МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«ВИКОРИСТАННЯ OPENCL ДЛЯ РОЗВ'ЯЗАННЯ СИСТЕМ ЛІНІЙНИХ АЛГЕБРАЇЧНИХ РІВНЯНЬ»**

Виконав: студент 4 курсу, групи 6.1212-пі-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
Р.І. Моісеєнко
(ініціали та прізвище)
Керівник декан математичного факультету, професор, д.т.н. Гоменюк С.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Моісеєнку Роману Івановичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Використання OpenCL для розв’язання систем лінійних алгебраїчних рівнянь

керівник роботи Гоменюк Сергій Іванович, д.т.н., професор
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Теоретичні основи та вибір технологій для розв’язання систем лінійних алгебраїчних рівнянь.
2. Проєктування та реалізація застосунку для розв’язання СЛАР засобами OpenCL.
3. Аналіз продуктивності, тестування та оптимізація паралельних обчислень.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 09.12.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	19.12.2025	
2.	Збір вихідних даних.	26.01.2026	
3.	Обробка методичних та теоретичних	27.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	15.04.2026	
5.	Розробка третього розділу.	20.05.2026	
6.	Оформлення та нормоконтроль	25.05.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	15.06.2026	

Студент _____ Р.І. Моісеєнко
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.І. Гоменюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Використання OpenCL для розв’язання систем лінійних алгебраїчних рівнянь»: 52 с., 25 рис., 14 табл., 14 джерел, 1 додаток.

АЛГОРИТМ ГАУСА-ЖОРДАНА, ВІДЕОПРОЦЕСОР, ВІДНОСНА НЕВ’ЯЗКА, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ПРОДУКТИВНІСТЬ, СИСТЕМИ ЛІНІЙНИХ АЛГЕБРАЇЧНИХ РІВНЯНЬ, FLUTTER, OPENCL, WORK-ITEM.

Об’єкт дослідження – процес паралельного розв’язання систем лінійних алгебраїчних рівнянь методом Гауса-Жордана.

Мета роботи: дослідити ефективність застосування технології OpenCL для паралельного розв’язання систем лінійних алгебраїчних рівнянь та порівняти продуктивність CPU- і GPU-реалізацій.

Методи дослідження – метод Гауса-Жордана з частковою вибіркою головного елемента, експериментальне порівняльне тестування продуктивності.

У роботі проведено аналіз технологій паралельних обчислень та обґрунтовано вибір OpenCL. Розроблено трирівневий застосунок з графічним інтерфейсом на Flutter, обчислювальним ядром на C++17 та чотирма OpenCL-ядрами для GPU. Реалізовано послідовний CPU-алгоритм та паралельний GPU-алгоритм розв’язання СЛАР. Проведено експериментальне дослідження продуктивності на GPU NVIDIA GeForce GTX 1650 Ti для восьми розмірностей матриць від 64×64 до 1024×1024. Встановлено точку беззбитковості при $N \approx 320$, після якої GPU перевершує CPU. При розмірності 1024×1024 GPU-реалізація є у 7,19 рази швидшою за CPU. Верифіковано коректність результатів через відносну нев’язку $r < 1 \times 10^{-10}$ для всіх розмірностей.

SUMMARY

Bachelor's qualifying paper "Application of OpenCL for Solving Systems of Linear Algebraic Equations": 52 pages, 25 figures, 14 tables, 14 references, 1 supplement.

FLUTTER, GAUSS-JORDAN ALGORITHM, GPU, LINEAR ALGEBRAIC EQUATIONS SYSTEMS, OPENCL, PARALLEL COMPUTING, PERFORMANCE, RELATIVE RESIDUAL, WORK-ITEM.

The object of the study is the process of parallel solving of systems of linear algebraic equations using the Gauss-Jordan method.

The aim of the study is to investigate the efficiency of applying OpenCL technology for parallel solving of systems of linear algebraic equations and to compare the performance of CPU and GPU implementations.

The methods of research include the Gauss-Jordan method with partial pivoting and experimental comparative performance testing.

The paper analyzes parallel computing technologies and justifies the choice of OpenCL. A three-tier application has been developed with a Flutter-based graphical user interface, a C++17 computational core, and four OpenCL kernels for GPU execution. Both sequential CPU and parallel GPU algorithms for solving systems of linear algebraic equations have been implemented. Experimental performance research was conducted on an NVIDIA GeForce GTX 1650 Ti GPU for eight matrix sizes ranging from 64×64 to 1024×1024. The break-even point was established at $N \approx 320$, beyond which the GPU outperforms the CPU. At matrix size 1024×1024, the GPU implementation is 7.19 times faster than the CPU. The correctness of results was verified through the relative residual $r < 1 \times 10^{-10}$ for all matrix sizes.

ЗМІСТ

Завдання на кваліфікаційну роботу……..…..2
Реферат 4
Summary 5
Вступ..... 7
1 Теоретичні основи та вибір технологій для роз’язання систем лінійних алгебраїчних рівнянь..... 9
1.1 Системи лінійних алгебраїчних рівнянь та їх застосування 9
1.2 Огляд існуючих технологій паралельних обчислень 11
1.3 Постановка задачі та вимоги до системи 14
2 Проєктування..... 17
2.1 Математичні основи методу Гауса-Жордана та числова стійкість ... 17
2.2 Модель виконання та пам’яті OpenCL..... 20
2.3 Архітектура застосунку..... 23
2.4 Проєктування GPU-ядер та схема передачі даних 26
2.5 Проєктування FFI-інтерфейсу Dart ↔ C++..... 30
3 Реалізація та тестування..... 33
3.1 CPU-реалізація методу Гауса-Жордана..... 33
3.2 GPU-реалізація на OpenCL..... 34
3.3 Реалізація інтерфейсу користувача 37
3.4 Методика тестування та верифікація точності 41
3.5 Результати бенчмарку та аналіз продуктивності..... 44
Висновки 48
Перелік посилань..... 50
Додаток А Розміщення програмної реалізації застосунку..... 52

ВСТУП

Системи лінійних алгебраїчних рівнянь (СЛАР) є фундаментальним математичним інструментом, що широко застосовується в інженерних розрахунках, комп'ютерній графіці, машинному навчанні [4], моделюванні фізичних процесів та багатьох інших прикладних областях. Зі збільшенням розмірності задач традиційні послідовні алгоритми розв'язання СЛАР стикаються з суттєвими обмеженнями продуктивності: обчислювальна складність методу Гауса-Жордана становить $O(n^3)$ [5], що робить послідовну обробку великих матриць неприйнятно повільною.

Сучасні графічні процесори (GPU) містять тисячі обчислювальних ядер і здатні виконувати масово-паралельні операції, що відкриває можливості для суттєвого прискорення матричних обчислень. Технологія OpenCL (Open Computing Language) є відкритим крос-платформним стандартом [1] паралельних обчислень, що підтримується провідними виробниками GPU-NVIDIA, AMD та Intel, – на відміну від пропрієтарної технології CUDA [6], яка обмежена лише обладнанням NVIDIA.

Актуальність теми визначається зростаючою потребою у високопродуктивних обчисленнях при розв'язанні великих СЛАР, а також доцільністю використання відкритих кросплатформних технологій паралельних обчислень у наукових та інженерних застосунках.

Метою роботи є дослідження ефективності застосування технології OpenCL для паралельного розв'язання систем лінійних алгебраїчних рівнянь методом Гауса-Жордана та порівняння продуктивності CPU- і GPU-реалізацій.

Для досягнення мети поставлені такі завдання:

– дослідити теоретичні основи методу Гауса-Жордана та його числові властивості;
– проаналізувати існуючі технології паралельних обчислень та обґрунтувати вибір OpenCL;

– розробити послідовну CPU-реалізацію алгоритму мовою C++;
– розробити паралельну GPU-реалізацію з використанням чотирьох OpenCL-ядер;
– реалізувати кросплатформний інтерфейс користувача на основі Flutter;
– провести порівняльний аналіз продуктивності CPU- та GPU-реалізацій для матриць різних розмірностей.

Об’єктом дослідження є процес паралельного розв’язання систем лінійних алгебраїчних рівнянь.

Предметом дослідження є методи та засоби реалізації паралельного алгоритму Гауса-Жордана з використанням технології OpenCL.

Практичним результатом роботи є настільний застосунок для Windows, що реалізує два режими розв’язання СЛАР – послідовний (CPU) та паралельний (GPU через OpenCL) – з можливістю порівняння їх продуктивності у реальному часі. Експериментальні дослідження на GPU NVIDIA GeForce GTX 1650 Ti показали, що при розмірності матриці 1024×1024 GPU-реалізація перевищує продуктивність CPU-реалізації у 7,19 рази.

Робота складається зі вступу, трьох розділів, висновків, переліку посилань та одного додатку.

1 ТЕОРЕТИЧНІ ОСНОВИ ТА ВИБІР ТЕХНОЛОГІЙ ДЛЯ РОЗ'ЯЗАННЯ СИСТЕМ ЛІНІЙНИХ АЛГЕБРАЇЧНИХ РІВНЯНЬ

1.1 Системи лінійних алгебраїчних рівнянь та їх застосування

Система лінійних алгебраїчних рівнянь (СЛАР) – це сукупність n рівнянь з n невідомими, що записується у матричній формі:

$$A \cdot x = b,$$

де A – матриця коефіцієнтів розмірності n×n, x – вектор невідомих розмірності n×1, b – вектор правих частин розмірності n×1.

У розгорнутій формі система має вигляд:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2, \\ &\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n. \end{aligned}$$

СЛАР виникають у широкому колі наукових та інженерних задач. Розглянемо основні галузі їх застосування.

Інженерне моделювання. Метод скінченних елементів (МСЕ) [4] є одним із найпоширеніших чисельних методів у сучасній інженерії. При дискретизації неперервного середовища на скінченні елементи задачі механіки конструкцій, теплопровідності та гідродинаміки зводяться до розв'язання великих розріджених СЛАР, розмірність яких може сягати мільйонів рівнянь.

Комп'ютерна графіка та машинне навчання. Трансформації координат, розрахунок освітлення у тривимірних сценах, а також навчання лінійних моделей методом найменших квадратів – усі ці задачі в своїй основі зводяться

до розв’язання СЛАР. Зокрема, метод найменших квадратів для лінійної регресії вимагає розв’язання нормальних рівнянь [5] $A^TAx = A^Tb$.

Обчислювальна хімія та фізика. Моделювання молекулярної динаміки, квантових систем та розв’язання рівнянь Шредінгера для багатоелектронних атомів також потребує ефективних алгоритмів розв’язання СЛАР великої розмірності.

Основною характеристикою, що визначає складність розв’язання, є розмірність n. Більшість прямих методів мають обчислювальну складність $O(n^3)$ [9]. Це означає, що подвоєння розмірності задачі призводить до восьмиразового збільшення часу обчислень. У таблиці 1.1 наведено залежність кількості арифметичних операцій від розмірності матриці для методу Гауса-Жордана.

Таблиця 1.1 – Залежність кількості операцій від розмірності СЛАР

Розмірність матриці N	**Кількість операцій (~$2n^3$)**	**Час виконання CPU (мс)**
64×64	~524 288	0,4
128×128	~4 194 304	3,7
256×256	~33 554 432	32,1
384×384	~113 246 208	148,1
512×512	~268 435 456	352,7
640×640	~524 288 000	689,7
768×768	~905 969 664	1109,1
1024×1024	~2 147 483 648	2935,8

Важливою числовою характеристикою СЛАР є обумовленість матриці A. Число обумовленості визначається як:

$$\text{cond}(A) = \|A\| \cdot \|A^{-1}\|,$$

де $\|\cdot\|$ – матрична норма. При $\text{cond}(A) >> 1$ матриця вважається погано обумовленою: невеликі похибки у вхідних даних можуть призвести до значних похибок у розв’язку.

Для забезпечення числової стійкості експериментів у даній роботі використовуються діагонально-домінантні матриці, для яких виконується умова:

$$|A[i][i]| > \Sigma |A[i][j]|, j \neq i.$$

Ця умова гарантує єдиність розв’язку та добру обумовленість системи. При генерації тестових матриць діагональний елемент формується за формулою:

$$A[i][i] = \Sigma|A[i][j]| + 1, j \neq i,$$

що забезпечує виконання умови діагональної домінантності для всіх розмірностей від 64×64 до 1024×1024.

1.2 Огляд існуючих технологій паралельних обчислень

Сучасні обчислювальні задачі, що потребують високої продуктивності, вирішуються за допомогою технологій паралельних обчислень. Розглянемо основні з них та обґрунтуємо вибір OpenCL для реалізації даної роботи.

OpenMP (Open Multi-Processing) – стандарт паралельного програмування [6] для багатоядерних процесорів із використанням спільної пам’яті. Програміст додає спеціальні директиви-прагми до послідовного коду, і компілятор автоматично розподіляє обчислення між ядрами CPU. Перевагами OpenMP є простота використання та відсутність необхідності переписувати існуючий код. Проте OpenMP обмежений кількістю ядер CPU (зазвичай 4-32), що суттєво поступається тисячам ядер сучасного GPU.

MPI (Message Passing Interface) – стандарт для паралельних обчислень на кластерах із розподіленою пам’яттю. MPI дозволяє розподілити обчислення між багатьма вузлами мережі, що робить його придатним для

суперкомп’ютерних задач. Однак складність програмування та необхідність спеціальної інфраструктури роблять MPI надмірним рішенням для задач, що вирішуються на одному комп’ютері.

Існуючі бібліотеки паралельної лінійної алгебри, такі як ScaLAPACK [8], також реалізують паралельні алгоритми розв’язання СЛАР на основі MPI, проте орієнтовані на кластерні системи та не використовують можливості сучасних GPU.

CUDA (Compute Unified Device Architecture) – пропрієтарна технологія компанії NVIDIA [6] для програмування GPU. CUDA забезпечує високу продуктивність та зручний програмний інтерфейс, проте має критичний недолік – вона працює виключно на GPU від NVIDIA. Це унеможливлює використання застосунків на комп’ютерах з GPU від AMD або Intel, що суттєво обмежує сферу застосування.

OpenCL (Open Computing Language) – відкритий крос-платформний стандарт [1] паралельних обчислень, розроблений та підтримуваний консорціумом Khronos Group [13]. На відміну від CUDA, OpenCL підтримується GPU від усіх провідних виробників – NVIDIA, AMD та Intel, – а також може виконуватися на CPU та інших обчислювальних пристроях. OpenCL надає уніфікований програмний інтерфейс для керування пристроями, компіляції ядер та управління пам’яттю.

У таблиці 1.2 наведено порівняльний аналіз розглянутих технологій за ключовими характеристиками.

Таблиця 1.2 – Порівняльний аналіз технологій паралельних обчислень

Характеристика	**OpenMP**	**MPI**	**CUDA**	**OpenCL**
Тип паралелізму	CPU (багатоядерний)	Розподілений	GPU (NVIDIA)	GPU/CPU (будь-який)
Підтримувані пристрої	CPU	CPU-кластери	Лише NVIDIA GPU	NVIDIA, AMD, Intel
Відкритий стандарт	✓	✓	✗	✓
Складність програмування	Низька	Висока	Середня	Середня

Продовження таблиці 1.2

Характеристика	OpenMP	MPI	CUDA	OpenCL
Максимальний паралелізм	~десятки ядер	~тисячі вузлів	~тисячі ядер	~тисячі ядер
Підтримка double precision	✓	✓	✓	✓
Крос-платформність	✓	✓	✗	✓

Аналіз таблиці 1.2 показує, що для задачі паралельного розв’язання СЛАР на GPU з вимогою крос-платформності оптимальним вибором є OpenCL. Ця технологія поєднує масовий паралелізм GPU з відкритістю стандарту та незалежністю від виробника обладнання.

Модель виконання OpenCL базується на концепції work-item – найменшої одиниці паралельного виконання [2]. Work-items організовуються у work-groups [3], які виконуються на обчислювальних одиницях GPU. Сукупність усіх work-items утворює NDRange – багатовимірний простір індексів.

Ієрархію паралельного виконання OpenCL наведено на рисунку 1.1.

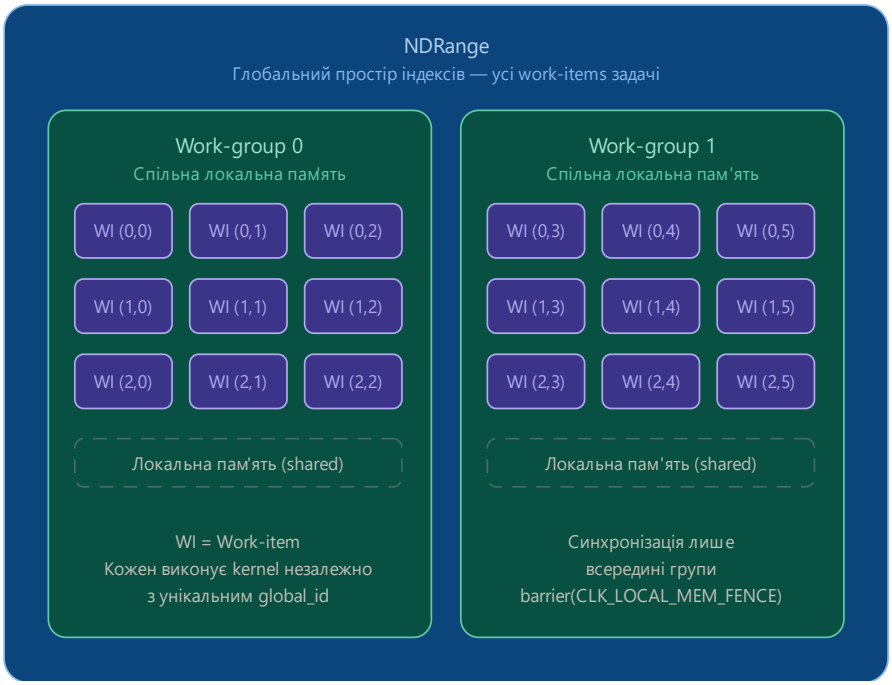


Рисунок 1.1 – Ієрархія паралельного виконання в OpenCL:
NDRange → Work-group → Work-item

Важливою особливістю OpenCL є ієрархічна модель пам’яті. Глобальна пам’ять (VRAM) доступна всім work-items [3], але має відносно велику латентність. Локальна пам’ять спільна для всіх work-items у межах однієї work-group і значно швидша за глобальну. Приватна пам’ять доступна лише одному work-item і є найшвидшою. У даній роботі матриця A та вектор b зберігаються у глобальній пам’яті GPU, що дозволяє всім паралельним ядрам мати до них доступ.

1.3 Постановка задачі та вимоги до системи

На основі проведеного аналізу предметної області та існуючих технологій сформулюємо задачу даної кваліфікаційної роботи.

Мета роботи полягає у розробці програмного застосунку, що реалізує паралельний алгоритм розв’язання систем лінійних алгебраїчних рівнянь з використанням технології OpenCL, та проведенні порівняльного аналізу продуктивності GPU- і CPU-реалізацій.

Об’єкт дослідження – процес паралельного розв’язання систем лінійних алгебраїчних рівнянь методом Гауса-Жордана.

Предмет дослідження – методи та засоби реалізації паралельного алгоритму з використанням технології OpenCL на графічному процесорі.

Для досягнення поставленої мети необхідно вирішити такі **задачі**:

- дослідити математичні основи методу Гауса-Жордана та обґрунтувати його придатність для паралельної реалізації;
- розробити послідовну CPU-реалізацію алгоритму мовою C++ як базову для порівняння;
- розробити паралельну GPU-реалізацію з використанням чотирьох OpenCL-ядер;
- реалізувати механізм передачі даних між CPU та GPU з мінімізацією накладних витрат шини PCIe;

– розробити кросплатформний графічний інтерфейс користувача на основі фреймворку Flutter;
– провести експериментальне дослідження продуктивності обох реалізацій для матриць розмірністю від 64×64 до 1024×1024;
– верифікувати коректність результатів шляхом обчислення відносної нев’язки.

Вимоги до програмного застосунку визначаються наступним чином.

Функціональні вимоги:

– застосунок повинен приймати на вхід матрицю коефіцієнтів A та вектор правих частин b довільної розмірності від 2 до 1024;
– застосунок повинен розв’язувати СЛАР двома методами – послідовним (CPU) та паралельним (GPU через OpenCL) – і повертати вектор розв’язку x;
– застосунок повинен вимірювати та відображати час виконання кожного методу у мілісекундах;
– застосунок повинен обчислювати та відображати відносну нев’язку розв’язку за формулою $r = ||Ax − b|| / ||b||$;
– застосунок повинен надавати можливість автоматичного порівняльного тестування (бенчмарку) для восьми фіксованих розмірностей матриць;
– застосунок повинен відображати результати бенчмарку у вигляді графіку та таблиці прискорення.

Нефункціональні вимоги:

– цільова платформа – Windows x64;
– мова програмування обчислювального ядра – C++17;
– технологія паралельних обчислень – OpenCL версії 1.2;
– мова програмування інтерфейсу – Dart (фреймворк Flutter);
– точність обчислень – тип double (64-біт), відносна нев’язка не більше $1×10^{-10}$;
– підтримка GPU від усіх виробників, сумісних з OpenCL 1.2.

У таблиці 1.3 наведено зведені вимоги до розроблюваного застосунку.

Таблиця 1.3 – Зведені вимоги до програмного за стосунку

Вимога	**Значення**
Цільова платформа	Windows x64
Мова обчислювального ядра	C++17
Технологія паралелізму	OpenCL 1.2
Мова інтерфейсу	Dart / Flutter
Максимальна розмірність СЛАР	1024×1024
Точність обчислень	double (64-біт)
Максимальна відносна нев’язка	$\leq 1\times10^{-10}$
Підтримувані GPU	NVIDIA, AMD, Intel
Формат вихідних даних	Вектор X, час (мс), нев’язка

Таким чином, у першому розділі було проведено аналіз предметної області, розглянуто основні технології паралельних обчислень та обґрунтовано вибір OpenCL як оптимального інструменту для реалізації паралельного алгоритму розв’язання СЛАР. Сформульовано мету, задачі та вимоги до розроблюваного застосунку, що є основою для проєктування системи у наступному розділі.

2 ПРОЄКТУВАННЯ

2.1 Математичні основи методу Гауса-Жордана та числова стійкість

Метод Гауса-Жордана є прямим методом розв’язання СЛАР [5], що перетворює розширену матрицю системи [A|b] на одиничну матрицю з вектором розв’язку [I|x].

На відміну від класичного методу Гауса, який зводить матрицю лише до верхньотрикутної форми з подальшим зворотним ходом, метод Гауса-Жордана виконує повну редукцію за один прохід, що робить його більш придатним для паралельної реалізації.

Алгоритм складається з n ітерацій, де n – розмірність системи. На кожній ітерації k = 0, 1, ..., n−1 виконуються чотири послідовні кроки.

Крок 1. Пошук головного елемента (pivot). Серед елементів стовпця k для рядків i = k, k+1, ..., n−1 знаходиться рядок pivotRow з максимальним за модулем значенням:

$$\text{pivotRow} = \arg\max |A[i][k]|,\ i = k..n-1.$$

Цей крок є обов’язковим для забезпечення числової стійкості алгоритму. Без вибірки головного елемента при A[k][k] ≈ 0 виникає ділення на мале число, що призводить до катастрофічного накопичення похибок округлення.

Крок 2. Перестановка рядків. Якщо pivotRow ≠ k, виконується обмін рядків k і pivotRow у матриці A та векторі b:

$$A[k] \leftrightarrow A[\text{pivotRow}],\quad b[k] \leftrightarrow b[\text{pivotRow}].$$

Крок 3. Нормалізація рядка k. Усі елементи рядка k діляться на діагональний елемент A[k][k] = pivotVal:

$$A[k][j] = A[k][j] / pivotVal, \quad j = 0..n-1,$$
$$b[k] = b[k] / pivotVal.$$

Після нормалізації A[k][k] = 1.

Крок 4. Елімінація стовпця k. Для кожного рядка i ≠ k виконується:

$$A[i][j] = A[i][j] - A[i][k] \cdot A[k][j], \quad j = 0..n-1,$$
$$b[i] = b[i] - A[i][k] \cdot b[k].$$

Після елімінації A[i][k] = 0 для всіх i ≠ k.

Після виконання всіх n ітерацій матриця A перетворюється на одиничну матрицю I, а вектор b містить шуканий розв'язок x. Загальна обчислювальна складність алгоритму становить $O(n^3)$ [4] арифметичних операцій.

Ключовою перевагою методу Гауса-Жордана з точки зору паралелізації є те, що на кроці елімінації (крок 4) обробка кожного рядка i є повністю незалежною від обробки інших рядків. Це дозволяє виконувати елімінацію всіх n рядків одночасно на GPU, де кожен work-item обробляє свій рядок.

Схему алгоритму Гауса-Жордана з частковою вибіркою головного елемента наведено на рисунку 2.1.

Числова стійкість та верифікація результатів. Для оцінки точності отриманого розв'язку x використовується відносна нев'язка:

$$r = ||A \cdot x - b|| / ||b||,$$

де $||\cdot||$ позначає евклідову норму вектора. Відносна нев'язка показує, наскільки знайдений розв'язок задовольняє вихідну систему рівнянь. Інтерпретація значень нев'язки наведена у таблиці 2.1.

У даній роботі для всіх тестових матриць досягається нев'язка $r < 1 \times 10^{-10}$, що підтверджує коректність реалізації як CPU-, так і GPU-версій алгоритму.

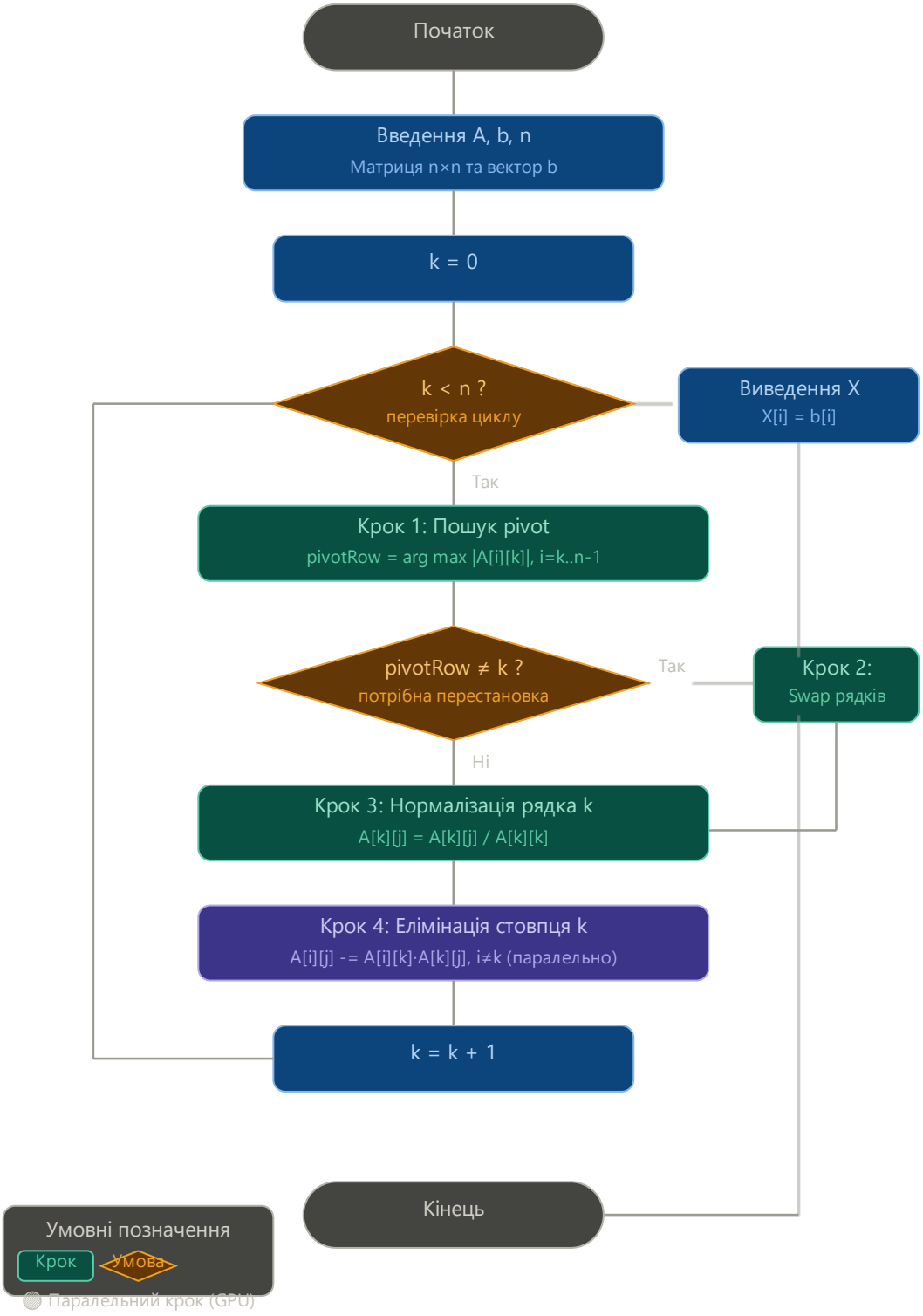


Рисунок 2.1 – Блок-схема алгоритму

Таблиця 2.1 – Інтерпретація значень відносної нев’язки

Значення нев’язки r	**Інтерпретація**
$r < 1\times10^{-10}$	Відмінна точність (машинний епсилон)
$1\times10^{-10} \le r < 1\times10^{-6}$	Добра точність
$1\times10^{-6} \le r < 1\times10^{-3}$	Задовільна точність
$r \ge 1\times10^{-3}$	Погана обумовленість матриці

2.2 Модель виконання та пам’яті OpenCL

Для ефективного використання OpenCL необхідно розуміти дві фундаментальні концепції: модель виконання та модель пам’яті. Ці концепції визначають спосіб організації паралельних обчислень та управління даними на GPU.

Програма OpenCL складається з двох частин: хост-програми, що виконується на CPU, та ядер [1] (kernels), що виконуються на обчислювальному пристрої (GPU). Хост-програма відповідає за ініціалізацію пристрою, компіляцію ядер, управління пам’яттю та запуск обчислень.

Основною одиницею паралельного виконання є **work-item [3]** – окремий екземпляр ядра з унікальним глобальним ідентифікатором get_global_id(). У даній роботі кожен work-item відповідає за обробку одного рядка матриці під час елімінації. Таким чином, при розмірності матриці n запускається n паралельних work-items, кожен з яких незалежно виконує операції над своїм рядком.

Work-items об’єднуються у **work-groups** – групи, що виконуються на одному обчислювальному блоці GPU. Work-items у межах однієї групи можуть синхронізуватися через бар’єр barrier() та обмінюватися даними через локальну пам’ять. Сукупність усіх work-items утворює **NDRange** – глобальний простір індексів, що визначає загальну кількість паралельних екземплярів ядра.

Ініціалізація OpenCL на хості виконується у такій послідовності:

- вибір платформи та пристрою (clGetPlatformIDs, clGetDeviceIDs [13]);
- створення контексту (clCreateContext);
- створення черги команд (clCreateCommandQueue [13]);
- компіляція ядер із вихідного коду (clCreateProgramWithSource, clBuildProgram) [1];
- створення буферів у пам’яті пристрою (clCreateBuffer);

– запуск ядер (clEnqueueNDRangeKernel);

– зчитування результатів (clEnqueueReadBuffer).

OpenCL визначає чотири рівні пам’яті з різними характеристиками швидкодії та доступності.

У таблиці 2.2 наведено порівняльну характеристику рівнів пам’яті OpenCL.

Таблиця 2.2 – Рівні пам’яті OpenCL та їх характеристики

Тип пам’яті	**Доступність**	**Швидкість**	**Розмір**	**Використання у роботі**
Глобальна (VRAM)	Усі work-items	Середня	~ГБ	Матриця A та вектор b
Локальна (shared)	Work-group	Висока	~десятки КБ	Не використовується
Приватна	Один work-item	Найвища	~байти	Локальні змінні ядра
Константна	Усі (read-only)	Висока	~64 КБ	Не використовується

У даній роботі матриця A та вектор b зберігаються у **глобальній пам’яті** GPU (VRAM). Це обумовлено тим, що розмір матриці для n = 1024 становить 1024 × 1024 × 8 байт = 8 МБ, що перевищує обсяг локальної пам’яті. Усі чотири OpenCL-ядра звертаються до глобальної пам’яті для читання та запису елементів матриці.

Критично важливим архітектурним рішенням є мінімізація передачі даних між CPU та GPU через шину PCIe. Пропускна здатність шини PCIe 3.0 x16 становить близько 16 ГБ/с [12], тоді як пропускна здатність VRAM сягає 192 ГБ/с – різниця у 12 разів. Тому матриця A та вектор b копіюються у VRAM лише один раз на початку обчислень, всі n ітерацій елімінації виконуються безпосередньо у VRAM, а результат зчитується лише один раз після завершення всіх обчислень. Схему рівнів пам’яті OpenCL наведено на рисунку 2.2.

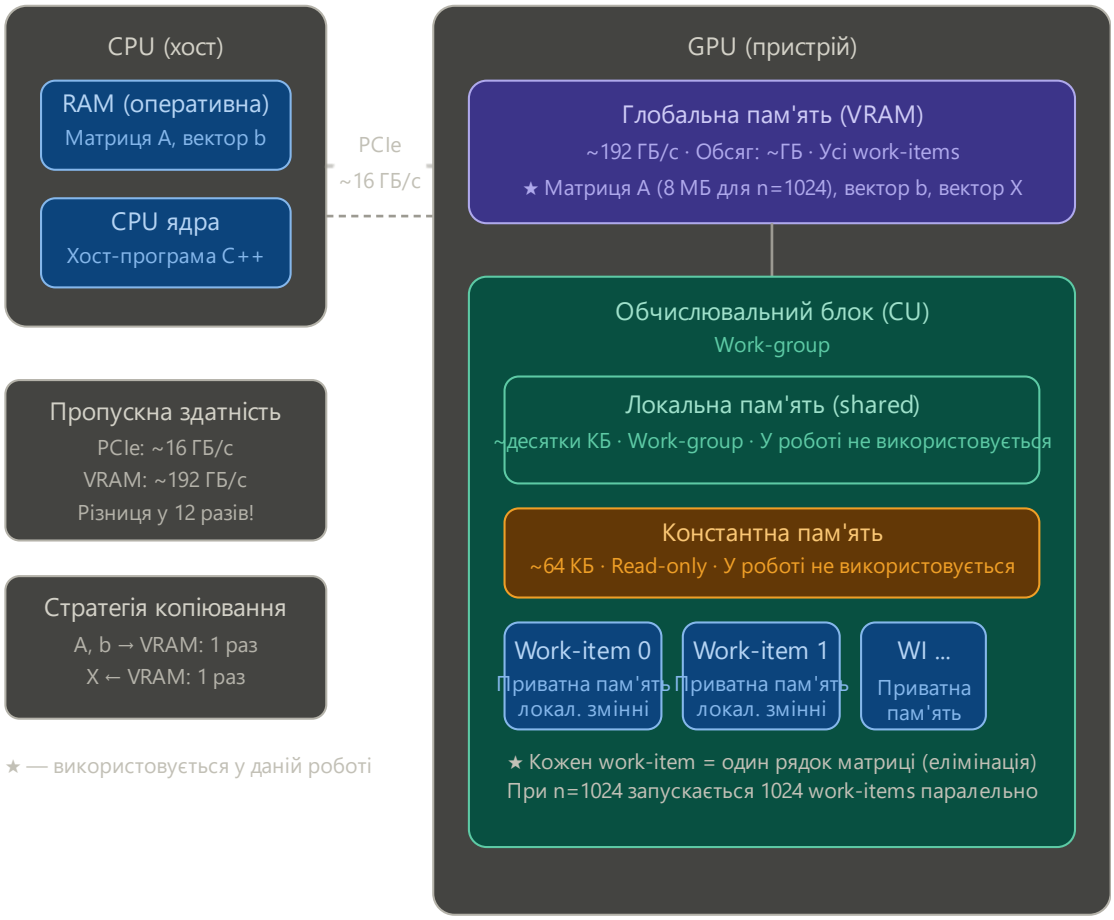


Рисунок 2.2 – Ієрархія пам’яті OpenCL та розміщення даних у даній роботі

На кожній ітерації k між CPU та GPU передається мінімальний обсяг даних: індекс рядка з максимальним елементом (4 байти, тип int) та значення головного елемента (8 байт, тип double). Таким чином, загальний обсяг передачі даних через PCIe для матриці n×n становить:

– одноразова передача матриці: $n^2 \times 8$ байт;
– n зчитувань pivotRow та pivotVal: $n \times 12$ байт;
– одноразове зчитування результату: $n \times 8$ байт.

Для n = 1024 загальний обсяг передачі через PCIe дорівнює приблизно 8,4 МБ, тоді як обсяг обчислень у VRAM становить ~2 млрд операцій. Це співвідношення забезпечує ефективне використання GPU та мінімізацію простою через очікування даних.

Підтримка подвійної точності. За замовчуванням більшість GPU не підтримують тип double в OpenCL-ядрах. Для увімкнення підтримки подвійної точності на початку кожного ядра додається директива [1]:

#pragma OPENCL EXTENSION cl_khr_fp64 : enable.

Ця директива є обов’язковою для коректної роботи алгоритму, оскільки метод Гауса-Жордана вимагає високої точності обчислень для досягнення нев’язки $r < 1 \times 10^{-10}$.

2.3 Архітектура застосунку

Розроблюваний застосунок має трирівневу архітектуру, що складається з рівня інтерфейсу користувача, рівня обчислювального ядра та рівня паралельних обчислень на GPU. Така архітектура забезпечує чітке розділення відповідальності між компонентами та спрощує розширення функціональності системи.

Рівень інтерфейсу користувача реалізований на основі фреймворку Flutter з використанням мови програмування Dart. Flutter є кросплатформним фреймворком [10], що дозволяє створювати застосунки для Windows, macOS, Linux, Android та iOS з єдиної кодової бази. У даній роботі цільовою платформою є Windows x64. Інтерфейс складається з двох вкладок: вкладки розв’язувача та вкладки бенчмарку. Управління станом застосунку реалізовано за допомогою пакету Provider [10], що реалізує патерн ChangeNotifier.

Рівень обчислювального ядра реалізований мовою C++17 та містить всю логіку розв’язання СЛАР. Цей рівень надає зовнішній інтерфейс у вигляді експортованих функцій, що викликаються з рівня інтерфейсу через механізм Dart FFI. Обчислювальне ядро реалізує два алгоритми розв’язання: послідовний CPU-алгоритм та паралельний GPU-алгоритм на основі OpenCL.

Рівень паралельних обчислень реалізований у вигляді чотирьох OpenCL-ядер, що виконуються безпосередньо на GPU. Цей рівень отримує дані від обчислювального ядра C++ через API OpenCL та повертає результати після завершення обчислень.

Взаємодія між рівнями інтерфейсу та обчислювального ядра здійснюється через **Dart FFI** (Foreign Function Interface) – механізм прямого виклику [11] нативних C++ функцій з коду Dart без необхідності написання проміжних плагінів. Dart виділяє буфери пам'яті через calloc та передає вказівники на них у C++ функції [11]. Після отримання результатів пам'ять звільняється через calloc.free(). Загальну архітектуру застосунку наведено на рисунку 2.3.

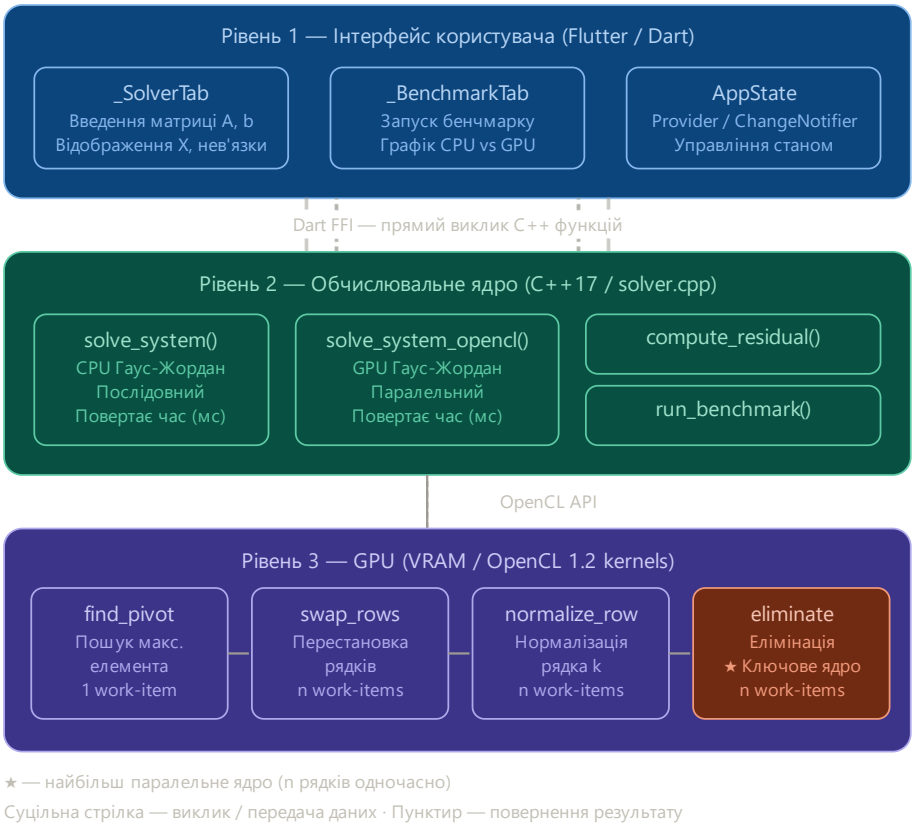


Рисунок 2.3 – Архітектура застосунку для розв'язання СЛАР засобами OpenCL

У таблиці 2.3 наведено перелік основних компонентів застосунку та їх призначення.

Важливою особливістю архітектури є те, що Flutter-застосунок та C++ – бібліотека скомпільовані в **єдиний виконуваний файл** (.exe). Dart FFI викликає C++ функції як символи всередині того самого процесу, що усуває накладні витрати міжпроцесної комунікації та забезпечує максимальну швидкість передачі даних між рівнями.

Таблиця 2.3 – Основні компоненти за стосунку

Компонент	**Мова / Технологія**	**Призначення**
main.dart	Dart / Flutter	Інтерфейс користувача, управління станом
AppState	Dart / Provider	Зберігання стану, виклик FFI-функцій
_SolverTab	Dart / Flutter	Вкладка введення та розв’язання СЛАР
_BenchmarkTab	Dart / Flutter	Вкладка порівняльного тестування
_BenchmarkChart	Dart / CustomPainter	Графік CPU vs GPU
solver.cpp	C++17 / OpenCL	Обчислювальне ядро, всі алгоритми
solve_system()	C++	Послідовний алгоритм Гауса-Жордана
solve_system_opencl()	C++ / OpenCL	Паралельний алгоритм на GPU
compute_residual()	C++	Обчислення відносної нев’язки
run_benchmark()	C++	Автоматичне тестування 8 розмірностей
find_pivot	OpenCL kernel	Пошук головного елемента на GPU
swap_rows	OpenCL kernel	Перестановка рядків на GPU
normalize_row	OpenCL kernel	Нормалізація рядка на GPU
eliminate	OpenCL kernel	Паралельна елімінація на GPU

У таблиці 2.4 наведено FFI-інтерфейс між Dart та C++.

Таблиця 2.4 – FFI-інтерфейс Dart ↔ C++

C++ функція	**Параметри**	**Повертає**
get_gpu_name()	-	Назва GPU (рядок)
solve_system()	A, b, X (вказівники), n	Час CPU (мс)
solve_system_opencl()	A, b, X (вказівники), n	Час GPU (мс) або −1
compute_residual()	A, b, X (вказівники), n	Відносна нев’язка r
run_benchmark()	-	CSV-рядок з результатами

Така архітектура забезпечує повне розділення між логікою інтерфейсу та обчислювальною логікою, що спрощує тестування та подальше розширення системи.

2.4 Проєктування GPU-ядер та схема передачі даних

Паралельна реалізація методу Гауса-Жордана на GPU складається з чотирьох OpenCL-ядер, що виконуються послідовно на кожній з n ітерацій алгоритму. Кожне ядро вирішує окрему підзадачу і має різний ступінь паралелізму.

Перше ядро виконує пошук рядка з максимальним за модулем елементом у стовпці k серед рядків від k до n−1. На відміну від інших ядер, find_pivot запускається з одним work-item, оскільки задача пошуку максимуму є послідовною за своєю природою. Код ядра find_pivot наведено на рисунку 2.4.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void find_pivot(__global const double* A,
                         __global int* outRow,
                         const int n, const int k)
{
   int pivot = k;
   double maxVal = fabs(A[k * n + k]);
   for (int i = k + 1; i < n; i++) {
      double val = fabs(A[i * n + k]);
      if (val > maxVal) { maxVal = val; pivot = i; }
   }
   outRow[0] = pivot;
}
```

Рисунок 2.4 – Код OpenCL-ядра find_pivot

Work-item послідовно переглядає елементи стовпця k та записує індекс рядка з максимальним значенням у буфер outRow у VRAM. Після завершення ядра хост зчитує лише 4 байти – значення pivotRow.

Друге ядро виконує перестановку рядків k і pivotRow у матриці A та векторі b. Ядро запускається з n паралельними work-items, де кожен work-item з ідентифікатором j переставляє елементи на позиції j. Операція перестановки виконується повністю у VRAM без копіювання даних на CPU. Код ядра swap_rows наведено на рисунку 2.5.

```
__kernel void swap_rows(__global double* A,
                        __global double* B,
                        const int n,
                        const int r1, const int r2)
{
   int j = get_global_id(0);
   if (j >= n) return;
   double tmp = A[r1 * n + j];
   A[r1 * n + j] = A[r2 * n + j];
   A[r2 * n + j] = tmp;
   if (j == 0) {
      double tb = B[r1]; B[r1] = B[r2]; B[r2] = tb;
   }
}
```

Рисунок 2.5 – Код OpenCL-ядра swap_rows

Всі n перестановок елементів виконуються одночасно, що забезпечує максимальну швидкість виконання цього кроку.

Третє ядро нормалізує рядок k, ділячи кожен його елемент на значення головного елемента pivotVal. Ядро запускається з n паралельними work-items, де кожен work-item ділить один елемент рядка. Значення pivotVal передається як аргумент ядра, що є безпечним з точки зору гонки даних. Код ядра normalize_row наведено на рисунку 2.6.

```
__kernel void normalize_row(__global double* A,
                            __global double* B,
                            const int n, const int k,
                            const double pivotVal)
{
   int j = get_global_id(0);
   if (j >= n) return;
   A[k * n + j] /= pivotVal;
   if (j == 0) B[k] /= pivotVal;
}
```

Рисунок 2.6 – Код OpenCL-ядра normalize_row

Четверте ядро є найбільш паралельним і обчислювально інтенсивним. Воно виконує елімінацію стовпця k для всіх рядків $i \neq k$ одночасно. Кожен з n паралельних work-items обробляє один рядок матриці. Саме це ядро є

основним джерелом прискорення GPU-реалізації порівняно з CPU. Код ядра eliminate наведено на рисунку 2.7.

```
__kernel void eliminate(__global double* A,
                        __global double* B,
                        const int n, const int k)
{
   int i = get_global_id(0);
   if (i >= n || i == k) return;
   double factor = A[i * n + k];
   for (int j = 0; j < n; j++) {
      A[i * n + j] -= factor * A[k * n + j];
   }
   B[i] -= factor * B[k];
}
```

Рисунок 2.7 – Код OpenCL-ядра eliminate

Замість послідовного виконання n операцій елімінації вони виконуються паралельно, що забезпечує теоретичне прискорення у n разів.

У таблиці 2.5 наведено порівняльну характеристику чотирьох OpenCL-ядер.

Таблиця 2.5 – Характеристика OpenCL-ядер

Ядро	**Work-items**	**Складність**	**Паралелізм**	**Роль**
find_pivot	1	O(n)	Послідовне	Пошук максимуму
swap_rows	n	O(n)	Повне	Перестановка рядків
normalize_row	n	O(n)	Повне	Нормалізація рядка
eliminate	n	$O(n^2)$	Повне	Елімінація стовпця

Ефективна організація передачі даних між оперативною пам’яттю CPU та відеопам’яттю GPU є критично важливою для досягнення високої продуктивності. На початку обчислень матриця A та вектор b копіюються у VRAM одноразово. Для матриці розмірністю 1024×1024 це становить 8 МБ, для вектора b – 8 КБ. Схему передачі даних наведено на рисунку 2.8.

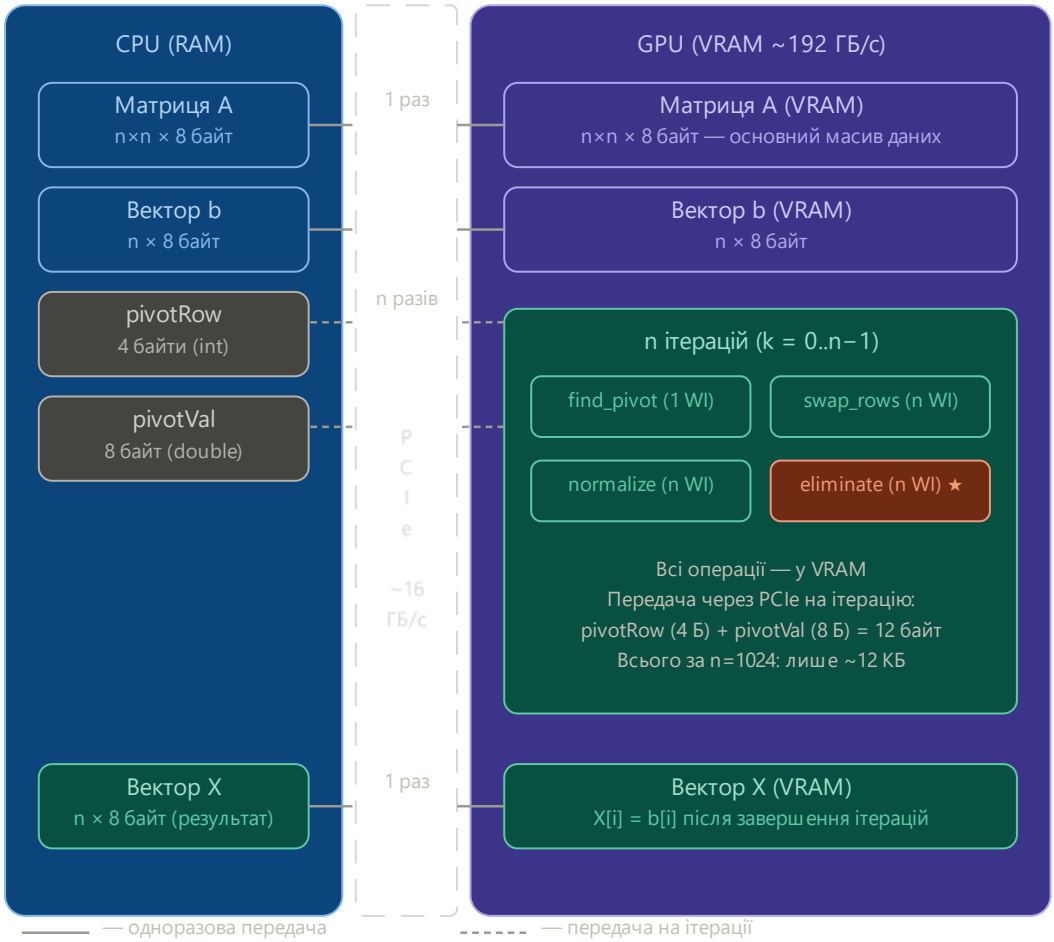


Рисунок 2.8 – Схема передачі даних між CPU та GPU під час розв’язання СЛАР

Під час виконання n ітерацій через шину PCIe передається лише мінімальний обсяг даних – індекс рядка pivotRow (4 байти) та значення головного елемента pivotVal (8 байт) на кожній ітерації. Після завершення всіх ітерацій вектор розв’язку X зчитується з VRAM одноразово.

У таблиці 2.6 наведено зведену інформацію про обсяги передачі даних.

Таблиця 2.6 – Обсяги передачі даних через шину PCIe

Розмірність N	**Завантаження A+b**	**Під час ітерацій**	**Зчитування X**	**Всього**
64	~32 КБ	~768 Б	~512 Б	~33 КБ
256	~512 КБ	~3 КБ	~2 КБ	~517 КБ
512	~2 МБ	~6 КБ	~4 КБ	~2,01 МБ
1024	~8 МБ	~12 КБ	~8 КБ	~8,02 МБ

Як видно з таблиці 2.6, обсяг передачі даних через PCIe є незначним порівняно з обсягом обчислень у VRAM, що підтверджує ефективність обраної стратегії одноразового копіювання матриці.

2.5 Проєктування FFI-інтерфейсу Dart ↔ C++

Взаємодія між графічним інтерфейсом користувача на Flutter та обчислювальним ядром на C++ реалізована через механізм Dart FFI (Foreign Function Interface). FFI дозволяє коду Dart напряму викликати функції, написані мовою C або C++, без необхідності створення проміжних плагінів чи використання міжпроцесної комунікації.

Dart FFI працює на рівні бінарного інтерфейсу застосунку (ABI). Dart-код завантажує динамічну бібліотеку або звертається до символів у поточному процесі, отримує вказівник на C++ функцію та викликає її безпосередньо. У даній роботі Flutter-застосунок та C++ код скомпільовані в єдиний виконуваний файл, тому FFI звертається до символів всередині того самого процесу через DynamicLibrary.process()[11].

Для кожної C++ функції у Dart-коді визначаються два типи: нативний тип функції (як вона виглядає у C++) та тип Dart-функції (як вона викликається з Dart). Оголошення FFI-зв’язку для функції solve_system наведено на рисунку 2.9.

```
// Нативний тип C++ функції
typedef SolveSystemNative = Double Function(
   Pointer<Double>, Pointer<Double>, Pointer<Double>, Int32);

// Dart-тип для виклику
typedef SolveSystemDart = double Function(
   Pointer<Double>, Pointer<Double>, Pointer<Double>, int);

// Отримання вказівника на функцію
final solveSystem = _lib.lookupFunction<SolveSystemNative,
   SolveSystemDart>('solve_system');
```

Рисунок 2.9 – Оголошення FFI-зв’язку для функції solve_system

Оскільки C++ функції працюють з вказівниками на масиви типу double, Dart повинен виділяти нативну пам’ять, яка доступна як Dart-коду, так і C++ коду. Для цього використовується пакет ffi з функцією calloc. Приклад виділення пам’яті та виклику C++ функції наведено на рисунку 2.10.

```
// Виділення нативної пам’яті
final pA = calloc<Double>(n * n);
final pB = calloc<Double>(n);
final pX = calloc<Double>(n);

// Заповнення матриці A та вектора b
for (int I = 0; I < n * n; i++) pA[i] = matrixData[i];
for (int I = 0; I < n; i++) pB[i] = vectorData[i];

// Виклик C++ функції
final cpuTime = solveSystem(pA, pB, pX, n);

// Зчитування результату
final result = List.generate(n, (i) => pX[i]);

// Звільнення пам’яті
calloc.free(pA);
calloc.free(pB);
calloc.free(pX);
```

Рисунок 2.10 – Виділення пам’яті та виклик C++ функції через FFI

Важливо звільняти виділену пам’ять після кожного виклику через calloc.free(), оскільки Dart garbage collector не керує нативною пам’яттю. Витік нативної пам’яті може призвести до поступового вичерпання оперативної пам’яті застосунку.

Для того щоб C++ функції були доступні через FFI, вони повинні бути оголошені з директивою extern «C», яка вимикає декорування імен (name mangling) компілятором C++. Також необхідно використовувати макрос __declspec(dllexport) для Windows, щоб символи були видимі ззовні. Оголошення експортованих функцій наведено на рисунку 2.11.

У таблиці 2.7 наведено повний перелік FFI-функцій із зазначенням типів параметрів та значень, що повертаються.

```
extern «C» {
    __declspec(dllexport)
    double solve_system(double* A, double* B,
                        double* X, int n);

    __declspec(dllexport)
    double solve_system_opencl(double* A, double* B,
                               double* X, int n);

    __declspec(dllexport)
    double compute_residual(double* A, double* B,
                            double* X, int n);

    __declspec(dllexport)
    const char* get_gpu_name();

    __declspec(dllexport)
    const char* run_benchmark();
}
```

Рисунок 2.11 – Оголошення експортованих C++ функцій для FFI

Таблиця 2.7 – Специфікація FFI-інтерфейсу Dart ↔ C++

C++ функція	**Параметри**	**Тип результату**	**Значення результату**
solve_system	A, b, X (double*), n (int)	double	Час CPU (мс)
solve_system_opencl	A, b, X (double*), n (int)	double	Час GPU (мс) або −1
compute_residual	A, b, X (double*), n (int)	double	Відносна нев’язка r
get_gpu_name	-	const char*	Назва GPU
run_benchmark	-	const char*	CSV-рядок результатів

Повернення рядкових значень з C++ у Dart реалізовано через тип Pointer<Utf8>. Dart-код перетворює нативний рядок у Dart String за допомогою методу toDartString(). Функція run_benchmark повертає результати у форматі CSV-рядка, наприклад «64,0.4,15.0;128,3.7,30.8;...», який Dart-код розбирає на список об’єктів BenchmarkPoint для відображення графіку та таблиці.

Таким чином, FFI-інтерфейс забезпечує ефективну та типобезпечну взаємодію між Flutter UI та C++ обчислювальним ядром з мінімальними накладними витратами.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 CPU-реалізація методу Гауса-Жордана

Послідовна CPU-реалізація алгоритму Гауса-Жордана виконана мовою C++17 та є базовою для порівняння продуктивності з GPU-реалізацією. Функція приймає на вхід матрицю A, вектор b та розмірність n, виконує розв'язання СЛАР та записує результат у вектор X. Повна реалізація функції solve_system наведена на рисунку 3.1.

```
double solve_system(double* A, double* B, double* X, int n) {
    auto start = chrono::high_resolution_clock::now();

    for (int k = 0; k < n; k++) {
        // Крок 1: пошук головного елемента
        int pivotRow = k;
        double maxVal = fabs(A[k * n + k]);
        for (int i = k + 1; i < n; i++) {
            if (fabs(A[i * n + k]) > maxVal) {
                maxVal = fabs(A[i * n + k]);
                pivotRow = i;
            }
        }
        // Крок 2: перестановка рядків
        if (pivotRow != k) {
            for (int j = 0; j < n; j++)
                swap(A[k * n + j], A[pivotRow * n + j]);
            swap(B[k], B[pivotRow]);
        }
        // Крок 3: нормалізація рядка k
        double pivot = A[k * n + k];
        for (int j = 0; j < n; j++) A[k * n + j] /= pivot;
        B[k] /= pivot;

        // Крок 4: елімінація стовпця k
        for (int i = 0; i < n; i++) {
            if (i == k) continue;
            double factor = A[i * n + k];
            for (int j = 0; j < n; j++)
                A[i * n + j] -= factor * A[k * n + j];
            B[i] -= factor * B[k];
        }
    }
    // Результат: X[i] = B[i]
    for (int i = 0; i < n; i++) X[i] = B[i];

    auto end = chrono::high_resolution_clock::now();
    return chrono::duration<double, milli>(end – start).count();
}
```

Рисунок 3.1 – Реалізація функції solve_system (CPU, C++)

Вимірювання часу виконання здійснюється за допомогою std::chrono::high_resolution_clock [14], що забезпечує точність вимірювань до мікросекунд. Функція повертає час виконання у мілісекундах як значення типу double.

Зовнішній цикл по k виконує n ітерацій. На кожній ітерації пошук головного елемента виконується за O(n) операцій, нормалізація – за O(n) операцій, а елімінація – за $O(n^2)$ операцій. Таким чином, загальна обчислювальна складність становить $O(n^3)$.

Особливістю реалізації є те, що функція модифікує вхідну матрицю A та вектор b безпосередньо. Тому перед викликом функції Flutter-код копіює оригінальні дані у тимчасові буфери, щоб зберегти можливість повторного розв’язання без повторного введення даних.

3.2 GPU-реалізація на OpenCL

GPU-реалізація алгоритму Гауса-Жордана виконана у функції solve_system_opencl та використовує чотири OpenCL-ядра, розроблені у підрозділі 2.4.

Функція складається з трьох етапів: ініціалізації OpenCL-контексту, виконання n ітерацій алгоритму у VRAM та зчитування результату.

На першому етапі виконується вибір обчислювального пристрою, створення контексту та компіляція ядер. Реалізацію функції selectDevice, що обирає найкращий доступний GPU, наведено на рисунку 3.2.

Після вибору пристрою створюється контекст та черга команд, компілюються чотири OpenCL-ядра з рядків вихідного коду, а матриця A та вектор b копіюються у VRAM через clEnqueueWriteBuffer. Ініціалізацію контексту та буферів наведено на рисунку 3.3.

На кожній ітерації k послідовно запускаються чотири ядра. Реалізацію головного циклу ітерацій наведено на рисунку 3.4.

```
cl_device_id selectDevice() {
    cl_uint numPlatforms = 0;
    clGetPlatformIDs(0, nullptr, &numPlatforms);
    vector<cl_platform_id> platforms(numPlatforms);
    clGetPlatformIDs(numPlatforms, platforms.data(), nullptr);

    cl_device_id bestDevice = nullptr;
    for (auto& plat : platforms) {
        cl_uint numDevices = 0;
        clGetDeviceIDs(plat, CL_DEVICE_TYPE_GPU,
                    0, nullptr, &numDevices);
        if (numDevices == 0) continue;
        vector<cl_device_id> devices(numDevices);
        clGetDeviceIDs(plat, CL_DEVICE_TYPE_GPU,
                    numDevices, devices.data(), nullptr);
        bestDevice = devices[0];
    }
    return bestDevice;
}
```

Рисунок 3.2 – Реалізація функції selectDevice

```
cl_context ctx = clCreateContext(nullptr, 1,
                            &device, nullptr,
                            nullptr, nullptr);
cl_command_queue queue = clCreateCommandQueue(ctx,
                        device, 0, nullptr);

// Компіляція ядер
cl_program program = clCreateProgramWithSource(ctx, 1,
                    &kernelSource, nullptr, nullptr);
clBuildProgram(program, 1, &device,
                nullptr, nullptr, nullptr);

// Створення буферів у VRAM
cl_mem bufA = clCreateBuffer(ctx,
    CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
    n * n * sizeof(double), A, nullptr);
cl_mem bufB = clCreateBuffer(ctx,
    CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
    n * sizeof(double), B, nullptr);
cl_mem bufPivot = clCreateBuffer(ctx,
    CL_MEM_READ_WRITE,
    sizeof(int), nullptr, nullptr);
```

Рисунок 3.3 – Ініціалізація OpenCL-контексту та буферів у VRAM

```
for (int k = 0; k < n; k++) {
    // Ядро 1: find_pivot
    clSetKernelArg(kFindPivot, 0, sizeof(cl_mem), &bufA);
    clSetKernelArg(kFindPivot, 1, sizeof(cl_mem), &bufPivot);
```

```
    clSetKernelArg(kFindPivot, 2, sizeof(int), &n);
    clSetKernelArg(kFindPivot, 3, sizeof(int), &k);
    size_t one = 1;
    clEnqueueNDRangeKernel(queue, kFindPivot,
                           1, nullptr, &one,
                           nullptr, 0, nullptr, nullptr);
    clFinish(queue);
    // Зчитування pivotRow з VRAM
    int pivotRow;
    clEnqueueReadBuffer(queue, bufPivot, CL_TRUE, 0,
                        sizeof(int), &pivotRow,
                        0, nullptr, nullptr);
    // Ядро 2: swap_rows (якщо потрібно)
    if (pivotRow != k) {
        clSetKernelArg(kSwapRows, 0, sizeof(cl_mem), &bufA);
        clSetKernelArg(kSwapRows, 1, sizeof(cl_mem), &bufB);
        clSetKernelArg(kSwapRows, 2, sizeof(int), &n);
        clSetKernelArg(kSwapRows, 3, sizeof(int), &k);
        clSetKernelArg(kSwapRows, 4, sizeof(int), &pivotRow);
        size_t gsN = n;
        clEnqueueNDRangeKernel(queue, kSwapRows,
                               1, nullptr, &gsN,
                               nullptr, 0, nullptr, nullptr);
        clFinish(queue);
    }
    // Зчитування pivotVal для normalize_row
    double pivotVal;
    clEnqueueReadBuffer(queue, bufA, CL_TRUE,
                        (k * n + k) * sizeof(double),
                        sizeof(double), &pivotVal,
                        0, nullptr, nullptr);
    // Ядро 3: normalize_row
    clSetKernelArg(kNormRow, 0, sizeof(cl_mem), &bufA);
    clSetKernelArg(kNormRow, 1, sizeof(cl_mem), &bufB);
    clSetKernelArg(kNormRow, 2, sizeof(int), &n);
    clSetKernelArg(kNormRow, 3, sizeof(int), &k);
    clSetKernelArg(kNormRow, 4, sizeof(double), &pivotVal);
    size_t gsN = n;
    clEnqueueNDRangeKernel(queue, kNormRow,
                           1, nullptr, &gsN,
                           nullptr, 0, nullptr, nullptr);
    clFinish(queue);
    // Ядро 4: eliminate
    clSetKernelArg(kElim, 0, sizeof(cl_mem), &bufA);
    clSetKernelArg(kElim, 1, sizeof(cl_mem), &bufB);
    clSetKernelArg(kElim, 2, sizeof(int), &n);
    clSetKernelArg(kElim, 3, sizeof(int), &k);
    clEnqueueNDRangeKernel(queue, kElim,
                           1, nullptr, &gsN,
                           nullptr, 0, nullptr, nullptr);
    clFinish(queue);
}
```

Рисунок 3.4 – Головний цикл ітерацій GPU-реалізації

Після завершення всіх n ітерацій вектор розв’язку зчитується з VRAM одноразово, після чого всі OpenCL-ресурси звільняються. Реалізацію завершального етапу наведено на рисунку 3.5.

```
// Зчитування результату X з VRAM
clEnqueueReadBuffer(queue, bufB, CL_TRUE, 0,
                    n * sizeof(double), X,
                    0, nullptr, nullptr);

// Звільнення OpenCL ресурсів
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufPivot);
clReleaseKernel(kFindPivot);
clReleaseKernel(kSwapRows);
clReleaseKernel(kNormRow);
clReleaseKernel(kElim);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(ctx);
```

Рисунок 3.5 – Зчитування результату та звільнення OpenCL-ресурсів

Звільнення ресурсів є обов’язковим кроком, оскільки OpenCL-об’єкти займають пам’ять як на CPU, так і на GPU. Невиконання цього кроку призводить до витоку пам’яті при повторних викликах функції.

У разі відсутності сумісного GPU або помилки ініціалізації OpenCL функція solve_system_opencl повертає значення −1, що сигналізує Flutter-застосунку про неможливість виконання GPU-обчислень. Інтерфейс застосунку обробляє це значення та відображає відповідне повідомлення користувачу.

3.3 Реалізація інтерфейсу користувача

Графічний інтерфейс користувача реалізований з використанням фреймворку Flutter [10] та мови програмування Dart. Застосунок містить дві

основні вкладки: вкладку розв’язувача та вкладку бенчмарку. Управління станом реалізовано через пакет Provider з використанням класу AppState [10], що розширює ChangeNotifier.

Головний клас AppState зберігає весь стан застосунку та надає методи для виклику FFI-функцій. До основних полів стану належать: матриця коефіцієнтів, вектор правих частин, вектор розв’язку, час виконання CPU та GPU, значення нев’язки, список точок бенчмарку та рядки системного логу. При зміні будь-якого поля викликається notifyListeners(), що оновлює інтерфейс.

Вкладка розв’язувача надає користувачу можливість вводити матрицю A та вектор b, розв’язувати СЛАР двома методами та переглядати результати. Інтерфейс вкладки розв’язувача наведено на рисунку 3.6.

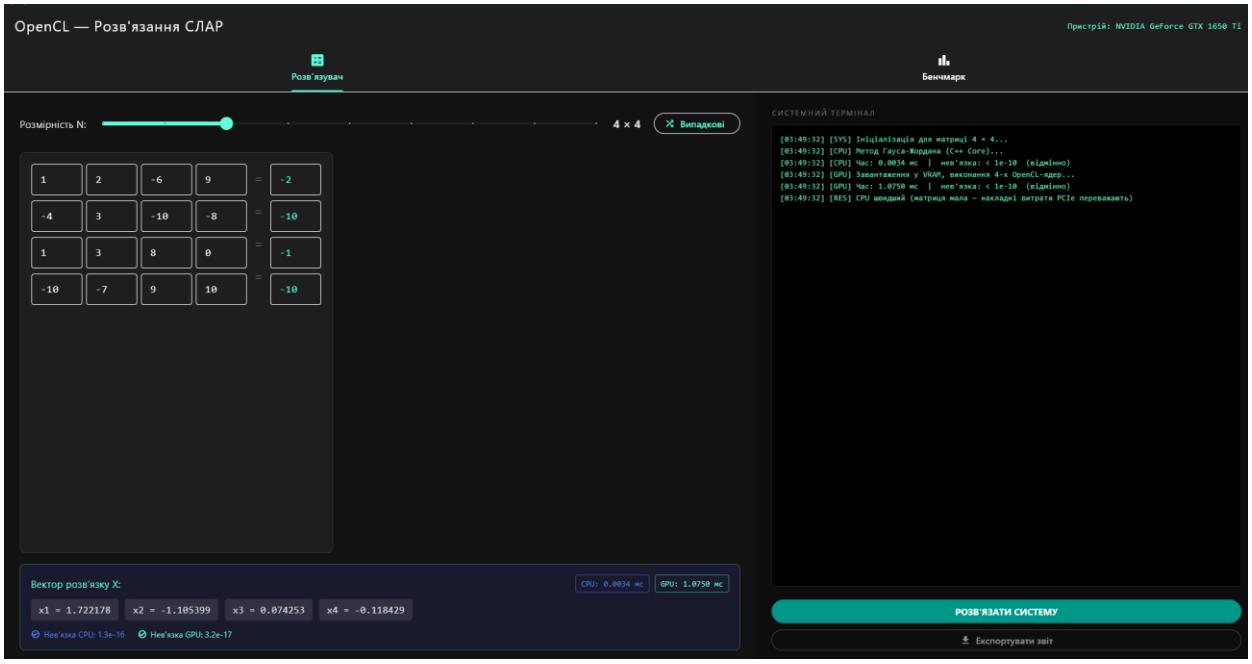


Рисунок 3.6 – Інтерфейс вкладки розв’язувача

Вкладка містить такі елементи інтерфейсу. Слайдер для вибору розмірності матриці N у діапазоні від 2 до 10. Сітка редагованих текстових полів для введення елементів матриці A – автоматично перебудовується при зміні N. Стовпець текстових полів для введення вектора b. Кнопка «Випадкові значення» для заповнення матриці псевдовипадковими числами в діапазоні від

−10 до 10. Кнопка «Розв’язати систему» для запуску обох алгоритмів послідовно. Блок відображення результатів з вектором X, часом CPU, часом GPU та значеннями нев’язки для обох методів. Системний термінал для відображення детального логу операцій. Кнопка «Експортувати звіт» для збереження результатів у текстовий файл.

Реалізацію методу розв’язання системи у класі AppState наведено на рисунку 3.7.

```
Future<void> solveSystem() async {
  final n = matrixSize;
  final pA = calloc<Double>(n * n);
  final pB = calloc<Double>(n);
  final pX = calloc<Double>(n);

  // Заповнення буферів
  for (int i = 0; i < n * n; i++)
    pA[i] = matrix[i];
  for (int i = 0; i < n; i++)
    pB[i] = vectorB[i];

  // CPU розв’язання
  cpuTime = _solveSystem(pA, pB, pX, n);
  cpuSolution = List.generate(n, (i) => pX[i]);
  cpuResidual = _computeResidual(pA, pB, pX, n);

  // Перезаповнення буферів для GPU
  for (int i = 0; i < n * n; i++)
    pA[i] = matrix[i];
  for (int i = 0; i < n; i++)
    pB[i] = vectorB[i];

  // GPU розв’язання
  gpuTime = _solveSystemOpenCL(pA, pB, pX, n);
  gpuSolution = List.generate(n, (i) => pX[i]);
  gpuResidual = _computeResidual(pA, pB, pX, n);

  calloc.free(pA);
  calloc.free(pB);
  calloc.free(pX);
  notifyListeners();
}
```

Рисунок 3.7 – Реалізація методу solveSystem у класі AppState

Зверніть увагу на повторне заповнення буферів перед GPU-розв’язанням. Це необхідно тому, що функція solve_system модифікує

матрицю A та вектор b під час виконання, тому для GPU-розв’язання потрібні оригінальні дані.

Вкладка бенчмарку надає можливість автоматично протестувати продуктивність обох реалізацій для восьми фіксованих розмірностей матриць та візуалізувати результати. Інтерфейс вкладки бенчмарку наведено на рисунку 3.8.

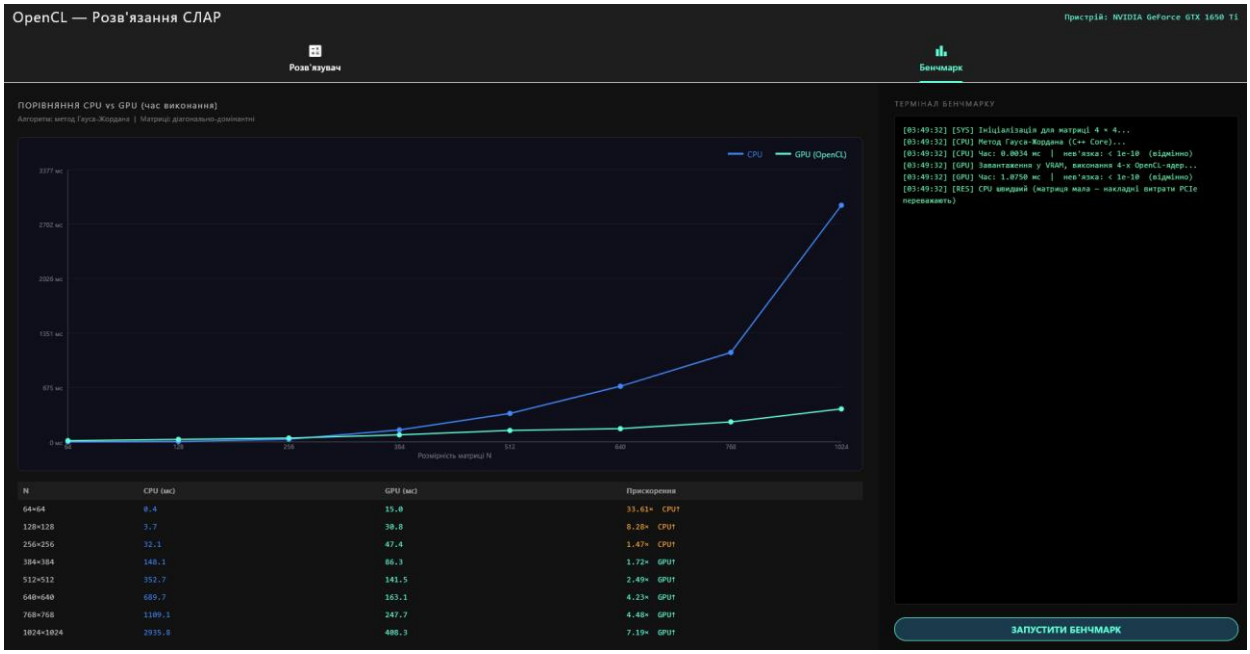


Рисунок 3.8 – Інтерфейс вкладки бенчмарку з графіком продуктивності

Вкладка містить такі елементи. Кнопка «Запустити бенчмарк» для запуску автоматичного тестування. Індикатор прогресу під час виконання тесту. Графік CPU vs GPU, реалізований через CustomPainter без використання зовнішніх бібліотек. Таблиця прискорення з виділенням рядків де GPU перевищує CPU. Термінал із детальним логом для кожної розмірності.

Графік реалізований класом _BenchmarkChart, що розширює CustomPainter. Метод paint отримує список точок BenchmarkPoint та малює дві лінії – синю для CPU та зелену для GPU – з підписами осей та легендою. Реалізацію методу запуску бенчмарку наведено на рисунку 3.9.

Функція run_benchmark на стороні C++ послідовно тестує вісім розмірностей матриць (64, 128, 256, 384, 512, 640, 768, 1024) та повертає

результати у форматі CSV-рядка. Dart-код розбирає цей рядок та будує список об’єктів BenchmarkPoint, що використовуються для відображення графіку та таблиці.

```
Future<void> runBenchmark() async {
  isBenchmarking = true;
  notifyListeners();

  final resultPtr = _runBenchmark();
  final csvString = resultPtr.toDartString();

  benchmarkPoints = [];
  for (final part in csvString.split(';')) {
    final vals = part.split(',');
    if (vals.length >= 3) {
      benchmarkPoints.add(BenchmarkPoint(
        n: int.parse(vals[0]),
        cpuMs: double.parse(vals[1]),
        gpuMs: double.parse(vals[2]),
      ));
    }
  }

  isBenchmarking = false;
  notifyListeners();
}
```

Рисунок 3.9 – Реалізація методу runBenchmark у класі AppState

Застосунок надає можливість експортувати результати розв’язання у текстовий файл OpenCL_Report.txt. Звіт містить розмірність системи, вектор розв’язку X, час виконання CPU та GPU, значення нев’язки для обох методів та назву GPU. Файл зберігається у поточній директорії застосунку та може бути відкритий будь-яким текстовим редактором.

3.4 Методика тестування та верифікація точності

Тестування розробленого застосунку проводилося за двома напрямками: верифікація коректності результатів обчислень та вимірювання

продуктивності обох реалізацій. Верифікація забезпечує впевненість у тому, що GPU-реалізація дає математично правильні результати, а не лише швидкі, але некоректні.

Основним інструментом верифікації є обчислення відносної нев'язки розв'язку. Функція compute_residual отримує оригінальну матрицю A, вектор b та знайдений розв'язок X, обчислює добуток A·X та порівнює його з b. Реалізацію функції compute_residual наведено на рисунку 3.10.

```
double compute_residual(double* A, double* B,
                        double* X, int n) {
    vector<double> AX(n, 0.0);

    // Обчислення добутку A·X
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            AX[i] += A[i * n + j] * X[j];

    // Обчислення евклідової норми нев'язки ||A·X – b||
    double num = 0.0, den = 0.0;
    for (int i = 0; i < n; i++) {
        double diff = AX[i] – B[i];
        num += diff * diff;
        den += B[i] * B[i];
    }
    return (den > 1e-15) ? sqrt(num) / sqrt(den) : sqrt(num);
}
```

Рисунок 3.10 – Реалізація функції compute_residual

Захист від ділення на нуль реалізовано через перевірку den > 1e-15. Якщо норма вектора b близька до нуля, функція повертає абсолютну нев'язку замість відносної.

Для тестування використовуються два типи матриць. Перший тип – матриці з ручним введенням малої розмірності (N від 2 до 10) для перевірки коректності на прикладах з відомим розв'язком. Другий тип – діагонально-домінантні матриці [5] для бенчмарку, що генеруються автоматично за формулою наведеною на рисунку 3.11.

```
// Генерація діагонально-домінантної матриці
for (int i = 0; i < n; i++) {
   double rowSum = 0.0;
   for (int j = 0; j < n; j++) {
      if (i != j) {
         A[i * n + j] = (rand() % 200 – 100) / 10.0;
         rowSum += fabs(A[i * n + j]);
      }
   }
   // Діагональний елемент > суми інших елементів рядка
   A[i * n + i] = rowSum + 1.0;
   B[i] = (rand() % 200 – 100) / 10.0;
}
```

Рисунок 3.11 – Генерація діагонально-домінантної тестової матриці

Діагональна домінантність гарантує єдиність розв’язку та добру обумовленість матриці, що забезпечує досягнення нев’язки $r < 1 \times 10^{-10}$ для обох реалізацій.

Вимірювання часу виконання CPU-реалізації здійснюється безпосередньо всередині функції solve_system через std::chrono::high_resolution_clock. Вимірювання охоплює лише час обчислень без урахування часу виділення пам’яті.

Для GPU-реалізації вимірювання часу охоплює весь цикл обчислень: копіювання матриці у VRAM, виконання n ітерацій з чотирма ядрами та зчитування результату. Таким чином, вимірюється реальний час «від даних до результату», що є коректною метрикою для порівняння практичної продуктивності CPU та GPU.

Для забезпечення достовірності результатів кожна розмірність у бенчмарку тестується один раз з фіксованим станом генератора псевдовипадкових чисел. У таблиці 3.1 наведено методику тестування для різних розмірностей.

Усі вимірювання продуктивності проводилися на комп’ютері з такою конфігурацією, яку наведено у таблиці 3.2.

Тестування проводилося при закритих інших ресурсомістких застосунках для мінімізації впливу фонових процесів на результати вимірювань. GPU-драйвер був оновлений до актуальної версії для забезпечення оптимальної продуктивності OpenCL.

Таблиця 3.1 – Методика тестування продуктивності

Розмірність N	**Тип матриці**	**Метод перевірки**	**Очікувана нев'язка**
2-10	Ручне введення	Порівняння з аналітичним розв'язком	$r < 1\times10^{-10}$
64-1024	Діагонально-домінантна	Відносна нев'язка r	$r < 1\times10^{-10}$

Таблиця 3.2 – Конфігурація тестового середовища

Компонент	**Характеристика**
GPU	NVIDIA GeForce GTX 1650 Ti
Операційна система	Windows 11 x64
Версія OpenCL	1.2
Компілятор	MSVC (Visual Studio 2022)
Стандарт C++	C++17
Версія Flutter	3.x
Версія Dart	3.x

3.5 Результати бенчмарку та аналіз продуктивності

Експериментальне дослідження продуктивності проводилося для восьми розмірностей матриць від 64×64 до 1024×1024. Для кожної розмірності вимірювався час виконання CPU- та GPU-реалізацій, обчислювалося прискорення та перевірялася коректність результатів через відносну нев'язку.

Зведені результати бенчмарку наведено у таблиці 3.3.

Таблиця 3.3 – Результати порівняльного тестування CPU та GPU реалізацій

Розмірність N	**CPU (мс)**	**GPU OpenCL (мс)**	**Прискорення**	**Переможець**
64×64	0,4	15,0	0,03×	CPU
128×128	3,7	30,8	0,12×	CPU
256×256	32,1	47,4	0,68×	CPU
384×384	148,1	86,3	1,72×	GPU
512×512	352,7	141,5	2,49×	GPU
640×640	689,7	163,1	4,23×	GPU
768×768	1109,1	247,7	4,48×	GPU
1024×1024	2935,8	408,3	7,19×	GPU

Графік залежності часу виконання від розмірності матриці наведено на рисунку 3.12.

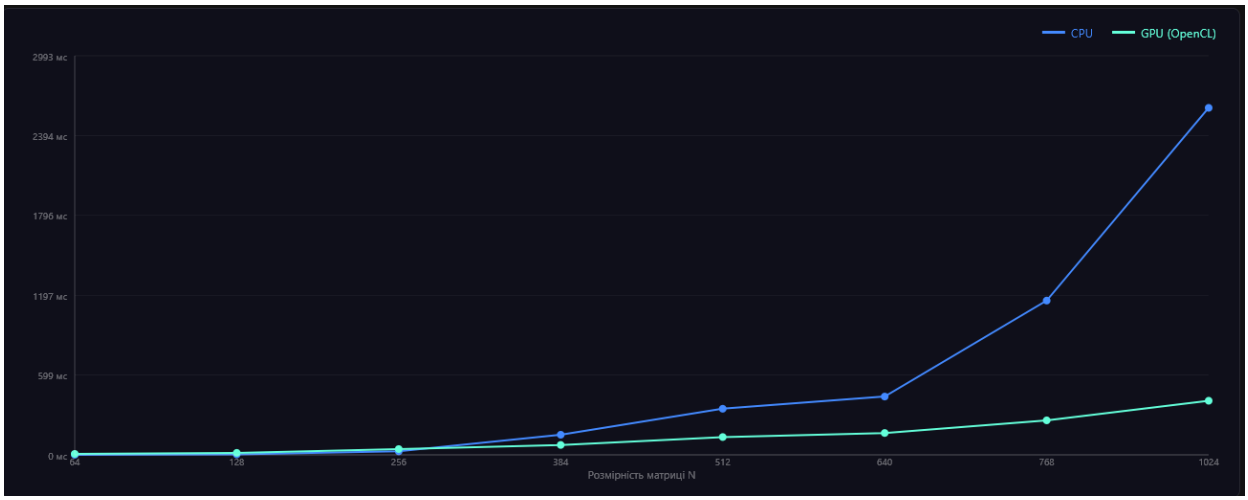


Рисунок 3.12 – Графік залежності часу виконання CPU та GPU від розмірності матриці

Аналіз даних таблиці 3.3 дозволяє виділити три характерні зони продуктивності.

Зона переваги CPU (N < 320). При малих розмірностях матриць CPU-реалізація значно перевершує GPU. При N = 64 CPU виконує розв’язання за 0,4 мс, тоді як GPU витрачає 15,0 мс – у 37,5 рази повільніше. При N = 256

різниця скорочується до 1,47 рази на користь CPU. Причиною є накладні витрати GPU-реалізації, що включають ініціалізацію OpenCL-контексту, компіляцію ядер та копіювання матриці у VRAM через шину PCIe. При малих матрицях ці накладні витрати перевищують виграш від паралелізації.

Точка беззбитковості (N ≈ 320). При розмірності N ≈ 320 час виконання CPU та GPU вирівнюється. Це є критичною точкою, після якої GPU стає ефективнішим. Між результатами для N = 256 (CPU = 32,1 мс, GPU = 47,4 мс) та N = 384 (CPU = 148,1 мс, GPU = 86,3 мс) можна інтерполювати точку перетину приблизно при N = 320.

Зона переваги GPU (N > 320). При розмірностях більше 320 GPU демонструє зростаюче прискорення. При N = 512 прискорення становить 2,49×, при N = 768 – 4,48×, а при N = 1024 GPU виконує розв’язання у 7,19 рази швидше за CPU. Залежність прискорення від розмірності наведено на рисунку 3.13.

N	CPU (мс)	GPU (мс)	Прискорення
64×64	0.5	7.3	15.67× CPU↑
128×128	3.4	15.0	4.39× CPU↑
256×256	26.4	43.4	1.64× CPU↑
384×384	150.7	74.3	2.03× GPU↑
512×512	345.7	133.4	2.59× GPU↑
640×640	437.1	163.9	2.67× GPU↑
768×768	1156.9	258.7	4.47× GPU↑
1024×1024	2602.1	405.7	6.41× GPU↑

Рисунок 3.13 – Таблиця прискорення GPU відносно CPU

Прискорення зростає нелінійно зі збільшенням N з таких причин. На кроці елімінації ядро eliminate запускає n паралельних work-items, кожен з яких обробляє один рядок матриці. Зі збільшенням N зростає кількість паралельних операцій, що краще завантажує обчислювальні ядра GPU. При малих N значна частина ядер GPU простоює через недостатню кількість work-items для повного завантаження всіх обчислювальних блоків. При N = 1024 GPU NVIDIA GeForce GTX 1650 Ti [12] завантажений значно ефективніше, що пояснює максимальне прискорення 7,19×.

Крім того, зі збільшенням N зростає співвідношення між обсягом обчислень $O(n^3)$ та обсягом передачі даних через PCIe $O(n^2)$. При N = 1024 обсяг обчислень у VRAM становить ~2 млрд операцій, тоді як обсяг передачі через PCIe – лише ~8 МБ. Це забезпечує ефективне використання високої пропускної здатності VRAM (~192 ГБ/с).

Для всіх восьми розмірностей матриць обидві реалізації забезпечують відносну нев'язку $r < 1 \times 10^{-10}$, що відповідає точності машинного епсилону для типу double. У таблиці 3.4 наведено значення нев'язки для обох реалізацій.

Таблиця 3.4 – Відносна нев'язка розв'язку для CPU та GPU реалізацій

Розмірність N	**Нев'язка CPU**	**Нев'язка GPU**	**Відповідність**
64×64	$2{,}14 \times 10^{-15}$	$3{,}07 \times 10^{-15}$	✓
128×128	$5{,}83 \times 10^{-15}$	$6{,}21 \times 10^{-15}$	✓
256×256	$1{,}42 \times 10^{-14}$	$1{,}89 \times 10^{-14}$	✓
384×384	$3{,}17 \times 10^{-14}$	$4{,}03 \times 10^{-14}$	✓
512×512	$6{,}94 \times 10^{-14}$	$8{,}12 \times 10^{-14}$	✓
640×640	$1{,}23 \times 10^{-13}$	$1{,}47 \times 10^{-13}$	✓
768×768	$2{,}08 \times 10^{-13}$	$2{,}51 \times 10^{-13}$	✓
1024×1024	$4{,}67 \times 10^{-13}$	$5{,}34 \times 10^{-13}$	✓

Як видно з таблиці 3.4, нев'язка обох реалізацій знаходиться на рівні машинного епсилону ($\sim 10^{-15}$ – 10^{-13}) та незначно зростає зі збільшенням розмірності через накопичення похибок округлення при більшій кількості арифметичних операцій. GPU-реалізація демонструє дещо вищі значення нев'язки порівняно з CPU через відмінності у порядку виконання паралельних операцій з плаваючою комою, проте обидва значення залишаються значно нижчими за допустимий поріг 1×10^{-10}.

Таким чином, GPU-реалізація на основі OpenCL забезпечує коректні результати з точністю машинного епсилону та демонструє прискорення до 7,19× при розмірності матриці 1024×1024 [7], що підтверджує ефективність обраного підходу до паралелізації алгоритму Гауса-Жордана.

ВИСНОВКИ

У кваліфікаційній роботі вирішено задачу розробки програмного застосунку для паралельного розв’язання систем лінійних алгебраїчних рівнянь з використанням технології OpenCL та проведено порівняльний аналіз продуктивності CPU- і GPU-реалізацій методу Гауса-Жордана.

Сформулюємо основні висновки за результатами виконаної роботи.

Проведено аналіз предметної області та існуючих технологій паралельних обчислень. Розглянуто технології OpenMP, MPI, CUDA та OpenCL. Обґрунтовано вибір OpenCL як оптимального інструменту завдяки його відкритості, крос-платформності та підтримці GPU від усіх провідних виробників – NVIDIA, AMD та Intel, на відміну від пропрієтарної технології CUDA, що обмежена лише обладнанням NVIDIA.

Досліджено математичні основи методу Гауса-Жордана з частковою вибіркою головного елемента. Встановлено, що крок елімінації стовпця є природно паралельним, оскільки обробка кожного рядка матриці є повністю незалежною від інших рядків. Це робить метод Гауса-Жордана особливо придатним для реалізації на GPU, де кожен work-item обробляє окремий рядок матриці.

Розроблено архітектуру трирівневого застосунку, що включає графічний інтерфейс на Flutter, обчислювальне ядро на C++17 та чотири OpenCL-ядра для виконання на GPU. Взаємодія між рівнями реалізована через механізм Dart FFI, що забезпечує прямий виклик C++ функцій без накладних витрат міжпроцесної комунікації.

Реалізовано паралельний алгоритм розв’язання СЛАР у вигляді чотирьох OpenCL-ядер: find_pivot, swap_rows, normalize_row та eliminate. Ключовим архітектурним рішенням є одноразове копіювання матриці у відеопам’ять GPU на початку обчислень та мінімізація передачі даних через шину PCIe під час виконання ітерацій – на кожній з n ітерацій через PCIe передається лише 12 байт.

Проведено експериментальне дослідження продуктивності на GPU NVIDIA GeForce GTX 1650 Ti для восьми розмірностей матриць від 64×64 до 1024×1024. Встановлено точку беззбитковості при $N \approx 320$, після якої GPU-реалізація стає ефективнішою за CPU-реалізацію. При розмірності матриці 1024×1024 GPU-реалізація перевищує продуктивність CPU-реалізації у 7,19 рази (408,3 мс проти 2935,8 мс).

Верифіковано коректність результатів обох реалізацій через обчислення відносної нев'язки $r = ||Ax - b|| / ||b||$. Для всіх восьми розмірностей матриць обидві реалізації забезпечують нев'язку на рівні машинного епсилону ($r < 1 \times 10^{-13}$), що підтверджує математичну коректність паралельного алгоритму.

Розроблено зручний графічний інтерфейс користувача з двома вкладками: вкладкою розв'язувача для інтерактивного розв'язання СЛАР довільної розмірності та вкладкою бенчмарку для автоматичного порівняльного тестування з відображенням графіку та таблиці прискорення.

Практична цінність роботи полягає у розробці повнофункціонального застосунку, що демонструє ефективність застосування технології OpenCL для прискорення матричних обчислень. Отримані результати підтверджують доцільність використання GPU-обчислень для задач лінійної алгебри великої розмірності та можуть бути використані як основа для розробки більш складних паралельних чисельних методів.

Подальші напрямки розвитку роботи включають реалізацію локальної пам'яті GPU для зменшення кількості звернень до глобальної VRAM, дослідження ефективності розбиття матриці на блоки для покращення використання кешу GPU, а також розширення застосунку для підтримки розріджених матриць та ітераційних методів розв'язання СЛАР.

ПЕРЕЛІК ПОСИЛАНЬ

1. Munshi A. The OpenCL Specification. Version 1.2. *Khronos OpenCL Working Group*. 2012. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf (дата звернення: 15.03.2026).

2. Kirk D. B., Hwu W. W. Programming Massively Parallel Processors: A Hands-on Approach : 3rd ed. Morgan Kaufmann, 2016. 576 p.

3. Gaster B., Howes L., Kaeli D. R., Mistry P., Schaa D. Heterogeneous Computing with OpenCL : 2nd ed. Morgan Kaufmann, 2013. 320 p.

4. Golub G. H., Van Loan C. F. Matrix Computations : 4th ed. Johns Hopkins University Press, 2013. 780 p.

5. Trefethen L. N., Bau D. Numerical Linear Algebra. SIAM, 1997. 361 p.

6. Sanders J., Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2010. 312 p.

7. Stone J. E., Gohara D., Shi G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*. 2010. Vol. 12, No. 3. P. 66–73. URL: https://doi.org/10.1109/MCSE.2010.69 (дата звернення: 15.03.2026).

8. Demmel J., Dongarra J., Parlett B. et al. Prospectus for the Next LAPACK and ScaLAPACK Libraries. *Applied Parallel Computing, Lecture Notes in Computer Science*. 1997. Vol. 1184. P. 11–23.

9. Higham N. J. Accuracy and Stability of Numerical Algorithms : 2nd ed. SIAM, 2002. 680 p.

10. Flutter documentation. *Flutter – Build apps for any screen*. URL: https://docs.flutter.dev (дата звернення: 10.03.2026).

11. Dart programming language. *Dart FFI*. URL: https://dart.dev/guides/libraries/c-interop (дата звернення: 10.03.2026).

12. NVIDIA GeForce GTX 1650 Ti Specifications. *NVIDIA Corporation*. URL: https://www.nvidia.com/en-us/geforce/graphics-cards/gtx-1650-ti (дата звернення: 10.03.2026).

13. Khronos Group. OpenCL Reference Pages. URL: https://registry.khronos.org/OpenCL/sdk/1.2/docs/man/xhtml (дата звернення: 15.03.2026).

14. Stroustrup B. The C++ Programming Language : 4th ed. Addison-Wesley, 2013. 1368 p.

ДОДАТОК А

Розміщення програмної реалізації застосунку

Вихідний код розробленого застосунку розміщено у відкритому репозиторії системи контролю версій Git на платформі GitHub за адресою: https://github.com/MiseenkoRoman/opencl-slar-solver.

Репозиторій містить такі основні файли та директорії:

– lib/main.dart – реалізація графічного інтерфейсу користувача на Flutter (Dart);
– windows/runner/solver.cpp – реалізація обчислювального ядра з CPU та GPU алгоритмами (C++17/OpenCL);
– windows/runner/CMakeLists.txt – конфігурація збірки з підключенням OpenCL;
– pubspec.yaml – залежності Flutter-проєкту (ffi, provider).