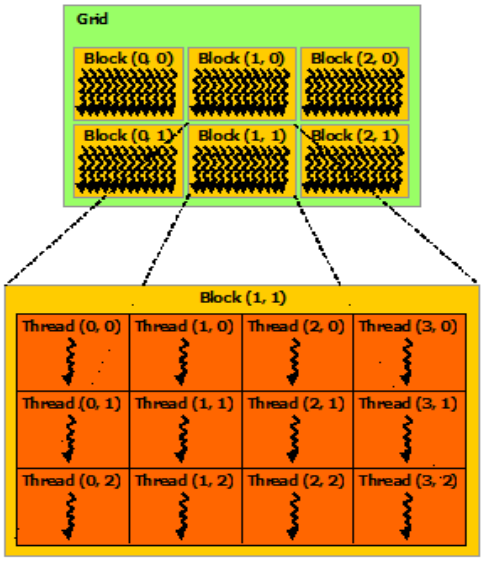


Рис 6. Архітектура CUDA [3].

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:

myKernel <<< gridSize, blockSize, sharedMemSize>>> (params),

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 *myKernel* - функція ядра. 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 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

2.6 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 методи *bidiagonalize(MatrixD A, Ring ring)* та *diagonalize(MatrixD A, Ring r),* 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 –діагональна матриця, W – правий множник. Матриця A, що передається в функцію заповнюється згенерованими числами типу Double в діапазоні від 0 до 10. 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).

Далі, в частині з діагоналізацією, ми приймаємо бідіагональну матрицю та намагаємось обнулити ненульові елементи над діагоналлю. Математична складова описана в розділі 1.3. В цьому ж розділі розглянемо на прикладі, як відбувається сам процес. Нехай маємо матрицю A, отриману з функції *biDiagonalize*.

$$A = \begin{pmatrix} d1 & ud1 & 0 & 0 \\ 0 & d2 & ud2 & 0 \\ 0 & 0 & d3 & ud3 \\ 0 & 0 & 0 & d4 \end{pmatrix}$$

Перші дві ітерації стосуватимуться першого та другого квадратів матриці. Ми говоримо одразу про дві ітерації, оскільки вони пов’язані між собою спільним елементом, а тому, не можуть бути виконані одночасно.

$$A_1 = \begin{pmatrix} \boldsymbol{d1} & ud1 & 0 & 0 \\ \boldsymbol{0} & \boldsymbol{d2} & ud2 & 0 \\ 0 & \boldsymbol{0} & d3 & ud3 \\ 0 & 0 & 0 & d4 \end{pmatrix}$$

Побудувавши матрицю поворту на A1 ми обнулимо елемент ud1, проте зіпсуємо елемент під d1. Значення d1 також зміниться, як і значення d2. Відповідно, нова матриця набуде такого вигляду:

$$A_2 = \begin{pmatrix} d1' & 0 & 0 & 0 \\ x & \boldsymbol{d2'} & ud2 & 0 \\ 0 & 0 & d3 & ud3 \\ 0 & 0 & 0 & d4 \end{pmatrix}$$

Далі ми виконуватимемо аналогічні операції і до наступних блоків (квадратів), повторюючи цей процес зверху до низу діагоналі, допоки усі значення над та під діагоналлю не будуть обнулені.

Разом з цим, виконуються обчислення матриць L та R за наведеним в розділі 1.3 принципом.

3.2 Бідіагоналізація з використанням алгоритму Хаусхолдера

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 subtractVectors

Функція віднімання двох векторів. Є можливість використання вбудованої функції додавання, додаючи помножений на -1 вектор, але цей підхід показує себе повільніше ніж власноруч розроблений кернел для віднімання. Виконується на сітці великої розмірності з фіксованим розміром блоку 512. Розмірність сітки визначається за формулою: `(int) Math.ceil((double) (N * N) / blockSizeX)`.

```
extern "C"
__global__ void subtractVectors(int n, double *a, double *b, double *res)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n)
    {
        res[i] = a[i] - b[i];
    }
}
```

3.2.2.3 setToZero

Функція обнулення n значень в масиві, розташованому на відеокарті. Потрібна для того, щоб працювати з вектором, ігноруючи вже пройдені елементи. Виконується на сітці великої розмірності з фіксованим розміром блоку 512.

```
extern "C"
__global__ void setToZero(int n, double *a, int toIndex)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n && i < toIndex)
    {
        a[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]));
}
```

3.3 Діагоналізація матриці

3.3.1 Опис програми

В розділі 3.1 було описано ідею алгоритму, а також показано, що деякі блоки не можуть бути опрацьовані паралельно. Проте, якщо намалювати ще наступний блок, то стає помітно, що його вже можна обробляти паралельно, з блоком який знаходиться через один від нього. Це стосується як попередніх, так і наступних.

$$A = \begin{pmatrix} d1 & ud1 & 0 & 0 \\ 0 & d2 & ud2 & 0 \\ 0 & 0 & d3 & ud3 \\ 0 & 0 & 0 & d4 \end{pmatrix}$$

Для наочності, розглянемо розширену матрицю, на якій виділимо блоки, які можуть бути обчислені паралельно.

$$B = \begin{pmatrix} d1 & ud1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & d2 & ud2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d3 & ud3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d4 & ud4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & d5 & ud5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & d6 & ud6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d7 & ud7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & d8 \end{pmatrix}$$

Якщо проаналізувати цю ситуацію, стає зрозуміло, що на кожному кроці ми можемо одночасно обробляти N / 2 блоків, де N це розмір матриці. Кожним набором блоків займається якийсь потік. Звідси робимо висновок, що кількість потоків, яка нам необхідна обчислюється за такою ж формулою: N / 2. Якщо ж матриця не квадратна, тоді кількість потоків схиляється в менший бік. Відповідно, потоки починають працювати поступово, один за одним, через кожні два кроки. З цього можна зробити висновок, що кожен потік починає працювати після затримки, яку можна обчислити за формулою номер потоку помножений 2.

Щодо обчислень матриць L та R, то за рахунок того, що потоки працюють з різницею в два кроки, в нас нема потреби синхронізувати значення цих матриць

між різними потоками, а тому, є можливість так само паралельно обчислювати їх в кожному потоці відповідно.

3.3.2 Робота з пам'яттю

Дуже важливим аспектом для підвищення швидкодії є вдале використання доступної пам'яті. Якщо ефективно її використовувати, можна буде більше користуватись кешем, а відповідно сильно пришвидшити обчислення, зекономивши на постійному читанні з загальної пам'яті. Саме для цього, матриця зберігається в вигляді одновимірного масиву. Проте, бувають випадки, коли спільної пам'яті якогось блоку потоків недостатньо для збереження дуже великої матриці. Наприклад, в сучасних відеокарт в середньому 48 Кб виділено під спільну пам'ять блоку. Проаналізувавши які елементи матриці беруть участь в алгоритмі, ми бачимо, що це лише три діагоналі. Відповідно, є можливість та потреба зберігати лише їх. Для цього, було написано кернел, що приводить матрицю, яка розташована в вигляді масиву на відеокарті до масиву, що зберігає лише три діагоналі в собі. Сам кернел має такий вигляд:

```
extern "C"
__global__ void to_bd_matrix(double* m, unsigned long s, double* r)
{
    for (unsigned long i = 0; i < s - 1; ++i) {
        unsigned long id = i * 3;
        r[id] = m[i * s + i];
        r[id + 1] = m[(i + 1) * s + i];
        r[id + 2] = m[i * s + i + 1];
    }
    r[3 * (s - 1)] = m[s * s- 1];
}
```

Він приймає матрицю m розміру s та приводить її до скороченого вигляду і записує результат в масив r. Кількість елементів в результуючому векторі обчислюється за формулою $(N-1)*3+1$, де N – розмір матриці (кількість рядків або стовпчиків, якщо брати квадратну). Такий підхід суттєво економить місце, завдяки чому, в 48Кб ми можемо помістити квадратну матрицю розміру 2^{11} заповнену елементами типу Double, а це вже досить немала матриця.

Для більших матриць ми будемо використовувати відповідно декілька блоків. Наприклад для квадратної матриці розміру 4096 знадобиться два блоки. Якщо ми знаємо, що в один блок поміщається 6144 елементи типу Double, тоді для матриці розміром 4096, з 12286 елементів після приведення до скороченого вигляду, в другому блоці буде 6143 значення. Ще одне значення буде заповнене нулем, щоб кількість елементів в обох блоках була однаковою. Розглянемо тепер детальніше, чому ми дублюємо один елемент. Проте, варто зазначити, що в деяких випадках швидше виконувати програму на декількох блоках, але не на великій кількості блоків. Наприклад, програма для матриці розміру 128 найефективніше показала себе на двох блоках з тридцяти двома групами по 4 потоки.

3.3.3 Синхронізація даних між блоками

Оскільки дані можуть бути поділені між різними блоками, елементи, що є спільними на стику блоків, потрібно синхронізувати між потоками цих блоків. Потоки одного блоку не мають доступу до пам’яті іншого блоку. Тому, ми будемо використовувати глобальну пам’ять відеокарти. Розглянемо простіший приклад:

$$\begin{pmatrix} d1 & ud1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & d2 & ud2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d3 & ud3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d4 & ud4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & d5 & ud5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & d6 & ud6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d7 & ud7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & d8 \end{pmatrix}$$

Припустимо, в нас є матриця 8 на 8, а максимальна кількість елементів, що можна помістити в один блок – 13. В такому випадку, ми помістимо 13 елементів в перший блок, а ще 9, дублюючи спільний елемент d5 в другий. Інші 4 значення заповнимо нулями.

Під час алгоритму, другий блок почне працювати лише тоді, коли перший блок оновить значення d5. Відповідно, для досягнення такого функціоналу нам потрібні два методи – перший для оновлення значень в глобальній пам’яті, а інший для зчитування його звідти. Відповідно перша записує нове значення, та

переконується, що воно записано. Друга ж, буде зчитувати з глобальної пам’яті доти, поки не побачить, що потрібний елемент квадрату вже онулений. Це означає, що цей квадрат вже опрацьований і можна вступати в роботу наступному блоку. Якщо дивитись на приклад вище, як тільки елемент ud4 стане нулем, ми зможемо розпочати роботу другого блоку. Саме тому, ми синхронізуємо не одне значення, а одразу три. Якщо ж значення є нулем одразу, блоки починають роботу незалежно. При цьому, самі потоки в межах одного блоку синхронізуються методом __syncthreads().

3.3.4 Оптимізація множень

Розглянемо тепер як ми можемо оптимізувати множення, які потрібно виконувати протягом роботи алгоритму. Для цього, розберемо який ефект має множення матриці G на матрицю A.

$$A = \begin{pmatrix} -2 & 4.33 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix};$$
$$a = -2;$$
$$b = 4.33;$$
$$r = \sqrt{(a^2 + b^2)};$$
$$c = a/r;$$
$$s = b/r;$$
$$G = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$
$$A' = A \cdot G;$$
$$out:$$

$$\begin{pmatrix} 4.77 & 0 & 0 & 0 \\ 0.45 & -0.21 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Рис 7. Множення матриці A на матрицю повороту.

Як видно з рисунку 7, в матриці A змінюється лише квадрат, відповідний заповненому квадрату в матриці повороту. А тому, ми можемо не множити цілі матриці, а лише перемножити між собою відповідні блоки. Звідси маємо той факт, що матрицю повороту можна зберігати не повністю, а лише значення синуса та косинуса.

Аналогічно, якщо подивитись ефект множення матриці G на L та R на G (розділ 1.3), то стає зрозуміло, що не потрібно повністю множити ці матриці, а достатньо лише відповідні елементи, які зазнають впливу. При цьому, оскільки в нас один потік обробляє два послідовні кроки, є сенс виконати одразу обидва ці кроки. В результаті, функції застосування матриці Ґівенса мають такий вигляд:

1) Обробка нижнього елементу відносно діагоналі

```
extern "C" __device__ void givens_lower(double* s_bdm, ulong pos, ulong offset, uint
thread_cnt, uint unit_thread_cnt, volatile double* L, ulong l_size, double eps)
{
    unsigned long id = pos * 3;
    double b = s_bdm[id + 1];

    if (abs(b) < eps)
        return;

    double a = s_bdm[id];
    double r = sqrt(sqr(a) + sqr(b));
    double c = 1, s = 0;

    if (r > eps)
    {
        c = a / r;
        s = b / r;
    }

    if (threadIdx.x < thread_cnt) {
        double c11 = c * s_bdm[id] + s * s_bdm[id + 1];
        double c12 = c * s_bdm[id + 2] + s * s_bdm[id + 3];
        double c21 = -s * s_bdm[id] + c * s_bdm[id + 1];
        double c22 = -s * s_bdm[id + 2] + c * s_bdm[id + 3];

        s_bdm[id] = c11;
        s_bdm[id + 1] = c21;
        s_bdm[id + 2] = c12;
        s_bdm[id + 3] = c22;
    }
```

```
    uint unit_thread_id = (uint)(threadIdx.x / thread_cnt);
    ulong share_size = (ulong)(l_size / unit_thread_cnt);
    ulong start = share_size * unit_thread_id;
    ulong stop = share_size * (unit_thread_id + 1);

    ulong id1 = l_size * (pos + offset);
    ulong id2 = l_size * (pos + offset + 1);

    for (int i = start; i < stop; ++i)
    {
        double v1 = L[id1 + i];
        double v2 = L[id2 + i];

        L[id1 + i] = c * v1 + s * v2;
        L[id2 + i] = -s * v1 + c * v2;
    }
}
```

2) Обробка верхнього елементу відносно діагоналі

```
extern "C" __device__ void givens_upper(double* s_bdm, unsigned long pos, ulong
offset, uint thread_cnt, uint unit_thread_cnt, volatile double* R, ulong r_size,
double eps)
{
    unsigned long id = pos * 3;
    double b = s_bdm[id + 2];

    if (abs(b) < eps)
        return;

    double a = s_bdm[id];
    double r = sqrt(sqr(a) + sqr(b));
    double c = 1, s = 0;

    if (r > eps)
    {
        c = a / r;
        s = b / r;
    }

    if (threadIdx.x < thread_cnt) {
        double c11 = s_bdm[id] * c + s_bdm[id + 2] * s;
        double c12 = s_bdm[id] * -s + s_bdm[id + 2] * c;
        double c21 = s_bdm[id + 1] * c + s_bdm[id + 3] * s;
        double c22 = s_bdm[id + 1] * -s + s_bdm[id + 3] * c;

        s_bdm[id] = c11;
        s_bdm[id + 1] = c21;
        s_bdm[id + 2] = c12;
```

```
        s_bdm[id + 3] = c22;
    }

    uint unit_thread_id = (uint)(threadIdx.x / thread_cnt);
    ulong share_size = (ulong)(r_size / unit_thread_cnt);
    ulong start = share_size * unit_thread_id;
    ulong stop = share_size * (unit_thread_id + 1);

    ulong id1 = r_size * (pos + offset);
    ulong id2 = r_size * (pos + offset + 1);

    for (int i = start; i < stop; ++i)
    {
        double v1 = R[id1 + i];
        double v2 = R[id2 + i];

        R[id1 + i] = v1 * c + v2 * s;
        R[id2 + i] = v1 * -s + v2 * c;
    }
}
```

Обидві функції приймають однаковий набір параметрів: вказівник на саму матрицю, розташовану в спільній пам’яті конкретного блоку; номер квадрату, який обробляємо; для матриць L або R також передається зміщення розташування в глобальній пам’яті; вказівник на саму матрицю R або L (важливо зазначити, що ми використовуємо ключове слово volatile, щоб зазначити, що в цьому випадку не потрібно використовувати значення з кешу, а зчитувати його з пам’яті заново; і наостанок передаємо бажану точність, з якою потрібно проводити обчислення. Крім того, передаємо загальну кількість потоків в цьому блоці та кількість потоків для множень на праву / ліву матрицю. Відповідно, кожен з них опрацьовує якусь свою частину матриці.

Сам же процес такий:

1) Обчислюємо позицію елемента, який будемо зануляти
2) Перевіряємо, чи абсолютне значення елемента, який ми обробляємо менше за бажану точність. Якщо так, тоді нічого далі не робимо, закінчуємо метод
3) Інакше, обчислюємо синус та косинус для матриці G

4) Застосовуємо знайдені значення до головної матриці
5) Обчислюємо позиції елементів для матриць L / R
6) Застосовуємо значення матриці повороту до матриць L або R відповідно

3.3.5 Завершення обчислень

Алгоритм завершує свою роботу, як тільки усі елементи стають меншими за бажану точність обчислень. Це перевіряється таким чином, що потоки повинні пройти свою частину матриці, не виконавши при цьому жодних дій. Для цього ми використовуємо змінну в яку записуємо кількість ітерацій без дій. Як тільки їхня кількість дорівнює розміру матриці ми можемо завершувати роботу. Потоки закінчують працювати одночасно.

Проте, перш ніж завершити роботу, за допомогою ще однієї глобальної змінної ми перевіряємо чи блок, що працював до поточного, вже завершив обчислення. Для першого блоку ця умова відповідно одразу істинна.

Розділ 4. Опис проведених експериментів

4.1 Загальні відомості

Експерименти проведено для кожної частини окремо, та для всієї програми, де ці частини працюють одна за одною з передачею результатів від першої до другої. Ці дані досліджено на предмет залежності часу обчислень від розміру матриць, а також оцінено та досліджено точність обчислень. Крім цього, досліджувалась залежність часу від точності для ітераційної частини алгоритму. Також, проведено порівняння з послідовним алгоритмом на центральному процесорі.

4.2 Опис технічних характеристик

Усі обчислення проводились на графічному процесорі GIGABYTE RTX 3070 GAMING OC з такими характеристиками:

Графічний процесор	GA104-300-A1
Архітектура	Ampere
Пам’ять	8 GB типу GDDR6; 256 bit; 448.0 GB/s
Кількість CUDA ядер	5888
Спільна пам’ять блоку	48 KB
Максимальна кількість потоків	1024

Інші характеристики системи:

Центральний процесор	Intel Core i5-12600K 3.7 GHz
Оперативна пам’ять	G.Skill 16 GB (2x8 GB) DDR4 3600 MHz
Накопичувач	WD SN850 SSD 1 TB

Обчислення проводились на операційній системі Windows 11 Pro 21H2 x64 (build 22000.739).

4.3 Експерименти

4.3.1 Дослідження часу та похибки обчислень

Перш за все було проведено експерименти для оцінки часу виконання алгоритму та оцінки похибки обчислень. Для цього генерувались матриці різних розмірів, з діапазоном значень від 0 до 100 типу *Double*. Для кожної згенерованої матриці проводилось по два запуски, оцінювався час виконання кожної частини алгоритму та загальний час, а також брався середній час і діапазон відхилень. В цей час враховується весь процес обчислень, та не враховується процес перевірки похибки обчислень. Такі ж метрики проводились для похибки обчислень. В таблиці 4.1 наведено детальні дані по часу обчислень. Час наведено в секундах.

Таблиця 4.1 – Оцінка часу виконання

Розмір матриці	128	256	512	1024	2048
Бідіагоналізація	0.051	0.098	0.219	0.878	6.557
Відхилення (+/- t)	0.005	0.009	0.022	0.009	0.008
Діагоналізація	0.267	1.646	22.756	161.360	689,072
Відхилення (+/- t)	0.056	0.031	0.803	14.302	76.037
Алгоритм	0.318	1.744	22.943	162.238	692,756
Відхилення (+/- t)	0.053	0.02	0.793	14.31	76.03

Щоб наочніше оцінити залежність часу виконання від розміру матриці, розглянемо декілька графіків. На рисунку 8 ми спостерігаємо плавний ріст часу зі збільшенням розміру матриці. Також, можемо зазначити, що процес бідіагоналізації проходить досить швидко – квадратна матриця розміру 1024

обчислюється менше секунди. Щодо відхилення по часу прогонів, то для бідіагоналізації воно зовсім мале, а для діагоналізації незначне.

Рис 8.Час бідіагоналізації.

Розглянемо тепер графік залежності часу діагоналізації від розміру матриці. Як бачимо з таблиці 4.1, цей процес складає переважну частину усього алгоритму, тому побудуємо графік одразу з двома кривими, для порівняння.

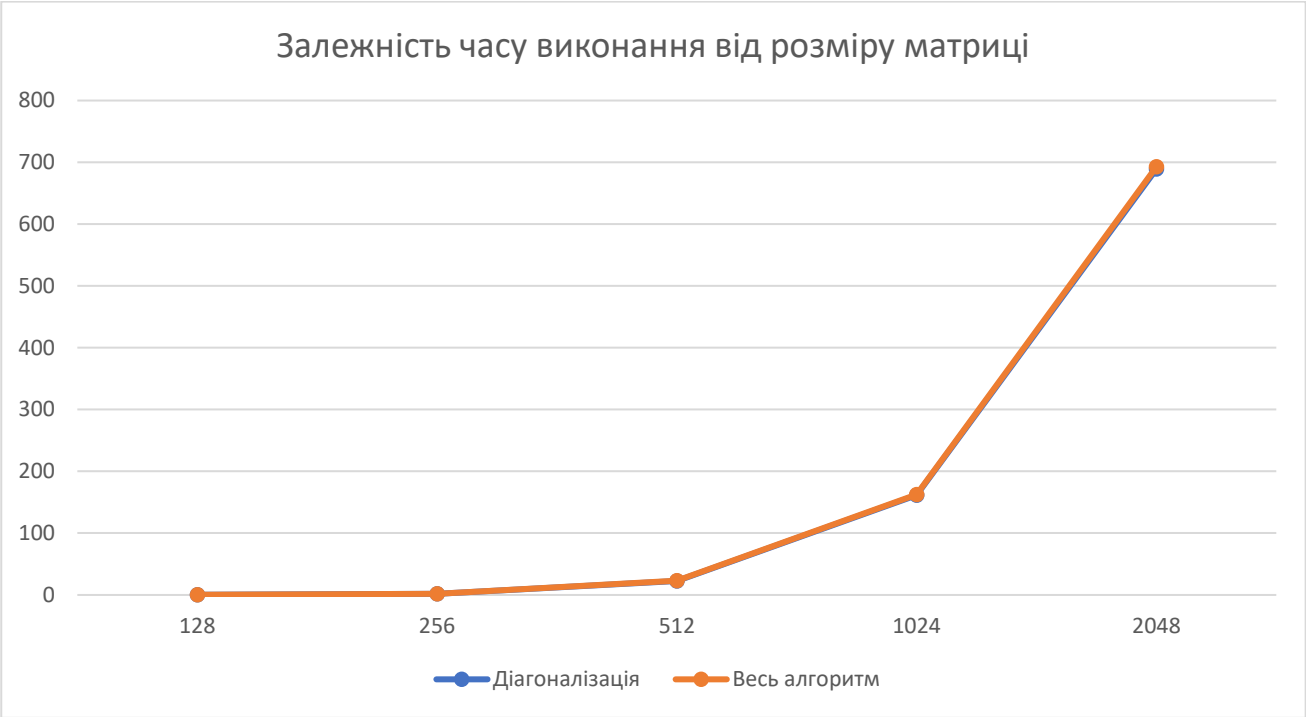


Рис 9. Час діагоналізації та всього алгоритму

Рисунок 9 дуже добре показує, наскільки близькі за часом ці криві – друга повністю перекриває першу. Сама залежність часу трохи різкіша ніж для бідіагоналізації, проте загальна тенденція схожа. А от різниця в часі виконання велика – більше 5 разів на для матриць 128 x 128 та більше ніж 180 разів на розмірі 1024 x 1024. Що цікаво, зі збільшенням розміру з 512 до 1024 час обчислень виріс в 7 разів, що не є добре, але, зі збільшенням з 1024 до 2048 час збільшився лише в 4 рази. Це покращення досягнуто за рахунок того, що матриця розміру 2048 виконується на більшій кількості блоків, відповідно більше потоків працюють з частиною матриці, але втрачається час на синхронізації значень.

Щодо похибки обчислень, то аналізуючи детальні дані наведені в таблиці 4.2 помітно, що абсолютна похибка обчислень досить повільно змінюється, залежно від розміру матриці. Особливо повільні зміни відбуваються під час бідіагоналізації. Проте, варто зазначити, що діагоналізацію можна налаштувати таким чином, щоб алгоритм працював, поки не буде досягнуто бажаної точності або не дійде до моменту, коли вже не може краще порахувати.

Таблиця 4.2 – Оцінка абсолютної похибки обчислень

Розмір матриці	128	256	512	1024	2048
Бідіагоналізація	4.47E-13	3.69E-13	6.25E-13	1.00E-12	1.25E-12
Відхилення (+/- err)	1.065E-13	4.25E-14	5.63E-14	1.42E-14	1.66E-13
Діагоналізація	5.28E-11	4.51E-11	7.74E-11	2.89E-10	2.96E-9
Відхилення (+/- err)	7.22E-13	6.16E-12	3.29E-12	7.56E-11	1.97E-9
Алгоритм	7.007E-12	8.64E-12	1.07E-11	3.06E-11	4.57E-10
Відхилення (+/- err)	6.94E-13	2.41E-13	2.28E-13	7.53E-12	1.35E-10

За рахунок того, що ми маємо можливість налаштовувати бажану точність обчислень, було проведено дослідження часу виконання алгоритму для точності 10E-3, що є значно меншою точністю ніж в попередніх експериментах, де вона складала 10E-10 (за можливості досягти такої точності). Відповідно, час виконання ітераційної частини зменшився. В таблиці 4.3 наведено конкретні приклади та пришвидшення.

Таблиця 4.3 – Дослідження часу ітераційної частини для різної дозволеної похибки обчислень

Розмір матриці	128	256	512	1024	2048
10E-10	0.318	1.744	22.943	162.238	692,756
10E-3	0.229	0.822	10.172	32.908	145,453
Пришвидшення в n разів	1.38	2.12	2.25	3.59	4.76

Як бачимо пришвидшення в 2-4 рази. При цьому, таки пришвидшення тримається як при обчисленнях на різній кількості блоків та потоків.

Щодо часової складності алгоритму, то очікувана складність дорівнює n^3. Після проведення експериментів, за допомогою системи MathPartner [13] було підібрано відповідну криву, яка оцінює часову складність згідно отриманого часу обчислень.

Ми використали ось такий шматок коду:

```
SPACE = R64[x];
\set2D( 0, 2500,  0, 1500);
f= 692.756/(2048^(1.8)) x^(1.8);
svd=\tablePlot([[128, 256, 512, 1024, 2048],
        [0.318,1.744,22.943,162.238,692.756]]);
complexity=\plot(f);
\showPlots([svd,complexity]);
```

Після його виконання було отримано відповідний графік:

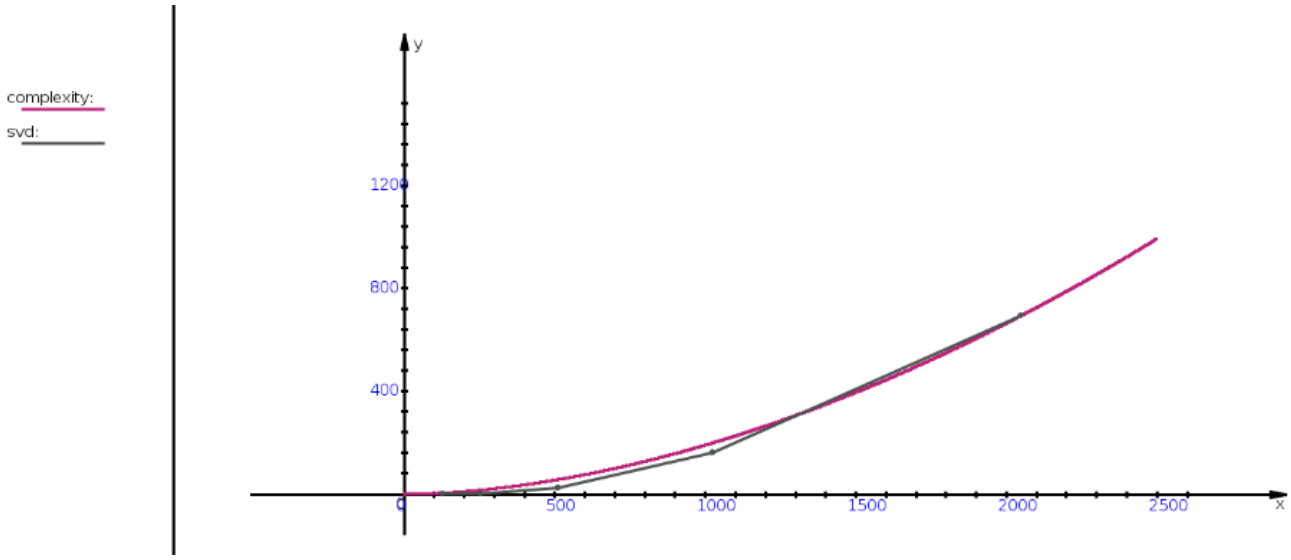


Рис 10. Оцінка часової складності алгоритму

Отримана формула кривої $1282.401/\left(2048^{(1.8)}\right)x^{(1.8)}$. Як бачимо, степінь насправді виявилась навіть меншою, ніж очікувалось. Це пов’язано з тим, що на

більшу матрицю в нас використовується більше ядер, відповідно збільшення часу менше, ніж очікується.

4.3.2 Порівняння з послідовним алгоритмом

Ми провели дослідження для квадратних матриць розміру 128 та 256, заповнених числами типу *Double* в діапазоні від 0 до 100, як і для програми, що виконувалась на графічному процесорі і порівняли результати, щоб розуміти ефективність таких обчислень на GPU. В таблиці 4.4 наведено результати експериментів, а саме, час виконання (в секундах) всього алгоритму SVD на GPU, CPU та визначене пришвидшення.

Таблиця 4.4 – Порівняння часу обчислень на CPU та GPU

Розмір матриці	128	256
CPU	110.291	1802.114
GPU	0.318	1.744
Пришвидшення в n разів	346,82	1033,32

Наведена вище таблиця підтверджує те, що один графічний процесор володіє дуже великими потужностями для паралельних та інтенсивних обчислень. Навіть на невеликій матриці розміру 128, нам знадобиться дуже гарно реалізована програма і процесор з 346-ма ядрами, щоб зрівнятись з можливостями відеокарти. На матриці більшого розміру різниця взагалі колосальна.

4.3.3 Порівняння з CUDA API

Також проведено порівняння з CUDA API [18]. Були проведені експерименти з точністю 10E-10 та матрицями з числами типу *Double* з діапазоном значень від 0 до 100. Як бачимо по даних в таблиці 4.5, різниця поки дуже серйозна. Якщо на матрицях невеликого розміру різниця ще відносно невелика і її легко подолати, за рахунок оптимізації виділення блоків та потоків, то на матрицях великого розміру

стає дуже помітною проблема того, що при використанні декількох блоків потрібно навчитись використовувати групи допоміжних (додаткових) потоків до тих, що працюють над множеннями правої та лівої матриць.

Таблиця 4.5 – Порівняння часу обчислень з CUDA API

Розмір матриці	128	256	512	1024	2048
SVD	0.318	1.744	22.943	162.238	692,756
CUDA API SVD	0.038	0.113	0. 335	1.224	7.462
Пришвидшення в n разів	8.36	15.43	68.46	132.54	92.83

Висновки

В даній роботі розглянуто, досліджено, запропоновано та реалізовано алгоритм сингулярного розкладу матриць на графічному процесорі. Перша частина алгоритму використовує метод Хаусхолдера для приведення матриці до дводіагонального вигляду, та показує чудові результати по швидкості та точності обчислень. Підхід трансформацій покращено та оптимізовано під виконання на відеокарті для зменшення кількості запису та зчитувань з пристрою та відповідно максимальної ефективності обчислень.

В другій частині алгоритму застосовано новий підхід в реалізації стандартного ітераційного процесу, який вдало використовує присутні в графічному процесорі ресурси, адаптовуючись під конкретні параметри конкретного пристрою. Запропоновано підхід до поділу матриці на декілька блоків, синхронізації даних між блоками та потоками, а також, запропоновано ефективний підхід по збереженню матриці в пам’яті.

Проведено ретельне дослідження з детальним описом умов та способів оцінки часу виконання та визначення точності обчислень. Результати показали колосальне пришвидшення відносно виконання цього алгоритму на центральному процесорі. Також, вони підтвердили, що ідея виділяти групи потоків в кожному блоці є дуже перспективною, адже швидкість обчислень помітно зростає з застосуванням оптимальних параметрів. Але, як показали порівняння з альтернативними способами реалізації, цей підхід з групами потоків потрібно покращувати для декількох блоків, оскільки саме на великих матрицях з’являється помітне відставання по часу обчислень порівняно з CUDA API наприклад.

Отже, запропонований підхід показав чудові результати, але ще залишає за собою простір для покращень та оптимізації в майбутніх дослідженнях.

Список публікацій за темою магістерської роботи

1. *Малашонок Г.І., Сухарський С.С.* Алгоритм обчислення дводіагональної матриці ортогональним розкладанням на графічному процесорі. Наукові записки НаУКМА. Комп'ютерні науки. 2021. Том 4. [Електронний ресурс]. – режим доступу: http://nrpcomp.ukma.edu.ua/article/view/246581. Дата звернення: 2022.06.14

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: 2022.06.12
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: 2022.06.18
4. J. Lambers, “The SVD Algorithm”, CME 335, Lecture 6. [Електронний ресурс]. – режим доступу: https://web.stanford.edu/class/cme335/lecture6.pdf. Дата звертання: 2022.06.03
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: 2022.06.05
6. Cornell University, “Numerical linear algebra and matrix factorizations”, [Електронний ресурс]. – режим доступу: http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/Householder.html. Дата звертання: 2022.06.12
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: 2022.06.15
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: 2022.05.02
12. Опис проекту JCuda. [Електронний ресурс]. – режим доступу: http://www.jcuda.org/tutorial/TutorialIndex.html. Дата звертання: 2022.01.10
13. Система комп’ютерної алгебри MathPartner [Електронний ресурс]. - http://mathpar.ukma.edu.ua/. Дата звернення: 2022.06.13

14. Новинний ресурс Nauka.ua [Електронний ресурс]. – режим доступу: https://nauka.ua/news/superkompyuter-vpershe-dosyag-efektivnosti-v-odin-ekzaflops. Дата звернення: 2022.06.14

15. *Малашонок Г.І., Сухарський С.С.* Алгоритм обчислення дводіагональної матриці ортогональним розкладанням на графічному процесорі. Наукові записки НаУКМА. Комп'ютерні науки. 2021. Том 4. [Електронний ресурс]. – режим доступу: http://nrpcomp.ukma.edu.ua/article/view/246581. Дата звернення: 2022.06.14

16. *Малашонок Г.І., Семилітко М.Ю.* Паралельний SVD алгоритм для тридіагональної матриці на відеокарті з використанням архітектури Nvidia CUDA. Наукові записки НаУКМА. Комп'ютерні науки. 2021. Том 4. [Електронний ресурс]. – режим доступу: http://nrpcomp.ukma.edu.ua/article/view/246582. Дата звернення: 2022.06.14

17. *Malaschonok, G.I., Sidko, A.A. Supercomputer Environment for Recursive Matrix Algorithms. Program Comput Soft 48, 90–101 (2022). https://doi.org/10.1134/S0361768822020086*

18. CUDA API SVD. [Електронний ресурс]. – режим доступу: https://github.com/NVIDIA/CUDALibrarySamples/blob/master/cuSOLVER/gesvdj/cusolver_gesvdj_example.cu . Дата звернення: 2022.07.01