

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



**Паралельний SVD алгоритм для трьохдіагональної матриці на
відеокарті з використанням архітектури Nvidia CUDA
Текстова частина
магістерської роботи
за спеціальністю „Інженерія програмного забезпечення” 123**

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

(підпис)

“ ____ ” _____ 2021 р.

Виконав студент Семилітко М.Ю.

“ ____ ” _____ 2021 р.

Київ 2021

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

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики, к.ф.-м.н.
_____ С. С. Гороховський
(підпис)
„____” _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р.н. магістерської програми Інженерія програмного забезпечення Семилітку Миколі Юрійовичу
Розробити Паралельний SVD алгоритм для трьохдіагональної матриці на відеокарті з використанням архітектури Nvidia CUDA

Зміст текстової частини до магістерської роботи:

Зміст

Анотація

Вступ

1 Огляд SVD алгоритму для трьохдіагональної матриці та архітектури Nvidia CUDA

2 Розробка паралельного алгоритму SVD алгоритму для трьохдіагональної матриці

3 Тестування реалізованого алгоритму

Висновки

Список літератури

Додатки

Дата видачі „____” _____ 2020 р.

Керівник

Г. І. Малашонок,

(підпис)

Завдання отримав

М.Ю. Семилітко

Тема: Паралельний SVD алгоритм для трьохдіагональної матриці на відеокарті з використанням архітектури Nvidia CUDA

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	01.11.2020	
2.	Огляд технічної літератури за темою роботи	15.11.2020	
3.	Виконання аналізу сучасних рішень	29.11.2020	
3.	Розробка архітектури паралельного алгоритму	27.12.2020	
4.	Реалізація паралельного алгоритму засобами мови програмування Java	17.01.2021	
5.	Реалізація паралельного алгоритму на архітектурі Nvidia CUDA	25.01.2021	
6.	Тестування розробленого алгоритму та виправлення помилок	27.03.2021	
7.	Написання пояснювальної записки	24.04.2021	
8.	Створення слайдів для доповіді та написання доповіді	01.05.2021	
9.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи	14.05.2021	
10.	Корегування роботи за результатами попереднього захисту	17.05.2021	
11.	Остаточне оформлення пояснювальної записки та слайдів	31.05.2021	
12.	Захист магістерської роботи (проекту)	17.06.2021	

Студент М. Ю. Семилітко

Керівник Г. І. Малапонок

“ ____ ” _____ 2021 р.

ЗМІСТ

Анотація	6
ВСТУП	7
РОЗДІЛ 1. Огляд SVD алгоритму для трьохдіагональної матриці та архітектури NVIDIA CUDA	9
1.1 Огляд SVD алгоритму для трьохдіагональної матриці	9
1.2 Огляд архітектури Nvidia CUDA	11
РОЗДІЛ 2. Розробка паралельного алгоритму SVD алгоритму для трьохдіагональної матриці	15
2.1 Опис роботи базового паралельного алгоритму	15
2.2 Оптимізація роботи з пам'яттю.	18
2.2.1 Оптимізація зберігання трьохдіагональної матриці для одного блоку	19
2.2.2 Оптимізація зберігання трьохдіагональної матриці для декількох блоків	20
2.3 Оптимізація множення для кроків алгоритму	22
2.3.1 Оптимізація множення для трьохдіагональної матриці	22
2.3.2 Оптимізація множення для матриць L та R	25
2.4 Синхронізація значень трьохдіагональних матриць та ітерацій потоків між блоками	27
2.4.1 Розробка методу синхронізації даних між блоками	28
2.4.2 Синхронізація блоків та значень матриці для паралельного алгоритму	30
2.4.3 Завершення роботи алгоритму	32
РОЗДІЛ 3. Тестування реалізованого алгоритму	34

3.1 Тестування швидкості роботи алгоритму для різних розмірів матриці	34
3.2 Тестування швидкості роботи алгоритму для різних значень точності обчислень	43
3.3 Порівняння швидкості роботи алгоритму за різної кількості обчислювальних потоків	46
3.4 Порівняння швидкості роботи алгоритму з існуючими рішеннями	49
Висновки по роботі та рекомендації для подальших досліджень	51
Список літератури	54
Додаток А. Схема паралельного SVD алгоритму на відеокарті	55
Додаток Б. Програмний код паралельного SVD алгоритму на відеокарті	56
Додаток В. Програмний код API для розробленого алгоритму на C++	61

Анотація

Дана робота пропонує реалізацію паралельного алгоритму SVD на відеокарті з використання архітектури Nvidia CUDA для роботи з великими матрицями. Для цього було досліджено роботу послідовного алгоритму, розроблена модель паралельного алгоритму на Java, який враховує особливості роботи відеокарти і реалізовані та протестовані алгоритми для відеокарти з використанням різних типів пам'яті відеокарти, які можна використовувати в програмах на Java та C/C++.

Ключові слова: Сингулярний розклад матриці, SVD, Nvidia CUDA, Java, C++.

ВСТУП

Актуальність. Алгоритм SVD (Singular Value Decomposition — сингулярний розклад матриці) використовується в рекомендаційних системах, машинному навчанні, обробці зображення, в різних алгоритмах роботи з матрицями, які можуть бути дуже великого розміру та Big Data, тому, враховуючи особливості роботи цього алгоритму, він може виконуватися на великій кількості обчислювальних потоків, які мають тільки відеокарти. Такий підхід дозволить зменшити час обчислень, а отже й зменшити кількість ресурсів та грошових витрат.

Архітектура CUDA (Compute Unified Device Architecture) – це сама передова архітектура обчислень на відеокарті, яка дозволяє виконувати велику кількість різних обчислень паралельно. Вона також використовується для машинного навчання, а також, в аналізі аерогеодезичних даних, в банківській сфері [1].

Оскільки відеокарти, які підтримують архітектуру CUDA, є невід’ємною частиною суперкомп’ютерів, паралельні алгоритми саме на цій архітектурі набувають все більшої та більшої популярності, а сама технологія неспинно розвивається, тому ця архітектура була вибрана для прискорення роботи алгоритму.

Мета дослідження. Розробити паралельний алгоритм SVD для трьохдіагональної матриці на відеокарті використовуючи архітектуру Nvidia CUDA.

Завдання дослідження. Проаналізувати роботу послідовного алгоритму, розробити модель паралельної версії алгоритму для процесора засобами мови програмування Java, враховуючи особливості роботи з пам’яттю на графічному чіпі та на отриманих результатах розробити та протестувати алгоритм сингулярного розкладу матриці для відеокарти на правильність обчислень, їх швидкість та затримки по роботі з пам’яттю для різних розмірів матриць.

Об'єкт дослідження. Паралельний алгоритм SVD для трьохдіагональної матриці.

Предмет дослідження. Використання архітектури NVIDIA CUDA для швидкого вирішення матричних задач.

Джерела дослідження. Електронні версії друкованої літератури, програмна документація, довідники посилань на API (application programming interface - прикладний програмний інтерфейс), електронні ресурси, в тому числі спеціалізовані форуми, вихідні коди програм та бібліотек, відео-інструкції.

Наукова новизна одержаних результатів. Запропонована модифікація алгоритму може виконуватися на великій кількості обчислювальних потоків, а також має оптимізовану роботу з різними типами пам'яті відеокарт NVIDIA, що також має значний вплив на швидкість обчислень.

Практичне значення одержаних результатів. Отриманий алгоритм має високу швидкість роботи та базується на розповсюдженій архітектурі, тому він може використовуватись для зменшення часу обчислень, економії обчислювальних та грошових ресурсів.

РОЗДІЛ 1. Огляд SVD алгоритму для трьохдіагональної матриці та архітектури NVIDIA CUDA

1.1 Огляд SVD алгоритму для трьохдіагональної матриці

SVD алгоритм для трьохдіагональної матриці M приводить до діагонального виду та повертає 3 матриці: L , діагональну матрицю M' та R , добуток яких повертає початкову трьохдіагональну матрицю M .

Трьох діагональна матриця має ненульові елементи на головній діагоналі та на діагоналі, яка знаходиться над або під головною діагоналлю. На рисунку 1.1 показаний приклад трьохдіагональної матриці M з ненульовими елементами над головною діагоналлю.

$$\begin{pmatrix} a & b & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & c & d & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & e & f & \dots & 0 & 0 & 0 \\ & & & & \dots & & & \\ 0 & 0 & 0 & 0 & \dots & 0 & g & h \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & i \end{pmatrix}$$

Рисунок 1.1 – Приклад трьохдіагональної матриці M з ненульовими елементами над головною діагоналлю.

Алгоритм складається з двох кроків, які виконуються для кожного квадрату на головній діагоналі, які зображені на рисунку 1.2. Ці кроки виконуються до тих пір, поки всі елементи нижньої та верхньої діагоналей не стануть нульовими.

$$\begin{pmatrix} a & b & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & c & d & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & e & f & \dots & 0 & 0 & 0 \\ & & & & \dots & & & \\ 0 & 0 & 0 & 0 & \dots & 0 & g & h \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & i \end{pmatrix}$$

Рисунок 1.2 – Елементи матриці, які оброблюються алгоритмом.

На першому кроці, до першого квадрату з рисунку 1.2 застосовується матриця обертання Гівенса, що обнулює значення над або під головною діагоналлю.

Для цього обчислюється косинус та синус для матриці повороту за такою формулою:

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

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

$$c = a/r,$$

$$s = b/r,$$

і матриця Гівенса матиме наступний вигляд (рисунок 1.3) :

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

Рисунок 1.3 – Матриця обороту Гівенса для першого квадрату матриці М.

При множенні вхідної матриці на матрицю Гівенса, вхідна матриця прийме наступний вигляд (рисунок 1.4), де значення верхнього правого елемента квадрата стало нульовим, а значення нижнього стало ненульовим:

$$\begin{pmatrix} a' & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ b' & c' & d & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & e & f & \dots & 0 & 0 & 0 \\ & & & \dots & & & & \\ 0 & 0 & 0 & 0 & \dots & 0 & g & h \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & i \end{pmatrix}$$

Рисунок 1.4 – Матриця М після множення на матрицю Гівенса.

На другому кроці отримана матриця повороту також множиться на матрицю L або R , в залежності від того, обнулюємо ми значиння над або під головною діагоналлю, тобто, якщо ми обнулюємо елемент над головною діагоналлю, тоді матриця Гівенса множиться на матрицю R , інакше на матрицю L . На початку роботи алгоритму матриці L та R – це одиничні матриці, такого ж розміру, як і вхідна матриця M .

Є випадки, коли на 1 кроці в квадраті 2 недіагональних елемента нульові. В такому разі крок можна не виконувати, тобто матрицю повороту не обчислюється і множиться на матриці M та L або R .

Коли всі елементи на верхній та нижній діагоналі стануть нульовими, алгоритм завершує свою роботу.

1.2 Огляд архітектури Nvidia CUDA

CUDA – це паралельна обчислювальна платформа та модель програмування, розроблена NVIDIA для загальних обчислень на графічних процесорах (GPU). Завдяки CUDA розробники можуть різко прискорити обчислювальні програми, використовуючи потужність графічних процесорів. У прискорених GPU додатках послідовна частина робочого навантаження працює на центральному процесорі, який оптимізований для однопотокової продуктивності, тоді як обчислювальна частина програми паралельно працює на тисячах ядер графічного процесора. Використовуючи CUDA, розробники програмують на популярних мовах, таких як C, C++, Fortran, Python та MATLAB [2].

В архітектурі CUDA є 3 основні абстракції:

1. Ієрархія блоків потоків
2. Спільна пам'ять
3. Синхронізація з використанням бар'єрів [3]

Обчислювальні ядра – це потоки, які знаходяться в середині блоку потоків. Кожен потік має локальну пам'ять, а також доступ до спільної пам'яті блоку та глобальної пам'яті відеокарти як показано на рисунку 1.5. Кількість потоків в блоці залежить від покоління та архітектури.

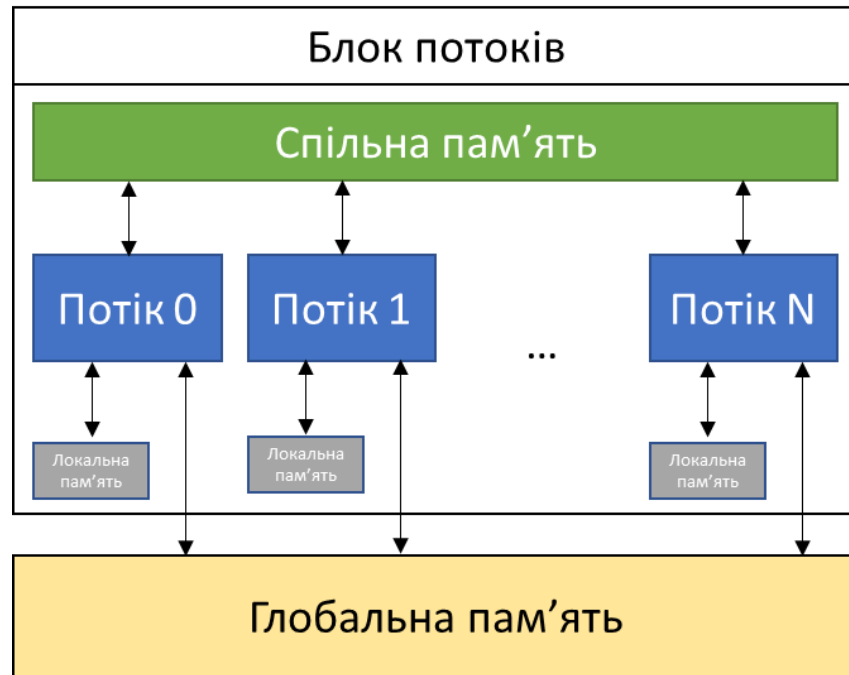


Рисунок 1.5 – Приклад організації роботи блоку потоків

Суттєва різниця між спільною пам'яттю блока та глобальною полягає в тому, що глобальна пам'ять знаходиться на платі за межами кристалу відеокарти, а спільна пам'ять знаходиться на ньому (рисунку 1.6). Для того щоб збільшити швидкість обчислень можна використовувати спільну пам'ять блоку замість глобальної, затримки якої набагато менші [4], але є обмеження по розміру, яке залежить від покоління та архітектури [5].

Також на рисунку 1.5 показано, що спільна пам'ять блоку і глобальна пам'ять з'єднані через обчислювальні потоки, тобто немає прямого з'єднання, тому треба вручну зчитувати дані з глобальної пам'яті в спільну пам'ять блоків перед початком обчислень і завантажувати нові значення після обчислень, якщо необхідно. Необхідно зазначити, що таке архітектурне рішення сильно

вплинуло і поставило ще 1 підзадачу для проектування алгоритму, а саме оновлення значень в спільній пам'яті між блоками.

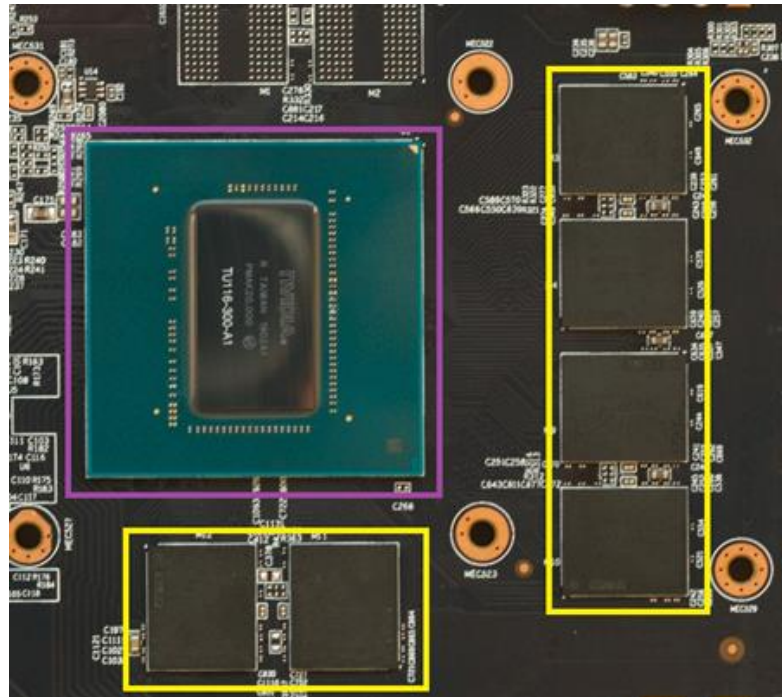


Рисунок 1.6 – Частина плати відеокарти, де в фіолетовому квадраті знаходиться чіп, а в жовтих прямокутниках глобальна пам'ять [6]

Всі потоки в блоці можна синхронізувати використовуючи функцію `__syncthreads()`, яка працює як бар'єр, тому, якщо ця функція використовується, тоді всі потоки в блоці мають одночасно закінчити свою роботу, інакше виникне взаємне блокування (deadlock).

Програми на CUDA можуть використовувати більшу кількість потоків для обчислень, ніж є в одному блоці потоків – для цього використовується декілька блоків з потоками, в яких кількість задіяних потоків однакова. Організація блоків потоків та потоків в середині блоків може бути трьох видів:

1. у вигляді вектора;

2. у вигляді матриці;
3. у вигляді паралелепіпеда.

Це зроблено для зручності розробки і ніяк не впливає на швидкість роботи програми [7].

Блоки потоків також можна синхронізувати за допомогою спеціальних груп, які з'явилися в нових версія CUDA, але для більш старих версій можна використовувати тільки потокові мультипроцесори (Streaming Multiprocessor), недоліком яких є менший обсяг спільної пам'яті ніж у звичайних потоків в середині блоку [7].

В цьому розділі було розглянуто принцип роботи послідовного та паралельного алгоритму SVD для трьохдіагональної матриці, а також представлені ключові складові архітектури CUDA, які будуть використані під час імплементації алгоритму.

В наступному розділі буде детально описані всі етапи розробки алгоритму та проблеми, які вирішувались.

РОЗДІЛ 2. Розробка паралельного алгоритму SVD алгоритму для трьохдіагональної матриці

2.1 Опис роботи базового паралельного алгоритму

Щоб зрозуміти, які операції над матрицею можна робити паралельно розглянемо роботу послідовного алгоритму.

На вході маємо трьохдіагональну матрицю, у якої елементи над діагоналлю ненульові (рисунок 2.1). Проведемо декілька ітерацій для першого кроку описаного в розділі 1.1.

	1	2	3	4	5	6	7	8	9	10	
1	x	a									1
2	0	x	a								2
3		0	x	a							3
4			0	x	a						4
5				0	x	a					5
6					0	x	a				6
7						0	x	a			7
8							0	x	a		8
9								0	x	a	9
10									0	x	10
	1	2	3	4	5	6	7	8	9	10	

Рисунок 2.1 – Приклад трьохдіагональної матриці, у якої елементи над діагоналлю ненульові

На першій та другій ітерації працюємо над першим квадратом як зображено на рисунку 2.2 та 2.3.

	1	2	3	4	5	6	7	8	9	10	
1	x	a									1
2	0	x	a								2
3		0	x	a							3
4			0	x	a						4
5				0	x	a					5
6					0	x	a				6
7						0	x	a			7
8							0	x	a		8
9								0	x	a	9
10									0	x	10
	1	2	3	4	5	6	7	8	9	10	

Рисунок 2.2 – Приклад першої ітерації над матрицею

	1	2	3	4	5	6	7	8	9	10	
1	y	0									1
2	b	x	a								2
3		0	x	a							3
4			0	x	a						4
5				0	x	a					5
6					0	x	a				6
7						0	x	a			7
8							0	x	a		8
9								0	x	a	9
10									0	x	10
	1	2	3	4	5	6	7	8	9	10	

Рисунок 2.3 – Приклад другої ітерації над матрицею

Як бачимо їх опрацювання не може бути виконане паралельно. Але на третій ітерації можна побачити, що перший квадрат діагоналі знову може бути оброблений (рисунок 2.4).

	1	2	3	4	5	6	7	8	9	10	
1	y	0									1
2	b	y	0								2
3		b	x	a							3
4			0	x	a						4
5				0	x	a					5
6					0	x	a				6
7						0	x	a			7
8							0	x	a		8
9								0	x	a	9
10									0	x	10
	1	2	3	4	5	6	7	8	9	10	

Рисунок 2.4 – Приклад третьої ітерації над матрицею

Через ще 2 ітерації перший квадрат знову може бути опрацьований (рисунок 2.5). Отже можемо зробити висновок, що на ітерації $size - 1$, де $size$ – це розмір матриці, може бути оброблено $size / 2$ квадратів на діагоналі (рисунок 2.6). Отже, оптимальна кількість потоків для матриці обчислюється за формулою (2.1).

$$\text{кількість потоків} = \text{розмір матриці} / 2 \quad (2.1)$$

	1	2	3	4	5	6	7	8	9	10	
1	z	c									1
2	0	z	0								2
3		b	y	0							3
4			b	y	a						4
5				0	x	a					5
6					0	x	a				6
7						0	x	a			7
8							0	x	a		8
9								0	x	a	9
10									0	x	10
	1	2	3	4	5	6	7	8	9	10	

Рисунок 2.5 – Приклад п'ятої ітерації над матрицею

	1	2	3	4	5	6	7	8	9	10	
1	p	e									1
2	0	p	0								2
3		b	t	0							3
4			d	t	a						4
5				0	z	c					5
6					0	z	a				6
7						0	y	0			7
8							b	y	a		8
9								0	x	a	9
10									0	x	10
	1	2	3	4	5	6	7	8	9	10	

Рисунок 2.6 – Приклад дев'ятої ітерації над матрицею

Якщо розмір матриці непарний, тоді кількість потоків округлюється в меншу сторону.

При цьому перший потік починає обчислення без затримки, другий з затримкою в 2 ітерації, для третього 6, і так далі, тобто:

$$\text{час затримки} = \text{номер потоку} \times 2 \quad (2.2)$$

де час затримки – це кількість пустих ітерацій для потоку, та нумерація потоків починається з нуля.

Тепер розглянемо, як змінюються матриці L та R при виконанні другого кроку. Нехай маємо квадратні матриці L та R, розмір яких 6, і вони складаються

з будь-яких елементів та матрицю повороту Гівенса G. Перемноживши L та R на матрицю G, отримаємо результати показані на рисунках 2.7 та 2.8:

$$\begin{matrix} & \mathbf{G} & & \mathbf{L} & & \mathbf{L'} \\ \begin{pmatrix} 0.5 & 0.3 & 0 & 0 & 0 & 0 \\ 0.3 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} & \times & \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 1 \\ 2 & 3 & 4 & 5 & 1 & 2 \\ 3 & 4 & 5 & 1 & 2 & 3 \\ 4 & 5 & 1 & 2 & 3 & 4 \\ 5 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 1 \end{pmatrix} & = & \begin{pmatrix} 1.1 & 1.9 & 2.7 & 3.5 & 2.8 & 1.1 \\ 1.3 & 2.1 & 2.9 & 3.7 & 2 & 1.3 \\ 3 & 4 & 5 & 1 & 2 & 3 \\ 4 & 5 & 1 & 2 & 3 & 4 \\ 5 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 1 \end{pmatrix}
 \end{matrix}$$

Рис 2.7 – Результат множення матриць G та L

$$\begin{matrix} & \mathbf{R} & & \mathbf{G} & & \mathbf{R'} \\ \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 1 \\ 2 & 3 & 4 & 5 & 1 & 2 \\ 3 & 4 & 5 & 1 & 2 & 3 \\ 4 & 5 & 1 & 2 & 3 & 4 \\ 5 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 1 \end{pmatrix} & \times & \begin{pmatrix} 0.5 & 0.3 & 0 & 0 & 0 & 0 \\ 0.3 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} 1.1 & 1.3 & 3 & 4 & 5 & 1 \\ 1.9 & 2.1 & 4 & 5 & 1 & 2 \\ 2.7 & 2.9 & 5 & 1 & 2 & 3 \\ 3.5 & 3.7 & 1 & 2 & 3 & 4 \\ 2.8 & 2 & 2 & 3 & 4 & 5 \\ 1.1 & 1.3 & 3 & 4 & 5 & 1 \end{pmatrix}
 \end{matrix}$$

Рисунок 2.8 – Результат множення матриць R та G

З результатів видно, що при множенні матриць G на L та R на G, для матриці L змінюються ті рядки, які і в трьохдіагональній матриці, а в матриці R ті ж стовпчики. Отже, якщо всі потоки будуть синхронізовано пересуватись по діагоналі матриці з кроком в 2 елементи, тоді синхронізувати доступ до матриць M, L та R нам не потрібно.

2.2 Оптимізація роботи з пам'яттю.

В умовах, коли ми маємо обмеження по розміру пам'яті необхідно, щоб вона використовувалась максимально ефективно. Для цього було проаналізовано як працює алгоритм за різними матрицями, які частини матриці можна зберігати в пам'яті блоку, а в які краще зчитувати з глобальної пам'яті з більшими затримками.

Також, не слід забувати про апаратне кешування даних, при якому рейтинг влучності по кешу може позитивно вплинути на швидкодію. Тому, якщо матриця повністю зберігається в пам'яті, тоді краще її зберігати у вигляді одновимірного масиву і вручну обчислювати всі зміщення, тому що саме такий підхід дозволяє покращити рейтинг кешу [8].

Одна з вимог до API розробленого алгоритму - це автоматично забезпечувати найкращу конфігурацію по розподілу пам'яті між матрицями, кількість блоків та потоків в них для досягнення найбільшої швидкості.

2.2.1 Оптимізація зберігання трьохдіагональної матриці для одного блоку

Оскільки алгоритм використовує трьохдіагональну матрицю, можна не зберігати в пам'яті велику кількість нульових елементів, які не змінюються.

Розглянемо матрицю розміром 25 на 25, яка зображена на рисунку 2.9. Для такої матриці кількість елементів становить 625, при чому кількість елементів, які необхідні для обчислень і можуть змінюватися всього 73. Оскільки елементами матриці є числа з плаваючою комою подвійної точності (double), для якого розмір 8 байт, такий підхід економить 552 байти або, іншими словами, зменшує розмір трьохдіагональної матриці більше ніж в 7 разів.

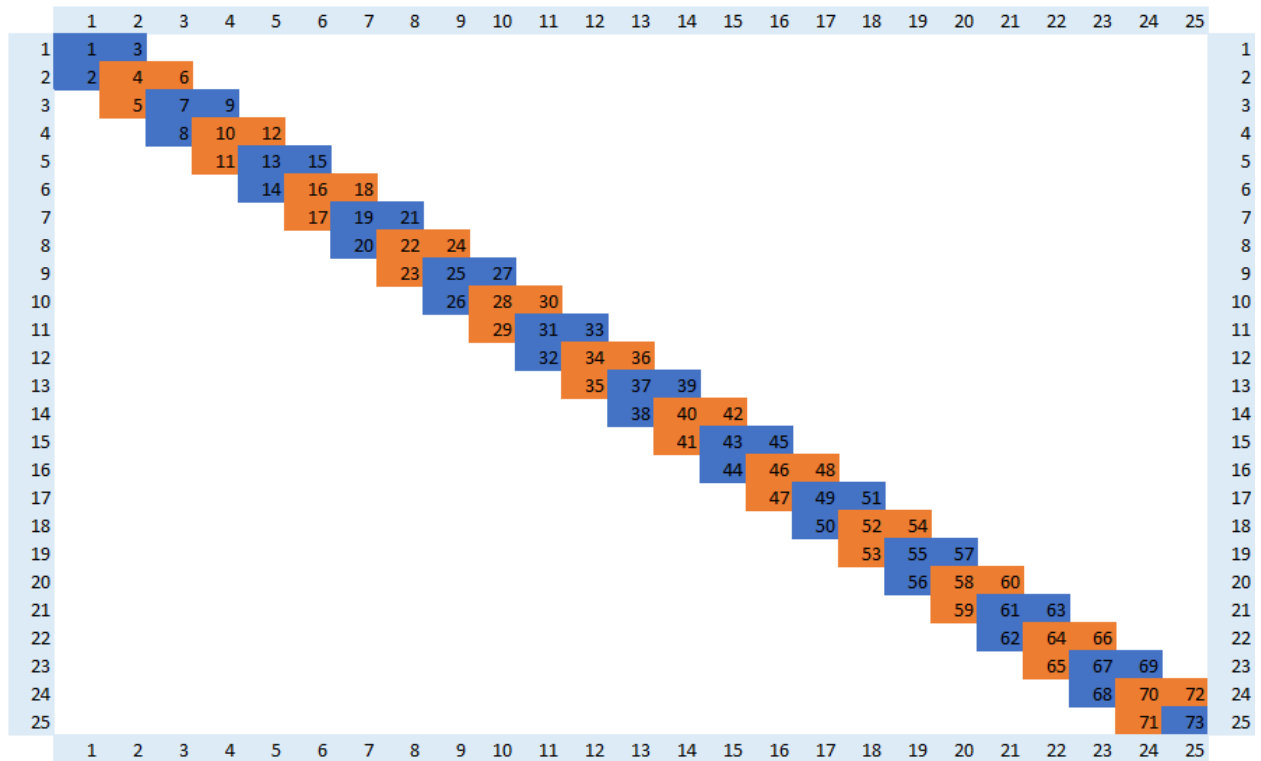


Рисунок 2.9 Приклад тридіагональної матриці розміру 25 на 25

Тому в алгоритмі матриця M буде представлена у вигляді одновимірного масиву, де елементи розташовані наступним чином (рисунок 2.10):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72
73																							

Рисунок 2.10 – Приклад збереження тридіагональної матриці розміру 25 на 25 у вигляді вектору

В даному векторі значення головної діагоналі розташовані на i -й позиції, елементи під діагоналлю на $i + 1$ позиції та елементи над діагоналлю на $i + 2$ позиції.

2.2.2 Оптимізація зберігання трьохдіагональної матриці для декількох блоків

Тепер розглянемо випадок, при якому матриця дуже великого розміру не може вміститися в 1 блоці спільної пам'яті. В останніх поколіннях відеокарт

розмір спільної пам'яті блоків дещо коливається, тому візьмемо за основу звичний розмір для попереднього покоління в 48Кбайт, в який поміщається 6144 елементів з плаваючою комою подвійної точності типу double. Враховуючи оптимізацію по зберіганню трьохдіагональної матриці, в один блок поміщається квадратна матриця розміру 2046 на 2046, тобто такий сценарій досить ймовірний для деяких напрямків, таких як Big Data.

Розглянемо приклад для матриці з рисунку 2.9, розмір якої 25 на 25, і нехай маємо фіксований розмір блоку 28 елементів типу double або 224 байти. Тоді для зберігання такої матриці будуть задіяні 3 блоки як показано на рисунку 2.11.

Через те, що алгоритм працює з квадратами, які знаходяться на діагоналі, блоки мають спільні елементи, наприклад, перший та другий блок мають спільний елемент під номером 28 (рисунок 2.11), другий та третій блок мають

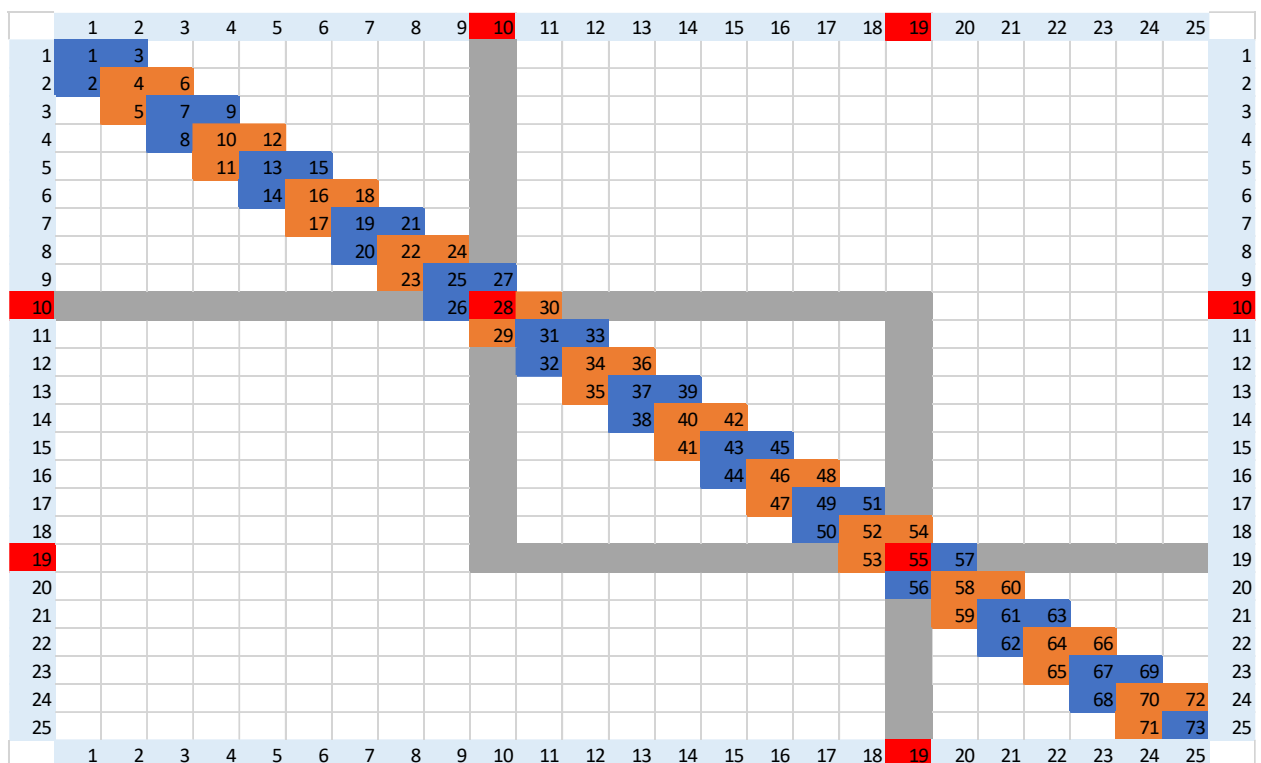


Рисунок 2.11 – Приклад збереження тридіагональної матриці розміру 25 на 25 в трьох блоках розміру 224 байти.

спільний елемент 55 (рисунок 2.11), значення яких необхідно синхронізувати, тому що після обчислень актуальне значення залишається в середині спільної пам'яті блоку, до якої інші блоки доступу не мають. Також при використанні декількох блоків, необхідно, щоб вони синхронно рухались. Детальний опис синхронізації значень матриці та потоків між блоками буде в наступних розділах.

Оскільки сумарний розмір блоків більший ніж розмір оптимізованої матриці, для третього блоку пусті місця будуть заповнені нулями. Вигляд матриці по блокам зображений на рисунку 2.12.

Block 1																											
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Block 2																											
28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55
Block 3																											
55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	0	0	0	0	0	0	0	0	0

Рисунок 2.12 – Приклад збереження оптимізованої тридіагональної матриці розміру 25 на 25 в трьох блоках розміру 224 байти.

2.3 Оптимізація множення для кроків алгоритму

Для прискорення роботи алгоритму необхідно не тільки оптимізувати роботу з пам'яттю, а ще й зменшити, або взагалі, прибрати з алгоритму кроки, які ніяк не впливають на результат.

Оскільки матриця повороту Гівенса – це майже одинична матриця, у якій 4 елементи є ненульовими і вони знаходяться на діагоналі, і згадуючи властивість одиничної матриці, де множення довільної матриці на одиничну відповідної розмірності дає в результаті ту ж саму матрицю, є сенс оптимізувати ці множення, оскільки в усіх кроках множаться матриці M , L та R множаться саме на цю матрицю.

2.3.1 Оптимізація множення для трьохдіагональної матриці

Розглянемо як змінюються елементи трьохдіагональної матриці під час множення на матрицю повороту Гівенса (рисунок 2.13).

$$\begin{matrix} & \mathbf{M} & & \mathbf{G} & & \mathbf{M}' \\ \begin{pmatrix} 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 4 & 2 & 0 & 0 & 0 \\ 0 & 0 & 6 & 3 & 0 & 0 \\ 0 & 0 & 0 & 8 & 4 & 0 \\ 0 & 0 & 0 & 0 & 10 & 5 \\ 0 & 0 & 0 & 0 & 0 & 12 \end{pmatrix} & \times & \begin{pmatrix} 0.89 & -0.4 & 0 & 0 & 0 & 0 \\ 0.45 & 0.89 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} 2.2 & -0 & 0 & 0 & 0 & 0 \\ 1.8 & 3.6 & 2 & 0 & 0 & 0 \\ 0 & 0 & 6 & 3 & 0 & 0 \\ 0 & 0 & 0 & 8 & 4 & 0 \\ 0 & 0 & 0 & 0 & 10 & 5 \\ 0 & 0 & 0 & 0 & 0 & 12 \end{pmatrix}
 \end{matrix}$$

Рисунок 2.13 – Приклад множення трьохдіагональної матриці на матрицю повороту

При множенні матриць, змінилися тільки елементи, які знаходяться в квадраті, над яким проводяться перетворення, тому можна не множити всі елементи матриць, а всього елементи, які знаходяться в квадраті. З цього можна зробити ще один висновок, що не потрібно зберігати всю матрицю повороту, а достатньо лише двох елементів – синуса та косинуса повороту. Отже для всіх оптимізацій маємо 2 функції:

- 1) Функція для обробки квадрату, коли елемент над діагоналлю не є нульовим:

```

__device__ void Givents_Top(double* s_bdm, unsigned long pos)
{
    unsigned long rid = pos * 3;
    if (abs(s_bdm[rid + 2]) < eps)
    {
        return;
    }
    double r = sqrt(sqr(s_bdm[rid]) + sqr(s_bdm[rid + 2]));
    double c = 1, s = 0;
    if (r > eps)
    {
        c = s_bdm[rid] / r;
        s = s_bdm[rid + 2] / r;
    }

    double c11 = s_bdm[rid] * c + s_bdm[rid + 2] * s;
    double c12 = s_bdm[rid + 2] * c - s_bdm[rid] * s;
    double c21 = s_bdm[rid + 1] * c + s_bdm[rid + 3] * s;
    double c22 = s_bdm[rid + 3] * c - s_bdm[rid + 1] * s;
    s_bdm[rid] = c11;
    s_bdm[rid + 1] = c21;
    s_bdm[rid + 2] = c12;
    s_bdm[rid + 3] = c22;
}

```

```
}
```

2) Функція для обробки квадрату, коли елемент під діагоналлю не є нульовим:

```
__device__ void Givents_Bot(double* s_bdm, ulong pos, double eps
= 1e-8)
{
    unsigned long rid = pos * 3;
    if (abs(s_bdm[rid + 1]) < eps)
    {
        return;
    }
    double r = sqrt(sqr(s_bdm[rid]) + sqr(s_bdm[rid + 1]));
    double c = 1, s = 0;
    if (r > eps)
    {
        c = s_bdm[rid] / r;
        s = s_bdm[rid + 1] / r;
    }
    double c11 = s_bdm[rid] * c + s_bdm[rid + 1] * s;
    double c12 = s_bdm[rid + 2] * c + s_bdm[rid + 3] * s;
    double c21 = s_bdm[rid + 1] * c - s_bdm[rid] * s;
    double c22 = s_bdm[rid + 3] * c - s_bdm[rid + 2] * s;
    s_bdm[rid] = c11;
    s_bdm[rid + 1] = c21;
    s_bdm[rid + 2] = c12;
    s_bdm[rid + 3] = c22;
}
```

Функції приймають наступні параметри:

- `double* s_bdm` – вказівник на трьохдіагональну матрицю, яка знаходиться в спільній пам'яті блоку
- `ulong pos` – номер квадрату в трьохдіагональній матриці
- `double eps` – точність обчислень

Функції працюють наступним чином:

- 1) обчислення зміщення для елементів в трьохдіагональній матриці
- 2) перевірка значення елемента, який буде обнулений; якщо абсолютне значення менше точності, тоді перетворення не відбувається, інакше переходимо до пункту 3)
- 3) обчислення синуса та косинуса для матриці повороту

- 4) множення елементів
- 5) запис нових значень в тріохдіагональну матрицю

2.3.2 Оптимізація множення для матриць L та R

На рисунках 2.7 та 2.8 показано, що для матриць L та R множення на матрицю Гівенса змінює тільки рядок та стовбець відповідно, тому повністю проводити множення матриць сенсу немає і в алгоритмі відбувається обрахунок тільки тих елементів, які змінилися.

Оскільки множення матриці M на G та L, R на G відбуваються в одному потоці, можна провести 2 крок алгоритму відразу після першого. Тоді маємо такі функції для обробки матриць:

- 1) Функція для обробки квадрату, коли елемент над діагоналлю не є нульовим:

```
__device__ void Givents_Top(double* s_bdm, unsigned long pos,
ulong offset, volatile double* R, ulong r_size, double eps = 1e-
8)
{
    unsigned long rid = pos * 3;
    if (abs(s_bdm[rid + 2]) < eps)
    {
        return;
    }
    double r = sqrt(sqr(s_bdm[rid]) + sqr(s_bdm[rid + 2]));
    double c = 1, s = 0;
    if (r > eps)
    {
        c = s_bdm[rid] / r;
        s = s_bdm[rid + 2] / r;
    }
    double c11 = s_bdm[rid] * c + s_bdm[rid + 2] * s;
    double c12 = s_bdm[rid + 2] * c - s_bdm[rid] * s;
    double c21 = s_bdm[rid + 1] * c + s_bdm[rid + 3] * s;
    double c22 = s_bdm[rid + 3] * c - s_bdm[rid + 1] * s;
    s_bdm[rid] = c11;
    s_bdm[rid + 1] = c21;
    s_bdm[rid + 2] = c12;
    s_bdm[rid + 3] = c22;
    ulong id1 = r_size * (pos + offset);
    ulong id2 = r_size * (pos + offset + 1);
```

```

for (int i = 0; i < r_size; ++i)
{
    double v1 = R[id1 + i];
    double v2 = R[id2 + i];
    R[id1 + i] = v1 * c - v2 * s;
    R[id2 + i] = v2 * c + v1 * s;
}
}

```

2) Функція для обробки квадрату, коли елемент під діагоналлю не є нульовим:

```

__device__ void Givents_Bot(double* s_bdm, ulong pos, ulong
offset, volatile double* L, ulong l_size, double eps = 1e-8)
{
    unsigned long rid = pos * 3;
    if (abs(s_bdm[rid + 1]) < eps)
    {
        return;
    }
    double r = sqrt(sqr(s_bdm[rid]) + sqr(s_bdm[rid + 1]));
    double c = 1, s = 0;
    if (r > eps)
    {
        c = s_bdm[rid] / r;
        s = s_bdm[rid + 1] / r;
    }
    double c11 = s_bdm[rid] * c + s_bdm[rid + 1] * s;
    double c12 = s_bdm[rid + 2] * c + s_bdm[rid + 3] * s;
    double c21 = s_bdm[rid + 1] * c - s_bdm[rid] * s;
    double c22 = s_bdm[rid + 3] * c - s_bdm[rid + 2] * s;
    s_bdm[rid] = c11;
    s_bdm[rid + 1] = c21;
    s_bdm[rid + 2] = c12;
    s_bdm[rid + 3] = c22;
    ulong id1 = l_size * (pos + offset);
    ulong id2 = l_size * (pos + offset + 1);
    for (int i = 0; i < l_size; ++i)
    {
        double v1 = L[id1 + i];
        double v2 = L[id2 + i];

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

```

Функції приймають наступні параметри:

- `double* s_bdm` – вказівник на трьохдіагональну матрицю, яка знаходиться в спільній пам'яті блоку
- `ulong pos` – номер квадрату в трьохдіагональній матриці
- `ulong offset` – зміщення для матриці L (R) в глобальній пам'яті відповідно до номеру блоку, в якому проводяться обчислення
- `volatile double* L` (R) – вказівник на матрицю L (R) в глобальній пам'яті. Ключове слово `volatile` говорить про те, що кешоване значення може бути застарілим і необхідно кожного разу перерахувати значення з глобальної пам'яті.
- `double eps` – точність обчислень

Функції працюють наступним чином:

- 1) обчислення зміщення для елементів в трьохдіагональній матриці
- 2) перевірка значення елемента, який буде обнулений; якщо абсолютне значення менше точності, тоді перетворення не відбувається, інакше переходимо до пункту 3)
- 3) обчислення синуса та косинуса для матриці повороту
- 4) множення елементів
- 5) запис нових значень в трьохдіагональну матрицю
- 6) обчислення зміщення відносно позиції квадрату для матриці L (R)
- 7) множення елементів та запис нових значень до матриці L (R) в глобальній пам'яті

2.4 Синхронізація значень трьохдіагональних матриць та ітерацій потоків між блоками

Однією із найважчих та найважливіших задач паралельного алгоритму, який використовує декілька блоків – це міжблочна синхронізація потоків та синхронізація спільних елементів матриці для блоків.

Для того щоб почати вирішувати проблему синхронізації значень матриці, треба зрозуміти яким чином можна синхронізувати потоки між блоками маючи глобальну пам'ять та функцію синхронізації потоків в середині блоку.

2.4.1 Розробка методу синхронізації даних між блоками

Розглянемо приклад синхронізації одного значення, спільного для двох блоків з одним потоком. Нехай маємо змінну лічильник в глобальній пам'яті, початкове значення якої нуль і яка змінюється за наступним алгоритмом:

```
__global__ void add_func(volatile double* counter, int* lock,
double* buf)
{
    extern __shared__ double m[2];
    m[0] = counter[0];
    for (int j = 0; j < 5; ++j)
    {
        if (blockIdx.x == 0)
        {
            for (int i = 0; i < 5000; ++i);
            if (first_cycle == false)
                m[0] = get_new_value(counter, m[0], 0);
            m[0] += 1;
            update_global_memory(counter, m[0], 0);
            double v = counter[0];
        }
        else
        {
            double v = get_new_value(counter, m[0], 0);
            m[0] = v * 2;
            update_global_memory(counter, m[0], 0);
        }
        __syncthreads();
    }
}
```

- 1) перший блок додає до змінної одиницю і записує нове значення в глобальну пам'ять
- 2) другий блок має зачекати поки перший додасть своє значення змінної лічильника, а потім подвоїть актуальне значення

3) перший блок чекає поки другий змінить значення змінної в глобальній пам'яті і потім до нового значення додає одиницю.

Такі дії виконуються п'ять ітерацій. В результаті значення змінної має стати рівним 62 і значення змінної представлені в таблиці 2.1

Таблиця 2.1 Очікувані значення змінної між ітераціями

Номер ітерацій	Блок 1	Блок 2
1	1	2
2	3	6
3	7	14
4	15	30
5	31	62

Функція запису в глобальну пам'ять працює наступним чином:

```
__device__ void update_global_memory(volatile double* bdm,
double value, ulong pos, double eps = 1e-8)
{
    while (abs(bdm[pos] - value) > eps)
    {
        bdm[pos] = value;
    }
}
```

Функція запису в глобальну пам'ять `update_global_memory` оновлює значення, а потім перевіряє, щоб це значення там там з'явилося. Такий процес буде продовжуватись, доки не з'явиться нове значення.

```
__device__ double get_new_value(volatile double* bdm, double
old_value, ulong pos)
{
    double new_val = bdm[pos];
    while (true)
    {
```

```

        if (abs(new_val - old_value) > 1e-15) return new_val;
        new_val = bdm[pos];
    }
}

```

Функція оновлення значення `get_new_value` зчитує значення з глобальної пам'яті до тих пір, поки не зчитується нове значення.

Використання даних функцій забезпечує коректний запис та оновлення даних при роботі з глобальною пам'яттю, тому їх використання дозволить синхронізувати потоки між блоками наступним чином:

- всі потоки в блоках синхронізуються за допомогою функції `__syncthreads()`
- 1 потік в середині блоку буде чекати на оновлене значення від іншого блоку, таким чином ніби зависаючи
- коли потоки зчитують нове значення, вони доходять до функції `__syncthreads()` і таким чином запускають інші потоки

2.4.2 Синхронізація блоків та значень матриці для паралельного алгоритму

Розглянемо як це працює на паралельному алгоритмі SVD, для матриці з рисунка 2.14.

Перший блок відразу починає працювати, в той час як другий та третій чекають оновлених значень 28 та 55 елементів відповідно. Тільки коли перший блок оновить значення 28 елементу, другий блок почне робити першу ітерацію тільки після 10 ітерації першого блоку як зображено на рисунку 2.15.

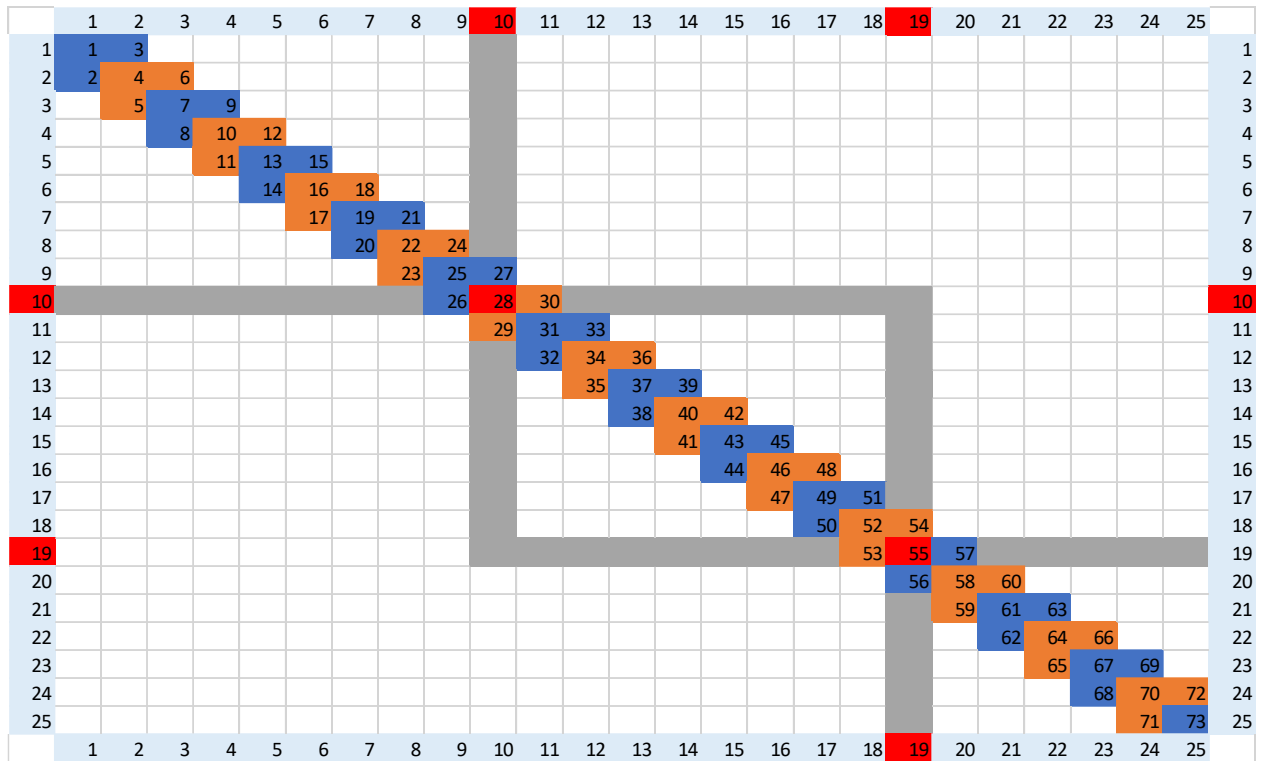


Рисунок 2.14 – Приклад трьохдіагональної матриці розміщеної в трьох блоках спільної пам'яті

Блок 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Блок 2	1										2	3	4	
Блок 3	1													

Рисунок 2.15 – Приклад кількості ітерацій для кожного блоку

Функції для читання та запису в глобальну пам'ять відрізняються від описаних в розділі 2.4.1. Відмінність полягає в тому, що значення тільки 28 елементу недостатньо, тому що є випадки, коли значення може не змінитись. Тому блоки звіряються не по новому значенню, а перевіряють чи виконана потрібна ітерація. Іншими словами другий блок хоче опрацювати свій перший квадрат з номерами 28, 29, 30, 31 (рис 2.14) і при цьому елемент 30 ненульовий, тому він зчитує значення 27 елементу з першого блоку і перевіряє чи є там нуль. Якщо нуля немає, тоді він чекає поки він там з'явиться. Після цього він зчитує значення 28 елементу, проводить розрахунки і записує 28, 29 і 30 елемент в спільну пам'ять, для того щоб потім з першого блоку знав, коли можна починати роботу.

Тобто ми синхронізуємо не 1 значення, а 3 для того щоб врахувати всі можливі ситуації. Якщо 2 елемента, над і під діагоналлю є нулями, тоді значення на перетині блоків більше не зміниться і блоки починають працювати незалежно.

2.4.3 Завершення роботи алгоритму

Робота алгоритму закінчується тоді, коли останній блок завершить свою роботу. Для того щоб кожен блок міг завершити свою роботу необхідні наступні умови:

- 1) Попередній блок завершив свою роботу (не стосується першого блоку). Ця умова перевіряється за допомогою змінної лічильника в глобальній пам'яті.
- 2) Кожен потік в блоці має пройти по всій діагоналі і не зробити жодного перетворення матриці, тобто всі елементи над та під діагоналлю мають бути рівні нулю. Якщо при обробці свого квадрату потік не робить ітерації, тоді він збільшує локальний лічильник пустих ітерацій. Як тільки кількість пустих ітерацій рівна розміру матриці, потік збільшує значення лічильника змінної, за допомогою атомарного додавання, яка знаходиться в спільній пам'яті блоку і служить для підрахунку завершених потоків в блоці. Як тільки кількість завершених потоків більше або дорівнює кількості потоків в блоці, всі потоки в блоці завершують свою роботу. Через те що використовується функція `__syncthreads()`, яка працює як бар'єр, потоки мають завершити свою роботу одночасно.

В даному розділі був досліджений принцип роботи послідовного SVD алгоритму для трьохдіагональної матриці, розглянутий підхід до розробки паралельної модифікації, оптимізації роботи з пам'яттю та різних функцій, які використовуються алгоритмом.

Також були показані та вирішені проблеми, такі як синхронізація блоків та спільних значень матриці між блоками, а також принцип завершення роботи алгоритму.

В наступному розділі буде проводитись тестування розробленого алгоритму на швидкість роботи для різних розмірів матриці із різною точністю обчислень, а також буде порівняння з існуючими паралельними алгоритмами SVD.

Схема роботи алгоритму представлена в Додатку А, програмний код алгоритму в Додатку Б та код API в Додатку В.

РОЗДІЛ 3. Тестування реалізованого алгоритму

В даному розділі буде досліджена швидкість роботи алгоритму за різних розмірів матриці, кількості блоків та потоків, час виконання для різних точностей обчислень та їх похибки, а також, проведено аналіз затримок по пам'яті.

Всі тестування проводилися в системі з наступними характеристиками:

- Процесор – Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz
- Оперативна пам'ять – 32 GB типу DDR4 3200 MHz
- Накопичувач – Samsung SSD 860 EVO 1TB
- Відеокарта – NVIDIA GeForce GTX 1660 Ti:
 - розмір глобальної пам'яті – 6 GB GDDR6
 - пропускна здатність пам'яті – 288 GB/s
 - розмір спільної пам'яті блоку – 64 KB
 - максимальна кількість потоків в блоці – 624

3.1 Тестування швидкості роботи алгоритму для різних розмірів матриці

Для того, щоб дослідити та оцінити швидкість роботи алгоритму, він буде протестований на матрицях з випадковими значеннями різного розміру, при цьому діапазони можливих значень в матриці будуть змінюватися. Такий підхід дозволить оцінити такі параметри:

- загальний час виконання
- кількість ітерацій
- час на виконання 1 ітерації

Загальний час виконання – це час алгоритму на відеокарті. Час на створення матриці та запису необхідних даних у пам'ять відеокарти не враховується.

Кількість ітерацій – це сума кількості операцій над матрицею для кожного потоку. Для цього створюється масив, розмір якого дорівнює кількості всіх потоків, куди, після завершення роботи, кожен потік запише свою

кількість ітерацій. Після цього, на процесорі, підраховується сума всіх значень з масиву.

Середній час виконання 1 ітерації – це відношення загального часу виконання до всієї кількості ітерацій. В цей час також враховується затримка по синхронізації. Оскільки алгоритм може мати ітерації без обчислень над матрицею, цікаво дослідити, як буде змінюватися цей час для різних розмірів матриці.

Для кожного діапазону значень виконувалось 5 ітерацій для різних матриць, після чого кількість ітерацій та час виконання рахувались як середнє арифметичне і з отриманих значень рахувався середній час ітерації.

Тестування для розміру матриці 20 на 20 елементів виконувалося на 1 блоці з 10 потоками і з точністю обчислень 10^{-8} . Результати представлені в Табл. 3.1.

Таблиця 3.1 Результати для розміру матриці 20 на 20 елементів на 10 потоках і з точністю обчислень 10^{-8} :

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	19946	85	0.00425
[0.00, 5.00]	23278	87	0.00373
[0.00, 10.00]	32320	121	0.00376
[0.00, 50.00]	86214	234	0.00271
[0.00, 1000.00]	35802	122	0.00341

Тестування для розміру матриці 50 на 50 елементів виконувалося на 1 блоці з 25 потоками і з точністю обчислень 10^{-8} . Результати представлені в Табл. 3.2.

Таблиця 3.2 Результати для розміру матриці 50 на 50 елементів на 25 потоках і з точністю обчислень 10^{-8} :

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	241623	524	0.00217
[0.00, 5.00]	180934	486	0.00269
[0.00, 10.00]	526510	954	0.00181
[0.00, 50.00]	696956	1182	0.0017
[0.00, 1000.00]	243615	684	0.00281

Тестування для розміру матриці 100 на 100 елементів виконувалося на 1 блоці з 50 потоками і з точністю обчислень 10^{-8} . Результати представлені в Табл. 3.3.

Таблиця 3.3 Результати для розміру матриці 100 на 100 елементів на 50 потоках і з точністю обчислень 10^{-8} :

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	496900	1034	0.00208
[0.00, 5.00]	3205170	4471	0.0014
[0.00, 10.00]	1332300	2664	0.002
[0.00, 50.00]	2103700	3288	0.00156
[0.00, 1000.00]	4120330	6739	0.00163

Тестування для розміру матриці 200 на 200 елементів виконувалося на 1 блоці з 100 потоками і з точністю обчислень 10^{-8} . Результати представлені в Табл. 3.4.

Таблиця 3.4 Результати для розміру матриці 200 на 200 елементів на 100 потоках і з точністю обчислень 10^{-8} :

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	1145220	2612	0.00228
[0.00, 5.00]	3294780	5047	0.00153
[0.00, 10.00]	1154460	2943	0.00255
[0.00, 50.00]	2988360	5855	0.00196
[0.00, 1000.00]	4945880	9331	0.00189

Тестування для розміру матриці 500 на 500 елементів виконувалося на 1 блоці з 250 потоками і з точністю обчислень 10^{-8} . Результати представлені в Табл. 3.5.

Таблиця 3.5 Результати для розміру матриці 500 на 500 елементів на 250 потоках і з точністю обчислень 10^{-8} :

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	4004050	9391	0.00235
[0.00, 5.00]	14768500	25234	0.00171
[0.00, 10.00]	6393350	14917	0.00233
[0.00, 50.00]	8896750	19424	0.00218
[0.00, 1000.00]	49579550	72864	0.00147

Тестування для розміру матриці 1000 на 1000 елементів виконувалося на 1 блоці з 500 потоками і з точністю обчислень 10^{-8} . Результати представлені в Табл. 3.6.

Таблиця 3.6 Результати для розміру матриці 1000 на 1000 елементів на 500 потоках і з точністю обчислень 10^{-8} :

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	23802900	38635	0.00162
[0.00, 5.00]	42754900	72422	0.00169
[0.00, 10.00]	42652600	67994	0.00159
[0.00, 50.00]	40920200	83327	0.00203
[0.00, 1000.00]	35695900	79761	0.00223

Тестування для розміру матриці 5000 на 5000 елементів виконувалося на 3 блоках з 624 потоками і з точністю обчислень 10^{-8} . Результати представлені в Табл. 3.7.

Таблиця 3.7 Результати для розміру матриці 5000 на 5000 елементів на 3 блоках з 624 потоками і з точністю обчислень 10^{-8} :

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	701971171	394510	0.00056
[0.00, 5.00]	2385725142	1031725	0.00043
[0.00, 10.00]	1537243293	785861	0.00051
[0.00, 50.00]	3035029987	1400095	0.00046
[0.00, 1000.00]	2688581725	3958828	0.00147

Тестування для розміру матриці 10000 на 10000 елементів виконувалося на 5 блоках з 624 потоками і з точністю обчислень 10^{-8} . Результати представлені в Табл. 3.8.

Таблиця 3.8 Результати для розміру матриці 10000 на 10000 елементів на 5 блоках з 624 потоками і з точністю обчислень 10^{-8} :

Діапазон значень	Кількість ітерацій	Час виконання, мс	Середній час 1 ітерації, мс
[0.00, 1.00]	1644255050	1012986	0.00062
[0.00, 5.00]	3099582752	1853057	0.00059
[0.00, 10.00]	2786180563	4200257	0.0015
[0.00, 50.00]	1979320996	3097922	0.00156
[0.00, 1000.00]	2382750772	3601241	0.00151

Розглянемо кількість ітерацій (рисунок 3.1) та час виконання алгоритму (рисунок 3.2) для матриць розміру 20, 50, 100 та 200.

Для матриці 20 на 20, для діапазону значень $[0, 1]$ час виконання є найменшим, а зі збільшенням діапазону значень час обчислень зростає, при цьому для діапазону $[0, 50]$ цей час найбільший. Це можна пояснити кількістю ітерацій, яка знадобилася для досягнення заданої точності обчислень.

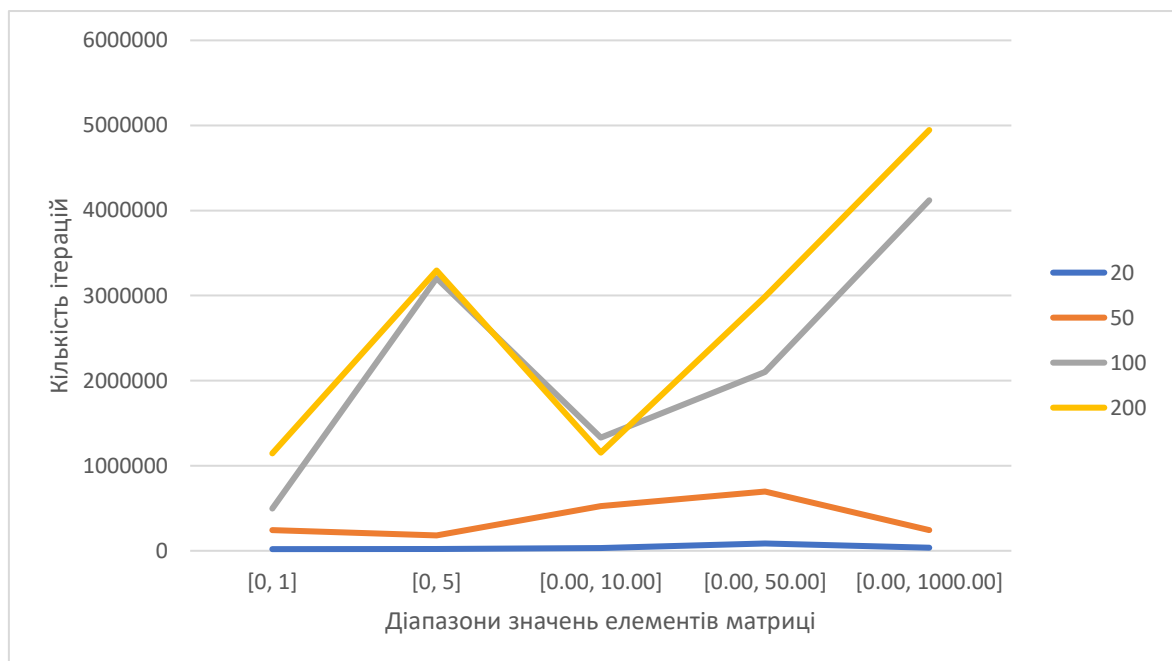


Рисунок 3.1 – Кількість ітерацій для матриць розміром 20, 50, 100, 200

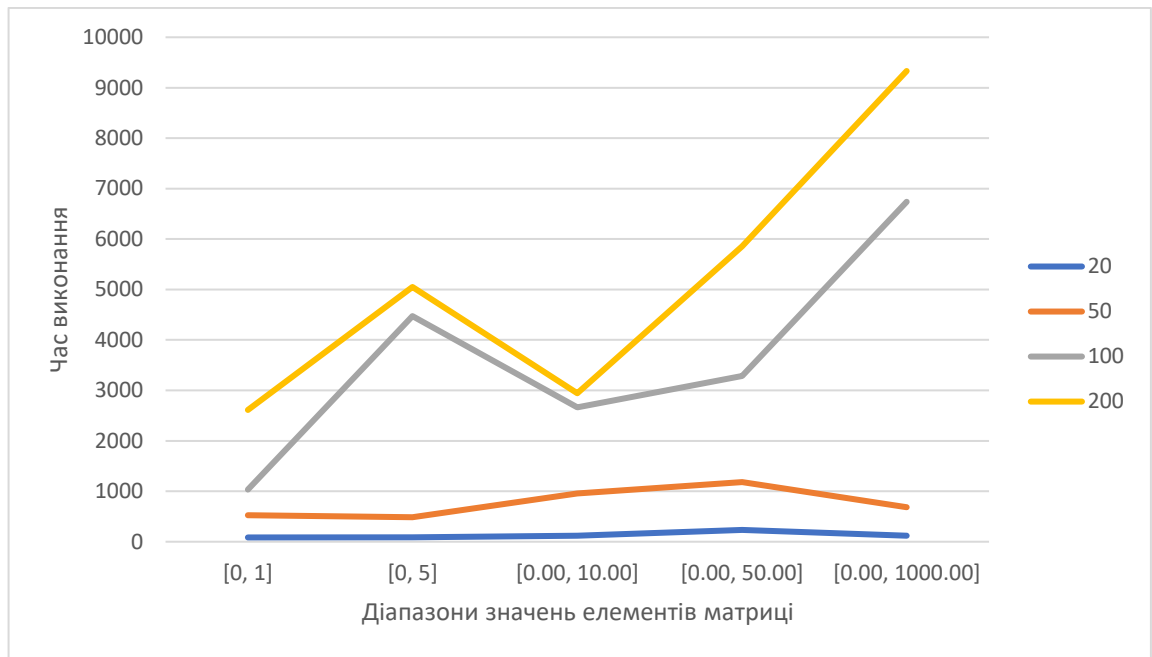


Рисунок 3.2 – Час виконання алгоритму для матриць розміром 20, 50, 100, 200

Час виконання та кількість ітерацій для матриці 50 на 50 збільшились в порівнянні з матрицею 20 на 20 елементів, при чому, середній час 1 ітерації зменшився, тому що використовувалось більше обчислювальних потоків.

Тенденція зберігається. Якщо порівнювати з матрицею 50 на 50, то для матриці 100 на 100 кількість ітерацій зростає швидше ніж час виконання – це за рахунок збільшення кількості потоків.

Необхідно зауважити, що в порівнянні з матрицею 100 на 100, для приблизно однакової кількості ітерацій, при розмірі матриці 200 на 200, час виконання збільшився. Це пояснюється тим, що деякі операції відбуваються довше і середній час ітерації зростає.

Для матриць більшого розміру є випадки, коли кількість ітерацій для матриці меншого розміру перевищує кількість ітерацій для матриці більшого розміру (рис. 3.3), при цьому час виконання алгоритму більшої матриці буде більшим (рис. 3.4), через те, що у матриці меншого розміру час 1 ітерації був меншим – багато елементів вже задовільняли точності і операції над ними не проводилися.

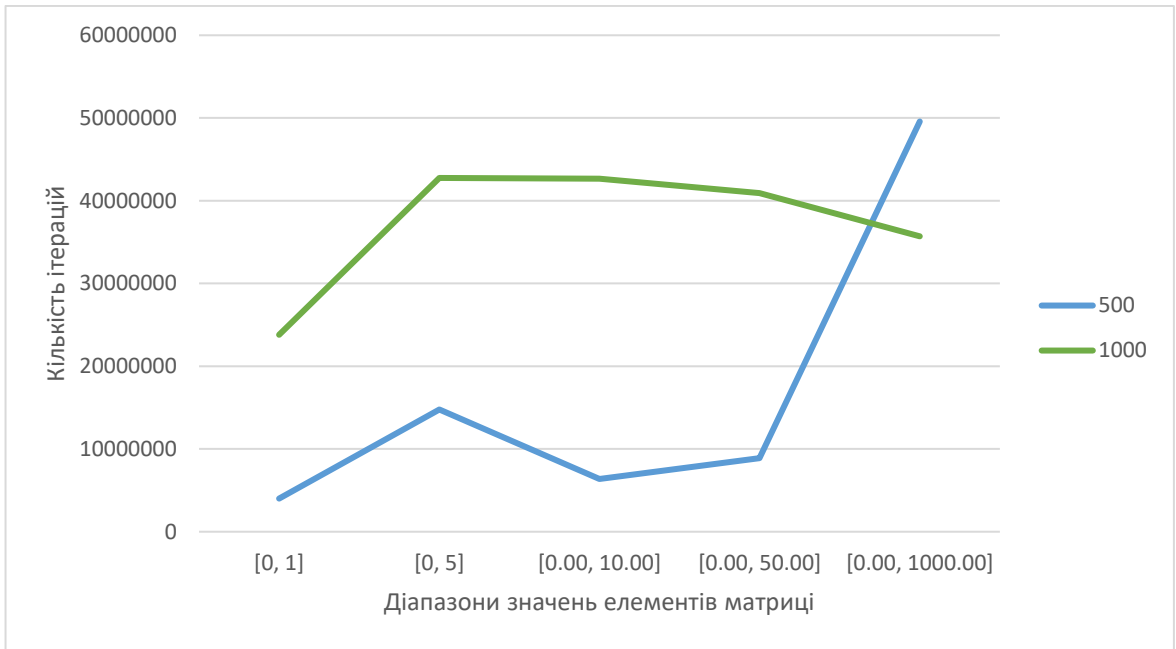


Рисунок 3.3 – Кількість ітерацій для матриць розміром 500 та 1000

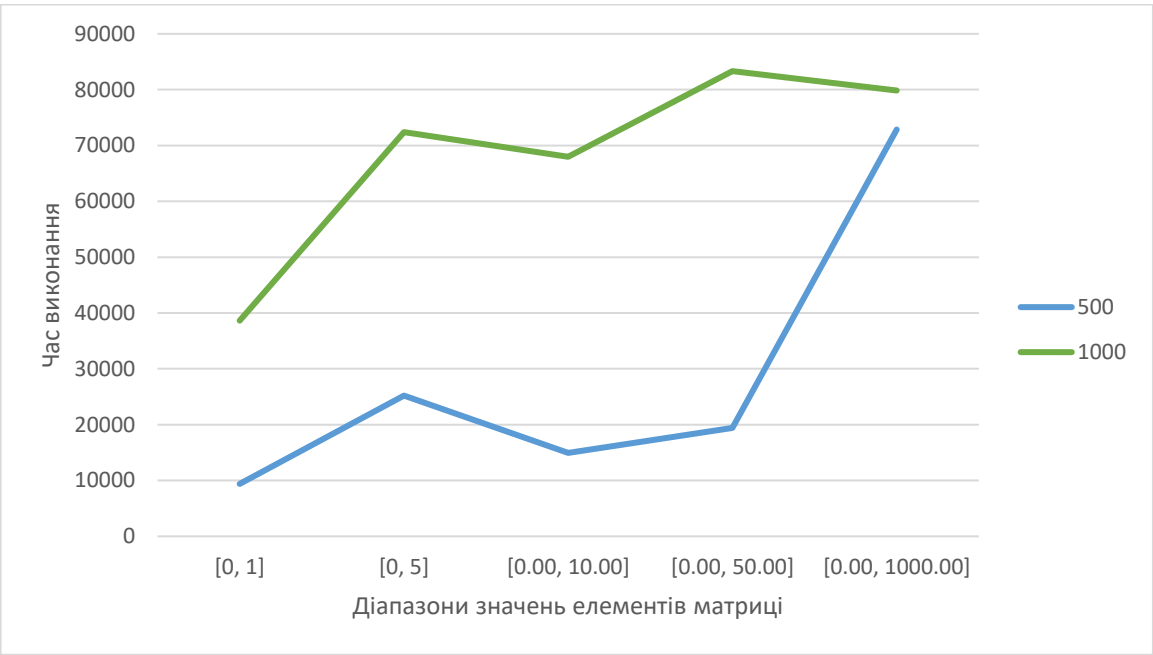


Рисунок 3.4 – Час виконання алгоритму для матриць розміром 500 та 1000

Для матриць розміру 5000 та 10000 кількість ітерацій перевищувала 10^9 для точності 10^{-8} . Для того, щоб прискорити роботу алгоритму для таких матриць, необхідно щоб кількість потоків в блоці була рівна половині розміру

матриці в блоці, а в нашому випадку їх було 624 з можливих 1000. Далі буде проведено аналіз швидкості роботи алгоритму для фіксованого розміру матриці з різною кількістю потоків, для того щоб дослідити як можна покращити роботу алгоритму. В нашому випадку, для таких матриць час виконання залежить більше від кількості ітерацій ніж, власне, від розміру матриці (рисунок 3.5 та рисунок 3.6).

Діапазон значень елементів матриці може вплинути на швидкість роботи. Для всіх розмірів матриць найменший час обчислень та кількість ітерацій саме для значень з проміжку $[0, 1]$. Зі збільшенням цього діапазону кількість ітерацій збільшується, але за яким законом сказати важко, але для матриць великого розміру діапазон сильніше впливає на час виконання, наприклад, на рисунку 3.6, для матриці 10000 на 10000, можна бачити стрімке зростання кількості ітерацій для діапазону $[0, 10]$, яка зросла більше ніж в 2 рази, а потім трималась на цьому рівні.

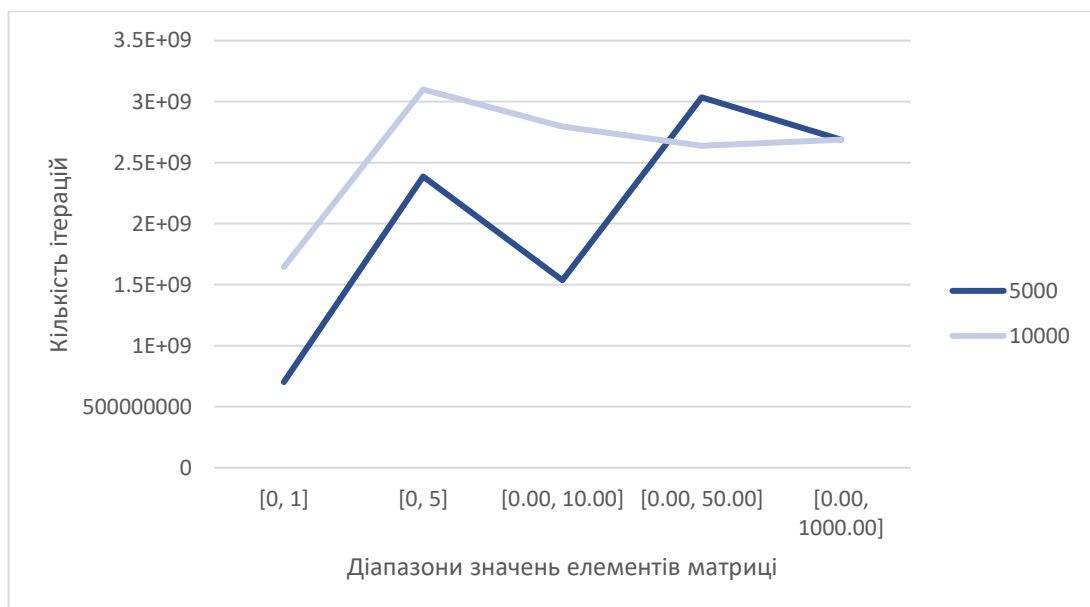


Рисунок – 3.5 Кількість ітерацій для матриць розміром 5000 та 10000



Рисунок 3.6 – Час виконання алгоритму для матриць розміром 5000 та 10000

3.2 Тестування швидкості роботи алгоритму для різних значень точності обчислень

Мета цього тесту – дослідити залежність кількості ітерації (часу обчислень) для матриць різного розміру з різною точністю обчислень.

Тестування проводилось за наступним сценарієм:

1. Генерується трьохдіагональна матриця розміру N .
2. Виконується SVD алгоритм для заданої точності.
3. Обчислення похибки:
 - а. Отримані матриці L , M' , R множаться на відеокарті. В результаті маємо отримати матрицю M^* , близьку по значенням до вхідної M .
 - б. Порівнюються всі елементи вхідної матриці M та отриманої M^* . Максимальне відхилення вважається похибкою обчислень.
4. Збереження результатів ітерації.
5. Продовження обчислень з більшою точністю.

Такий сценарій дозволяє точно визначити залежність кількості ітерацій від точності обчислень.

Результати для матриці розміру 200 на 200 елементів представлені в таблиці 3.9.

Таблиця 3.9 Результати для розміру матриці 200 на 200 елементів:

Кількість ітерацій	Точність обчислень	Похибка	Час виконання, 1мс	Середній час 1 ітерації, мс
783400	1.00E-02	9.00E-03	2311	0.002949962
1095800	1.00E-04	9.00E-05	3279	0.002992334
2878500	1.00E-06	8.00E-07	6990	0.002428348
13741200	1.00E-08	1.00E-08	20561	0.001496303
30175400	1.00E-10	1.00E-09	48619	0.001611213

З отриманих результатів видно, що при збільшенні точності обчислень з 10^{-6} до 10^{-8} час обчислень збільшився в 3 рази, а похибка зменшилась на порядок менше.

Результати для матриці розміру 500 на 500 елементів представлені в таблиці 3.10.

Таблиця 3.10 Результати для розміру матриці 500 на 500 елементів:

Кількість ітерацій	Точність обчислень	Похибка	Час виконання, 1мс	Середній час 1 ітерації, мс
5280750	1.00E-02	9.00E-03	9897	0.001874166
12939250	1.00E-04	1.00E-04	25850	0.001997797
36864250	1.00E-06	9.00E-07	74534	0.00202185
404350250	1.00E-08	1.00E-08	530041	0.001310846
470843000	1.00E-10	2.00E-09	634310	0.001347179

Як і для матриці розміру 200 на 200, збільшенні точності обчислень з 10^{-6} до 10^{-8} збільшило час обчислень в декілька разів, при цьому похибка зменшилась на порядок менше.

Результати для матриці розміру 1000 на 1000 елементів представлені в таблиці 3.11.

Таблиця 3.11 Результати для розміру матриці 1000 на 1000 елементів:

Кількість ітерацій	Точність обчислень	Похибка	Час виконання, 1мс	Середній час 1 ітерації, мс
36641500	1.00E-02	1.00E-02	64966	0.001773017
97980500	1.00E-04	9.00E-05	310887	0.003172948
129814000	1.00E-06	1.00E-08	338894	0.002610612
315327500	1.00E-08	1.00E-08	572256	0.001814799
365050500	1.00E-10	2.00E-09	829576	0.002272497

Для матриці розміру 1000 ситуація дещо змінилась. Точність обчислень 10^{-6} забезпечило таку ж точність як обчислення при точності 10^{-8} , а точність 10^{-10} зменшило похибку всього на порядок.

На рис. 3.7 видно, що час роботи алгоритму зростає лінійно зі збільшенням точності обчислень. На графіку зелена ламана – це час обчислень, а синя пряма – це графік лінійної функції.

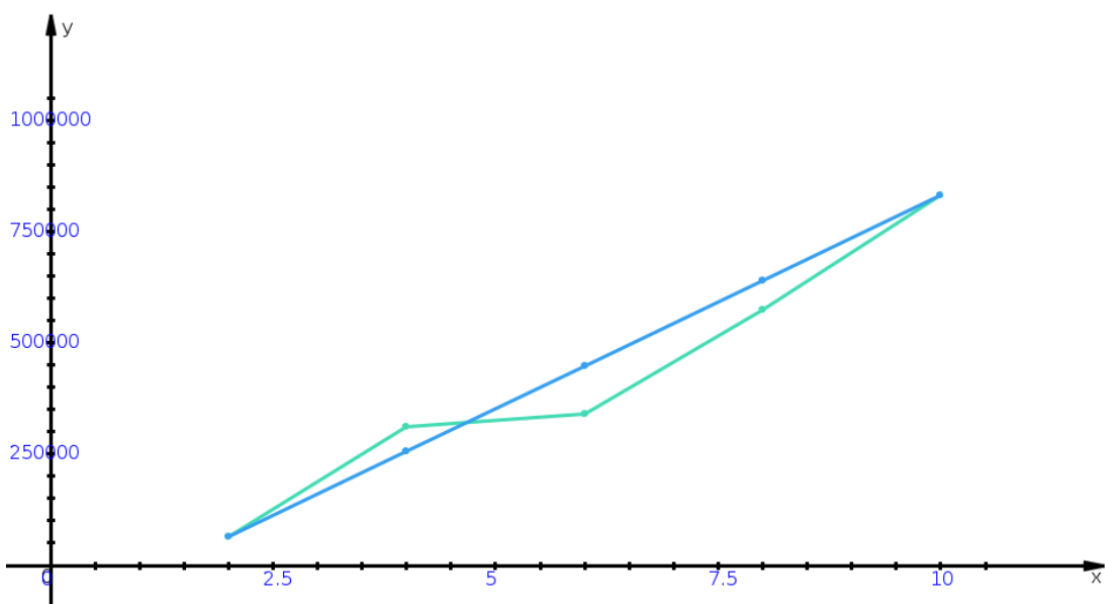


Рисунок 3.7 – Графік залежності часу роботи алгоритму від точності обчислень для матриці розміру 1000 на 1000

Отже, з отриманих результатів, найменшу похибку та непогану швидкість обчислень в порівнянні з іншими дає точність 10^{-6} (рисунок 3.8). Також, похибка обчислень досягає свого максимуму за точності 10^{-10} і зі збільшенням точності обчислень вона не буде зменшуватись через накопичення помилок.

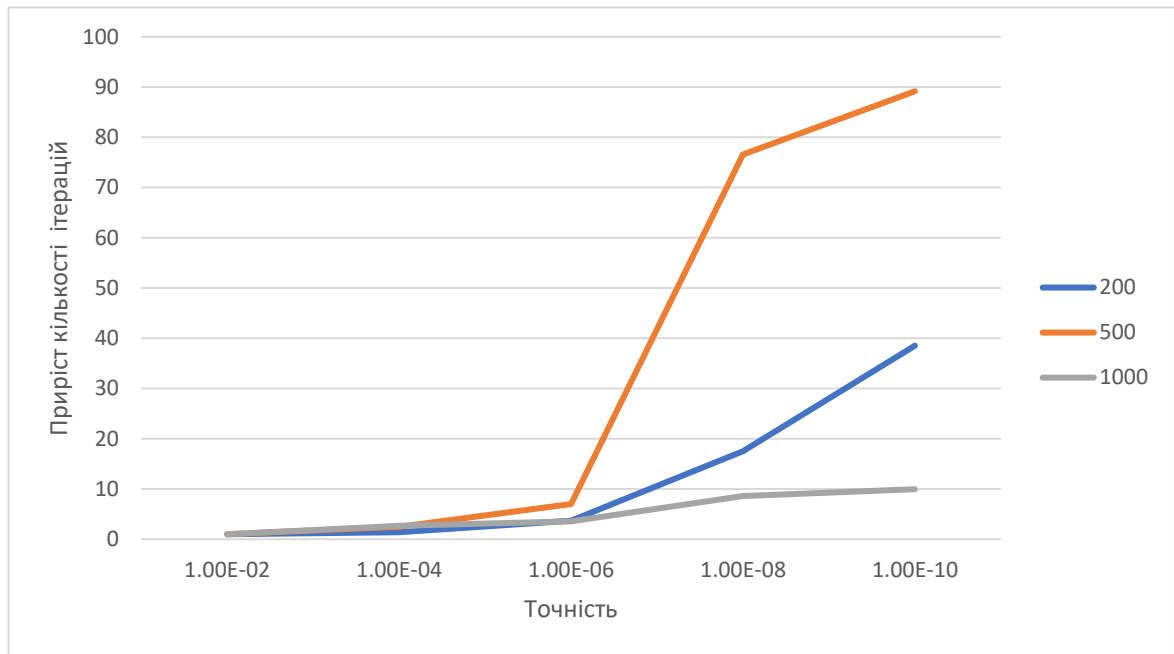


Рисунок 3.8 – Приріст кількості ітерацій зі збільшенням точності обчислень для матриць розміру 200, 500 та 1000

3.3 Порівняння швидкості роботи алгоритму за різної кількості обчислювальних потоків

Метою даного тестування є дослідження приросту швидкості роботи алгоритму зі збільшенням кількості обчислювальних потоків.

Тестування відбувалось за наступним алгоритмом:

1. Створення матриці розміру N.
2. Запуск алгоритму на k потоках.
3. Збереження результатів.

Кроки 2-3 повторюються для заданих значень кількості потоків.

Коефіцієнту прискорення – це відношення часу виконання алгоритму між k_i та k_0 кількістю потоків:

$$t_i/t_0,$$

де t_0 – це час виконання для початкової кількості потоків, t_i – це час виконання для кількості потоків з i -того рядка таблиці.

Ефективність прискорення – це відношення коефіцієнту прискорення до приросту кількості потоків:

$$\frac{t_i/t_0}{k_i/k_0}$$

де k_0 – це початкова кількість потоків, відносно якої буде рахуватися прискорення, k_i – це кількість потоків для i -того рядку в таблиці.

Ефективність прискорення буде рівне 1, якщо при збільшенні кількості потоків в N разів, час виконання також зменшиться в N разів.

В таблиці 3.12 наведені результати коефіцієнту прискорення та ефективності 1 потоку для матриці розміру 500 на 500 елементів для різної кількості потоків.

Таблиця 3.12 Результати для розміру матриці 500 на 500 елементів:

Кількість потоків	Час виконання, мс	Коефіцієнт прискорення	Ефективність прискорення
100	12278	1	1
200	7788	1.576528	0.788263996
250	6773	1.812786	0.725114425

В таблиці 3.13 наведені результати коефіцієнту прискорення та ефективності 1 потоку для матриці розміру 1000 на 1000 елементів для різної кількості потоків.

Таблиця 3.13 Результати для розміру матриці 1000 на 1000 елементів:

Кількість потоків	Час виконання, мс	Коефіцієнт прискорення	Ефективність прискорення
100	83337	1	1
200	57027	1.46136	0.730680204
250	50334	1.65568	0.662272023
500	35560	2.34356	0.468712036

Отже, з отриманих результатів, кількість потоків істотно впливає на швидкість роботи алгоритму (рисунок 3.9). Оскільки кількість потоків не може бути більшою за половинний розмір матриці, а великі матриці не можуть бути записані в 1 блок, необхідно підбирати такі розміри блоків, при яких кількість потоків буде максимальною.

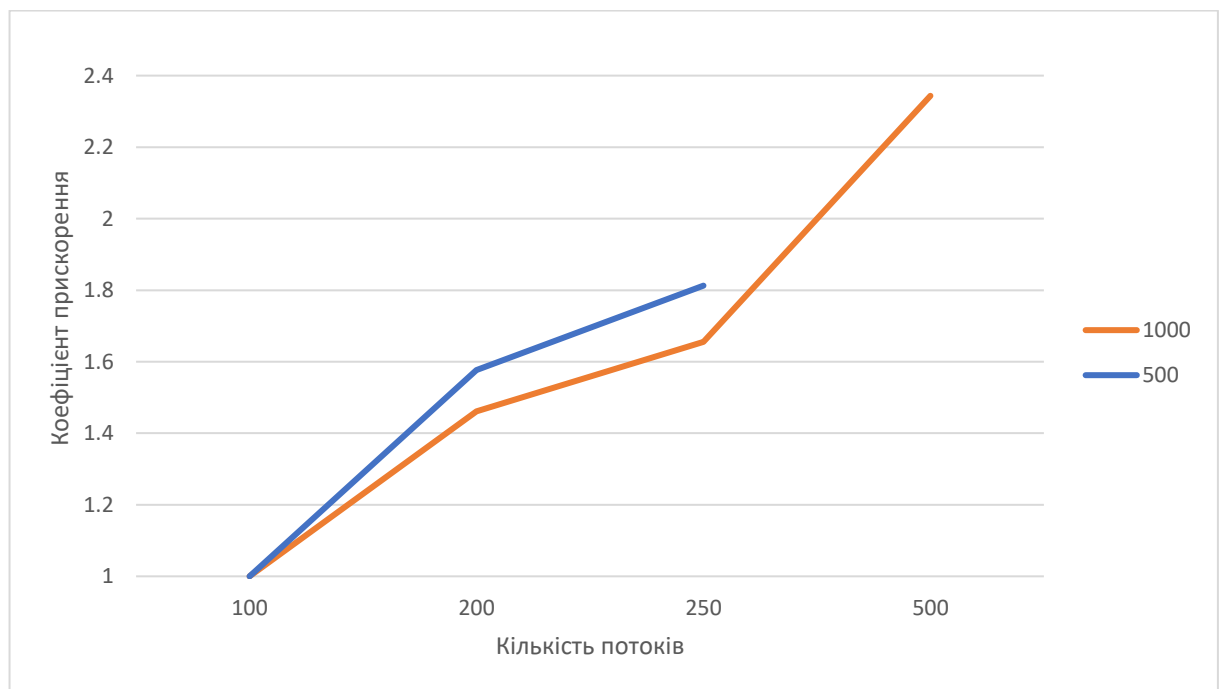


Рисунок 3.9 – Графік коефіцієнту прискорення для матриці

3.4 Порівняння швидкості роботи алгоритму з існуючими рішеннями

Сьогодні існує багато модифікацій SVD алгоритму, але результати тестування не є очевидними. Наприклад, на рисунку 3.10 засоби бібліотеки CuSolver, який базується на методі Якобі, для матриці 500 на 500 обчислюють за 3 секунди, але інформація про точність обчислень та кількість ітерацій не надається. [9]

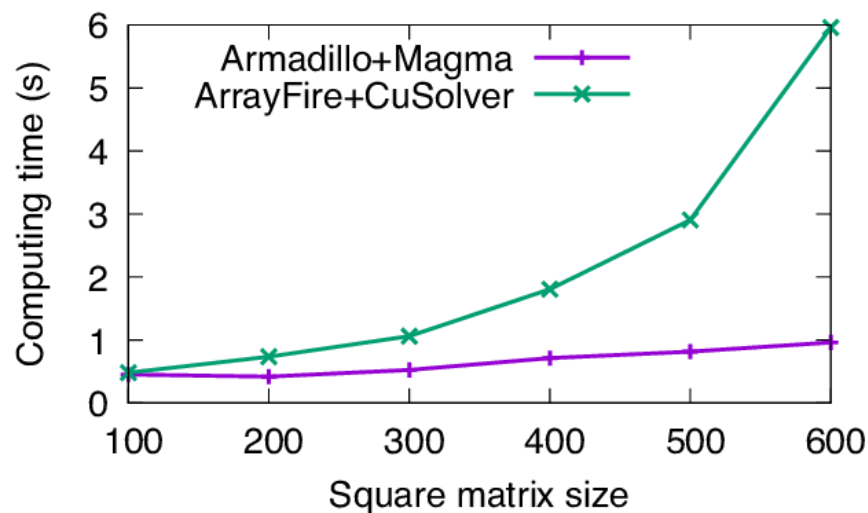


Рисунок 3.10 – Результат SVD бібліотеки CuSolver [10]

В іншій статті про SVD алгоритм з використанням архітектури CUDA, працює за схожим алгоритмом, але виконує фіксовану кількість ітерацій $12 \cdot N \cdot N$, де N – розмір матриці і має наступні результати (рисунок 3.11).

Для матриці 512 на 512 кількість ітерацій буде складати 3145728, а приблизний час 1 ітерації буде $1.224 \cdot 10^{-7}$. Для матриці 1000 на 1000 кількість ітерацій – $12 \cdot 10^6$, а приблизний час 1 ітерації $1.08 \cdot 10^{-7}$. Для матриці 4000 на 4000 приблизний час ітерації $1.042 \cdot 10^{-7}$, тобто ще менший. Як досягаються такі результати для відносно старої відеокарти – не очевидно та відтворити такі результати в цьому алгоритмі не вдалося.

Проблема в порівнянні з іншими існуючими рішеннями полягає в тому, що для них не надається ні точність обчислень, ні кількість ітерацій, а іноді й сам час обчислень.

SIZE	Diagonalization Intel MKL	Diagonalization GTX 280	Diagonalization 8800
128×128	0.010	0.017	0.041
512×512	0.5439	0.385	0.381
$1K \times 1K$	6.417	1.3	1.347
$2K \times 2K$	49.1	5.14	5.29
$3K \times 3K$	159.413	11.6	11.821
$4K \times 4K$	354.3	20	21.7
$8K \times 32$	0.022	0.007	—
$8K \times 256$	0.564	0.159	—
$8K \times 512$	2.239	0.530	—
$8K \times 2K$	100.000	8.2	—

Рисунок 3.11 Результати в секундах алгоритму бідіагоналізації із статті
Singular Value Decomposition on GPU using CUDA [11]

На відміну від цих результатів, в даній роботі представлений докладний опис алгоритму оцінки похибки (як найбільшого елемента в матриці, яка отримана як різниця вихідної матриці і твори знайдених співмножників (L , M' , R)). Також робота подає у повному обсязі інші деталі виконаних експериментів, що дозволить іншим дослідникам виконати повноцінне порівняння з нашими результатами.

В даному розділі було представлено результати тестування на швидкість та точність обчислень для різних розмірів матриці, розглянуто як впливає швидкість роботи кількість потоків та точність обчислень, а також проведене порівняння з існуючими версіями алгоритму.

Висновки по роботі та рекомендації для подальших досліджень

В даній роботі був розроблений та протестований паралельний SVD алгоритм для трьохдіагональної матриці на відеокарті з використанням архітектури NVIDIA CUDA. Для цього був проведений огляд архітектури паралельних обчислень на відеокарті, досліджена послідовна версія алгоритму, і на базі цього, розроблений і експериментально досліджений паралельний алгоритм на архітектурі NVIDIA CUDA.

Під час розробки алгоритму були вирішені наступні проблеми:

- оптимізація збереження матриць в пам'яті та алгоритмів множення
- використання спільної пам'яті відеокарти для підвищення швидкості роботи алгоритму
- синхронізація роботи блоків і спільних значень між блоками
- завершення роботи алгоритму

Розроблений алгоритм працює для будь-яких розмірів матриць, обмеження лише в апаратних можливостях графічного чіпу. Розроблений API підтримує такі мови програмування як C++ та Java, а також автоматично визначає оптимальну конфігурацію для алгоритму, базуючись на апаратних характеристиках відеокарти.

Алгоритм був протестований на різних розмірах матриць, з різними параметрами для оцінки таких показників, як:

- швидкість роботи та кількість ітерацій в залежності від розміру матриці
- швидкість роботи та кількість ітерацій в залежності від значень елементів матриці
- швидкість роботи та кількість ітерацій в залежності від заданої користувачем точності обчислень
- коефіцієнт прискорення для різної кількості потоків

З отриманих результатів, точність обчислень збільшує кількість ітерацій лінійно для точності 10^{-10} . Також API дозволяє задати власне значення точності.

До переваг розробленого алгоритму можна віднести:

- висока точність обчислень
- підтримку багатьох версій CUDA
- оптимізовану роботу з пам'яттю та алгоритмів множення
- API для C++ та Java, яке автоматично визначає конфігурацію для роботи та дозволяє їх змінювати
- невелике навантаження відеокарти
- лінійна залежність часу роботи обчислень від точності

Недоліком алгоритму можна розглядати велику кількість пустих ітерацій, які виникають внаслідок того, що деякі елементи матриці стають умовним нулем раніше чим інші. У такому випадку матриці можна розбити на декілька менших розміру і проводити обчислення на них незалежно. Для матриць невеликих розмірів, коли число потоків рівне половині розміру блоку – поліпшити ситуацію неможливо. Але можна запропонувати для великих матриць, в умовах нехватки потоків, в деяких моментах, коли з'явиться багато нульових елементів, перерозподілити їх. Це може бути предметом подальших досліджень.

Досягнення даної роботи:

1. розроблений та експериментально досліджений паралельний SVD алгоритм для трьохдіагональної матриці на відеокарті
2. вперше точно описані умови експерименту, спосіб оцінки похибки та результати, що дозволяють іншим дослідникам повторити та порівняти свої експерименти

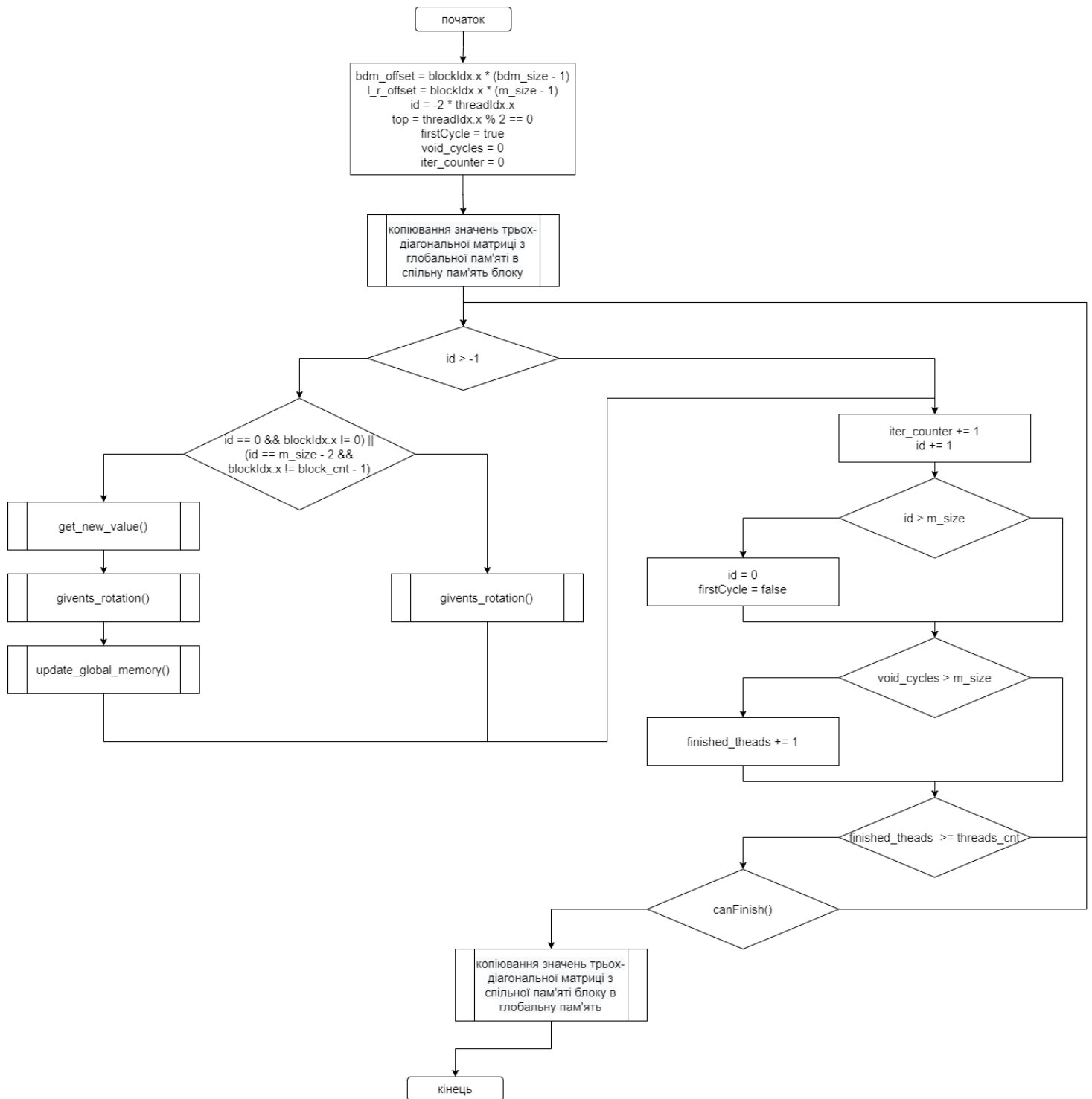
3. вперше експериментально знайдені границі для максимальної точності обчислень при заданій машинній арифметиці та розмірі матриці.

Завдяки архітектури NVIDIA CUDA, алгоритм працює набагато швидше ніж на процесорі, але є можливості для подальшого вдосконалення.

Список літератури

1. CUDA-Almanac : веб-сайт. URL: <https://www.nvidia.ru/docs/IO/141194/CUDA-Almanac-September.pdf>
2. Nvidia Developer CUDA zone : веб-сайт. URL: <https://developer.nvidia.com/cuda-zone>
3. Introduction to GPUs : веб-сайт. URL: <https://nyu-cds.github.io/python-gpu/02-cuda/>
4. Parallel Thread Execution ISA Version 7.3 : веб-сайт. URL: <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#operand-costs>
5. CUDA C++ Programming Guide : веб-сайт. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications>
6. https://3dnews.ru/assets/external/illustrations/2019/03/14/984208/pcb_front.jpg
7. NVIDIA CUDA C Programming Guide. Nvidia CUDA™, v.3.11. 2010
http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf
8. CUDA C++ Best Practices Guide : веб-сайт. URL: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>
9. Paolo Viviani, Marco Aldinucci, Massimo Torquati, Roberto d'Ippolito. Multiple back-end support for the Armadillo linear algebra interface. // Computer Science Department, University of Torino, Italy. 2016
10. <https://docs.nvidia.com/cuda/cusolver/index.html>
11. Sheetal Lahabar, P J Narayanan. Singular Value Decomposition on GPU using CUDA. // International Institute of Information Technology Hyderabad, INDIA. 2009

Додаток А. Схема паралельного SVD алгоритму на відеокарті



Додаток Б. Програмний код паралельного SVD алгоритму на відеокарті

```

unsigned long rid = pos * 3;

if (abs(s_bdm[rid + 1]) < eps)
{
    return;
}

double r = sqrt(sqr(s_bdm[rid]) + sqr(s_bdm[rid + 1]));
double c = 1, s = 0;
if (r > eps)
{
    c = s_bdm[rid] / r;
    s = s_bdm[rid + 1] / r;
}

double c11 = s_bdm[rid] * c + s_bdm[rid + 1] * s;
double c12 = s_bdm[rid + 2] * c + s_bdm[rid + 3] * s;
double c21 = s_bdm[rid + 1] * c - s_bdm[rid] * s;
double c22 = s_bdm[rid + 3] * c - s_bdm[rid + 2] * s;

s_bdm[rid] = c11;
s_bdm[rid + 1] = c21;
s_bdm[rid + 2] = c12;
s_bdm[rid + 3] = c22;

ulong id1 = l_size * (pos + offset);
ulong id2 = l_size * (pos + offset + 1);
for (int i = 0; i < l_size; ++i)
{
    double v1 = L[id1 + i];
    double v2 = L[id2 + i];

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

__device__ void Givents_Top(double* s_bdm, unsigned long pos, ulong offset, volatile
double* R, ulong r_size, double eps)
{
    unsigned long rid = pos * 3;
    if (abs(s_bdm[rid + 2]) < eps)
    {
        return;
    }
    double r = sqrt(sqr(s_bdm[rid]) + sqr(s_bdm[rid + 2]));
    double c = 1, s = 0;
    if (r > eps)
    {
        c = s_bdm[rid] / r;
        s = s_bdm[rid + 2] / r;
    }

    double c11 = s_bdm[rid] * c + s_bdm[rid + 2] * s;
    double c12 = s_bdm[rid + 2] * c - s_bdm[rid] * s;
    double c21 = s_bdm[rid + 1] * c + s_bdm[rid + 3] * s;
    double c22 = s_bdm[rid + 3] * c - s_bdm[rid + 1] * s;

    s_bdm[rid] = c11;

```



```

s_bdm[rid + 1] = c21;
s_bdm[rid + 2] = c12;
s_bdm[rid + 3] = c22;

ulong id1 = r_size * (pos + offset);
ulong id2 = r_size * (pos + offset + 1);
for (int i = 0; i < r_size; ++i)
{
    double v1 = R[id1 + i];
    double v2 = R[id2 + i];

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

__device__ bool canFinish(volatile int* finish_blocks)
{
    if (*finish_blocks == blockIdx.x)
        return true;
    return false;
}

__device__ void update_global_memory(volatile double* bdm, double value, ulong offset,
uint id, double sval, bool top, double eps)
{
    ulong pos = offset;
    ulong null_pos, spos;

    if (id != 0)
    {
        offset += id * 3;
        pos = offset + 3;
    }

    if (top)
    {
        null_pos = offset + 2;
        spos = offset + 1;
    }
    else
    {
        null_pos = offset + 1;
        spos = offset + 2;
    }

    while (abs(bdm[pos] - value) > eps)
    {
        bdm[pos] = value;
    }
    while (abs(bdm[null_pos]) > eps)
    {
        bdm[null_pos] = 0;
    }
    while (abs(bdm[spos] - sval) > eps)
    {
        bdm[spos] = sval;
    }
}

__device__ double get_new_value(volatile double* bdm, double old_value, ulong offset,
uint id, bool top, double eps)
{

```

```

    ulong pos = offset;
    if (id != 0)
        pos += (id + 1) * 3;

    ulong check_pos;
    if (top)
    {
        if (id == 0)
        {
            check_pos = pos - 1;
        }
        else
        {
            check_pos = pos + 1;
        }
    }
    else
    {
        if (id == 0)
        {
            check_pos = pos - 2;
        }
        else
        {
            check_pos = pos + 2;
        }
    }

    while(true)
    {
        double new_value = bdm[check_pos];
        if (abs(new_value) < eps)
            return bdm[pos];
    }
    return 0;
}

__global__ void svd_for_bidiagonal(double* bdm, ulong m_size, ulong bdm_size, uint
thread_cnt, double* L, double* R, ulong full_size, ulong block_cnt, int* finished_blocks,
ulong* counters, double eps)
{
    extern __shared__ double s_bdm[];
    ulong bdm_offset = blockIdx.x * (bdm_size - 1);
    ulong l_r_offset = blockIdx.x * (m_size - 1);
    ulong iter_counter = counters[blockDim.x * blockIdx.x + threadIdx.x];
    for (unsigned long i = 0; i < (bdm_size / thread_cnt) + 1; ++i)
    {
        unsigned long id = i * thread_cnt + threadIdx.x;
        if (id < bdm_size)
            s_bdm[id] = bdm[bdm_offset + id];
    }

    if (threadIdx.x == 0)
        s_bdm[bdm_size] = 0;

    __syncthreads();

    long id = -2 * threadIdx.x;
    bool top = threadIdx.x % 2 == 0;
    bool firstCycle = true;
    ulong void_cycles = 0;

    while (true)
    {

```

```

++iter_counter;
if (id > -1)
{
    if ((id == 0 && blockIdx.x != 0) || (id == m_size - 2 && blockIdx.x
!= block_cnt - 1))
    {
        ulong update_pos = id * 3;
        ulong sec_pos = id * 3 + 1;
        if (id != 0)
        {
            update_pos += 3;
        }

        double v;
        if (threadIdx.x != 0 || firstCycle == false || id == 0)
        {
            v = get_new_value(bdm, s_bdm[update_pos], bdm_offset,
id, top, eps);
            s_bdm[update_pos] = v;
        }

        if (top)
        {
            if (abs(s_bdm[id * 3 + 2]) < eps)
                ++void_cycles;
            else
            {
                svd_utils::Givents_Top(s_bdm, id, l_r_offset, R,
full_size, eps);

                void_cycles = 0;
                atomicExch((float*)&(s_bdm[bdm_size]), 0.);
            }
        }
        else
        {
            if (abs(s_bdm[id * 3 + 1]) < eps)
                ++void_cycles;
            else
            {
                svd_utils::Givents_Bot(s_bdm, id, l_r_offset, L,
full_size, eps);

                void_cycles = 0;
                atomicExch((float*)&(s_bdm[bdm_size]), 0.);
            }
            sec_pos += 1;
        }
        double sval = s_bdm[sec_pos];
        update_global_memory(bdm, s_bdm[update_pos], bdm_offset, id,
sval, top, eps);
    }
    else
    {
        if (top)
        {
            if (abs(s_bdm[id * 3 + 2]) < eps)
                ++void_cycles;
            else
            {
                svd_utils::Givents_Top(s_bdm, id, l_r_offset, R,
full_size, eps);

                void_cycles = 0;
                atomicExch((float*)&(s_bdm[bdm_size]), 0.);
            }
        }
    }
}

```

```

        else
        {
            if (abs(s_bdm[id * 3 + 1]) < eps)
                ++void_cycles;
            else
            {
                svd_utils::Givents_Bot(s_bdm, id, l_r_offset, L,
full_size, eps);

                void_cycles = 0;
                atomicExch((float*)&(s_bdm[bdm_size]), 0.);
            }
        }
    }

    if (++id == m_size - 1)
    {
        if (thread_cnt == 1)
            top = !top;
        id = 0;
        firstCycle = false;
    }

    if (void_cycles > m_size + 1)
    {
        atomicAdd(&(s_bdm[bdm_size]), 1.);
    }

    __syncthreads();

    ulong fcount = (ulong)atomicAdd(&(s_bdm[bdm_size]), 0.);
    if (fcount >= thread_cnt)
    {
        if (svd_utils::canFinish(finished_blocks))
        {
            if (threadIdx.x == 0)
                atomicAdd(finished_blocks, 1);
            break;
        }
    }
    atomicExch((float*)&(s_bdm[bdm_size]), 0.);
}

for (unsigned long i = 0; i < (bdm_size / thread_cnt) + 1; ++i)
{
    unsigned long id = i * thread_cnt + threadIdx.x;
    if (id < bdm_size)
        bdm[bdm_offset + id] = s_bdm[id];
}

counters[blockDim.x * blockIdx.x + threadIdx.x] = iter_counter;

__syncthreads();
}

```

Додаток В. Програмний код API для розробленого алгоритму на C++

```

cuda_svd::cuda_svd(double ** matrix, ulong rows, ulong columns, double eps)
{
    // save pointer to initial matrix
    this->matrix = matrix;
    // select size matrix from min for rows and columns
    this->size = rows < columns ? rows : columns;
    // create bidiagonal matrix
    this->bi_matrix = matrix_to_bidiagonal(this->matrix, this->size);
    this->bidiagonal_matrix_origin_size = (this->size - 1) * 3 + 1;
    this->eps = eps;
    calc_params();
}

cuda_svd::~cuda_svd()
{
    if (matrix)
    {
        for (ulong i = 0; i < this->size - 1; ++i)
            delete[] matrix[i];
        delete[] matrix;

        matrix = 0;
    }

    if (bi_matrix)
    {
        delete[] bi_matrix;
        bi_matrix = 0;
    }
    free_memory();
    free_cuda_memory();
}

void cuda_svd::free_cuda_memory()
{
    if (gpu_l)
    {
        cudaFree(gpu_l);
        gpu_l = 0;
    }
    if (gpu_r)
    {
        cudaFree(gpu_r);
        gpu_r = 0;
    }
    if (gpu_bgm)
    {
        cudaFree(gpu_bgm);
        gpu_bgm = 0;
    }
    if (gpu_finished_blocks)
    {
        cudaFree(gpu_finished_blocks);
        gpu_finished_blocks = 0;
    }
}

void cuda_svd::free_memory()
{
    if (L)

```

```

{
    delete[] L;
    L = 0;
}
if (R)
{
    delete[] R;
    R = 0;
}
if (finished_blocks)
{
    delete finished_blocks;
    finished_blocks = 0;
}
if (full_bgm)
{
    delete[] full_bgm;
    full_bgm = 0;
}
if (vec_matrix)
{
    delete[] vec_matrix;
    vec_matrix = 0;
}
}

void cuda_svd::calc_params()
{
    int cudaDeviceNum = 0;
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, cudaDeviceNum);

    sysCUAOptions.global_memory_size = prop.totalGlobalMem;
    sysCUAOptions.max_shared_memory_size = prop.sharedMemPerBlock;
    sysCUAOptions.max_threads_per_block = 625; // prop.maxThreadsPerBlock;
    sysCUAOptions.max_blocks = prop.maxBlocksPerMultiProcessor;

    ulong bi_matrix_size_in_bytes = this->bidiagonal_matrix_origin_size *
sizeof(double);

    if (bi_matrix_size_in_bytes <= sysCUAOptions.max_shared_memory_size)
    {
        this->one_block = true;
        svd_params.blocks_number = 1;
    }
    else
    {
        this->one_block = false;

        svd_params.threads_per_block = sysCUAOptions.max_threads_per_block;
        svd_params.blocks_number = bi_matrix_size_in_bytes /
sysCUAOptions.max_threads_per_block;
        if (svd_params.blocks_number > sysCUAOptions.max_blocks)
            svd_params.blocks_number = sysCUAOptions.max_blocks;
    }

    svd_params.block_size = this->size / svd_params.blocks_number + 2;
    svd_params.bidiagonal_matrix_size = this->bidiagonal_matrix_origin_size;
    svd_params.input_matrix_size = this->size;
    svd_params.block_bi_matrix_size = (svd_params.block_size - 1) * 3 + 1;
    svd_params.all_blocks_size = svd_params.block_bi_matrix_size *
svd_params.blocks_number;
}

```

```

        svd_params.new_matrix_size = (svd_params.all_blocks_size - 1) / 3 + 1;
        svd_params.threads_per_block = (svd_params.block_size - 1) / 2 <
sysCUDAOptions.max_threads_per_block ? (svd_params.block_size - 1) / 2 :
sysCUDAOptions.max_threads_per_block;

        if (svd_params.threads_per_block % 2 == 1 && svd_params.threads_per_block != 1)
            svd_params.threads_per_block -= 1;

        m_size = this->size;
        bd_block_size = svd_params.block_bi_matrix_size;
        all_blocks_size = svd_params.all_blocks_size;
        bd_size = svd_params.bidiagonal_matrix_size;
        new_matrix_size = svd_params.new_matrix_size;
        block_size = svd_params.block_size;
        blocks_number = svd_params.blocks_number;
        threads_in_block = svd_params.threads_per_block;
    }

void cuda_svd::allocate_memory()
{
    full_bgm = new double[all_blocks_size];
    vec_matrix = matrix_to_vec(this->matrix, this->size);
    memcpy(full_bgm, this->bi_matrix, bd_size * sizeof(double));

    memset(full_bgm + bd_size, 0, (all_blocks_size - bd_size) * sizeof(double));

    cudaError_t result;
    result = cudaMalloc(&gpu_bgm, all_blocks_size * sizeof(double));

    finished_blocks = new int;
    *finished_blocks = 0;

    result = cudaMalloc(&gpu_finished_blocks, sizeof(int));

    L = one_matrix(size);
    R = one_matrix(size);

    result = cudaMalloc(&gpu_l, size * size * sizeof(double));
    result = cudaMalloc(&gpu_r, size * size * sizeof(double));

    counter = new ulong[blocks_number * threads_in_block];
    memset(counter, 0, blocks_number * threads_in_block * sizeof(ulong));

    cudaMalloc(&gpu_counter, blocks_number * threads_in_block * sizeof(ulong));
}

void cuda_svd::copy_to_device()
{
    cudaError_t result;
    result = cudaMemcpy(gpu_bgm, full_bgm, all_blocks_size * sizeof(double),
cudaMemcpyHostToDevice);
    result = cudaMemcpy(gpu_finished_blocks, finished_blocks, sizeof(int),
cudaMemcpyHostToDevice);
    result = cudaMemcpy(gpu_l, L, size * size * sizeof(double),
cudaMemcpyHostToDevice);
    result = cudaMemcpy(gpu_r, R, size * size * sizeof(double),
cudaMemcpyHostToDevice);
    result = cudaMemcpy(gpu_counter, counter, blocks_number * threads_in_block *
sizeof(ulong), cudaMemcpyHostToDevice);
}

void cuda_svd::copy_from_device()
{
    cudaError_t result;

```

```

        result = cudaMemcpy(full_bgm, gpu_bgm, all_blocks_size * sizeof(double),
        cudaMemcpyDeviceToHost);
        result = cudaMemcpy(L, gpu_l, size * size * sizeof(double),
        cudaMemcpyDeviceToHost);
        result = cudaMemcpy(R, gpu_r, size * size * sizeof(double),
        cudaMemcpyDeviceToHost);
        result = cudaMemcpy(counter, gpu_counter, blocks_number * threads_in_block *
        sizeof(ulong), cudaMemcpyDeviceToHost);
    }

void cuda_svd::update_cycles()
{
    ulong sum = 0;
    for (ulong i = 0; i < blocks_number * threads_in_block; ++i)
        sum += counter[i];
    printf("Cycles %zd\n", sum);
}

void cuda_svd::mult_and_compare()
{
#ifdef DEBUG_TEST
    printf("Input matrix\n\n");
    print_matrix(this->matrix, m_size);
    for (uint j = 0; j < blocks_number; ++j)
    {
        printf("Block %d\n", j);
        for (uint i = 0; i < bd_block_size; ++i)
            printf("%5d ", j * (bd_block_size - 1) + i + 1);
        printf("\n");
        printf("\nfull bgm\n");
        for (uint i = 0; i < bd_block_size; ++i)
            printf("%3.2f ", full_bgm[j * (bd_block_size - 1) + i]);
        printf("\n");
    }
#endif // DEBUG_TEST

    double** new_matr = new double*[this->size];
    for (ulong i = 0; i < this->size; ++i)
    {
        new_matr[i] = new double[this->size];
        memset(new_matr[i], 0, this->size * sizeof(double));
    }

    from_bd_matrix(new_matr, full_bgm, this->size);
    double* new_matr_vec = matrix_to_vec(new_matr, this->size);
    double* Lt = transpose(L, size);
    double* lm = mult_api(Lt, new_matr_vec, this->size);
    double* fin = mult_api(lm, R, this->size);

    ulong max_dif_id = 0;
    double max_dif = fabs(fin[0] - vec_matrix[0]);
    for (ulong i = 1; i < sqr(this->size); ++i)
    {
        double dif = fabs(fin[i] - vec_matrix[i]);
        if (dif > max_dif)
        {
            max_dif = dif;
            max_dif_id = i;
        }
    }

#ifdef DEBUG_TEST
    printf("\nL\n");
    print_m(Lt, this->size);

```



```

    auto stop = std::chrono::high_resolution_clock::now();
    printf("Cuda stops with res %d\n", result);

    copy_from_device();
    update_cycles();

#ifdef TEST
    mult_and_compare();
#endif // TEST

    svd_time = std::chrono::duration_cast<std::chrono::milliseconds>(stop -
start).count();
    free_memory();
    free_cuda_memory();
    printf("for eps %e\ttime %ld\n", e, svd_time);

    return svd_time;
}

```