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

РОЗРОБКА QR АЛГОРИТМУ РОЗВ'ЯЗАННЯ СИСТЕМ ЛІНІЙНИХ РІВНЯНЬ НА СУПЕРКОМП'ЮТЕРІ ОСНАЩЕНОМУ GPU

Текстова частина
магістерської роботи
за спеціальністю „Інженерія програмного забезпечення" 121

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

____
(підпис)

"____" _________ 202_ р.

Виконав студент
Миронюк Тарас Андрійович

"____" _________ 202_ р.

Київ 2025

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

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики
к.н., доц. Гороховський С.С.
____
(підпис)
“_____” ___________202_ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р.н. магістерської програми Інженерія Програмного Забезпечення Миронюку Тарасу Андрійовичу

Дослідити Застосування QR алгоритмів для розв’язання систем лінійних рівнянь на суперкомп’ютері оснащеному GPU

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1 Аналіз предметної області.
2 Опис розробленого алгоритму та програмного забезпечення.
3 Оцінка розробленого QR алгоритму.
Висновки по роботі та рекомендації для подальших досліджень
Список літератури
Додатки

Дата видачі “_____” __________ 202_ р.

Керівник
Г.І. Малашонок, доктор фіз-мат наук, професор

____
(підпис)

Завдання отримав
Т.А. Миронюк ______________________
(підпис)

Тема: Розробка QR алгоритму розв'язання систем лінійних рівнянь на суперкомп'ютері, оснащеному GPU

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	31.10.2024	
2.	Огляд технічної літератури за темою роботи	01.11.2024-01.12.2024	
3.	Ознайомлення з технічною документацією CUDA	02.12.2024-02.01.2025	
4.	Ознайомлення з технічною документацією DAP	03.01.2025-03.02.2025	
5.	Огляд паралельної реалізації алгоритмів QR розкладу	04.02.2025-04.02.2025	
6.	Вивчення можливостей HIP API	06.03.2025-14.03.2025	
7.	Створення примітивного рішення на одному CPU GPU	15.03.2025-01.04.2025	
8.	Вибір методів інтеграції алгоритму в DAP	02.04.2025-09.04.2025	
9.	Інтеграція алгоритму з DAP	10.04.2025-28.06.2025	
11.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи	29.06.2025	
12.	Коригування роботи за результатами попереднього захисту	30.06.2025-06.06.2025	
14.	Захист магістерської роботи (проекту)	09.06.2025-10.06.2025	

Студент Миронюк Т.А.

Керівник Малашонок Г.І. “____”________________ 202_ р.

ЗМІСТ

Анотація 6

Вступ... 7

Актуальність теми та її практична значущість 7

Постановка задачі... 8

Розділ 1. Аналіз предметної області.. 10

1.1 Прямі методи розв'язання систем лінійних рівнянь 10

1.2. Бібліотеки лінійної алгебри для CPU 13

1.2.1. Базові підпрограми лінійної алгебри (BLAS) 13

1.2.2. LAPACK (Linear Algebra PACKage) 15

1.3 Архітектура та програмування GPU 16

1.3.1. Архітектура сучасних GPU 16

1.3.1. API для програмування GPU... 17

1.4 Версії BLAS з підтримкою прискорення від GPU 18

1.4.1. Бібліотеки NVIDIA для GPU.. 19

1.4.3. Інші реалізації BLAS із прискоренням на GPU 21

1.5. Бібліотеки для прискореного розв'язування СЛР............................ 22

1.5.1. Бібліотеки від виробників GPU 22

1.5.2. Інші спеціалізовані пакети 23

1.6. Розподілені обчислення та MPI 26

1.6.1. Message Passing Interface (MPI) 26

1.6.2. Технологія динамічного розпаралелювання DAP 27

1.6.3. Швидкий рекурсивний QR розклад 29

Розділ 2. Опис розробленого програмного забезпечення 30

2.1 Архітектура та ключові рішення бібліотеки QrGPU 31

2.2. Інтеграція бібліотеки QrGPU із DAP 37

Розділ 3. Експерементальні дослідження та аналіз результатів 42

3.1. Конфігурація тестового середовища 42

3.2. Методологія тестування 42

3.2.1. Генерація тестових даних 42

3.2.2. Сценарії тестування 43

3.2.3. Метрики оцінки 43

3.3. Аналіз продуктивності QrGPU на одному вузлі 43

3.3.1. Обмеження відеопам’яті та максимальна розмірність 43

3.4. Масштабування алгоритму з використанням технології DAP 44

3.4.1. Порівняння продуктивності 45

3.3.2. Вплив оптимізації (перенесення множення матриць на GPU). 46

3.4.2. Аналіз масштабованості та подолання апаратних обмежень ... 47

3.5. Аналіз результатів 47

висновки 50

Список використаних джерел 52

Додаток А 55

АНОТАЦІЯ

Дипломна робота присвячена розробці масштабованого та портативного QR-алгоритму для розв'язання великих систем лінійних рівнянь з розрідженими матрицями на гетерогенних обчислювальних архітектурах із GPU та розподіленою пам'яттю.

У першому розділі роботи проведено аналіз предметної області, включаючи прямі методи розв'язання систем лінійних рівнянь, огляд бібліотек лінійної алгебри (BLAS, LAPACK, cuSOLVER, hipSOLVER), архітектурні особливості GPU та API для їх програмування. Також розглянуто принципи розподілених обчислень та технологію DAP.

Другий розділ присвячений опису розробленого програмного забезпечення. Детально висвітлено архітектуру та ключові рішення бібліотеки QrGPU, яка реалізує QR-розклад та матричне множення розріджених матриць на GPU з використанням CUDA API та підтримкою платформ NVIDIA та AMD через HIP. Описано також інтеграцію бібліотеки QrGPU із системою DAP за допомогою JNI інтерфейсу.

Третій розділ містить результати експериментального дослідження та аналіз продуктивності розробленого алгоритму. Визначено оптимальний розмір даних для відвантаження на GPU, продемонстровано подолання обмежень відеопам'яті завдяки інтеграції з DAP, що дозволило обробляти матриці значно більших розмірів. Проаналізовано джерела накладних витрат та вплив QR розкладу на структуру розрідженості матриць.

У висновках узагальнено результати роботи, підкреслено високу масштабованість розробленого розподіленого QR алгоритму та його перспективність для розв'язання великомасштабних систем лінійних рівнянь на сучасних суперкомп'ютерах. Запропоновано напрямки подальшого розвитку, зокрема, адаптацію до реальних кластерів та відмову від JNI на користь прямої взаємодії через MPI.

ВСТУП

Актуальність теми та її практична значущість

Розв'язування систем лінійних рівнянь є фундаментальною проблемою у чисельній лінійній алгебрі, із застосуванням у різних галузях, таких як моделювання фізичних процесів[1] , астрономія [2], енергетика [3], комп'ютерна графіка, машинне навчання, криптографія та багато інших. Однак, із збільшенням розміру системи (тобто кількості рівнянь і змінних) також збільшується обчислювальна складність її розв'язання. Це робить розв'язання систем лінійних рівнянь великого масштабу високовитратним завданням.

Традиційно, такі системи розв'язувалися за допомогою центральних процесорів (CPU). Однак, з появою графічних процесорів (GPU) з уніфікованою архітектурою став очевидним їх потенціал для значного прискорення обчислень. GPU перетворилися зі спеціалізованих пристроїв для рендерингу відеоігор в потужні універсальні процесори загального призначення. Вони надають велику обчислювальну потужність та пропускну здатність пам'яті, що робить їх привабливими для високовитратних завдань.

У останні роки спостерігається значний приріст потужності графічних процесорів, переважно через зростання інтересу до та розвитку штучного інтелекту (AI) та машинного навчання (ML). Ці галузі вимагають значної обчислювальної потужності, а GPU виявилися дуже ефективними для цих завдань. Це призвело до швидкого розвитку та покращення архітектур GPU, з великими інвестиціями виробників у розробку все потужніших та ефективніших GPU. Цей стрибок в потужності GPU відкриває унікальну можливість для розробки нових алгоритмів та програмних пакетів, оптимізованих для цих передових GPU.

Незважаючи на потенційні переваги, використання GPU для розв'язання систем лінійних рівнянь не є простим. Архітектура GPU значно відрізняється від

традиційних CPU. Алгоритми, які ефективні на CPU, можуть бути не ефективними на GPU, і навпаки. Це пояснюється тим, що CPU орієнтовані на послідовне виконання складних операцій, тоді як GPU спеціалізуються на паралельному виконанні великої кількості простіших операцій. Також, у традиційних існуючих реалізаціях, GPU швидше за CPU для СЛР невеликого порядку (до кількох тисяч). Однак, для великих систем CPU демонструє вищу продуктивність [3]. Тому, є потреба у розробці нових алгоритмів або адаптації існуючих для ефективного використання обчислювальної потужності GPU.

Постановка задачі

Обмежуючись полем дійсних чисел, систему лінійних рівнянь з n невідомими можна записати у матричній формі як

$$Ax = b,$$

де $A \in R^{n \times n}$ - квадратна матриця коефіцієнтів, $x \in R^n$ - вектор невідомих, а $b \in R^n$ - вектор вільних членів. Задача полягає у знаходженні такого вектора x, що задовольняє рівняння.

У межах даного дослідження розглядається задача ефективного розв'язання систем лінійних рівнянь з великими розрідженими матрицями коефіцієнтів. Особливу увагу приділено тим випадкам, коли розмірність задачі перевищує можливості обробки в межах пам'яті одного графічного процесора (GPU), що робить недоцільним використання стандартних бібліотечних рішень.

Метою роботи є розробка та оптимізація QR-алгоритму для розв'язання систем лінійних рівнянь на суперкомп'ютерах з багатопроцесорною архітектурою та розподіленою пам'яттю, які оснащені GPU. Це дозволить розв'язувати задачі значно більшого масштабу, забезпечуючи при цьому ефективне використання апаратних ресурсів.

У рамках поставленої задачі передбачається:

- Провести аналіз архітектурних особливостей GPU, що впливають на продуктивність чисельних методів, зокрема QR-розкладу.
- Адаптувати QR-алгоритм для ефективного виконання на GPU, з урахуванням паралельності, оптимізації доступу до пам’яті та мінімізації витрат на передавання даних між CPU і GPU.
- Розробити методи розподілу даних і обчислень для розріджених матриць у середовищі з розподіленою пам’яттю, що дозволить працювати з обсягами даних, які перевищують можливості одного GPU.
- Реалізувати програмну версію QR-алгоритму, здатну ефективно функціонувати на багатопроцесорних системах із GPU.
- Провести експериментальне дослідження розробленого алгоритму, порівнюючи його продуктивність з традиційними CPU-орієнтованими підходами та існуючими GPU-рішеннями, з акцентом на масштабованість і ефективність при роботі з великими розрідженими матрицями.

Таким чином, дослідження спрямоване на створення масштабованого і продуктивного інструменту для розв’язання великих систем лінійних рівнянь, що враховує специфіку сучасних обчислювальних архітектур з GPU та розподіленою пам’яттю.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Прямі методи розв'язання систем лінійних рівнянь

Прямі методи є основними для розв'язання систем лінійних рівнянь у чисельних обчисленнях. Ці підходи зводять початкову систему до вигляду, який дозволяє знайти її розв'язок за скінченну кількість кроків. Тому, всі прямі методи складаються із прямого та зворотного ходу. Зазвичай зворотний хід є тривіальним, і зосереджуються на оптимізації прямого ходу. Далі буде надано короткий огляд розповсюджених прямих методів розв'язання СЛР: методів, що базуються на QR-, LU- розкладах, розкладі Холецького, а також методу Гауса та його варіацій для розріджених матриць - фронтального та мультифронтального методів. Ми порівняємо їхні особливості, переваги й обмеження, зосередившись на застосуванні до розріджених систем великої розмірності. Також обґрунтуємо вибір QR-розкладу як оптимального для цієї роботи.

Метод Гауса

Метод запропонований Карлом Фрідріхом Гаусом у XIX столітті, як алгоритм для приведення матриці до трикутного вигляду через елементарні операції над рядками. Основна мета — послідовне виключення змінних для отримання зворотно-підставної форми.

Метод широко застосовується як базовий алгоритм для побудови LU-розкладу, визначення рангу та обчислення оберненої матриці. Обчислювальна складність становить $O(n^3)$, що обмежує ефективність для великих щільних систем. Проте, його основною проблемою є накопичення похибки округлення під час прямого ходу, що суттєво знижує точність розв'язку, особливо для погано обумовлених систем (цьому можна запобігти ефективним вибором стратегій вибору ведучого елемента (pivoting) [4]). Для розріджених матриць без модифікацій метод призводить до значного заповнення раніше нульових

елементів (fill-in), тому для розріджених систем використовують модифікацію методу Гауса – фронтальний метод.

LU-розклад

LU-розклад полягає в представленні матриці A як добутку нижньотрикутної матриці L та верхньотрикутної U:

$$A = LU$$

Цей розклад ефективний для щільних квадратних матриць, де дозволяє швидко розв'язувати системи рівнянь шляхом прямої та зворотної підстановки. Проте він є нестійким для вироджених або близьких до вироджених матриць. Для покращення стійкості застосовують *LUP-розклад* із перестановками рядків.

У випадку розріджених матриць LU-розклад безперечно поступається QR-методу через вищу чутливість до заповнення та гіршу числову стійкість. Для отримання LU-розкладу матриці часто використовується Метод Гауса.

Розклад Холєцького

Метод розкладу симетричних позитивно визначених матриць. Розклад має вигляд:

$$A = LDL^T,$$

Де L - нижньотрикутна, а D - блочно-діагональнf матриця.

На практиці, матриця A попередньо масштабується для покращення числової стійкості. Це також зменшує щільність матриці L:

$$\bar{A} = PSASP^T,$$

Такий алгоритм щонайменше удвічі ефективніший за LU. Проте, область його застосування обмежується симетричними додатно-визначеними матрицями.

Метод надзвичайно популярний у задачах моделювання фізичних процесів та в статистичних обчисленнях (наприклад, у фільтрі Калмана).

Мультифронтальний метод

Мультифронтальний метод [5] це ефективний паралельний спосіб розв’язання великих розріджених симетричних додатно-визначених систем, що був розроблений Ієном Даффом і Джоном Рідом в 1983 році і з того часу широко використовується в багатьох наукових і інженерних застосуваннях [6]. Метод розбиває LU-розклад, або розклад Холецького в послідовність часткових розкладів щільніших і менших підматриць (фронтальних матриць), до яких вже можна ефективно застосовувати швидкі матричні алгоритми, такі як операції BLAS.

Метод активно використовується в бібліотеках MA57/HSL, STRUMPACK, MUMPS. Мультифронтальний метод досягає кращої ефективності враховуючи особливості розріджених матриць, та завдяки високому рівню паралелізму в його реалізації. Проте, його застосування, як і в LU та LDL^T, обмежене квадратними симетричними додатно-визначеними матрицями.

QR-розклад

QR-розклад був запропонований Алстоном Хаусгольдером, як чисельно стійка альтернатива LU. Він полягає в розкладанні заданої матриці A на добуток ортогональної матриці Q та верхньотрикутної матриці R так, що:

$$A = QR,$$

де матриця Q ортогональна, тобто її транспонована матриця також є її оберненою ($Q^T = Q^{-1}$), а матриця R верхньотрикутна (всі елементи нижче головної діагоналі дорівнюють нулю). Метод вирізняється числовою стійкістю (через ортогональність матриці відбиття Хаузхолдера). Це робить його бажаним методом для розв'язання систем лінійних рівнянь у багатьох практичних

застосуваннях, особливо коли матриця коефіцієнтів A майже сингулярна або погано умовлена. Також QR можливо застосувати до прямокутних матриць.

Основний недолік - обчислювальна вартість (часто вища за LU), а також збільшення щільності у розріджених випадках. Проте сучасні реалізації QR з використанням відбиття Хаузхолдера добре розпаралелюються. Саме тому QR-метод часто є кращим вибором у високопродуктивних обчисленнях, зокрема він використовується для розв’язання СЛР бібліотеками cuSOLVER, ScaLAPACK, MAGMA.

Висновки

Таким чином, попри ефективність LU, Холєцького та мультифронтальних методів у певних класах задач, QR-розклад обрано у цій роботі через його чисельну стабільність, здатність працювати з прямокутними матрицями, та наявність ефективних паралельних реалізацій, зокрема на графічних процесорах.

1.2. Бібліотеки лінійної алгебри для CPU

У чисельних методах надзвичайно важливе ефективне виконання векторних та матричних операцій. Бібліотеки BLAS (Basic Linear Algebra Subprograms) та LAPACK (Linear Algebra PACKage) стали де-факто стандартами для цих операцій.

1.2.1. Базові підпрограми лінійної алгебри (BLAS)

BLAS (Basic Linear Algebra Subprograms) — це специфікація (API), що визначає набір стандартизованих низькорівневих процедур для виконання базових операцій лінійної алгебри [7]. Операції BLAS класифікуються на три рівні, виходячи з обчислювальної складності та типів операндів:

- BLAS Level 1: Векторні операції із лінійною складністю. Приклади включають масштабування векторів, додавання, векторний та скалярний добуток, та ін.
- BLAS Level 2: Матрично-векторні операції, такі як множення матриці на вектор, складності $O(n^2)$.
- BLAS Level 2: Матрично-матричні операції, як множення матриць, зі складністю $O(n^3)$.

Широка популярність BLAS зумовлена високою ефективністю та стандартизованим інтерфейсом. Реалізації BLAS високо оптимізовані для різноманітних архітектур з метою досягнення максимальної продуктивності. Стандартизований інтерфейс також значно покращує сумісність та портативність обчислювальних бібліотек. Якщо програмний пакет написаний з використанням інтерфейсу BLAS, його можна використовувати на будь-якій машині, для якої існує та встановлено реалізацію BLAS.

Існує кілька високооптимізованих CPU-реалізацій BLAS, адаптованих для різних типів обладнання:

- Netlib BLAS: Базова, вільно доступна реалізація, написана на Fortran.
- ATLAS (Automatically Tuned Linear Algebra Software): Відкрите програмне забезпечення, створене для автоматичної генерації оптимізованих версій процедур BLAS під дане апаратне забезпечення.
- Intel MKL (Math Kernel Library): Бібліотека оптимізованих математичних процедур для наукових, інженерних та фінансових застосувань. Основні математичні функції в MKL оптимізовані спеціально для процесорів Intel.
- OpenBLAS: Відкрита оптимізована бібліотека BLAS, заснована на GotoBLAS2.

Завдяки BLAS розробники можуть досягти високої продуктивності своїх застосунків, не вдаючись до низькорівневої оптимізації операцій лінійної алгебри.

1.2.2. LAPACK (Linear Algebra PACKage)

LAPACK (Linear Algebra PACKage) - це повнофункціональна бібліотека, яка надає процедури для розв'язання більшості типових задач лінійної алгебри, як розв'язання систем лінійних рівнянь, знаходження власних значень та векторів, найменших квадратів, сингулярного розкладу, різних матричних розкладів (LU, Cholesky, QR), та ін. [8] LAPACK значною мірою спирається на ефективні реалізації BLAS для виконання низькорівневих обчислень.

Хоча бібліотека LAPACK не була початково задумана як API в тому ж сенсі, що й BLAS, її універсальний та добре структурований інтерфейс став де-факто стандартом. Багато інших бібліотек і пакетів, включно з реалізаціями для GPU та розподілених систем, перейняли її інтерфейс та структуру процедур, що значно полегшує перенесення коду між різними обчислювальними платформами.

Основні переваги LAPACK:

- Широкий спектр функцій: Охоплює практично всі стандартні задачі лінійної алгебри.
- Висока продуктивність: Завдяки використанню оптимізованих реалізацій BLAS.
- Надійність: Алгоритми LAPACK є перевіреними та стабільними.
- Портативність: Код, написаний з використанням LAPACK, легко переноситься на різні системи, де доступні її реалізації.

Існують різні реалізації LAPACK, зокрема та, що підтримується самим проєктом Netlib (Netlib LAPACK), а також інтегровані версії в таких пакетах, як Intel MKL та OpenBLAS.

1.3 Архітектура та програмування GPU

Сучасні графічні процесори (GPU) — це високопаралельні процесори, спеціально розроблені для ефективного рендеру та виконання загальних обчислень (General-Purpose computing on Graphics Processing Units, GPGPU). На відміну від центральних процесорів, їхня архітектура характеризується наявністю великої кількості спеціалізованих обчислювальних блоків та ієрархічною системою пам'яті, що дозволяє досягати надзвичайної пропускної здатності для паралельних задач.

1.3.1. Архітектура сучасних GPU

Основою GPU є обчислювальні блоки - процесори, здатні виконувати інструкції незалежно. GPU мають ієрархічну систему пам'яті, яка включає *глобальну*, *спільну*, *локальну* та *постійну* пам'ять. Глобальна пам'ять є найбільшою та найповільнішою, тоді як спільна та локальна пам'ять мають значно менший об'єм, але забезпечують набагато швидший доступ. Для прискорення доступу до даних графічні процесори також активно використовують кешування на різних рівнях.

Хоча більшість процесорів GPU призначені для рендеру, для загальних обчислень використовуються тільки основні обчислювальні процесори (шейдерні процесори). Тому подальший огляд зосередиться саме на них та відмінностях в їх архітектурі у провідних виробників.

Архітектура Nvidia

Nvidia є домінуючим виробником дискретних GPU, займаючи значну частку ринку. Сучасні архітектури Nvidia використовують шейдерні процесори CUDA, також відомі як CUDA-ядра (хоча в технічному сенсі, вони не є самостійними ядрами). Вони, разом з іншими типами процесорів, згруповані в потокові мультипроцесори (SM). Кожен SM має спільний кеш інструкцій, планувальник, блоки диспетчеризації та файл регістрів. У сучасних архітектурах,

починаючи з Turing, потоковий мультипроцесор також містить певну кількість тензорних процесорів, призначених для апаратного виконання операції множення матриць 4-байтних типів даних, що потрібно для виконання завдань машинного навчання в реальному часі, проте має низьку точність.

Архітектура AMD

AMD є іншим значним виробником GPU, із їх актуальною архітектурою RDNA. У цих процесорах використовуються обчислювальні блоки (Compute Units, CU), об’єднані у шейдерні процесори. Кожен CU, у свою чергу, містить кілька потокових процесорів і має власні кеші інструкцій та скалярних даних.

1.3.1. API для програмування GPU

Щоб спростити роботу із різноманітними архітектурами графічних процесорів, для написання програм використовуються стандартизовані API.

CUDA

CUDA є де-факто стандартом API для програмування GPU. Це власний API, а також розширення мови C++, призначені для програмування GPU від NVIDIA. CUDA можна використовувати двома основними способами: через програмний API, що надає набір C-подібних модулів та процедур, або через API драйвера, який забезпечує більший контроль над апаратним забезпеченням, але вимагає більше зусиль від програміста. Оскільки CUDA розроблена тією ж компанією, що виробляє апаратне забезпечення, на якому вона виконується, можна очікувати кращої відповідності обчислювальним характеристикам GPU, ширшого доступу до функцій пристрою та кращої продуктивності.

Альтернативні API та портативність: OpenCL та ROCm / HIP

Хоча CUDA є домінуючим API для NVIDIA GPU, існують інші важливі платформи, що забезпечують ширшу сумісність або альтернативні шляхи розробки.

OpenCL (Open Computing Language): Розроблений Khronos Group, OpenCL є відкритим стандартом для паралельного програмування з використанням CPU, GPU та інших типів процесорів. Він забезпечує кросплатформенну сумісність, дозволяючи коду працювати на пристроях від різних постачальників, що робить його гнучким вибором для застосунків, які мають функціонувати на різноманітних апаратних платформах.

ROCm (Radeon Open Compute platform) та HIP: ROCm - це відкрита програмна платформа від AMD, яка надає повний стек програмного забезпечення для програмування їхніх GPU. В рамках цієї екосистеми, HIP (Heterogeneous-Compute Interface for Portability) виступає як ключовий програмний інтерфейс та модель програмування. HIP розроблений з акцентом на портативність, дозволяючи розробникам писати код, що може бути скомпільований та виконуватися як для NVIDIA (використовуючи CUDA бекенд), так і для AMD (використовуючи ROCm бекенд). Таким чином, HIP функціонує як міст між двома основними екосистемами, забезпечуючи розробку по-справжньому портативних високопродуктивних застосунків.

Для спрощення міграції існуючих CUDA-застосунків на HIP, AMD надає HIPify pipeline [9]. Це інтерпретатор, що, в теорії, автоматично перекладає код із використанням CUDA API та синтаксису, на еквівалентний код HIP. Цей процес значно мінімізує ручні зміни коду, дозволяючи розробникам швидко адаптувати свої проєкти для роботи на платформах AMD ROCm, зберігаючи при цьому сумісність з NVIDIA CUDA. Така можливість є критично важливою для функціонування екосистеми ROCm, оскільки вже десятиліттями програми для GPU писалися переважно для CUDA.

1.4 Версії BLAS з підтримкою прискорення від GPU

Стандартні операції лінійної алгебри, визначені BLAS, були також адаптовані із використанням можливостей прискорення на GPU.

1.4.1. Бібліотеки NVIDIA для GPU

Компанія NVIDIA розробила власні бібліотеки, що використовують архітектуру CUDA для прискорення обчислень лінійної алгебри.

cuBLAS

cuBLAS — це бібліотека високопродуктивних обчислень на GPU, розроблена компанією NVIDIA [10]. Вона є реалізацією специфікації BLAS поверх для середовища виконання CUDA. cuBLAS включає API, що реалізує стандартні API BLAS (Рівні 1, 2, 3) та спеціалізовані API для загального множення матриць (GEMM), з підтримкою оптимізованих функцій злиття, призначених для GPU NVIDIA. Бібліотека також містить розширення для пакетних операцій (batched operations), виконання на кількох GPU та операцій зі змішаною та низькою точністю.

В основу роботи cuBLAS покладено стовпцевий формат зберігання даних (column-major order), що є традиційним для стандартів BLAS та LAPACK, успадкованим від Fortran. Хоча сучасні C-подібні мови програмування, зазвичай використовують рядковий формат (row-major order), cuBLAS надає можливості для роботи з матрицями, що зберігаються у цьому форматі, часто за допомогою параметрів функцій, які вказують на необхідність виконання операції над транспонованою матрицею. Параметр leading dimension (провідний вимір) є важливим для коректного доступу до елементів матриці.

Реалізація цих операцій у cuBLAS здійснюється з урахуванням архітектури GPU для досягнення максимальної продуктивності. Підходи до оптимізації включають використання масового паралелізму на рівні тисяч обчислювальних одиниць (процесорів), ефективне керування ієрархією пам'яті пристрою, а також застосування техніки tiling (розбиття матриць на менші блоки) для підвищення локальності даних. Функції Рівня 3 BLAS, зокрема множення матриць (GEMM), отримують найбільше прискорення на GPU завдяки високому ступеню паралелізму.

cuSPARSE

cuSPARSE - це бібліотека, розроблена для ефективної обробки розріджених матриць на графічних процесорах NVIDIA [11]. На відміну від щільних матриць, де більшість елементів є ненульовими, розріджені матриці містять переважну більшість нульових елементів, що дозволяє використання спеціалізованих методів зберігання для мінімізації обсягу пам'яті. cuSPARSE підтримує декілька поширених форматів збереження розріджених даних, кожен з яких має свої переваги залежно від структури розрідженості та типу виконуваних операцій. До основних підтримуваних форматів належать:

- CSR (Compressed Sparse Row): Стиснений за рядками формат.
- CSC (Compressed Sparse Column): Стиснений за стовпцями формат.
- COO (Coordinate Format): Координатний формат.
- BSR (Block Sparse Row): Блочно-стиснений за рядками формат.
- ELL (ELLPACK): ЕЛЛПАК-формат.

Бібліотека надає широкий спектр матричних операцій, адаптованих для роботи з цими розрідженими форматами, проте не реалізує повністю інтерфейс BLAS Level 3. Натомість, присутні спеціалізовані операції, такі як множення розрідженої матриці на щільний вектор (SpMV) або на щільну матрицю (SpMM), а також складніші функції, включаючи множення двох розріджених матриць (SpGEMM), розв'язання систем лінійних рівнянь з трикутними розрідженими матрицями (SpSV) та різні перетворення форматів.

Бібліотеки AMD ROCm/HIP для GPU

Платформа AMD ROCm також надає власний набір високопродуктивних бібліотек лінійної алгебри, які забезпечують функціонал аналогічний бібліотекам NVIDIA, і можуть використовуватися через інтерфейс HIP.

rocBLAS / hipBLAS

rocBLAS - це реалізація функціоналу BLAS для платформи AMD ROCm, аналогічно cuBLAS у NVIDIA. Доступ до rocBLAS часто здійснюється через шар сумісності hipBLAS, що є частиною екосистеми HIP. hipBLAS дозволяє розробникам, які використовують HIP, викликати BLAS-подібні функції, виклики які потім динамічно спрямовуються до бібліотеки rocBLAS на AMD GPU або до cuBLAS на NVIDIA GPU, забезпечуючи портативність та крос-платформенність коду [13].

rocSPARSE / hipSPARSE

Аналогічно, rocSPARSE є бібліотекою AMD для операцій з розрідженими матрицями на GPU, оптимізованою для архітектур AMD. Доступ до rocSPARSE також можливий через інтерфейс hipSPARSE, що дозволяє писати портативний код для роботи з розрідженими матрицями на різних архітектурах GPU [14].

1.4.3. Інші реалізації BLAS із прискоренням на GPU

Окрім пропрієтарних рішень від NVIDIA та AMD, існують також BLAS-бібліотеки, засновані на відкритому стандарті OpenCL, які пропонують кросплатформенну сумісність:

- clBLAS: Відкрита програмна бібліотека, що реалізує функції BLAS, написані на OpenCL.
- CLBlast: Сучасна, легка, продуктивна та налаштовувана бібліотека OpenCL BLAS, написана на C++11, призначена для широкого спектру пристроїв OpenCL.
- Intel clBLAS: Відкрита реалізація BLAS, що використовує OpenCL, призначена для прискорення математичних операцій за допомогою графічних процесорів Intel.

Хоча ці бібліотеки і надають гнучкість завдяки використанню OpenCL, основний фокус цієї роботи буде зосереджено на рішеннях CUDA та HIP/ROCm

через їх актуальність для високопродуктивних обчислень на відповідних апаратних платформах.

1.5. Бібліотеки для прискореного розв'язування СЛР

Поряд з базовими бібліотеками, що надаються реалізацію стандарту BLAS із апаратним прискоренням, існують більш високорівневі та спеціалізовані бібліотеки для ефективного розв'язування складних систем лінійних рівнянь безпосередньо на GPU.

1.5.1. Бібліотеки від виробників GPU

cuSOLVER - бібліотека від NVIDIA, яка призначена для виконання просунутих операцій лінійної алгебри безпосередньо на графічних процесорах. Вона є частиною інструментарію CUDA. Для розв'язання СЛР, cuSOLVER підтримує різні прямі методи:

cuSOLVER по суті реалізує API LAPACK, надаючи паралельні GPU версії його процедур. Для виконання цих операцій, cuSOLVER активно використовує інші базові бібліотеки CUDA Бібліотека складається з трьох основних компонентів, кожен з яких оптимізований для певного типу матриць:

- cuSolverDN (Dense): Призначений для роботи зі щільними матрицями. Цей компонент широко застосовує функції з cuBLAS.
- cuSolverSP (Sparse): Спеціалізований для розв'язання задач із розрідженими матрицями. Він спирається на cuSPARSE для ефективного зберігання розріджених даних.
- cuSolverRF (Refactorization): Розроблений для сценаріїв, де значення матриці змінюються, але шаблон ненульового заповнення залишається незмінним.

Ці компоненти надають відповідні класи та типи даних для ефективної роботи з різними видами матриць, а також численні допоміжні функції.

Аналогічно NVIDIA cuSOLVER, платформа AMD ROCm надає бібліотеку rocSOLVER. Як і інші бібліотеки ROCm, rocSOLVER може бути доступна через HIP-інтерфейс для забезпечення портативності коду.

1.5.2. Інші спеціалізовані пакети

SuperLU

SuperLU — це бібліотека загального призначення для розв'язання великих розріджених несиметричних систем лінійних рівнянь. Бібліотека написана мовою C, і доступна для програм на C та Fortran. Вона використовує MPI для зв'язку між процесами, OpenMP для виконання на процесорі, CUDA (NVIDIA) і HIP (AMD) для виконання розрахунків на графічних процесорах [15].

Для розв'язку системи SuperLU використовує LU-розкладання ($P_r D_r A D_c P_c = LU$), де D_r та D_c – діагональні матриці, які використовуються для масштабування матриці A, а P_c, P_r – матриці перестановки стовпців/рядків. Для розкладання використовується *метод Гауса*.

SuperLU доступна у трьох різних варіантах:

- SuperLU для однопоточних.
- SuperLU_MT (multi-threaded) для паралельних машин зі спільною пам'яттю.
- SuperLU_DIST (distributed memory) для машин з розподіленою пам'яттю.

Послідовна версія SuperLU використовує алгоритм GEPP – Gaussian Elimination with Partial Pivoting – Метод Гауса з динамічним вибором опорного елементу [15][16]. На відміну від неї, розподілена версія використовує новий алгоритм GESP – Gaussian Elimination with Static Pivoting – що передбачає статичний вибір опорного елементу [17], що поєднує числову стабільність часткового перестановлення з масштабованістю відсутності перестановлення, для балансування точності та ефективності на великій кількості процесорів [15].

Недоліком статичного підходу до вибору опорного елементу є втрата гарантії стійкості алгоритму та зменшення точності, оскільки динамічний підхід в методі Гауса допомагає контролювати зростання елементів. Проте, на машинах з розподіленою пам’яттю доступ до інших елементів матриці є дуже дорогою операцією, через що динамічний пошук опорного елементу не є оптимальним рішенням для такої системи [17].

Для розподілення матриці між процесорами алгоритм використовує підхід графового розділення, застосований до графу $|A| + |A^T|$, для розділення матриці таким чином, зберігає розрідженість і сприяє паралельному розкладання матриці [17][18]. Розділення призводить до утворення так-званого дерева розділу, яке представляє залежності між етапами обчислення. Паралельний алгоритм виконує обхід дерева знизу вгору і на кожному кроці отримує один стовбець матриці L та однин рядок матриці U. Якщо отриманий розв'язок не є достатньо точним, відбувається ітераційна корекція на основі методу ітерацій Річардсона.

MAGMA

MAGMA – Matrix Algebra on GPU and Multi-core Architectures – це набір бібліотек для лінійної алгебри нового покоління для гетерогенних обчислень, що дозволяє програмам повністю використовувати потужність поточних гібридних систем багатоядерних CPU та кластерів GPU для найшвидшого можливого вирішення задач при вказаних енергетичних обмеженнях. MAGMA включає реалізації для багатоядерних ЦП, графічних процесорів NVIDIA або AMD, понад 400 операцій, таких як розкладання матриць, розв’язок систем лінійних рівнянь, а також частину процедур бібліотеки BLAS, значно оптимізованих для графічних процесорів.

STRUMPACK

STRUMPACK – STRUctured Matrix PACKage – бібліотека, написана на мові C++, що надає підпрограми лінійної алгебри та розв'язувачі лінійних систем для розріджених і щільних *лінійних систем з ранговою структурою*.

Для розріджених матриць використовується алгоритм розв'язання великих розріджених лінійних систем запропонований Jianlin Xia [20]. Ми пропонуємо випадкові прямі, які інтегрують випадкові процеси в метод багатофронтальних рішень з ранговою структурою. Використання випадковості високо спрощує різноманітні ключові етапи структурованих розв'язків, де швидкі операції над вузькими матрицями-векторами замінюють традиційні складні операції над густими чи структурованими матрицями. Нові методи значно розширюють гнучкість та ефективність використання структурованих методів у розріджених розв'язках. Ми також розглядаємо різні техніки, такі як деякі методи графів, включення додаткових структур, концепцію зменшених матриць, використання інформації повторного використання та адаптивні схеми. Методи застосовні до різноманітних розріджених матриць із певними ранговими структурами. Зокрема, для матриць, які виникають від дискретизованих математичних задач із факторизацією, яка призводить до густого заповнення з деякими паттернами рангів поза діагоналлю, вартість факторизацій становить приблизно O(n) операцій з плаваючою. Ці розрахунки отримані на основі двох стратегій оптимізації та ідеї розрідженого рангу. Методи особливо корисні для приблизних розв'язків та попередньої умови. Числові тести на дискретизованих рівняннях математичної фізики та більш загальних проблемах використовуються для демонстрації ефективності та точності. Ідеї тут також мають потенціал для узагальнення до безматричних розріджених прямих розв'язників на основі множення матриці на вектор у майбутніх розробках.

У розріджених прямих вирішувачах, заснованих на LU-розкладання, коефіцієнти LU часто також можна добре апроксимувати за допомогою рангово-структурованого стиснення матриці, що призводить до надійних предобуславлювачів. Розріджений розв’язувач у STRUMPACK також можна використовувати як точний прямий розв’язувач, у цьому випадку він функціонує так само, як, наприклад, SuperLU або Superlu_Dist. Прямий розріджений

розв’язувач STRUMPACK забезпечує хорошу продуктивність і масштабованість розподіленої пам’яті, а також *чудову підтримку CUDA*.

SSIDS

SSIDS є пакетом з бібліотеки SPRAL (Sparse Parallel Robust Algorithm Library) – відкритої бібліотеки для алгебраїчних алгоритмів на розріджених матрицях, для мов С та Fortrun [6]. Вона розробляється групою чисельного аналізу при Rutherford Appleton Laboratory у Великій Британії. Пакет SSIDS (Sparse Symmetric Indefinite Direct Solver) призначений для розв’язку розріджених симетричних рівнянь розмірності $n \times n$ за допомогою багатофронтального методу. Бібліотека опціонально підтримує гібридне обчислення з використанням одного або декількох графічних процесорів NVIDIA. У загальному випадку, SSIDS обчислює розріджену декомпозицію Холецького. Також є можливість вткористання масштабованої матриці $\bar{A} = SAS$, де S - діагональна матриця масштабування.

1.6. Розподілені обчислення та MPI

Для розв’язання задач, які неможливо розмістити в оперативній пам'яті чи ефективно обробити на одному обчислювальному вузлі, застосовують розподілені обчислення: обчислювальне навантаження розподіляють між множиною взаємопов'язаних процесів у межах кластера чи суперкомп'ютера. Кожен процес має власну локальну пам'ять, а взаємодія між ними відбувається через повідомленнями. Такий підхід дозволяє масштабувати ресурсомісткі алгоритми, зокрема паралельний QR-розклад для великих матриць, залучаючи потужності десятків чи сотень процесорів.

1.6.1. Message Passing Interface (MPI)

Для вирішення проблем стандартизації в розподілених системах було розроблено інтерфейс передачі повідомлень Message Passing Interface (MPI). Цей

стандартизований API надає засоби для обміну даними між процесами, що виконуються паралельно в розподіленьму середовищах (наприклад, на суперкомп'ютері). Розробку та підтримку специфікації MPI здійснює MPI Forum - консорціум із представників академічних кіл та промисловості.

Стандарт постійно вдосконалюється: від першої версії MPI-1.0 (1994 р.) до нинішнього MPI-5.0 та новіших, які розширюють його функціонал. Актуальна документація доступна на офіційному сайті MPI Forum [21].

У високопродуктивних обчисленнях (HPC) MPI став де-факто стандартом завдяки високій продуктивності та масштабованості. Для нашого паралельного алгоритму QR-розкладу, орієнтованого на роботу з великими матрицями на суперкомп'ютерах, використання MPI критично важливе, оскільки він забезпечує ефективний розподіл даних, координацію обчислень між вузлами та можливість масштабування.

Серед реалізацій стандарту виділяється Open MPI [22]- високопродуктивна відкрита бібліотека, розроблена консорціумом наукових, академічних та промислових організацій. Вона інтегрує найкращі розробки спільноти та надає оптимізовані рішення для розробників та науковців.

1.6.2. Технологія динамічного розпаралелювання DAP

Для ефективного розпаралелювання рекурсивних алгоритмів на кластерах з розподіленою пам'яттю, зокрема для завдань з великими матрицями, розроблена технологія динамічного розпаралелювання під назвою ДАП-схема (Дроп-Амін-Пайн) [23]. Ця технологія є розвитком попередніх підходів, таких як децентралізоване управління DDP, усуваючи їхні обмеження, зокрема простої процесорів в очікуванні завершення обчислень поточного піддерева.

Основні об'єкти ДАП-технології:

- Дроп (Drop) є мінімальною передаваною одиницею обчислень — компактним підграфом алгоритму, що містить одну рекурсивну

вершину (R-вершину) з нелінійною складністю та суміжними нерекурсивними вершинами. Кожен дроп інкапсулює тип завдання, вхідні/вихідні дані та стан виконання.

- Амін (Amine) виступає контейнером стану для всіх дропів на одному рекурсивному рівні, зберігаючи топологію графа, дані рівнів і статуси підзадач.
- Пайн (Pine) це динамічний реєстр активних амінів у пам'яті процесора, де завершені аміни автоматично видаляються. Кожна обчислювальна задача (дроп) має адресу приписки, яка визначається номером процесора, номером аміна в пайні цього процесора та номером дропа в аміні. Ця адреса використовується для повернення результату обчислення.

Система динамічно балансує навантаження через три взаємопов'язані структури: чергу завдань (доступні дропи з глибиною рекурсії), мапу процесорів (відстеження відправлених завдань на дочірніх вузлах) та пул вільних ресурсів. Останній динамічно розподіляється між вузлами, які обробляють завдання з мінімальною глибиною вкладеності, запобігаючи простоям. Кожен вузол використовує два спеціалізовані потоки:

- Обчислювальний (CalcThread) виконує дроп-завдання: створює амін для нелістових задач, обробляє листові дропи послідовно та відправляє результати за pad-адресою.
- Диспетчерський (DispThread) керує комунікацією: контролює порти, приймає завдання/результати, обмінюється даними про вільні процесори та планує розсилку дропів. Для синхронізації диспетчерський потік періодично призупиняється (sleep), гарантуючи ресурси обчислювальному потоку.

Ця архітектура забезпечує ефективне динамічне планування без централізованого контролера, що критично важливо для масштабування на

великих кластерах. Схема застосовна до будь-яких блочно-рекурсивних алгоритмів (як для щільних, так і розріджених матриць), повністю усуваючи простої обчислювальних вузлів.

1.6.3. Швидкий рекурсивний QR розклад

Система DAP надає реалізацію швидкий рекурсивний алгоритму QR розкладу [24]. В основі цього алгоритму лежить використання обертань Гівенса. Основними такими дропами (базовими операціями або рекурсивними викликами, що формують алгоритм) є:

- QR розклад
- QP розклад: Спеціалізована процедура, що по суті є QR розкладом прямокутної матриці A розміром 2n×n. Її ключовою особливістю є те, що нижній блок цієї матриці вже є верхньотрикутною матрицею, що дозволяє застосовувати певні оптимізації під час обчислень.
- Матричне множення
- Множення із додаванням (AB+C): Комбінована операція множення двох матриць з додаванням до результату третьої матриці.

Завдяки своїй блочно-рекурсивній структурі та динамічному управлінню завданнями через DAP-технологію, цей алгоритм QR розкладу є дуже паралельним. Його діаграма виконання, що ілюструє послідовність та паралельність операцій, надана у Додатку А.

РОЗДІЛ 2. ОПИС РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Метою цієї роботи була розробка масштабованого та продуктивного інструменту для розв'язання великих систем лінійних рівнянь, що враховує можливості сучасних обчислювальних архітектур з GPU та розподіленою пам'яттю. Для цього було розробилено версію швидкого рекурсивного QR розкладу, інтегровану з технологією динамічного розпаралелювання DAP, із використанням апаратного прискорення на GPU.

Реалізація цього підходу вимагала подолання низки технічних викликів, пов'язаних із взаємодією між компонентами системи та апаратним забезпеченням. Ключовою проблемою став доступ до GPU безпосередньо з Java-системи DAP. Хоча існують Java-бібліотеки для CUDA (наприклад, JCUDA), додавання такої залежності було неприйнятним через ризик зростання складності, ускладнення розгортання та обмеження портативності.

Натомість, було обрано архітектуру з окремою процедурною бібліотекою на C++17, яка відповідає за всю взаємодію з GPU. Ця бібліотека реалізована як динамічна (shared library), що забезпечує гнучкий доступ до її функцій з JVM через Java Native Interface (JNI). Це рішення є оптимальним, оскільки дозволяє уникнути жорстких залежностей і гармонійно поєднується з GPU API (CUDA, HIP), які самі є динамічними бібліотеками.

Розроблена C++ бібліотека QrGPU (деталі в підрозділі 2.1) надає DAP дві ключові GPU-прискорені функції: матричне множення і QR розклад для розріджених матриць.

Інтеграція працює наступним чином: система DAP, виконуючи рекурсивний QR розклад, досягає "листів" рекурсивного дерева - невеликих завдань, які вже недоцільно (або неефективно) ділити далі [23]. Саме ці завдання передаються на виконання GPU через бібліотеку QrGPU.

Важливою перевагою QrGPU є її портативність. Хоча основна розробка велась з використанням CUDA, налаштуваний CMake-пайплайн з автоматичною трансляцією коду в HIP (Heterogeneous-Compute Interface for Portability) за допомогою інструменту hippify. Це дозволяє компілювати та запускати рішення як на GPU NVIDIA (з CUDA), так і на GPU AMD (з ROCm).

Одним із ключових технічних завдань, яке вирішує QrGPU, є перетворення форматів даних. Система DAP використовує власні формати матриць (MatrixS – розріджені, MatrixD – щільні), які несумісні з форматами JVM, C++ бібліотеки та GPU-бібліотек (наприклад, CSR для розріджених, спеціалізовані щільні формати для cuBLAS/rocBLAS). Бібліотека QrGPU ефективно перетворює вхідні матриці MatrixS з DAP у GPU-сумісні формати (наприклад, CSR для розріджених операцій) та організовує передачу даних. Для QR розкладу, який виконується cuBLAS/rocBLAS (щільні матриці), розріджені матриці додатково конвертуються у щільний формат безпосередньо на GPU. Розріджене матричне множення виконується безпосередньо у CSR-форматі за допомогою cuSPARSE/rocSPARSE. Після виконання обчислень, результати повертаються назад у DAP, з необхідним перетворенням форматів.

2.1 Архітектура та ключові рішення бібліотеки QrGPU

У рамках цієї дипломної роботи було розроблено високопродуктивну бібліотеку QrGPU, призначену для QR-розкладу розріджених матриць на графічних процесорах (GPU). Її головна перевага - крос-платформність: бібліотека однаково ефективно працює як на GPU AMD (через HIP API), так і на NVIDIA (через CUDA API). Для досягнення максимальної швидкодії та контролю над ресурсами ми обрали процедурний стиль програмування на C++17.

Представлення даних

Ядром бібліотеки стала універсальна структура Matrix<T> (див. Лістинг 1). Вона гнучко керує даними як на процесорі (CPU), так і на відеокарті (GPU),

підтримуючи ключові формати: щільний (Dense) та стиснутий рядковий (CSR). Використання std::variant дозволяє без значних накладних витрат перемикатися між чотирма типами сховищ: DenseHostStorage, DenseGpuStorage, CsrHostStorage та CsrGpuStorage.

Лістинг 1-Перемикання типу матриці часу компіляції

```
template<typename T> struct Matrix {
        using Storage = std::variant<
            DenseHostStorage, DenseGpuStorage, CsrHostStorage,
    CsrGpuStorage
        >;
        int64_t rows = 0;
        int64_t cols = 0;
        StorageFormat format;
        Storage data;
    };
```

Управління ресурсами

Незважаючи на обраний процедурний стиль програмування, архітектура бібліотеки QrGPU відходить від парадигми "однієї довгої функції", запроваджуючи інкапсулюючи проміжні стани у структури даних - типи сховища. Інкапсуляція в даному контексті потрібна щоб запобігти розділенню логіки управління ресурсами та станом даних між численними процедурами і функціями. Ці структури дозволяють зібрати воєдино всі елементи, що стосуються певного представлення матриці (наприклад, щільна матриця в пам'яті хоста), в одне логічне ціле. Це рішення є принципово важливим, оскільки дозволяє явно інкапсулювати стан даних матриці (її формат та розташування пам'яті), роблячи код значно чистішим, легшим для розуміння та менш схильним до помилок. Таким чином, такі структури, як DenseHostStorage та CsrGpuStorage,

стають не просто контейнерами, а носіями власності над відповідними ресурсами.

Таким чином, структури типу сховища, стають носіями власності над відповідними ресурсами. Це досягається завдяки принципу Resource Acquisition Is Initialization (RAII), який реалізовано через std::unique_ptr зі спеціалізованими видалячами (deleter-ами) — HipDeleter, CsrSpMatDeleter та DnMatDeleter. Таким чином, при створенні об'єкта однієї з цих структур, він автоматично отримує необхідні ресурси (наприклад, виділяє, або отримує виділену пам'ять на хості або GPU, налаштовує контексти). І навпаки, коли об'єкт виходить за межі області видимості, його деструктор автоматично викликається, забезпечуючи впорядковане та безпечне звільнення цих ресурсів (викликаючи hipFree або cudaFree та очищаючи дескриптори, як-от hipsparseSpMatDescr_t). Цей дисциплінований підхід до управління ресурсами важливий для бібліотек, що працюють з потенційно великими обсягами пам'яті та пристроє-специфічними ресурсами. Він мінімізує ризик витоків пам'яті та значно спрощує обробку помилок, що зрештою сприяє створенню більш надійної та підтримуваної системи. Інноваційність підходу полягає в тому, що, на відміну від низькорівневих GPU API, які не мають вбудованої концепції власності даних користувачем, цю концепцію було спроєктовано та реалізовано на рівні бібліотеки.

Опис реалізованих алгоритмів

Бібліотека QrGPU надає чотири основні алгоритми для перетворення матриць між різними форматами та розташуваннями пам'яті:

load_csr_host_to_gpu (Завантаження розрідженої матриці у форматі CSR з пам’яті хоста на GPU): Цей алгоритм відповідає за ефективне переміщення розрідженої матриці, що зберігається в CSR-форматі в оперативній пам'яті хоста, до глобальної пам'яті графічного процесора (GPU).

convert_csr_gpu_to_dense_gpu (Перетворення розрідженої матриці у форматі CSR в пам’яті GPU у щільну): Цей алгоритм виконує перетворення формату розрідженої матриці з розрідженого формату CSR на за допомогою бібліотечних засобів. Цей крок потрібен для підготовки розріджених даних до використання з бібліотеками лінійної алгебри (hipBLAS/cuBLAS), створених для роботи із щільними матрицями.

convert_dense_gpu_to_csr_gpu (Перетворення щільної матриці у пам’яті GPU у розріджену в форматі CSR): Цей алгоритм є оберненим до попереднього, дозволяючи перетворити щільну матрицю, що знаходиться на GPU, на розріджений CSR-формат, залишаючи дані на пристрої, бібліотечними засобами

unload_csr_gpu_to_host (Завантаження розрідженої матриці у форматі CSR з пам’яті GPU на хост): Цей алгоритм забезпечує переміщення результатів обчислень у форматі розрідженої матриці CSR-формату з глобальної пам'яті GPU до оперативної пам'яті хоста для подальшої обробки.

Також у QrGPU реалізовано два і обчислювальні алгоритми для виконання матричних операцій безпосередньо на GPU, використовуючи оптимізовані бібліотечні засоби:

decompose_dense_gpu (QR-розклад щільних матриць на GPU): Цей алгоритм виконує QR-розклад для щільних матриць у пам'яті GPU. Він використовує бібліотечні підпрограми, такі DnDgeqrf та DnDorgqr для обчислення ортогональної матриці Q та верхньої трикутної матриці R.

multiply_csr_gpu (Множення розріджених матриць на GPU): Даний алгоритм реалізує операцію множення для розріджених матриць у форматі CSR в пам'яті GPU. Для цього використовуються бібліотечна процедура Scsrmm.

Для користувача бібліотеки, для зручності, надано функції, що автоматизують перетворення даних, вивантаження та збірку результату а також обробку помилок, у формі конвеєрів (pipeline):

perform_full_qr_decomposition_pipeline): Цей конвеєр послідовно використовує процедури перетворення формату матриці та decompose_dense_gpu. Він починається із завантаження розрідженої матриці на GPU, її конвертації у щільний формат, виконання QR-розкладу щільної матриці, зворотного перетворення результатів у CSR-формат і завершується вивантаженням на CPU. Цей конвеєр інкапсулює всю послідовність операцій, забезпечуючи зручний інтерфейс для користувача.

perform_full_sparse_multiplication_pipeline: Аналогічно, для операції множення розріджених матриць, створений конвеєр, який включає завантаження операндів на GPU, використання multiply_csr_gpu та подальший збір результатів.

Гнучка збірка

Використана система збірки на основі CMake є інтуїтивною та адаптивною. Користувач може легко обрати цільовий бекенд (USE_HIP або USE_CUDA), а система автоматично знайде необхідні бібліотеки лінійної алгебри (HIPBLAS/HIPSOLVER/HIPSPARSE або їхні аналоги CUDA) через змінні оточення (ROCM_PATH/CUDA_PATH). Для компіляції в цілому використовується hipcc, що є макросом, який автоматично викликає nvcc чи amdclang++ залежно від обраної платформи, проте є можливість обрати nvcc примусово (гілка USE_CUDA).

Хоча HIP (Heterogeneous-compute Interface for Portability) сам по собі є крос-платформним рішенням, нативний CUDA-код є де-факто стандартом, а також часто дає вищу продуктивність на платформах NVIDIA. Тому було обрано такий гібридний підхід: спочатку весь функціонал був реалізований та оптимізований під CUDA. Потім, за допомогою інструменту HIPify, код було автоматично трансльовано у HIP-версію, та доопрацювано для усунення деяких конфліктів, що виникли під час роботи над системою. Це дозволило поєднати високу швидкодію та простоту підтримки коду, що забезпечує CUDA, та

водночас сумісність із платформами AMD завдяки повторному використанню коду.

QrGPU компілюється як динамічна бібліотека (Shared Library). Цей вибір ключовий для її створення та використання. По-перше, це зменшує використання оперативної пам’яті, оскільки бібліотека завантажується в пам'ять ОС лише один раз, навіть якщо її використовують кілька програм водночас. По-друге, це покращує сумісність зі стеком GPU, оскільки системні бібліотеки CUDA, ROCm та HIP самі є динамічними. Тому такий вибір природно вписується в цю модель. Ключовий момент в тому, що інтеграція QrGPU з Java через JNI безпосередньо залежить від її формату як динамічної бібліотеки.

Таким чином, використання динамічної бібліотеки робить QrGPU гнучкішою, ефективнішою і, що найважливіше, забезпечує для інтеграції з Java через JNI. Це архітектурне рішення прямо відповідає вимогам функціоналу та підтримки бібліотеки.

Обробка помилок

Для забезпечення стійкості бібліотеки до збоїв, впроваджено прозорий механізм обробки помилок. Спеціалізований клас GpuException та його відповідник у Java обгортці, що дозволяє піднімати помилки, що виникли під час виконання. Для автоматичної перевірки всіх викликів GPU API використовуємо глобальні обгортки:

- gpu_call для базових операцій (HIP/CUDA)
- blas_call для HIPBLAS/CUBLAS
- sparse_call для HIPSPARSE/CUSPARSE
- solver_call для HIPSOLVER/CUSOLVER

Якщо виклик завершується помилкою (наприклад, повертає не hipSuccess), обгортка генерує виняток GpuException. Це суттєво спрощує діагностику проблем.

Тестування

У процесі розробки застосовувався тест-орієнтований підхід (Test-Driven Development), за якого тестові сценарії створювались до написання основного коду. Тестування зосереджене переважно на unit-тестах для матриць невеликої розмірності, та на ключових компонентах - алгоритмам перетворення форматів, матричного множення, QR розкладу. Усі тести інтегровані в систему збирання за допомогою CTest, що забезпечує їх автоматичне виконання після кожної збірки

2.2. Інтеграція бібліотеки QrGPU із DAP

Для забезпечення взаємодії бібліотеки QrGPU з іншими програмними системами, зокрема DAP, було розроблено JNI-обгортку (Java Native Interface). Ця обгортка слугує мостом між нативним кодом C++ та віртуальною машиною Java (JVM), дозволяючи їй використовувати наші алгоритми.

Основним файлом, що містить реалізацію JNI-інтерфейсу, є jni_qr_gpu.cpp. У ньому реалізовані нативні методи Java, які безпосередньо взаємодіють з функціями обчислювальних конвеєрів QrGPU:

- decomposeSparse: Цей нативний метод відкриває доступ до функції perform_full_qr_decomposition_pipeline, дозволяючи виконувати повний QR-розклад розрідженої матриці безпосередньо з Java.
- multiplySparse: Цей метод є обгорткою для функції perform_full_sparse_multiplication_pipeline, яка забезпечує можливість множення розріджених матриць на GPU з Java-коду.

Одним із ключових аспектів інтеграції є перетворення форматів даних. DAP, зокрема, використовує власний формат розріджених матриць - "зубчастий масив" MatrixS. Щоб уникнути необхідності впровадження залежності від нього, ми вводимо клас із такою ж внутрішньою структурою, JavaSparseMatrix. Цей клас є уніфікованим представленням розріджених матриць як для вхідних даних, так і для результатів, що повертаються бібліотекою. При виклику нативних

методів, дані з JavaSparseMatrix перетворюються у CSR-формат який є найбільш подібним до MatrixS із тих, що нативно підтримується GPU-орієнтованими алгоритмами.

Клас QrGPU у пакеті qrgpu є основною точкою входу для Java-розробників. Він містить статичний блок, який завантажує нативну бібліотеку QrGPU_jni, та оголошення нативних методів decomposeSparse і multiplySparse, які приймають та повертають об'єкти JavaSparseMatrix (див. Лістинг 2).

```
package qrgpu;


public final class QrGPU {

    static {

        try {

            System.loadLibrary("QrGPU_jni");

        } catch (UnsatisfiedLinkError e) {

            System.err.println("Failed to load JNI library
'QrGPU_jni'");

        }

    }


    public static native JavaSparseMatrix[]
decomposeSparse(JavaSparseMatrix input);


    public static native JavaSparseMatrix
multiplySparse(JavaSparseMatrix[] input);

}
```

Лістинг 2 - Механізм завантаження нативної бібліотеки

Процес збірки JNI-обгортки та Java-класів повністю автоматизовано за допомогою системи збірки CMake. Це забезпечує зручність та послідовність у розгортанні бібліотеки. Автоматизована збірка включає кілька ключових етапів, які виконуються послідовно:

- Налаштування середовища збірки: Система автоматично визначає необхідні шляхи до інструментарію Java Development Kit.
- Збірка нативного JNI-модуля: Компілюється C++ код, що містить JNI-обгортку, створюючи динамічну бібліотеку (.so файл для Linux). Цей етап включає лінкування з основними бібліотеками QrGPU, а також з GPU-бібліотеками (hipBLAS, hipSPARSE, hipsolver у випадку вибору HIP).
- Компіляція Java: файл обгортки Java компілюється у байт-код (.class файли).
- Створення JAR архіву: Всі скомпільовані Java-класи упаковуються в єдиний архівний файл (.jar), що значно спрощує інтеграцію - достатньо додати цей .jar файл до classpath.

Основною перевагою такої системи є створення залежності на рівні CMake між нативними та Java компонентами. Вона гарантує, що Java класи будуть скомпільовані раніше ніж нативний модуль JNI, оскільки останній покладається на структуру класів обгортки. Така послідовність забезпечує надійність процесу збірки.

Інтеграція QrGPU в DAP була реалізована шляхом перевизначення механізму обчислення "листових" задач у дропах DAP. У типовій архітектурі DAP, коли рекурсивний матричний алгоритм досягає певного порогового малого розміру (leafSize), система викликає послідовний алгоритм (sequentialCalc). Для інтеграції було використано цей механізм. А саме, шляхом розширення класів дропів у DAP, як, наприклад, QRDecompositionGPU, який успадковується від

базового QRDecomposition (див. Лістинг 3). У цьому класі перевизначається метод sequentialCalc, який тепер замість звичайного послідовного обчислення на хості, викликає функціонал QrGPU. Також, для ефективного використання апаратного прискорення , поріг leafSize було значно збільшено, з типових 4 до експериментально підібраних 512 (для QR-розкладу), де система показує хороший приріст ефективності, зберігаючи стабільність.

Аналогічний підхід було застосовано для інтеграції інших GPU-прискорених операцій, таких як матричне множення та розклад QP (що також використовує QR-розклад на останньому етапі). Це дозволяє системі DAP, не змінюючи своєї архітектури динамічного розпаралелювання, ефективно використовувати обчислювальні можливості GPU для тих підзадач, де це дає найбільший приріст продуктивності, розширюючи її можливості для ресурсоємних матричних операцій.

Лістинг 3 - Виклик qrGPU з DAP

```
public class QRDecompositionGPU extends QRDecomposition {

    protected static int leafSize = 512; // Збільшений поріг
    @Override
    public void sequentialCalc(Ring ring) {

        MatrixS source = (MatrixS) inData[0];

        // Виклик QR-розкладу через обгортку QrGPU
        MatrixS[] QR = GpuBlockQR.compute(source, ring);

        outData[0] = QR[0];
        outData[1] = QR[1];
    }
}
```

РОЗДІЛ 3. ЕКСПЕРЕМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

Метою дослідження є оцінка продуктивності, масштабованості та обмежень реалізованого QR-алгоритму при виконанні на одному графічному процесорі та в розподіленому середовищі за допомогою технології DAP.

3.1. Конфігурація тестового середовища

Апаратне забезпечення:

- CPU: Intel Core i5-9400F
- Оперативна пам'ять: 32 ГБ DDR4
- GPU: AMD Radeon RX 6700 XT (12 ГБ VRAM),
- Шина даних: PCI Express 3.0x16

Програмне забезпечення:

- Операційна система: Ubuntu 22.04 LTS
- Платформа: AMD ROCm 6.3.3 (з реалізацією HIP)
- Компілятор C++17: hipcc (amdclang++)
- Стандартна бібліотека C++17: libhipcxx
- Open MPI 5.0.7

3.2. Методологія тестування

3.2.1. Генерація тестових даних

Для тестування використовувалися випадково згенеровані квадратні розріджені матриці A розмірністю N×N з елементами типу double. Щоб змоделювати різні практичні сценарії, було обрано три рівні щільності (відсоток ненульових елементів):

- 0.1%: Високорозріджені.

- 1%: Матриці з середньою розрідженістю (напр., аналіз соціальних мереж).
- 10%: Слаборозріджені.

3.2.2. Сценарії тестування

Дослідження проводилося за двома ключовими сценаріями:

- Прямий запуск QrGPU на одному GPU: Мета - визначити максимальну розмірність матриці, що може бути оброблена в межах відеопам'яті одного GPU, та оцінити базову продуктивність.
- Розподілений запуск QrGPU за допомогою DAP: Мета — продемонструвати здатність системи долати обмеження пам'яті одного вузла та розв'язувати задачі значно більшого масштабу.

3.*2.3. Метрики оцінки*

Оцінка ефективності проводилася за такими критеріями:

- Час виконання: Загальний час, витрачений на виконання QR-розкладу, виміряний за допомогою std::chrono::high_resolution_clock.
- Максимальна розмірність матриці: Граничний розмір матриці, при якому система залишається працездатною.

3.3. Аналіз продуктивності QrGPU на одному вузлі

На цьому етапі досліджувалися характеристики алгоритму при запуску на одній машині без залучення системи DAP.

3.3.1. Обмеження відеопам'яті та максимальна розмірність

Ключовим обмеженням для обробки великих матриць на одному GPU є обсяг відеопам'яті (VRAM). Під час виконання QR-розкладу, особливо з використанням бібліотек типу hipSOLVER, що працюють зі щільними

матрицями, відбувається значне споживання пам'яті. Навіть якщо вхідна матриця A розріджена, матриці Q та R зазвичай є щільними або значно щільнішими, що призводить до "розбухання" даних (ефект fill-in).

Експерименти показали, що при спробі обробити матриці, розмірність яких перевищувала $N = 16030\ (\sim 2^{14})$, виникала помилка, що свідчить про повне вичерпання 12 ГБ відеопам'яті.

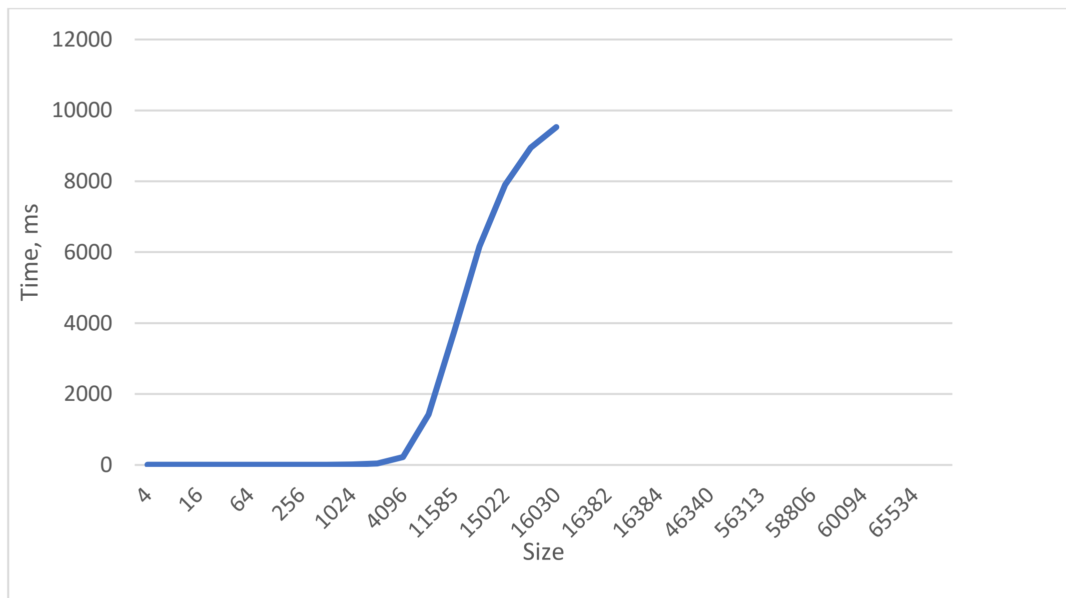


Рис. 3.1. Залежність часу виконання від розмірності матриці N для прямого запуску QrGPU.

Як видно з рис. 3.1, з наближенням до граничного розміру N, подальші обчислення стають неможливими. Це підтверджує фундаментальне обмеження підходу з одним GPU для задач великого масштабу.

3.4. Масштабування алгоритму з використанням технології DAP

Основною метою роботи було подолання обмежень одного GPU. Для цього було використано технологію динамічного розпаралелювання DAP.

3.4.1. Порівняння продуктивності

Запуск алгоритму через DAP дозволив обробляти матриці, розмірність яких значно перевищує можливості одного GPU. Експериментально вдалося успішно виконати QR-розклад для матриці розмірністю $N = 58806\ (\sim 2^{16})$. Подальше збільшення розміру обмежувалося вже не відеопам'яттю, а загальною стабільністю тестової системи та обсягом оперативної пам'яті для управління завданнями.

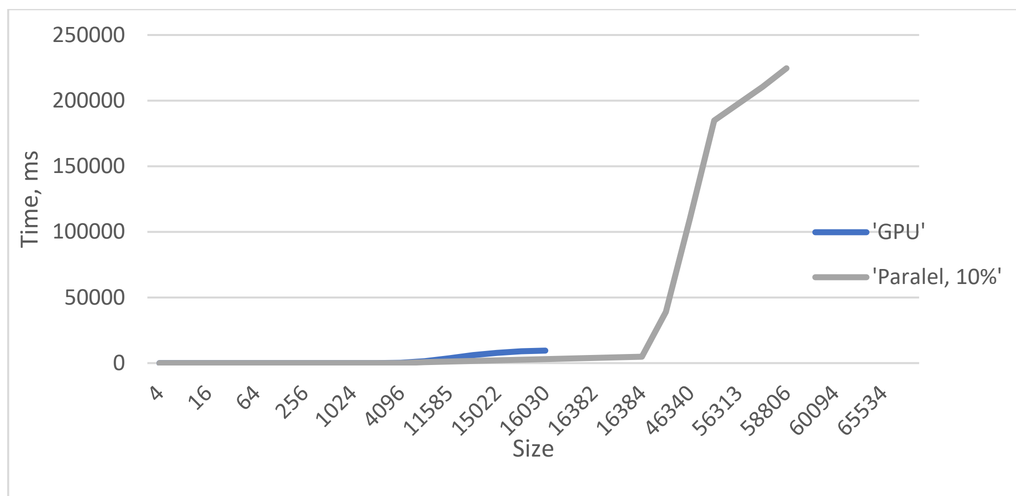


Рис. 3.2. Порівняння часу виконання QR розкладу для прямого запуску QrGPU та розподіленого запуску через DAP, матриці щільністю 10%.

На рис. 3.2 видно, що для матриць малого розміру підхід з DAP може бути повільнішим через накладні витрати на комунікацію та управління розподіленими завданнями (дропами). Однак його ключова перевага - здатність до масштабування - повністю нівелює цей недолік для великих задач, що і є ціллю даної роботи. З іншого боку видно значну перевагу нашого алгоритму – оптимізація для дуже розріджених матриць, як видно на рисунку 3.3:

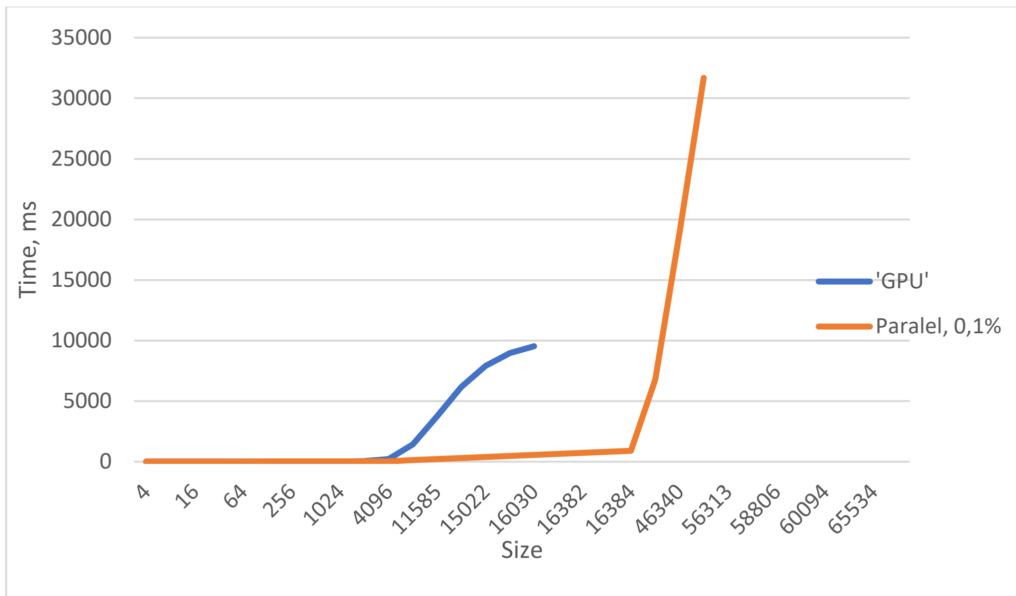


Рис. 3.3. Порівняння часу виконання QR розкладу для прямого запуску QrGPU та розподіленого запуску через DAP, матриці щільністю 0,1%.

3.3.2. Вплив оптимізації (перенесення множення матриць на GPU)

У ході розробки було реалізовано важливу оптимізацію: операції множення матриць, які є частиною рекурсивного QR-алгоритму, були перенесені з CPU на GPU. Це дозволило уникнути зайвих передач даних між CPU та GPU, які є вузьким місцем системи.

Порівняння продуктивності до та після оптимізації показало значний приріст швидкодії. Для матриць щільністю 10% середнє прискорення склало ~23%. Водночас для високорозріджених матриць (0.1%) ефект був менш вираженим. Це пояснюється тим, що накладні витрати на запуск GPU-ядра для множення дуже розріджених матриць можуть бути співставними з виграшем від паралельних обчислень.

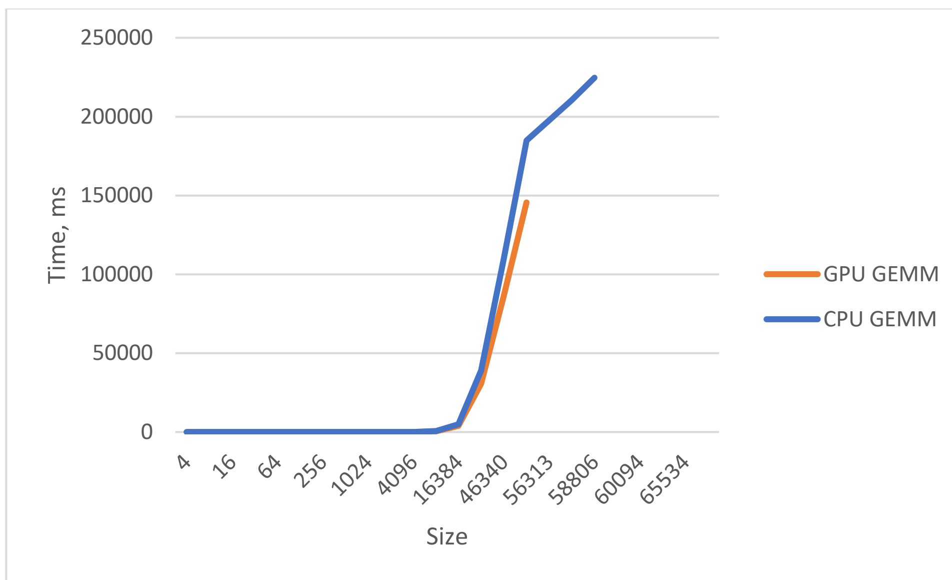


Рис. 3.4. Порівняння часу виконання до та після перенесення операції множення на GPU, для матриць із 1% щільності.

3.4.2. Аналіз масштабованості та подолання апаратних обмежень

Результати підтверджують, що розроблений підхід ефективно розв'язує поставлену задачу. Технологія DAP дозволяє динамічно розподіляти блочно-рекурсивні обчислення між доступними ресурсами, використовуючи GPU як потужний прискорювач для виконання окремих підзадач (обробки дропів).

Це дозволяє системі вийти за межі одного обчислювального вузла і теоретично масштабуватися на великі кластери. Обмеження, що спостерігалося, пов'язане з апаратною конфігурацією тестового стенда, а не з архітектурою самого алгоритму. На потужнішому суперкомп'ютері з більшим обсягом RAM та швидшими комунікаціями можна досягти обробки матриць ще більшого порядку.

3.5. Аналіз результатів

Експериментально доведено, що використання QrGPU на одному графічному процесорі обмежене обсягом відеопам'яті. Для тестової системи з 12 ГБ VRAM максимальна розмірність матриці float (8b) склала $N = 16030$ ($\sim 2^{14}$).

Застосування технології динамічного розпаралелювання DAP дозволило успішно подолати обмеження одного GPU та розв'язати задачу для матриці розмірністю $N = 58806$ ($\sim 2^{16}$).

Перенесення обчислень множення матриць з CPU на GPU дало значний приріст продуктивності (до 23%), що пропускна здатність шини PCIe обмежувальним фактором в цих експериментах.

З іншого боку, основні джерела накладних витрат зрозумілі:

- Виклики до операційної системи для використання JNI (перемикання контексту).
- Перетворення даних, через різні формати збереження матриць у різних частинах системи.
- Передавання даних між CPU/GPU.
- Підготовка даних для GPU: Оскільки hipSOLVER та hipBLAS працюють з щільними матрицями, а вхідні дані – розріджені.

Для матриць малого та середнього розміру вони часто займають більше часу, ніж власне розклад на GPU. Хоча для дуже великих матриць відносна частка цих витрат зменшується (оскільки домінує час обчислень), вони все одно залишаються важливим фактором, що обмежує загальну швидкодію бібліотеки.

Також, при роботі із розрідженими матрицями, варто пам’ятати, як QR-розклад впливає на розрідженість матриць. А саме, що результати втрачають початкову розрідженість. На практиці це означає, що ортогональна матриця Q часто виявляється майже повністю заповненою, а верхня трикутня матриця R, хоч і зберігає структуру, містить значно більше ненульових елементів, ніж початкова матриця A. Це не вада реалізації, а властивість самих алгоритмів QR-розкладу - особливо методів Хаусхолдера (Householder) та Гівенса (Givens rotations), які і використовує бібліотеки QrGPU. Під час обнулення елементів ці перетворення неминуче заповнюють нульові позиції. Наприклад, перетворення

Хаусхолдера для одного стовпця впливає на всі наступні, створюючи нові ненульові елементи там, де раніше були нулі.

ВИСНОВКИ

Дипломна робота «Розробка QR-алгоритму розв'язання систем лінійних рівнянь на суперкомп'ютері, оснащеному GPU» була спрямована на створення масштабованого і портативного інструменту для розв'язання великих систем лінійних рівнянь із розрідженими матрицями, що враховує специфіку сучасних гетерогенних обчислювальних архітектур із GPU і розподіленою пам'яттю. Практичною метою стало розробити таку реалізацію QR алгоритму, яка б дозволила обробляти матриці, розмірність яких перевищує обсяг відеопам'яті одного графічного процесора.

Під час виконання роботи проведено аналіз архітектурних особливостей GPU, зокрема обмежень пам'яті та пропускної здатності шини PCIe, які суттєво впливають на ефективність чисельних методів. Встановлено, що сучасні бібліотечні алгоритми QR розкладу на GPU (cuSOLVER, hipSOLVER) працюють виключно з щільними матрицями, і тому втрачають ефективність при роботі з розрідженими даними [12][25].

Для подолання цих обмеженнь запропоновано використати вже реалізований у системі DAP рекурсивний QR-алгоритм на основі обертань Гівенса [23][24]. Такий підхід дає змогу розбити вхідну розріджену матрицю на блоки та передавати лише малі підматриці (листи рекурсії) фіксованого розміру на GPU. При цьому перетворення матриці на щільну відбувається вже на GPU, щоб зменшити кількість даних, що передаються.

У ході роботи, створено процедурно бібліотеку QrGPU для мови C++, що реалізує виконання QR розкладу і матричного множення розріджених матриць на GPU, що дозволяє ефективно використовувати його ресурси при мінімізації накладних витрат. Бібліотека написана із використанням CUDA API, проте завдяки технології платформі HIP та інструменту hipify [9], ця бібліотека підтримує як платформу NVIDIA, так і AMD. Бібліотеку QrGPU було інтегровано із системою DAP методом створення їй JNI інтерфейсу.

У процесі експериментального дослідження встановлено, що оптимальний розмір листів становить 512×512. На одному GPU із 12 ГБ відеопам’яті, QR розклад був обмежений розміром матриці ~16 тис. Після інтеграції GPU у DAP, реалізацію вдалося успішно обробити систему з матрицею розміром 58 тис., що свідчить про подолання обмежень пам’яттю GPU. Водночас виявлено кілька джерел накладних витрат, зокрема: використання JNI, перетворення форматів даних, передача через PCIe та необхідність адаптації розріджених вхідних матриць до форматів, підтримуваних бібліотеками GPU.

Особливу увагу приділено впливу QR розкладу на структуру розрідженості. У процесі обчислень відбувається заповнення нульових елементів, що призводить до значного зростання кількості ненульових компонентів у матрицях Q та R. Це властивість алгоритмів QR розкладу, а не недолік реалізації, однак саме вона обмежує масштабування - оскільки збільшене споживання пам’яті на хості при роботі з великими системами є вузьким місцем при поточній реалізації.

Подальший розвиток роботи має полягати в адаптації поточної реалізації до умов реального кластеру, де на кожному вузлі є GPU. Окрім того, необхідно відмовитися від проміжного шару JNI й запровадити пряму взаємодію всіх процесів через MPI, що дозволить суттєво зменшити накладні витрати на переключення контекстів і прискорить передачу блоків.

Тим не менш, розроблений розподілений QR алгоритм демонструє високу масштабованість, що робить його перспективним інструментом для розв'язання систем лінійних рівнянь великого масштабу на сучасних гетерогенних суперкомп'ютерах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

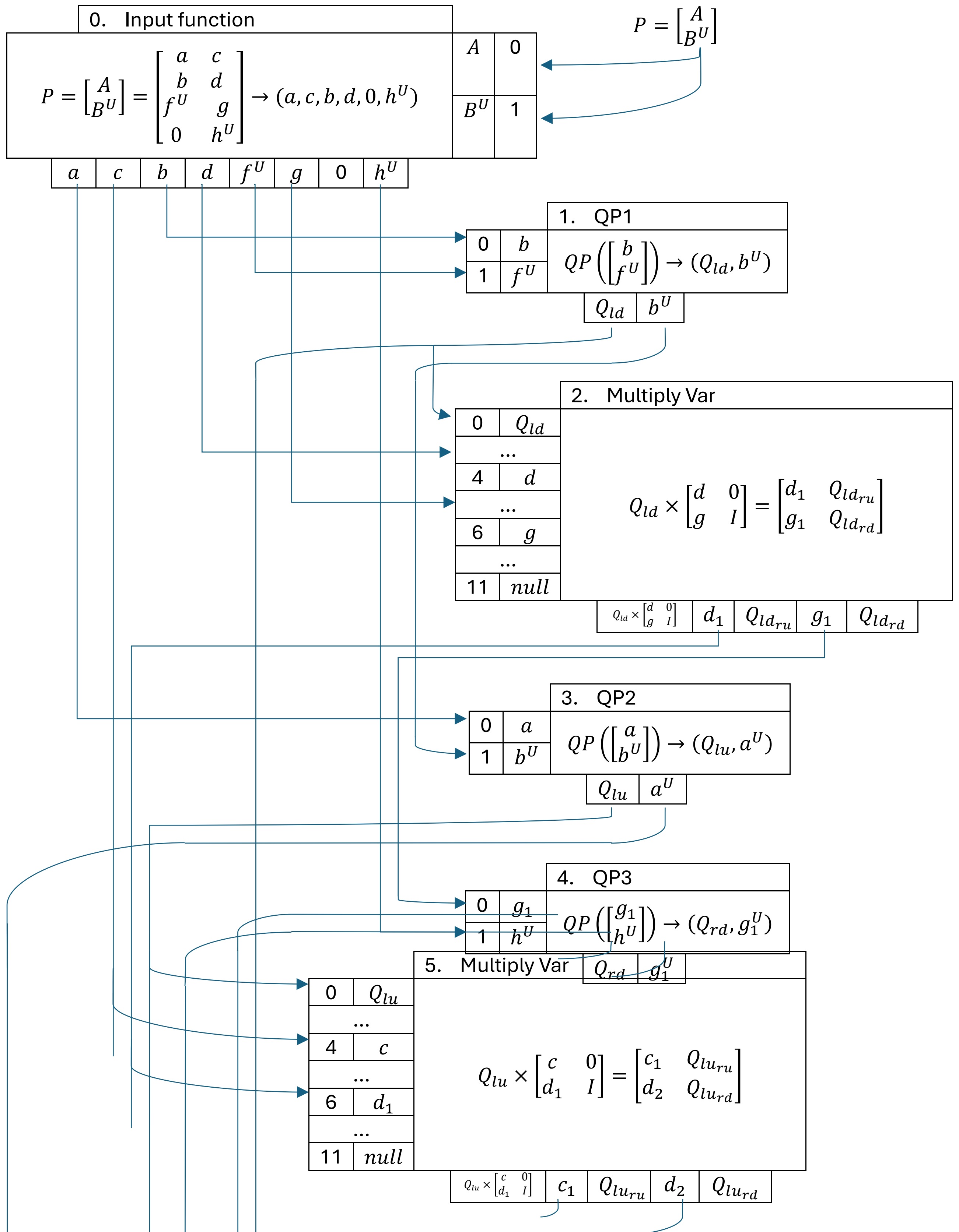
1. Tracy, K. (2022, August 12). A Square-Root Kalman Filter Using Only QR Decompositions. arXiv.org. https://arxiv.org/abs/2208.06452
2. Legendre, M., Schmidt, A., Moussaoui, S., & Lammers, U. (2013). Solving systems of linear equations by GPU-based matrix factorization in a Science Ground Segment. Astronomy and Computing, 3-4, 58–64. https://doi.org/10.1016/j.ascom.2013.11.004
3. Świrydowicz, K., Darve, E., Jones, W., Maack, J., Regev, S., Saunders, M. A., Thomas, S. J., & Peleš, S. (2022). Linear solvers for power grid optimization problems: A review of GPU-accelerated linear solvers. Parallel Computing, 111, 102870. https://doi.org/10.1016/j.parco.2021.102870
4. Zlatev, Z. (1980). On Some Pivotal Strategies in Gaussian Elimination by Sparse Technique. SIAM Journal on Numerical Analysis, 17(1), 18–30
5. Davis, T. A., & Duff, I. S. (1997). AN UNSYMMETRIC-PATTERN MULTIFRONTAL METHOD FOR SPARSE LU FACTORIZATION. SIAM Journal on Matrix Analysis and Applications, 18(1), 140–158.
6. Hogg, J. D., Ovtchinnikov, E., & Scott, J. A. (2016). A Sparse Symmetric Indefinite Direct Solver for GPU Architectures. ACM Transactions on Mathematical Software (TOMS), 42(1), 1–25. https://doi.org/10.1145/2756548
7. BLAS Technical Forum. (2001). BLAS Technical Forum Standard. netlib.org. http://www.netlib.org/blas/blast-forum/blas-report.pdf
8. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., & Sorensen, D. (1999). LAPACK Users' Guide (3rd ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719604
9. AMD. (2025, May 21). HIP Documentation, Release 6.4.1. In ROCm Documentation. Advanced Micro Devices, Inc. https://rocm.docs.amd.com/projects/HIP/en/latest/

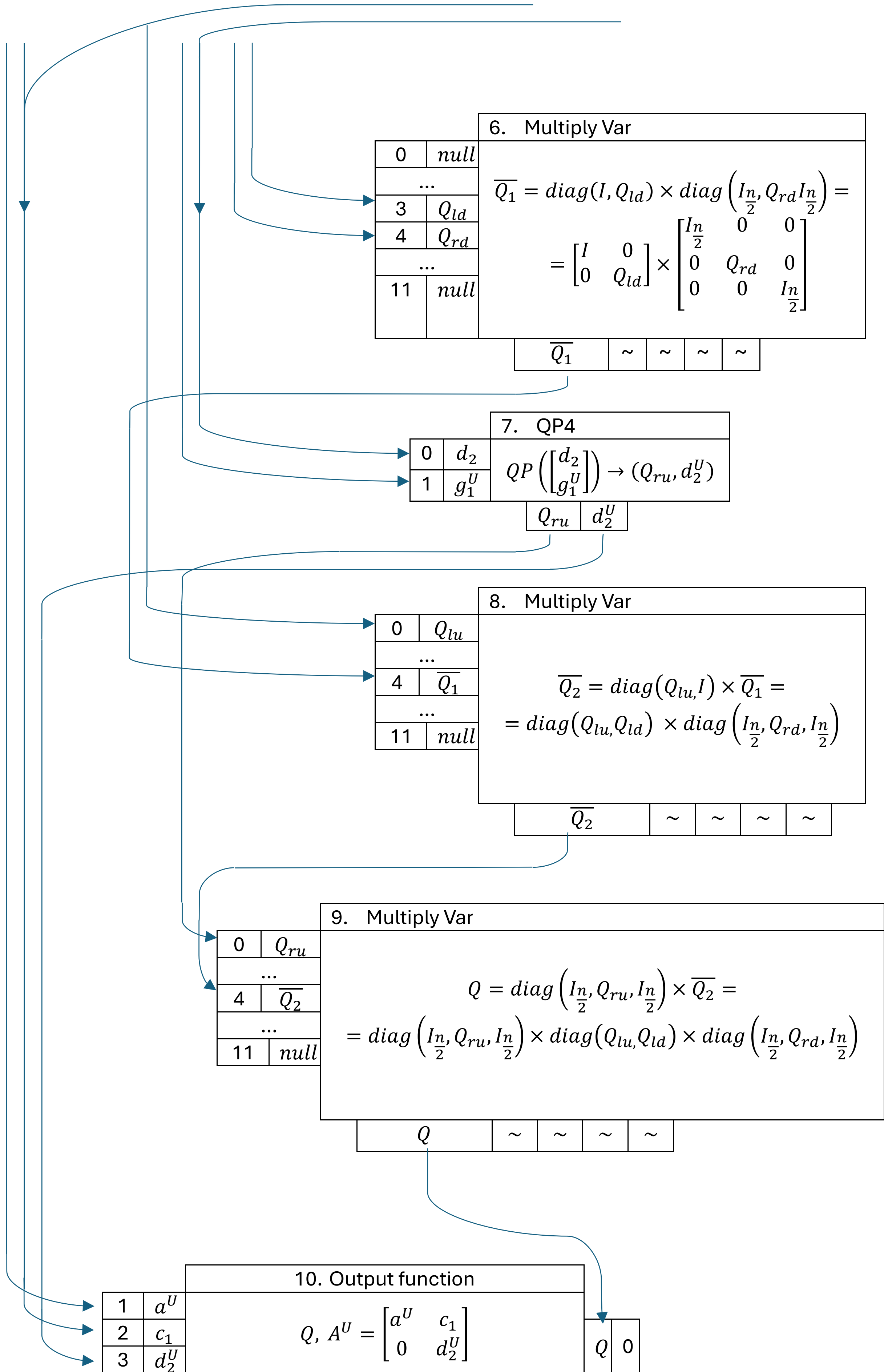
10. NVIDIA. (2025, June). cuBLAS Library, Release 12.9.1. In CUDA Toolkit Documentation. NVIDIA Corporation. https://docs.nvidia.com/cuda/cublas/index.html
11. NVIDIA. (2025, June). cuSPARSE Library, Release 12.9. In CUDA Toolkit Documentation. NVIDIA Corporation. https://docs.nvidia.com/cuda/cusparse/index.html
12. NVIDIA. (2025, June). cuSOLVER Library, Release 11.7.5.82. In CUDA Toolkit Documentation. NVIDIA Corporation. https://docs.nvidia.com/cuda/cusolver/index.html
13. AMD. (2025, May 21). hipBLAS Library, Release 6.4.1. In ROCm Documentation. Advanced Micro Devices, Inc. https://rocm.docs.amd.com/projects/hipBLAS/en/latest/
14. AMD. (2025, May 21). hipSPARSE Library, Release 6.4.1. In ROCm Documentation. Advanced Micro Devices, Inc. https://rocm.docs.amd.com/projects/hipSPARSE/en/latest/
15. Li, X. S., Demmel, J. W., Gilbert, J. R., & Grigori, L. (2018). *SuperLU Users' Guide* (LBNL-44289). Lawrence Berkeley National Laboratory. https://portal.nersc.gov/project/sparse/superlu/
16. Li, X. S. (1996). Sparse Gaussian Elimination on High Performance Computers [Doctoral dissertation, University of California at Berkeley]. University of California, Berkeley.
17. Li, X. S., & Demmel, J. W. (1998). Making Sparse Gaussian Elimination Scalable by Static Pivoting. In Proceedings of SC 98.
18. Grigori, L., Demmel, J. W., & Li, X. S. (2007). Parallel Symbolic Factorization for Sparse LU with Static Pivoting. SIAM Journal on Scientific Computing, 29(3), 1289–1314. https://doi.org/10.1137/050638102
19. Abdelfattah, A., Beams, N., Carson, R., Ghysels, P., Kolev, T., Stitt, T., Vargas, A., Tomov, S., & Dongarra, J. (2024). MAGMA: Enabling exascale performance with accelerated BLAS and LAPACK for diverse GPU architectures. The International Journal of High Performance Computing Applications.

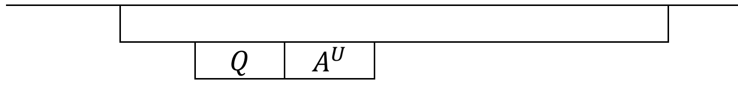
20. Rouet, F.-H., Li, X. S., Ghysels, P., & Napov, A. (2016). A Distributed-Memory Package for Dense Hierarchically Semi-Separable Matrix Computations Using Randomization. ACM Transactions on Mathematical Software, 42(4), Article 4. https://doi.org/10.1145/2930660
21. Message Passing Interface Forum. (2025, June). MPI: A Message-Passing Interface Standard Version 5.0. https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf
22. Open MPI Project. (2025, May 12). Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/
23. Малашонок, Г. І., & Сідько, А. А. (2018). Розподілені обчислення: ДАП-технологія розпаралелювання рекурсивних алгоритмів. Наукові записки НаУКМА. Комп'ютерні науки, 1, 25–32. http://nbuv.gov.ua/UJRN/NaUKMAkn_2018_1_8
24. Малашонок, Г., & Івашкевич, А. (2021). Quick Recursive QR Decomposition. У Proceedings of the Conference on Mathematical Foundations of Informatics MFOI2020.
25. AMD. (2025, May 21). hipSOLVER Library, Release 6.4.1. In ROCm Documentation. Advanced Micro Devices, Inc. https://rocm.docs.amd.com/projects/hipSOLVER/en/latest/

ДОДАТОК А

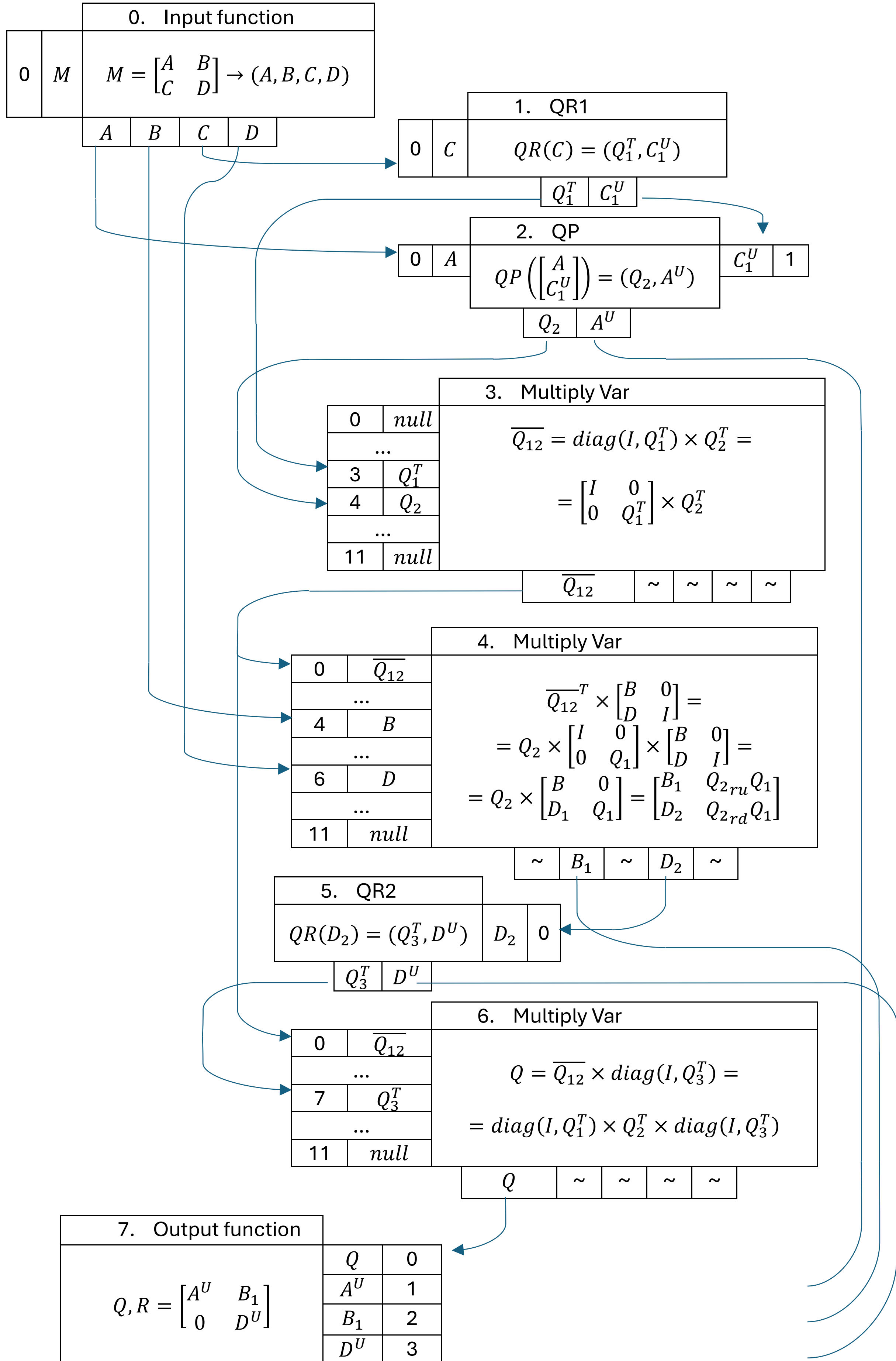
Схема рекурсивного QR+QP алгоритму




6. Multiply Var
0 null
...
3 Q_{ld}
4 Q_{rd}
...
11 null
$\overline{Q_1} = diag(I, Q_{ld}) \times diag\left(I_{\frac{n}{2}}, Q_{rd} I_{\frac{n}{2}}\right) = \begin{bmatrix} I & 0 \\ 0 & Q_{ld} \end{bmatrix} \times \begin{bmatrix} I_{\frac{n}{2}} & 0 & 0 \\ 0 & Q_{rd} & 0 \\ 0 & 0 & I_{\frac{n}{2}} \end{bmatrix}$
$\overline{Q_1}$ ~ ~ ~ ~
7. QP4
0 d_2
1 g_1^U
$QP\left(\begin{bmatrix} d_2 \\ g_1^U \end{bmatrix}\right) \to (Q_{ru}, d_2^U)$
Q_{ru} d_2^U
8. Multiply Var
0 Q_{lu}
...
4 $\overline{Q_1}$
...
11 null
$\overline{Q_2} = diag(Q_{lu}, I) \times \overline{Q_1} = diag(Q_{lu}, Q_{ld}) \times diag\left(I_{\frac{n}{2}}, Q_{rd}, I_{\frac{n}{2}}\right)$
$\overline{Q_2}$ ~ ~ ~ ~
9. Multiply Var
0 Q_{ru}
...
4 $\overline{Q_2}$
...
11 null
$Q = diag\left(I_{\frac{n}{2}}, Q_{ru}, I_{\frac{n}{2}}\right) \times \overline{Q_2} = diag\left(I_{\frac{n}{2}}, Q_{ru}, I_{\frac{n}{2}}\right) \times diag(Q_{lu}, Q_{ld}) \times diag\left(I_{\frac{n}{2}}, Q_{rd}, I_{\frac{n}{2}}\right)$
Q ~ ~ ~ ~
10. Output function
1 a^U
2 c_1
3 d_2^U
$Q,\ A^U = \begin{bmatrix} a^U & c_1 \\ 0 & d_2^U \end{bmatrix}$
Q 0


Q
A^U

QR Graph:



Q	R