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

Кваліфікаційна робота

Освітній ступінь - магістр

На тему **"Інтеграція прискорення на графічному процесорі у середовище виконання для блоково-рекурсивних матричних алгоритмів DAP"**

Виконав: студент 2-го року навчання освітньої програми «Комп'ютерні науки» спеціальності 122 Комп'ютерні науки

Комонов Кирило Максимович

Керівник: Малашонок Геннадій Іванович доктор фіз.-мат. наук, професор.

Кваліфікаційна робота захищена з оцінкою

Секретар ЕК

«____» _________________2025 р.

Київ - 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мережних технологій
доктор фіз-мат наук, професор
Малашонок Геннадій Іванович

____

«____» _________________2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту 2 р.н. магістерської програми Комп'ютерні науки

Комонову Кирилу Максимовичу

Інтегрувати прискорення на графічному процесорі у середовище виконання для блоково-рекурсивних матричних алгоритмів DAP.

Зміст текстової частини кваліфікаційної роботи:

Анотація

Вступ

1. Огляд множення матриць у середовищі паралельних обчислень блоково-рекурсивних алгоритмів DAP
2. Множення матриць на відеокарті
3. Імплементація алгоритмів
4. Експерименти

Висновки

Список використаної літератури

Додатки

Дата видачі « » 2024 р.

Керівник

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

Завдання отримав

К. М. Комонов ____________

Тема: “Інтеграція прискорення на графічному процесорі у середовище виконання для блоково-рекурсивних матричних алгоритмів DAP”

Календарний план виконання роботи:

№ п/п	Назва етапу дипломної роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу.	03.11.2024	
2.	Огляд технічної літератури за темою роботи.	01.01.2025	
3.	Виконати аналіз методів виконання обчислень на відеокартах.	14.01.2025	
4.	Програмування множення матриць на відеокарті.	01.02.2025	
5.	Інтеграція програми у DAP.	01.03.2025	
6.	Виконання експериментів на суперкомп’ютері.	14.04.2025	
7.	Написання пояснювальної роботи.	14.05.2025	
8.	Створення слайдів для доповіді.	28.05.2025	
9.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи.	30.05.2025	
10.	Корегування роботи за результатами попереднього захисту.	04.06.2025	
11.	Остаточне оформлення пояснювальної роботи та слайдів.	09.06.2025	
12.	Захист магістерської роботи.	11.06.2025	

Студент Комонов Кирило Максимович

Керівник Малашонок Геннадій Іванович «____»_______________2025 р.

Зміст

Анотація ... **6**

Перелік прийнятих скорочень ... **7**

Вступ ... **8**

Розділ 1. Огляд множення матриць у середовищі паралельних обчислень блоково-рекурсивних алгоритмів DAP ... **10**

1.1 Блоково-рекурсивні алгоритми множення матриць ... 10

1.1.3 Алгоритм «розділяй і володарюй» ... 10

1.1.4 Алгоритм Штрассена ... 11

1.2 Платформа DAP ... 11

1.2.1 Основні ідеї DAP ... 11

1.2.2 Потоки DAP ... 12

1.2.3 Важливі змінні ... 12

1.2.4 Множення щільних матриць у DAP ... 12

1.3 Огляд літератури ... 12

Розділ 2. Множення матриць на відеокарті ... **14**

2.1 CUDA ... 14

2.2 cuBLAS ... 14

2.3 Підготовка даних до передачі у відеопам’ять ... 15

2.4 Ефективне використання CUDA ... 15

2.4.1 Асинхронний API CUDA ... 15

2.4.2 Перемикання контекстів CUDA ... 17

Розділ 3. Імплементація алгоритмів ... **18**

3.1 Опис алгоритму ... 18

3.2 Інструменти координації ... 18

3.3 Ініціалізація ... 18

3.4 Координаційна модель диспетчер-підлеглі ... 20

3.4.1 Огляд координаційної моделі ... 20

3.4.2 CudaDispatcher ... 22

3.4.3 CudaExecutor ... 23

3.5 Інтеграція у DAP ... 23

3.5.1 Інтеграція CudaDispatcher ... 23

3.5.2 Інтеграція CudaExecutor ... 24

Розділ 4. Експерименти ... **26**

4.1 Опис експериментів 26
4.2 Середовище виконання експериментів 26
4.3 Використані метрики 27
4.4 Результати експериментів 27
4.4.1 Вузол g4301 27
4.4.2 Вузол n5013 31
4.4.3 Інтерпретація результатів експериментів 32
Висновки **34**
Список використаної літератури **35**
Додаток А **38**
Додаток Б **39**
Додаток В **40**
Додаток Г **41**
Додаток Д **42**
Додаток Е **43**

Анотація

Було розглянуто задачу множення щільних матриць блоково-рекурсивними алгоритмами в середовищі виконання паралельних обчислень DAP і використання CUDA для обчислень в кластерному середовищі. Запропоновано і реалізовано алгоритм координації доступу до графічних процесорів в кластерному середовищі. Множення щільних матриць використовуючи cuBLAS інтегровано у середовище DAP. Проведено експерименти для порівняння часу виконання програми. Проаналізовано отримані результати та зроблено висновки про подальший розвиток дослідження.

Ключові слова: DAP, CUDA, cuBLAS, MPI, dense matrix, matrix multiplication, HPC.

Перелік прийнятих скорочень

HPC (High Performance Computing) – високопродуктивні обчислення, тобто обчислення що виконуються на кластерах (суперкомп'ютерах).

CUDA (Compute Unified Data Architecture) – обчислювальна архітектура, однойменна бібліотека для виконання обчислень на відеокартах від компанії Nvidia.

MPI (Message Passing Interface) – бібліотека для комунікації у середовищі з розділеною пам'яттю, наприклад на кластері.

API (Application Programming Interface) – інтерфейс взаємодії комп'ютерних програм, пакетів, бібліотек між собою.

Java – мова програмування.

Дроп – об'єкт класу Drop, описує завдання яке необхідно виконати над вхідними даними.

DAP (Drop Amine Pine) - назва середовища(платформи) обчислення блоково-рекурсивних матричних алгоритмів.

GPU (Graphic Processing Unit) – графічний процесор, синонім відеокарти (в контексті цієї роботи).

Вступ

Множення матриць - одна з найважливіших задач сьогодення. Ця задача є складовою алгоритмів, що використовуються в різних наукових сферах (фізичні симуляції, гідрологія, квантова механіка). В контексті комп'ютерних наук це використання нейронних мереж, лінійна алгебра, комп'ютерна графіка. Задачі, що потребують обчислень над матрицями великих розмірів, виконуються у середовищах високопродуктивних обчислень (HPC), також їх називають суперкомп'ютерами або кластерами. Сучасні кластери часто містять відеокарти, які можуть бути використані для прискорення певних алгоритмів. Множення матриць – один з таких алгоритмів.

Середовище виконання блоково-рекурсивних матричних алгоритмів DAP дозволяє виконувати матричні операції у кластерному середовищі. DAP наразі не підтримує прискорення на відеокартах. Тобто всі обчислення виконуються на класичних процесорах. Метою цієї роботи є інтеграція прискорення алгоритмів множення матриць на відеокартах у DAP, порівняння швидкодії отриманої програмної системи. Об'єктом дослідження є поєднання обчислень на відеокарті і середовища блоково-рекурсивних матричних алгоритмів DAP.

Для досягнення мети необхідно виконати наступні завдання:

1) Оглянути існуючі програмні інструменти для виконання обчислень на відеокарті.
2) Реалізувати виконання необхідних алгоритмів на відеокарті.
3) Інтегрувати це рішення у середовище DAP.
4) Провести експерименти і заміряти метрики, що нас цікавлять.
5) На основі результатів експериментів з'ясувати межі використання прискорення на відеокартах для задачі матричного множення.
6) Визначити, для якого діапазону розмірів вхідних даних використання графічного процесора надає перевагу у швидкодії.

У першому розділі розглянуто рекурсивні алгоритми множення матриць, платформу DAP, існуючі роботи з поєднання CUDA і MPI.

У другому розділі описано особливості використання графічного процесора для виконання обчислень. Розглянуто шляхи збільшення ефективності програм на відеокарті.

У третьому розділі описано імплементацію необхідних алгоритмів, особливості реалізованої програми. Запропоновано і реалізовано алгоритм координації процесорів щодо спільного використання відеокарт на вузлі кластера.

У четвертому розділі наведено деталі експериментів, що було проведено для оцінки результатів роботи. Було розраховано кратне збільшення швидкодії програми.

Ефективне використання графічних карт на кластері потребує координації між процесами. Тому, в рамках роботи реалізовано «диспетчер» доступу до відеокарт. Було досягнуто значного пришвидшення швидкодії програми для задачі множення щільних матриць завдяки використанню потужностей графічних процесорів.

Розділ 1. Огляд множення матриць у середовищі паралельних обчислень блоково-рекурсивних алгоритмів DAP

1.1 Блоково-рекурсивні алгоритми множення матриць

У цій роботі розглянуто блоково-рекурсивні алгоритми множення матриць. Це алгоритм «розділяй і володарюй» і алгоритм Штрассена. В контексті цих алгоритмів розглядаємо дві матриці X і Y розміру 2N на 2N, де N – довільне натуральне число. Розділимо кожну з матриць на чотири підматриці (Рисунок 1).

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Рисунок 1.1 – Розділення матриць на чотири підматриці.

Обидва алгоритми передбачають декомпозицію задачі множення матриці розмірами 2N на 2N у операції над матрицями розміром N на N. Розглянемо їх детальніше.

1.1.3 Алгоритм «розділяй і володарюй»

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Рисунок 1.2 – Схема алгоритму множення матриць «розділяй і володарюй»

У алгоритмі «розділяй і володарюй» ми маємо 8 операцій множення над матрицями розміру N. Асимптотична складність цього алгоритму – O(n^3), де n – розмір вхідних матриць [1].

1.1.4 Алгоритм Штрассена

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

$$\begin{aligned} P_1 &= A(F-H) & P_5 &= (A+D)(E+H) \\ P_2 &= (A+B)H & P_6 &= (B-D)(G+H) \\ P_3 &= (C+D)E & P_7 &= (A-C)(E+F) \\ P_4 &= D(G-E) & & \end{aligned}$$

Рисунок 1.3 - Схема алгоритму множення матриць Штрассена

У алгоритмі Штрассена ми маємо 7 операцій множення над матрицями розміру N. Асимптотична складність цього алгоритму – $O(n^{\log_2 7}) \approx O(n^{2.83})$, де n – розмір вхідних матриць [1]. Тобто алгоритм Штрассена оптимальніший за алгоритм «розділяй і володарюй».

1.2 Платформа DAP

Платформа DAP (Drop Amine Pine) [2] дозволяє виконувати рекурсивні алгоритми над матрицями у середовищі з розподіленою пам’яттю (на кластері). DAP реалізовано мовою програмування Java, для комунікації використовується MPI.

1.2.1 Основні ідеї DAP

Завдання, що потребує обчислення, представлено об’єктом класу Drop [2] (в подальшому об’єкти класу Drop або його нащадків називається дропом, або завданням). Обчислення дропу можна виконати одразу або розділити його на дропи з вхідними матрицями меншого розміру. Всі алгоритми реалізовані в DAP можуть бути представлені як направлені графи, де вершинами є дропи, а зв’язки між вершинами залежності між дропами. Такий граф, що описує алгоритм дропу, називається аміном [2] (Amine). Це дозволяє виконувати обчислення паралельно і поєднувати результати для отримання розв’язку початкової задачі. Дропи можуть виконуватись як на початковому процесі так і

бути надісланими на виконання іншим процесам. Існує механізм розподілу дропів між вільними процесами.

1.2.2 Потоки DAP

У платформі DAP на кожному процесі запущено два потоки CalcThread і DispThread. DispThread виконується швидко і має запускатись періодично. CalcThread має працювати постійно. CalcThread виконує обчислення дропів на поточному процесорі. Тобто робота CalcThread займає більшість часу роботи програми. DispThread керує обчисленнями: надсилає дропи на виконання іншим процесам, обмінюється інформацією про вільні процеси, отримує результати, обчислені у інших процесах.

1.2.3 Важливі змінні

Клас CalcThread містить такі змінні, важливі в контексті цієї роботи:

vokzal (вокзал) – список дропів, що потрібно обчислити, розділений на рівні за глибиною дропа (номер ітерації розгортки дропа);

ownTrack – список дропів, що буде обчислено локальним процесом, тобто ці завдання не будуть надіслані іншим процесам.

1.2.4 Множення щільних матриць у DAP

Алгоритм множення матриць «розділяй і володарюй» реалізовано класом MatrDMult4. При декомпозиції (розбитті) цей дроп перетворюється на чотири дропи MatrDMultiplyScalar. Дроп MatrDMultiplyScalar обчислює суму двох результатів множення матриць. Алгоритм множення матриць Штрассена реалізовано класом MatrDMultStrassWin7. Цей дроп розбивається на сім дропів MatrDMultStrassWin7.

1.3 Огляд літератури

Використання графічних процесорів для множення матриць є відомою практикою у сфері обчислень великої продуктивності (HPC) [3]. Так згаданий вище алгоритм Штрассена було імплементовано на CUDA [4]. У [4] автори дійшли висновку, що алгоритм Штрассена має як мінімум два рівні паралелізму.

Перший рівень – це розділення матриць на чотири частини і використання MPI для паралельного обчислення у різних процесах. Другий рівень – це можливість паралельно виконувати множення матриць на відеокарті. У [5] автори пропонують механізм віртуалізації графічного процесора, щоб мати можливість виконувати операції на графічному процесорі, що знаходиться на іншому вузлі.

Розділ 2. Множення матриць на відеокарті

2.1 CUDA

Compute Unified Data Architecture (CUDA)[6] – це API від Nvidia для виконання обчислень загального призначення на графічних картах їх компанії. CUDA стала де-факто стандартним рішенням в HPC і інших індустріях для використання відеокарт для загальних обчислень - GPGPU (General-Purpose Graphics Processing Unit) [7]. Відеокарти мають масово-паралельну архітектуру. Це дозволяє значно прискорювати алгоритми, що здатні до паралелізації, наприклад, множення матриць [8]. Робота виконана на основі DAP мовою програмування Java. CUDA не підтримує Java нативно, тому використовується бібліотека JCuda[9].

2.2 cuBLAS

CUDA Basic Linear Algebra Subroutine (cuBLAS) – це бібліотека від Nvidia для роботи над задачами лінійної алгебри [10]. В тому числі вона має функціонал, що дозволяє множити матриці між собою , виконувати додавання матриць. Так, для множення двох щільних матриць з 64-бітними числами з плаваючою крапкою між собою, ми використовуємо функцію "cublasDgemm". Функція cublasDgemm [11] виконує обчислення за формулою 2.1:

$$C = \alpha * op(A) * op(B) + \beta * C \quad (2.1)$$

де C – матриця в яку ми записуємо результат;

α – скалярний множник до A;

A – перша матриця в операції множення;

B – друга матриця в операції множення;

β – скалярний множник до C;

op(M) – може набувати значень M або M^T.

Для додавання матриць використовується функція "cublasDaxpy"[12].

2.3 Підготовка даних до передачі у відеопам'ять

У DAP щільні матриці представлені класом MatrixD, що містить двовимірний масив класу Element. Перед тим, як ми можемо працювати з цими даними на відеокарті, необхідно трансформувати ці дані у формат, що підтримується cuBLAS. Це має бути одновимірний масив примітивного типу (в даному випадку double), що містить всі елементи матриці у «column-major» порядку [10]. Ця трансформація даних відбувається у методі prepareMatrixForGPUTransfer класу CudaMatrixDWrapped. Після того, як дані трансформовано у необхідний формат, вони копіюються у відеопам'ять. Коли результат обчислень готовий, його копіюють у оперативну пам'ять і виконують зворотну трансформацію – методом transferToHost класу CudaMatrixDWrapped.

2.4 Ефективне використання CUDA

2.4.1 Асинхронний API CUDA

Виконання однієї операції на відеокарті, маючи вхідні дані в оперативній пам'яті і очікуючи результат в оперативній пам'яті, потребує трьох послідовних дій:

1) копіювання вхідних даних з оперативної пам'яті у відеопам'ять;
2) виконання операції на відеокарті;
3) копіювання результатів з відеопам'яті в оперативну пам'ять.

Згідно [13] ці етапи для різних операцій можуть виконуватись одночасно. Крім цього можливе виконання різних операцій на відеокарті одночасно. CUDA API має дублюючі функції (синхронні і асинхронні). При використанні синхронних функцій комп'ютер чекатиме виконання кожної з функцій. Саме при використанні асинхронних функцій відеокарта зможе виконувати різні операції одночасно. Отже, асинхронне виконання має перевагу у швидкодії перед синхронним. Для того, щоб скористатись перевагами одночасного виконання обчислень на відеокарті, використовується механізм потоків CUDA. Незалежні операції мають окремі потоки (CUDA Streams), що дозволяє їм виконуватись одночасно. При використанні асинхронного API CUDA виникає необхідність

вказувати залежність певних операцій від інших. Прикладом може слугувати метод completeOperationAsync класу MultiplyMatrixDMatricesClient (Рисунок 2.1).

```
60     public void completeOperationAsync(int taskId) {
61         LOG.info("Received start signal. TaskId: "+taskId + " "+this.hashCode());
62         this.finishPreparations();
63         this.status = TaskStatus.InProgress;
64         // Create CUDA streams
65         // Each input matrix has own stream
66         this.firstStream = new cudaStream_t();
67         this.secondStream = new cudaStream_t();
68         this.resultStream = new cudaStream_t();
69         JCuda.cudaStreamCreate(firstStream);
70         JCuda.cudaStreamCreate(secondStream);
71         JCuda.cudaStreamCreate(resultStream);
72         // Create CUDA events
73         cudaEvent_t firstEvent = new cudaEvent_t();
74         JCuda.cudaEventCreate(firstEvent);
75         cudaEvent_t secondEvent = new cudaEvent_t();
76         JCuda.cudaEventCreate(secondEvent);
77         this.hostResourcesForInputCanBeFreedEvent = secondEvent;
78         // Move both matrices to VRAM independently
79         this.wrappedFirst.placeOnGpu(firstStream);
80         this.wrappedSecond.placeOnGpu(secondStream);
81         JCuda.cudaEventRecord(firstEvent, firstStream);
82         JCuda.cudaEventRecord(secondEvent, secondStream);
83         wrappedResult.initOnGpu(new CUstream(resultStream));
84         JCuda.cudaStreamWaitEvent(secondStream, firstEvent, flags: 0); // Second stream waits for first stream
85         JCuda.cudaStreamWaitEvent(resultStream, secondEvent, flags: 0); // Result stream waits for second stream
86         wrappedResult.multiplyAndStore(wrappedFirst, wrappedSecond, shouldClearMyself: false, resultStream);
87         cudaEvent_t multiplyingFinished = firstEvent;
88         JCuda.cudaEventRecord(multiplyingFinished, resultStream); // Record event on finishing multiply
89         // First and second streams wait for multiplyingFinished
90         JCuda.cudaStreamWaitEvent(firstStream, multiplyingFinished, flags: 0);
91         JCuda.cudaStreamWaitEvent(secondStream, multiplyingFinished, flags: 0);
92         this.wrappedFirst.releaseGpuResources(new CUstream(firstStream));
93         this.wrappedSecond.releaseGpuResources(new CUstream(secondStream));
94     }
```

Рисунок 2.1 – Використання асинхронного CUDA API у коді програми

У цьому методі ми починаємо операцію множення двох матриць. Операція переміщення вхідних матриць завершується викликом функції placeOnGpu. Для кожної матриці при виклику методу placeOnGpu ми передаємо аргументом різні потоки. Також маємо третій потік – resultStream для матриці, в якій буде збережено результат операції. На рядку №83 ми викликаємо метод initOnGpu, передаючи йому resultStream. Поки що жодних залежностей не вказано, тобто всі три операції зможуть потенційно виконуватись одночасно. Після викликів функції placeOnGpu для firstStream і secondStream викликаємо метод cudaEventRecord. На рядках №84-85 ми викликаємо cudaStreamWaitEvent метод двічі. Перший раз, щоб змусити secondStream чекати на завершення роботи

firstStream. Другий раз для того, щоб resultStream дочекався виконання всіх операцій від secondStream. Таким чином, на рядку №86 ми можемо бути впевненими, що всі три операції, що виконувались паралельно (переміщення даних з двох вхідних матриць та ініціалізація місця збереження результату) закінчили свою роботу перед викликом функції, що множить матриці (multiplyAndStore). Цю залежність потоків необхідно вказати, щоб уникнути виконання операцій на відеокарті з некоректними даними. Аналогічно, перед тим як викликати методи releaseGpuResources для вхідних матриць, ми маємо переконатись, що виконання операцій в resultStream завершено.

2.4.2 Перемикання контекстів CUDA

У DAP на кожному ядрі кластера передбачається запускати по одному процесу. У кластерному середовищі кількість ядер на процесорах зазвичай значно вище за кількість відеокарт. Так, на кластері ІК НАН України, вузол g4301 має сумарно дванадцять ядер на двох процесорах і вісім відеокарт, а вузол n5013 має 224 ядер на двох процесорах і одну відеокарту. Відповідно на кожну відеокарту припадає більше ніж один процес у DAP.

Кожен процес DAP при роботі з CUDA вимушений мати окремий CUDA контекст. Згідно [14] відеокарта може виконувати одночасно лише команди в одному контексті і переключення між контекстами має небажані негативні ефекти (затримка, витрати пам’яті). Рішення від Nvidia, яке пропонується – це використання Multiple-Process Service (MPS). Під час виконання даної роботи передбачалось, що MPS може бути присутнім чи відсутнім у кластері. У даній роботі негативні наслідки переключення контекстів взяті до уваги при розробці алгоритму координації реалізованому у класі CudaDispatcher.

Розділ 3. Імплементація алгоритмів

Програмний код DAP з внесеними змінами доступний у публічному репозиторії [15].

3.1 Опис алгоритму

Кожен процес може використовувати відеокарту для обчислень. Кожному процесу визначаємо до якої відеокарти у нього є доступ. Доступ до декількох відеокарт з одного процесу не підтримується. Для того, щоб уникнути спроб одночасного використання відеокарти різними процесами, використовується механізм координації доступу до ресурсів графічного процесора, реалізований у класі CudaDispatcher.

3.2 Інструменти координації

Для комунікації між процесами в середовищі DAP використовується OpenMPI версії 4.1.2 [16]. Для коректної взаємодії при використанні ресурсів графічного процесора було додано нові класи, які обмінюються новими повідомленнями. Крім цього комунікація, що стосується використання відеокарт, відбувається у окремому MPI комунікаторі. Повідомлення, що передаються, мають невеликий розмір (до 20 байтів). Під час ініціалізації використовуються групові операції (scatter, gather), бар'єр, двосторонні повідомлення (send, recv). Під час основної роботи середовища використовуються двосторонні асинхронні повідомлення (iSend, iRecv), методи перевірки наявності повідомлень (testAny, probeAny).

3.3 Ініціалізація

У контексті координації використання графічних процесорів за ініціалізацію відповідає клас AffinityFinder. Перед тим, як ділити відеокарти, потрібно зрозуміти, які процеси запущені на одному вузлі і які відеокарти присутні на цьому вузлі. Процес ініціалізації:

1) кожен процес виконує команду «hostname» і отримує назву свого вузла;
2) кожен процес обчислює значення хеш-функції від назви вузла;

3) всі процеси крім процеса з рангом нуль надсилають хеш-код назви вузла процесу з рангом нуль;
4) процес з рангом нуль обчислює мінімальний ранг для всіх унікальних значень хеш-коду, процеси, чий ранг є мінімальним для певних хеш-кодів, вважаються лідерами вузлів;
5) процес з рангом нуль надсилає всім процесам ранг їх лідера;
6) процес з рангом нуль записує ранги процесів, які мають такий же хеш-код, як і його – це його підлеглі;
7) процес з рангом нуль надсилає всім процесам-лідерам списки їх підлеглих;
8) всі процеси-лідери отримують списки своїх підлеглих і зберігають їх;
9) виконується команда бар'єру для синхронізації процесів;
10) глобальний комунікатор розділяється на низку комунікаторів згідно значення хеш-коду назви вузла (вся подальша комунікація у AffinityFinder відбувається у нових комунікаторах), нові комунікатори зберігаються на кожному процесі у змінну hostComm;
11) процеси-лідери дізнаються характеристики всіх доступних відеокарт на своєму вузлі;
12) процеси-лідери розподіляють всі процеси до якогось одного графічного процесора;
13) процес-лідер надсилає всім підлеглим номер відеокарти, яка була їм виділена у роботу;
14) виконується команда бар'єру.

У результаті ініціалізації у кожного процесу є об'єкт класу AffinityFinder, що містить такі поля:

hostComm – комунікатор вузла, у якому знаходиться процес;

myLeader – ранг процеса-лідера вузла, на якому знаходиться даний процес;

myGpu – номер відеокарти, яку виділено цьому процесу;

amILeader – булева змінна, що має значення true тільки, якщо цей процес є лідером свого вузла.

Процеси-лідери також мають такі поля:

subordinateRanks – ранги підпорядкованих процесів у глобальному комунікаторі;

subordinateGpus – номери графічних процесорів, визначених для кожного процесу;

initialFreeVRAM – інформація про початкову кількість вільної відеопам’яті.

3.4 Координаційна модель диспетчер-підлеглі

3.4.1 Огляд координаційної моделі

На кожному вузлі єдиний процес має об’єкт класу CudaDispatcher. Всі процеси на вузлі мають по об’єкту класу CudaExecutor. CudaExecutor комунікує з CudaDispatcher засобами MPI за виключенням процесу-лідера вузла. На процесі-лідері вузла присутні об’єкти обох класів, тому використовуються звичайні виклики API. CudaExecutor має завдання, які можна обчислити на відеокарті. CudaDispatcher координує використання відеокарт кожним процесом цього вузла через об’єкти класу CudaExecutor. Працює з відеокартою кожен процес, який отримав на це дозвіл і старт-сигнал. CudaDispatcher займається координацією цього процесу.

На рисунку 3.1 проілюстровано життєвий цикл виконання завдання на відеокарті. Отже, розглянемо детально життєвий цикл виконання завдання на відеокарті:

1) CudaExecutor отримує нове завдання, яке можна обчислити на графічному процесорі. CudaExecutor обчислює хеш-функцію від об’єкту завдання. Хеш-код виконує роль ідентифікатора на початковому етапі комунікації до отримання підтвердження. CudaExecutor надсилає CudaDispatcher

повідомлення з інформацією про завдання (хеш-код, максимальний обсяг матриці, загальні потреби у відеопам'яті, тип завдання).

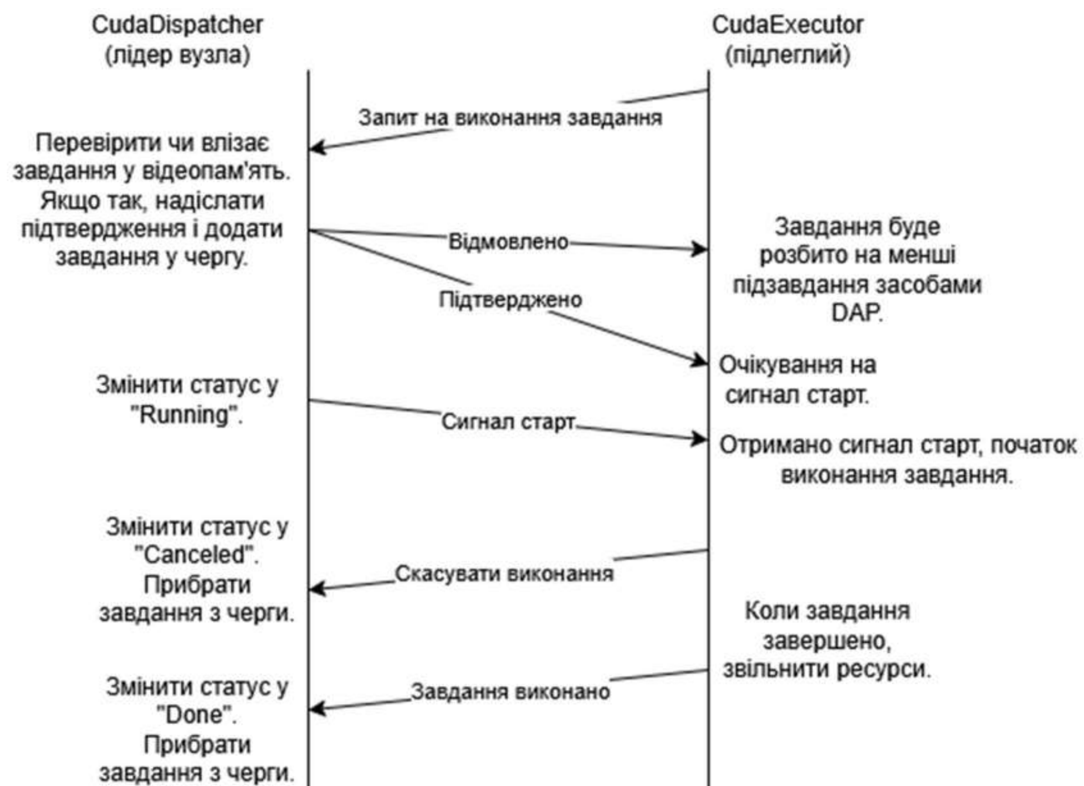


Рисунок 3.1 - Діаграма повідомлень між CudaDispatcher і CudaExecutor

2) CudaDispatcher отримує повідомлення з запитом на виконання завдання. Спершу перевіряється, чи можна виділити у відеопам'яті місце під всі необхідні матриці для цієї задачі, якщо ні, то надсилається відмова. Якщо місця достатньо, то перевіряємо, чи не перевищено максимальний обсяг пам'яті, необхідний для матриці (2^31-1 байтів). Обмеження пов'язане з тим, що клас ByteBuffer зберігає довжину буфера у змінній типу Integer. Кожна відеокарта має окремі черги завдань. Якщо перевищення обсягу пам'яті немає, додаємо завдання у чергу підтверджених потрібної відеокарти. Далі генеруємо унікальний ідентифікатор taskId для завдання. Незалежно від того, чи було підтверджене завдання чи ні, надсилаємо повідомлення процесу, що надіслав запит, з наступним вмістом (відповідь на запит, taskId, хеш-код).

3) Якщо CudaDispatcher відмовив завданню, його життєвий цикл завершується. Пізніше воно буде розбито на декілька підзавдань меншого розміру.
4) Якщо CudaDispatcher підтвердив, що завдання буде виконано, то CudaExecutor буде асинхронно очікувати на стартовий сигнал від CudaDispatcher або на сигнал переривання.
5) Якщо завдання перервано, його не буде виконано на відеокарті, інформація про це надсилається CudaDispatcher. Це кінець життєвого циклу завдання.
6) У певний момент CudaDispatcher надсилає сигнал про старт завдання.
7) CudaExecutor запускає завдання на відеокарті і періодично асинхронно перевіряє, чи завдання завершило виконання.
8) Коли CudaExecutor стало відомо, що завдання на відеокарті завершено, повідомлення про це надсилається CudaDispatcher. Це кінець життєвого циклу завдання.

3.4.2 CudaDispatcher

Для кожного наявного графічного процесора маємо окремі черги завдань: acceptedTasks, runningTasks, finishedTasks. Періодично викликається метод checkChanges, який перевіряє, чи можна надсилати старт-сигнал для завдань у черзі, не порушуючи наявних обмежень. Для кожної відеокарти checkChanges перевіряє очікувану кількість вільної відеопам’яті (від загального обсягу пам’яті віднімається сума витрат відеопам’яті для завдань що наразі запущені) і знаходить найбільше завдання, що може розміститись у вільній відеопам’яті. Для цього завдання надсилається стартовий сигнал процесу, що надіслав завдання. Для того, щоб уникнути постійного переключення контекстів CUDA, CudaDispatcher стартує завдання для процесу, що зараз використовує відеокарту, коли такі завдання закінчуються, CudaDispatcher обирає інший процес і його завдання найбільшого розміру.

3.4.3 CudaExecutor

DAP взаємодіє з функціоналом прискорення на відеокартах саме з CudaExecutor класом. Важливі методи класу:

checkMessages – перевіряє наявність повідомлень MPI з CudaDispatcher і оброблює повідомлення, що надходять;

startClient – переводить завдання у статус «InProgress» або «ReadyToStart» (в залежності від того, чи все підготовлено до початку виконання коду на відеокарті);

accelerateDrop – початковий метод для виконання завдання на графічному процесорі, що створює запит до CudaDispatcher;

slowProcess – метод, що виконує підготовчий етап для завдань, що перебувають у статусі «ReadyToStart», переводить статус у «InProgress» і запускає ці завдання асинхронно;

quickProcess – метод, що перевіряє, чи можна очищати проміжні дані у оперативній пам'яті і якщо так, виконує очищення.

cancelClient – змінює статус завдання у «Canceled», надсилає сигнал CudaDispatcher.

checkForResults – перевіряє, які завдання, запущені на відеокарті, виконані, копіює дані з відеопам'яті у оперативну пам'ять, трансформує дані у потрібний формат (MatrixD), повертає список виконаних завдань.

3.5 Інтеграція у DAP

3.5.1 Інтеграція CudaDispatcher

CudaDispatcher має постійно перевіряти наявність повідомлень від багатьох процесів, тому об'єкт цього класу має існувати у DispThread. DispThread теж регулярно оброблює повідомлення, тобто постійно викликається метод execute. У CudaDispatcher для обробки вхідних повідомлень і подальших дій створено метод loopStep. Тому код для перевірки наявності нових повідомлень для CudaDispatcher знаходиться поряд з кодом, що перевіряє наявність повідомлень для DispThread. CudaDispatcher і DispThread

використовують різні комунікатори і різні теги для повідомлень, щоб уникнути некоректної роботи середовища. Маємо цикл, що працює, доки є хоча б одне необроблене повідомлення, що надійшло CudaDispatcher.

3.5.2 Інтеграція CudaExecutor

3.5.2.1 accelerateDrop

У класі CalcThread (обчислювальний потік DAP) є методи putDropInTrack, putDropInVokzal, що додають обчислювальні завдання (об’єкти класу Drop) у ownTrack і vokzal змінні. В обидва методи додаємо код, що за можливості робить запит на прискорення завдань (метод accelerateDrop класу CudaExecutor). Тобто всі завдання, що можуть бути прискорені на відеокарті, ми спробуємо прискорити. для всіх об’єктів класу Drop, які ми прискорюємо, заповнюється змінна cudaClient посиланням на об’єкт класу MatrixDClient. Цей клас містить змінні і методи що стосуються взаємодії з CUDA API.

Було реалізовано можливість прискорення для завдань множення щільних матриць типу NumberR64 (тип double). Це дропи класів MatrDMult4, MatrDMultStrassWin7, MatrDMultiplyScalar. MatrDMult4 і MatrDMultStrassWin7 прискорюються ідентично – як множення двох матриць. MatrDMultiplyScalar – це сума множення двох пар матриць.

3.5.2.2 Порядок діставання дропів з вокзалу і ownTrack

Функція ProcFunc у CalcThread постійно намагається брати завдання на обчислення. Це відбувається у методі getTask класу CalcThread. До змін, зроблених в рамках цієї роботи, з вокзалу діставались завдання по черзі. У поточній роботі змінено порядок діставання дропів з вокзалу, за це відповідає метод chooseNextDrop. Після того як ми ініціюємо прискорення завдань на відеокарті, ми їх не прибираємо з вокзалу.

Метод ProcFunc виконує обчислення на класичному процесорі. Щоб уникнути виконання одного й того ж завдання на класичному процесорі і на відеокарті, коли завдання надходить у ProcFunc, надсилаємо сигнал скасування для переривання виконання на відеокарті. Ми хочемо зменшити кількість дропів, для яких буде надіслано сигнал скасування. Тому метод chooseNextDrop

має обрати, яке обчислювальне завдання повернути з вокзалу. Метод chooseNextDrop реалізує такий пріоритет повернення завдань з вокзалу:

1) завдання, які вже обчисленні на графічному процесорі (статус Done);
2) завдання, для яких не було ініційовано прискорення на відеокарті;
3) завдання, яким було відмовлено у виконанні прискорення (статус Rejected);
4) нове завдання, для якого ще не відомо, чи воно підтверджене чи ні (статуси New, RequestSent);
5) підтверджені завдання (статус Accepted);
6) завдання, що отримали стартовий сигнал, але які не почали виконання на відеокарті (статус ReadyToStart);
7) всі інші завдання (запущені на відеокарті).

На відміну від вокзалу з ownTrack завдання забираються у тому ж порядку, але є додаткова перевірка, яка не дозволяє повернення завдання, що було запущено на відеокарті.

3.5.2.3 Підготовка даних до обчислення на відеокарті

Як було вказано у розділі 2.3 для того, щоб виконувати операції на відеокарті, ми маємо трансформувати дані і перенести їх у відеопам'ять. Ці операції можуть бути тривалими для матриць великого обсягу, тому їх не можна виконувати у DispThread. Ці операції запускаються методом slowProcess класу CudaExecutor. Метод slowProcess викликається у CalcThread перед тим, як виконувати ProcFunc. Метод quickProcess класу CudaExecutor запускається поряд з slowProcess у CalcThread.

3.5.2.4 Отримання результатів обчислених на графічному процесорі

Метод checkForResults класу CudaExecutor перевіряє, які операції на відеокарті вже виконано. Для цих завдань дані необхідно записати у поле outData класу Drop. За це відповідає функція dropAfterCudaProcess класу CalcThread. Виклик методу dropAfterCudaProcess для завдань, повернутих методом checkForResults відбувається поряд з quickProcess.

Розділ 4. Експерименти

4.1 Опис експериментів

Метою експериментів є дізнатись, як змінилась швидкодія операції множення щільних матриць у DAP з впровадженням прискорення на графічному процесорі. У DAP для щільних матриць реалізовано два блоково-рекурсивних алгоритми: MatrDMult4, MatrDMultStrassWin7. Обидва вони отримали прискорення на відеокарті. Для вимірювання метрик було використано існуючий у DAP функціонал вимірювання часу виконання, витрат оперативної пам’яті. Цей функціонал було модифіковано, щоб вимірювати додаткову метрику, пов’язану з обслуговуванням обчислень на графічному процесорі. Крім цього, наступною метою є дізнатись в яких межах розмірів вхідних матриць, використання прискореної версії DAP зменшує час виконання програми.

Всі запуски експериментів використовували максимальну можливу кількість оперативної пам’яті. Для запусків з більше ніж одним процесом, загальний обсяг оперативної пам’яті ділився на кількість процесорів і передавався аргументом “-Xmx” до Java.

4.2 Середовище виконання експериментів

Обчислення були виконані за підтримки обчислювального комплексу СКІТ Інституту кібернетики НАН України [17]. Для запуску експериментів було використано вузли g4301, n5013 з розділу scit5ai. Дані про характеристики вузлів (Таблиця 4.1) були отримані шляхом виконання команд «scontrol show node», «nvidia-smi».

Таблиця 4.1 – Характеристики вузлів у кластері

Вузол	**Обсяг оперативної пам'яті (Гб)**	**Обсяг відеопам'яті (Гб)**	**Кількість ядер процесора**	**Кількість відеокарт**	**Відеокарта**
g4301	56	12	12	8	RTX 2080Ti
n5013	256	48	224	1	RTX A6000

На вузлі g4301 встановлено два сокети з шестиядерними процесорами Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz. На вузлі n5013 існують два сокети з процесорами AMD EPYC 9734 112-Core Processor, що підтримують два потоки на ядро.

4.3 Використані метрики

Для оцінки швидкодії DAP було обрано такі метрики:

- загальний час, витрачений на обчислення;
- час, витрачений на підготовчі дії для роботи на відеокарті (трансформація даних матриць, переміщення даних вхідних матриць з оперативної пам’яті у відеопам’ять, переміщення результатів обчислень назад);
- максимальний обсяг оперативної пам’яті, що було використано під час обчислень на одному процесі.

Варто зауважити, що для передачі даних у відеопам’ять використовуються буфери, виділені поза купою Java. Пам’ять, виділена таким чином (поза купою Java , не буде зареєстрована у метриці витрат оперативної пам’яті.

4.4 Результати експериментів

4.4.1 Вузол g4301

Розглянемо результати експериментів для вузла g4301.
Для запусків без прискорення на відеокарті (Рисунок 4.1) бачимо, що алгоритм Штрассена спрацьовує швидше за алгоритм «розділяй і володарюй» (що очікувано, враховуючи різницю у обчислюваній складності). Також бачимо, що запуск на дванадцяти ядрах процесора спрацьовує швидше ніж на одному (теж очікувано).

Для запусків з прискоренням на відеокартах бачимо, що дані практично ідентичні для різних алгоритмів і запуски на дванадцяти процесорах ефективніші за запуски на одному процесорі (Рисунок 4.2). Дослідивши логи детальніше, видно, що у запусках з прискоренням головний дроп виконувався на графічному процесорі повністю (без розбиття на менші дропи). Це пояснює майже однакові дані для різних алгоритмів. Також видно, що для запусків на

дванадцяти процесорах лише один з процесів виконував обчислення, бо початковий дроп не був розбитий і розподілений між усіма дропами.

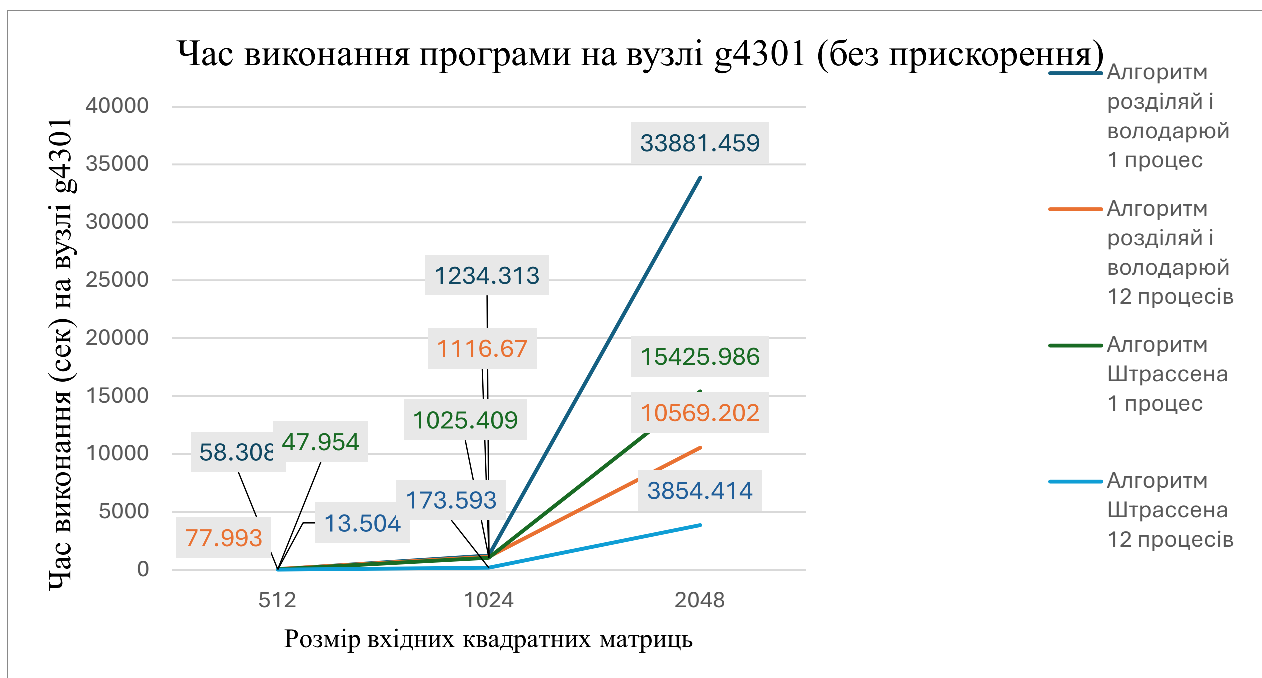


Рисунок 4.1 – Час виконання програми без прискорення на вузлі g4301

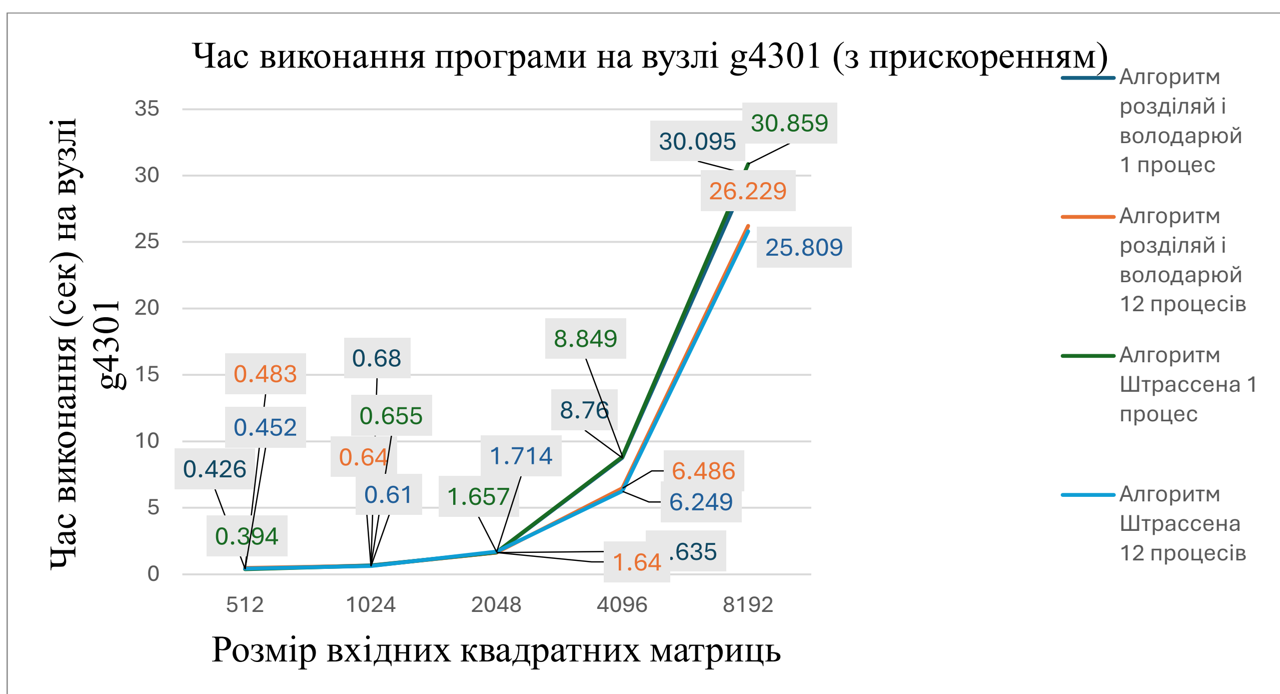


Рисунок 4.2 – Час виконання програми з прискоренням на вузлі g4301

При порівнянні даних для запусків з прискоренням на відеокартах з даними запусків без прискорення бачимо значне пришвидшення виконання програми (Таблиця 4.2).

Таблиця 4.2 – Час виконання програми на вузлі g4301

Алгоритм	**Вузол**	**Чи є GPU**	**Кількість процесорів**	**Розмір вхідних матриць**	**Час виконання (сек)**	**Час обслуговання GPU (сек)**	**RAM (mb)**
MatrDMult4	g4301	Ні	1	512	58.308	-	54
MatrDMult4	g4301	Ні	1	1024	1234.313	-	152
MatrDMult4	g4301	Ні	1	2048	33881.459	-	543
MatrDMult4	g4301	Ні	1	4096	Не виконалось за 24 години		
MatrDMult4	g4301	Ні	12	512	77.993	-	36
MatrDMult4	g4301	Ні	12	1024	1116.67	-	124
MatrDMult4	g4301	Ні	12	2048	10569.202	-	437
MatrDMult4	g4301	Ні	12	4096	Не виконалось за 24 години		
MatrDMult4	g4301	Так	1	512	0.426	0.173	32
MatrDMult4	g4301	Так	1	1024	0.68	0.251	95
MatrDMult4	g4301	Так	1	2048	1.635	0.596	347
MatrDMult4	g4301	Так	1	4096	8.76	5.79	1363
MatrDMult4	g4301	Так	1	8192	30.095	16.478	5445
MatrDMult4	g4301	Так	1	16384	Закінчилась оперативна пам'ять		
MatrDMult4	g4301	Так	12	512	0.483	0.28	29
MatrDMult4	g4301	Так	12	1024	0.64	0.322	92
MatrDMult4	g4301	Так	12	2048	1.64	0.89	344
MatrDMult4	g4301	Так	12	4096	6.486	3.602	1353
MatrDMult4	g4301	Так	12	8192	26.229	14.891	5385
MatrDMult4	g4301	Так	12	16384	Закінчилась оперативна пам'ять		
MatrDMultStrassWin7	g4301	Ні	1	512	47.954	-	113
MatrDMultStrassWin7	g4301	Ні	1	1024	1025.409	-	253
MatrDMultStrassWin7	g4301	Ні	1	2048	15425.986	-	935
MatrDMultStrassWin7	g4301	Ні	12	512	13.504	-	52
MatrDMultStrassWin7	g4301	Ні	12	1024	173.593	-	239
MatrDMultStrassWin7	g4301	Ні	12	2048	3854.414	-	881
MatrDMultStrassWin7	g4301	Так	1	512	0.394	0.167	32
MatrDMultStrassWin7	g4301	Так	1	1024	0.655	0.247	95
MatrDMultStrassWin7	g4301	Так	1	2048	1.657	0.639	347
MatrDMultStrassWin7	g4301	Так	1	4096	8.849	5.789	1363
MatrDMultStrassWin7	g4301	Так	1	8192	30.859	16.259	5445
MatrDMultStrassWin7	g4301	Так	1	16384	Закінчилась оперативна пам'ять		
MatrDMultStrassWin7	g4301	Так	12	512	0.452	0.249	29
MatrDMultStrassWin7	g4301	Так	12	1024	0.61	0.287	92
MatrDMultStrassWin7	g4301	Так	12	2048	1.714	0.971	344
MatrDMultStrassWin7	g4301	Так	12	4096	6.249	3.39	1353
MatrDMultStrassWin7	g4301	Так	12	8192	25.809	14.566	5385
MatrDMultStrassWin7	g4301	Так	12	16384	Закінчилась оперативна пам'ять		

Час виконання відрізняється дуже сильно. Так, найменший коефіцієнт прискорення – 30 (тобто найгірше прискорення у 30 разів) (Таблиця 4.3). Коефіцієнт прискорення – це співвідношення часу виконання програми з

використанням прискорення на відеокартах до часу виконання програми без прискорення.

Таблиця 4.3 – Коефіцієнти прискорення швидкодії програми з впровадженням обчислень на графічному процесорі (на вузлі g4301)

Алгоритм	**Вузол**	**Кількість процесів**	**Розмір вхідних матриць**	**Коефіцієнт прискорення (округлений до одиниць)**
MatrDMult4	g4301	1	512	137
MatrDMult4	g4301	1	1024	1815
MatrDMult4	g4301	1	2048	20723
MatrDMult4	g4301	12	512	161
MatrDMult4	g4301	12	1024	1745
MatrDMult4	g4301	12	2048	6445
MatrDMultStrassWin7	g4301	1	512	122
MatrDMultStrassWin7	g4301	1	1024	1566
MatrDMultStrassWin7	g4301	1	2048	9310
MatrDMultStrassWin7	g4301	12	512	30
MatrDMultStrassWin7	g4301	12	1024	285
MatrDMultStrassWin7	g4301	12	2048	2249

Деякі запуски без прискорення не виконались за 24 години і були перервані середовищем вузла. Запуски, що завершились помилкою OutOfMemoryError, вказано у таблиці 4.2. Для вузла g4301 граничний розмір матриць, для якого вистачало оперативної пам’яті, – це 8192 (за результатом експериментальних запусків). Матриці розміром 8192 все ще повністю вміщаються у відеопам’ять, тому їх множення виконується однією операцією.

4.4.2 Вузол n5013

Вузол n5013 має більший об'єм оперативної пам'яті ніж вузол g4301(Таблиця 4.1). Як бачимо (таблиця 4.4), граничний ліміт розміру вхідних матриць – 16384 для алгоритму «розділяй і володарюй». Детальніше оглянувши логи програми, приходимо до висновку, що лише для експерименту з вхідними матрицями розміру 16384 початковий дроп не може бути виконано повністю на відеокарті без розбиття. Під час виконання експерименту з розміром 16384 початковий дроп розбивається на чотири дропи типу MatrDMultiplyScalar, що одночасно виконуються на графічному процесорі. Причина, за якої початкових дроп не було виконано одразу, – це обмеження на обсяг пам'яті, яку ми виділяємо командою allocateDirect і зберігаємо у ByteBuffer.

Таблиця 4.4 – Час виконання на вузлі n5013 на одному процесі з прискоренням відеокартою

Алгоритм	Вузол	Розмір вхідних матриць	Час виконання (сек)	Час обслуговання GPU (сек)	RAM (mb)
MatrDMult4	n5013	512	0.19	0.073	39
MatrDMult4	n5013	1024	0.29	0.101	102
MatrDMult4	n5013	2048	0.78	0.224	352
MatrDMult4	n5013	4096	2.83	1.278	1362
MatrDMult4	n5013	8192	10.4	3.649	5431
MatrDMult4	n5013	16384	93.2	45.98	23102
MatrDMult4	n5013	32768	Закінчилась оперативна пам'ять		
MatrDMultStrassWin7	n5013	512	0.19	0.069	39
MatrDMultStrassWin7	n5013	1024	0.31	0.101	102
MatrDMultStrassWin7	n5013	2048	0.82	0.247	352
MatrDMultStrassWin7	n5013	4096	2.92	1.354	1362
MatrDMultStrassWin7	n5013	8192	10.4	3.658	5431
MatrDMultStrassWin7	n5013	16384	Закінчилась оперативна пам'ять		

4.4.3 Інтерпретація результатів експериментів

Головна причина запуску експериментів з одним процесом у кластерному середовищі – це можливість виділити максимальний об’єм оперативної пам’яті (56Гб і 256Гб для різних вузлів). Навіть з виділенням всієї оперативної пам’яті вузла одному процесу, матриці розмірів, що не перевищують граничний розмір, зазвичай достатньо невеликі, щоб операція їх множення могла бути повністю виконана на відеокарті (виконання початкового дропу без розбиття). Винятком з цього спостереження є експеримент на вузлі n5013 з вхідним розміром 16384. У цьому експерименті початковий дроп розбивається на чотири дропи меншого типу, які одночасно виконуються на графічному процесорі.

Під час виконання цієї роботи, існувало припущення, що матриці до певного граничного розміру зможуть швидко виконуватись на відеокарті, а матриці більші за граничний розмір будуть розбиватись на дропи меншого розміру. Після чого, дропи менших розмірів будуть одночасно обчислюватись на різних процесах, на відеокартах (тобто в тому числі на відеокартах що асоційовано з іншими процесами) і стандартним методом - на багатьох ядрах процесорів. Для цього очікуваного сценарію і було розробленого алгоритм координації доступу до відеокарт, ефективне використання CUDA API.

В результаті експериментів, було виявлено що цей граничний розмір для обох вузлів (g4301, n5013) – це 8192. Але, дослідити поведінку програми ускладнено тим, що у нас закінчується оперативна пам’ять при роботі з матрицями розмірів більшого за граничний (8192), крім експерименту-винятку згаданого вище.

Для підтримки обчислень на матрицях більших розмірів ніж поточний граничний розмір (8192), пропонується скористатись альтернативним (до MatrixD) класом репрезентації матриць FileMatrixD (або, для розріджених матриць класом SFileMatrix). Ці класи не тримають вміст матриці постійно у оперативній пам’яті, основним місцем зберігання даних є файлова система. Використання FileMatrixD і SFileMatrix класів, може бути перспективним шляхом збільшення граничного розміру матриць, що підтримуються DAP.

Було встановлено, що на всьому діапазоні вхідних розмірів щільних матриць (які можна використовувати в DAP) на вузлі g4301, прискорення на відеокартах значно (як мінімум у 30 разів) пришвидшує швидкодію програми (Таблиця 4.2).

Висновки

Було розглянуто блоково-рекурсивні алгоритми для множення матриць «розділяй і пануй» і алгоритм Штрассена; ознайомлено з їх представленням у середовищі DAP. Обрано бібліотеки CUDA, cuBLAS для реалізації алгоритмів множення щільних матриць на відеокарті.

Запропоновано і реалізовано алгоритм диспетчера доступу до відеокарт для кластерного середовища. Було реалізоване прискорення на графічному процесорі алгоритмів «розділяй і пануй» і Штрассена. Для прискорення обчислень на відеокарті було використано асинхронне CUDA API для покращення ефективності програми.

Проведено експериментальні запуски DAP для обох алгоритмів множення на двох вузлах кластера з різною конфігурацією (різна кількість відеокарт, різні процесори). Виміряно значний приріст швидкодії програми у порівнянні з версією без прискорення на відеокартах. Було встановлено, що нова версія DAP з прискоренням на графічних процесорах працює швидше для задачі множення щільних матриць на всьому діапазоні можливих розмірів вхідних матриць, що наразі підтримується DAP.

Для подальших досліджень рекомендуються такі теми:

- поєднання поточного рішення прискорення на відеокартах з альтернативними способами зберігання матричних даних у DAP (наприклад, у файловій системі);
- оптимізація витрат оперативної пам'яті у DAP (у тому числі додавання логіки економії оперативної пам'яті у диспетчер відеокарт);
- реалізація прискорення на тензор-ядрах для матриць;
- реалізація прискорення інших (крім множення) алгоритмів на графічних процесорах.

Список використаної літератури

1) 2.5 Matrix multiplication // Algorithms / Авт.: S. Dasgupta, C. Papadimitriou, U. Vazirani. – [Б. м.], 2006. – С. 62–63.
2) Malaschonok G. I. Supercomputer Environment for Recursive Matrix Algorithms [Електронний ресурс] / Gennadi I. Malaschonok, Alla Sidko. – [Б. м. : б. в.], 2023. – 24 с. – (Препринт / National University of Kyiv Mohyla Academy ; arXiv:2303.11017). – Режим доступу: https://doi.org/10.48550/arXiv.2303.11017 (дата звернення: 08.06.2025). – Назва з екрана.
3) HPC Developer | NVIDIA Developer [Електронний ресурс] // NVIDIA Developer. – Режим доступу: https://developer.nvidia.com/hpc (дата звернення: 08.06.2025). – Назва з екрана.
4) Karunadasa N. P. Accelerating high performance applications with CUDA and MPI [Електронний ресурс] / N. P. Karunadasa, D. N. Ranasinghe // 2009 International Conference on Industrial and Information Systems (ICIIS 2009), Sri Lanka, 28–31 груд. 2009 р. – [Б. м.], 2009. – Режим доступу: https://doi.org/10.1109/iciinfs.2009.5429842 (дата звернення: 08.06.2025). – Назва з екрана.
5) A complete and efficient CUDA-sharing solution for HPC clusters [Електронний ресурс] / Antonio J. Peña [та ін.] // Parallel Computing. – 2014. – Т. 40, № 10. – С. 574–588. – Режим доступу: https://doi.org/10.1016/j.parco.2014.09.011 (дата звернення: 08.06.2025). – Назва з екрана.
6) NVIDIA CUDA [Електронний ресурс] // NVIDIA Docs. – Режим доступу: https://docs.nvidia.com/cuda/doc/index.html (дата звернення: 08.06.2025). – Назва з екрана.
7) Wang J. CUDA is Still a Giant Moat for NVIDIA [Електронний ресурс] / James Wang // Weighty Thoughts | James Wang | Substack. – Режим доступу: https://weightythoughts.com/p/cuda-is-still-a-giant-moat-for-nvidia (дата звернення: 08.06.2025). – Назва з екрана.

8) FUJIMOTO N. DENSE MATRIX-VECTOR MULTIPLICATION ON THE CUDA ARCHITECTURE [Електронний ресурс] / NORIYUKI FUJIMOTO // Parallel Processing Letters. – 2008. – Т. 18, № 04. – С. 511–530. – Режим доступу: https://doi.org/10.1142/s0129626408003545 (дата звернення: 08.06.2025). – Назва з екрана.
9) jcuda.org - JCuda [Електронний ресурс] // jcuda.org - Java bindings for CUDA. – Режим доступу: http://www.jcuda.org/jcuda/JCuda.html (дата звернення: 08.06.2025). – Назва з екрана.
10) 1. Introduction - cuBLAS 12.9 documentation [Електронний ресурс] // NVIDIA Documentation Hub - NVIDIA Docs. – Режим доступу: https://docs.nvidia.com/cuda/cublas/ (дата звернення: 08.06.2025). – Назва з екрана.
11) 1. Introduction - cuBLAS 12.9 documentation [Електронний ресурс] // NVIDIA Documentation Hub - NVIDIA Docs. – Режим доступу: https://docs.nvidia.com/cuda/cublas/index.html#cublas-t-gemm (дата звернення: 08.06.2025). – Назва з екрана.
12) 1. Introduction - cuBLAS 12.9 documentation [Електронний ресурс] // NVIDIA Documentation Hub - NVIDIA Docs. – Режим доступу: https://docs.nvidia.com/cuda/cublas/index.html#cublas-t-axpy (дата звернення: 08.06.2025). – Назва з екрана.
13) NVIDIA Corporation. Chapter 6.2.8 Asynchronous Concurrent Execution [Електронний ресурс] / NVIDIA Corporation // CUDA C++ Programming Guide. – [Б. м.], 2024. – С. 46–50. – Режим доступу: https://docs.nvidia.com/cuda/archive/12.6.1/pdf/CUDA_C_Programming_Guide.pdf (дата звернення: 08.06.2025). – Назва з екрана.
14) NVIDIA Corporation. 13.6. Multiple contexts [Електронний ресурс] / NVIDIA Corporation // CUDA C++ Best Practices Guide. – [Б. м.], 2024. – С. 65–66. – Режим доступу: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf (дата звернення: 08.06.2025). – Назва з екрана.

15) DAP01 runtime [Електронний ресурс] // Bitbucket | Git solution for teams using Jira. – Режим доступу: https://bitbucket.org/mathpar/dap01/src/master/src/main/java/com/mathpar/parallel/dapGpu/ (дата звернення: 08.06.2025). – Назва з екрана.

16) Open MPI: Version 4.1 [Електронний ресурс] // Open MPI: Open Source High Performance Computing. – Режим доступу: https://www.open-mpi.org/software/ompi/v4.1/ (дата звернення: 08.06.2025). – Назва з екрана.

17) Енергоефективний суперкомп'ютер СКІТ-4 / А.Л. Головинський, А.Л. Маленко, І.В. Сергієнко, В.Г. Тульчинський // Вісн. НАН України. — 2013. — № 2. — С. 50-59. http://dspace.nbuv.gov.ua/handle/123456789/43042

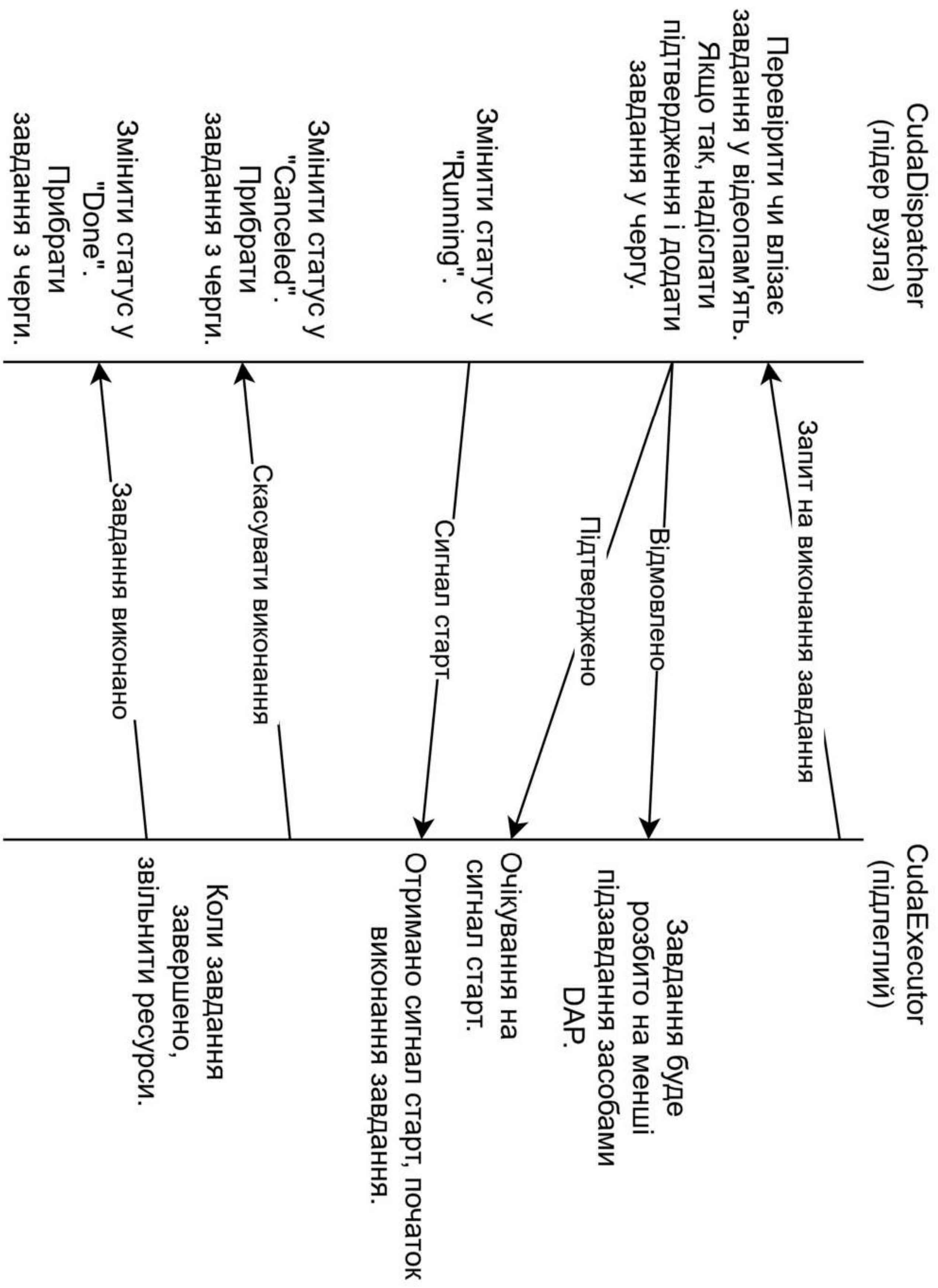
Додаток А

Рисунок 2.1 – Використання асинхронного CUDA API у коді програми

```java
public void completeOperationAsync(int taskId) {
    LOG.info("Received start signal. TaskId: "+taskId + " "+this.hashCode());
    this.finishPreparations();
    this.status = TaskStatus.InProgress;
    // Create CUDA streams
    // Each input matrix has own stream
    this.firstStream = new cudaStream_t();
    this.secondStream = new cudaStream_t();
    this.resultStream = new cudaStream_t();
    JCuda.cudaStreamCreate(firstStream);
    JCuda.cudaStreamCreate(secondStream);
    JCuda.cudaStreamCreate(resultStream);
    // Create CUDA events
    cudaEvent_t firstEvent = new cudaEvent_t();
    JCuda.cudaEventCreate(firstEvent);
    cudaEvent_t secondEvent = new cudaEvent_t();
    JCuda.cudaEventCreate(secondEvent);
    this.hostResourcesForInputCanBeFreedEvent = secondEvent;
    // Move both matrices to VRAM independently
    this.wrappedFirst.placeOnGpu(firstStream);
    this.wrappedSecond.placeOnGpu(secondStream);
    JCuda.cudaEventRecord(firstEvent, firstStream);
    JCuda.cudaEventRecord(secondEvent, secondStream);
    wrappedResult.initOnGpu(new CUstream(resultStream));
    JCuda.cudaStreamWaitEvent(secondStream, firstEvent, flags: 0); // Second stream waits for first stream
    JCuda.cudaStreamWaitEvent(resultStream, secondEvent, flags: 0); // Result stream waits for second stream
    wrappedResult.multiplyAndStore(wrappedFirst, wrappedSecond, shouldClearMyself: false, resultStream);
    cudaEvent_t multiplyingFinished = firstEvent;
    JCuda.cudaEventRecord(multiplyingFinished, resultStream); // Record event on finishing multiply
    // First and second streams wait for multiplyingFinished
    JCuda.cudaStreamWaitEvent(firstStream, multiplyingFinished, flags: 0);
    JCuda.cudaStreamWaitEvent(secondStream, multiplyingFinished, flags: 0);
    this.wrappedFirst.releaseGpuResources(new CUstream(firstStream));
    this.wrappedSecond.releaseGpuResources(new CUstream(secondStream));
}
```

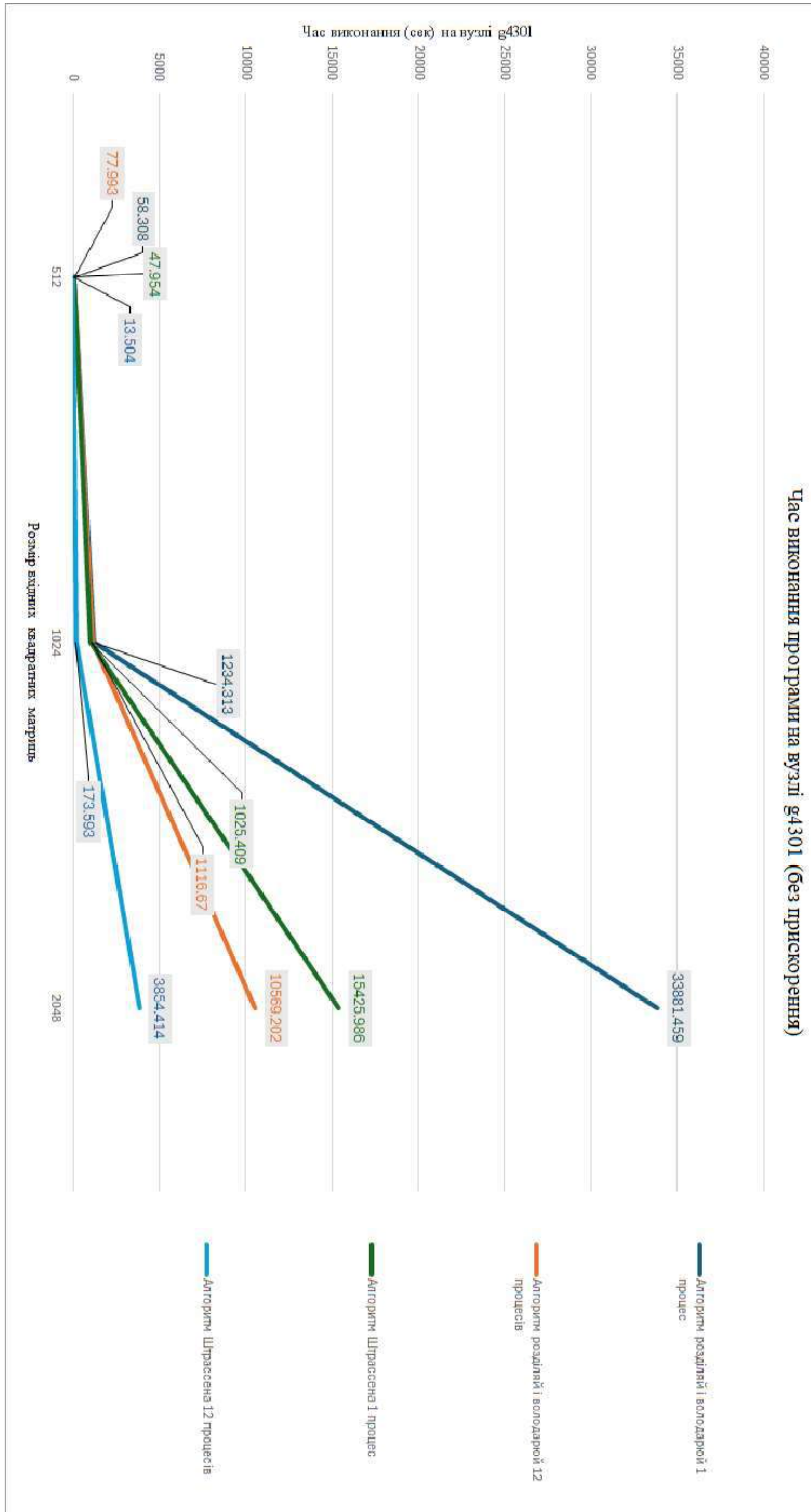
Додаток Б

Рисунок 3.1 – Діаграма повідомлень між CudaDispatcher і CudaExecutor



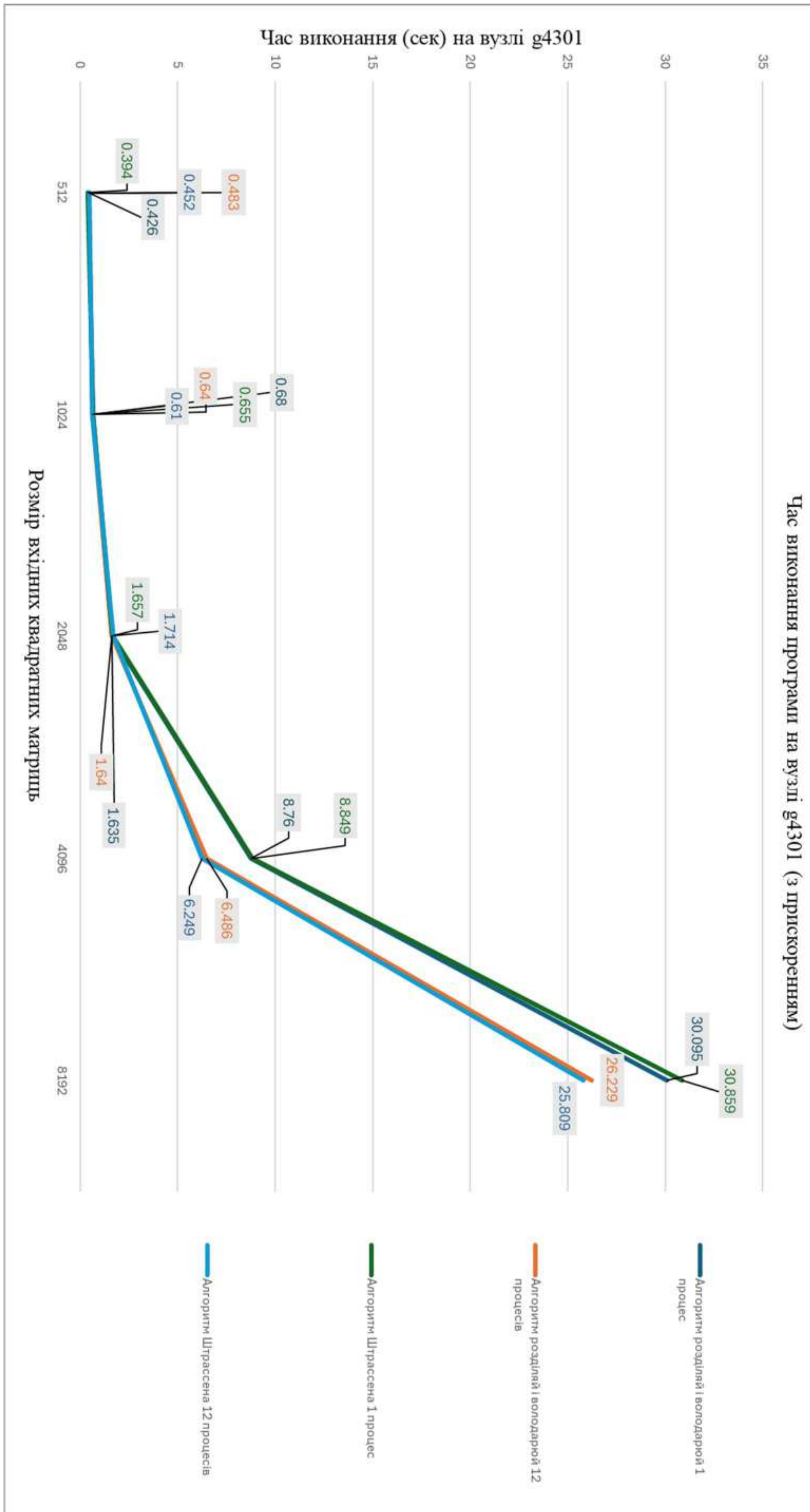
Додаток В

Рисунок 4.1 – Час виконання програми без прискорення на вузлі g4301



Додаток Г

Рисунок 4.2 – Час виконання програми з прискоренням на вузлі g4301



Додаток Д

Таблиця 4.2 – Час виконання програми на вузлі g4301

Алгоритм	**Вузол**	**Чи є GPU**	**Кількість процесорів**	**Розмір вхідних матриць**	**Час виконання (сек)**	**Час обслуговання GPU (сек)**	**RAM (mb)**
MatrDMult4	g4301	Ні	1	512	58.308	-	54
MatrDMult4	g4301	Ні	1	1024	1234.313	-	152
MatrDMult4	g4301	Ні	1	2048	33881.459	-	543
MatrDMult4	g4301	Ні	1	4096	Не виконалось за 24 години		
MatrDMult4	g4301	Ні	12	512	77.993	-	36
MatrDMult4	g4301	Ні	12	1024	1116.67	-	124
MatrDMult4	g4301	Ні	12	2048	10569.202	-	437
MatrDMult4	g4301	Ні	12	4096	Не виконалось за 24 години		
MatrDMult4	g4301	Так	1	512	0.426	0.173	32
MatrDMult4	g4301	Так	1	1024	0.68	0.251	95
MatrDMult4	g4301	Так	1	2048	1.635	0.596	347
MatrDMult4	g4301	Так	1	4096	8.76	5.79	1363
MatrDMult4	g4301	Так	1	8192	30.095	16.478	5445
MatrDMult4	g4301	Так	1	16384	Закінчилась оперативна пам'ять		
MatrDMult4	g4301	Так	12	512	0.483	0.28	29
MatrDMult4	g4301	Так	12	1024	0.64	0.322	92
MatrDMult4	g4301	Так	12	2048	1.64	0.89	344
MatrDMult4	g4301	Так	12	4096	6.486	3.602	1353
MatrDMult4	g4301	Так	12	8192	26.229	14.891	5385
MatrDMult4	g4301	Так	12	16384	Закінчилась оперативна пам'ять		
MatrDMultStrassWin7	g4301	Ні	1	512	47.954	-	113
MatrDMultStrassWin7	g4301	Ні	1	1024	1025.409	-	253
MatrDMultStrassWin7	g4301	Ні	1	2048	15425.986	-	935
MatrDMultStrassWin7	g4301	Ні	12	512	13.504	-	52
MatrDMultStrassWin7	g4301	Ні	12	1024	173.593	-	239
MatrDMultStrassWin7	g4301	Ні	12	2048	3854.414	-	881
MatrDMultStrassWin7	g4301	Так	1	512	0.394	0.167	32
MatrDMultStrassWin7	g4301	Так	1	1024	0.655	0.247	95
MatrDMultStrassWin7	g4301	Так	1	2048	1.657	0.639	347
MatrDMultStrassWin7	g4301	Так	1	4096	8.849	5.789	1363
MatrDMultStrassWin7	g4301	Так	1	8192	30.859	16.259	5445
MatrDMultStrassWin7	g4301	Так	1	16384	Закінчилась оперативна пам'ять		
MatrDMultStrassWin7	g4301	Так	12	512	0.452	0.249	29
MatrDMultStrassWin7	g4301	Так	12	1024	0.61	0.287	92
MatrDMultStrassWin7	g4301	Так	12	2048	1.714	0.971	344
MatrDMultStrassWin7	g4301	Так	12	4096	6.249	3.39	1353
MatrDMultStrassWin7	g4301	Так	12	8192	25.809	14.566	5385
MatrDMultStrassWin7	g4301	Так	12	16384	Закінчилась оперативна пам'ять		

Додаток Е

Таблиця 4.4 – Час виконання на вузлі n5013 на одному процесі з прискоренням відеокартою

Алгоритм	**Вузол**	**Розмір вхідних матриць**	**Час виконання (сек)**	**Час обслуговання GPU (сек)**	**RAM (mb)**
MatrDMult4	n5013	512	0.19	0.073	39
MatrDMult4	n5013	1024	0.29	0.101	102
MatrDMult4	n5013	2048	0.78	0.224	352
MatrDMult4	n5013	4096	2.83	1.278	1362
MatrDMult4	n5013	8192	10.4	3.649	5431
MatrDMult4	n5013	16384	93.2	45.98	23102
MatrDMult4	n5013	32768	Закінчилась оперативна пам'ять		
MatrDMultStrassWin7	n5013	512	0.19	0.069	39
MatrDMultStrassWin7	n5013	1024	0.31	0.101	102
MatrDMultStrassWin7	n5013	2048	0.82	0.247	352
MatrDMultStrassWin7	n5013	4096	2.92	1.354	1362
MatrDMultStrassWin7	n5013	8192	10.4	3.658	5431
MatrDMultStrassWin7	n5013	16384	Закінчилась оперативна пам'ять		