НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проектування

До захисту допущено:

Завідувач кафедри

_____ Вадим МУХІН

«___»_____2022 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою

«Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

зі спеціальності 122 «Комп'ютерні науки»

на тему: «Процедури вирішення СЛАР з розрідженими матрицями із одночасним застосуванням GPU та CPU обчислень»

Виконав:
студент IV курсу, групи ДА-81
Дурда Роман Євгенович _____

Керівник:
старший викладач, Булах Богдан Вікторович _____

Консультант з економічного розділу:
к.е.н., доцент, Рощина Надія Василівна _____

Консультант з нормконтролю:
к.т.н., доцент, Кирюша Богдан Анатолійович _____

Рецензент:
проф., д. т. н., Петро Іванович Бідюк _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2022

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут прикладного системного аналізу
Кафедра системного проектування

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма – «Інтелектуальні сервіс-орієнтовані розподілені обчислення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Вадим МУХІН

«___»_____2022 р.

ЗАВДАННЯ

на дипломну роботу студенту

Дурді Роману Євгеновичу

1. Тема роботи: «Процедури вирішення СЛАР з розрідженими матрицями із одночасним застосуванням GPU та CPU обчислень».
   Керівник роботи: Булах Богдан Вікторович, старший викладач, вчене звання, затверджені наказом по університету від «___»_____ 2022 р. №_____

2. Термін подання студентом роботи: 10 червня 2022 р.

3. Вихідні дані до роботи: реалізований вирішувач СЛАР з розрідженими матрицями на GPU і CPU, використовуючи C++ і OpenCL.

4. Зміст роботи: дослідження методів для вирішування СЛАР з розрідженими матрицями на GPU, та їх реалізація.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): діаграми для опису методів вирішування СЛАР, архітектури GPU та приклад роботи програми

6. Консультанти розділів роботи[1*]

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	к.е.н. доцент Рощина Надія Василівна		

7. Дата видачі завдання 20 березня 2022 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вивчення літератури за темою роботи.	28.03.2022	
2.	Підготовка першого розділу.	14.04.2022	
3.	Підготовна другого розділу.	23.04.2022	
4.	Підготовка третього розділу.	02.05.2022	
5.	Підготовка четвертого розділу та програмна реалізація	03.06.2022	
6.	Підготовка економічної частини.	10.06.2022	
7.	Оформлення розділів відповідно до нормконтролю.	13.06.2022	
8.	Підготовка презентації та доповіді.	17.06.2022	
9.	Оформлення дипломної роботи.	19.06.2022	

Студент Роман Дурда

Керівник Богдан Булах

* Якщо визначені консультанти. Консультантом не може бути зазначено керівника дипломної роботи.

АНОТАЦІЯ

до бакалаврської дипломної роботи Дурди Романа Євгеновича на тему:
«Процедури вирішення СЛАР з розрідженими матрицями із одночасним застосуванням GPU та CPU обчислень»

У даній дипломній роботі були розглянуті методи вирішення СЛАР з розрідженими матрицями, застосовуючи GPU і CPU, з використанням таких інструментів як мова програмування C++ та фреймворк для обчислення GPGPU – OpenCL.

Ми розглянули, які взагалі бувають методи для вирішення систем лінійних алгебраїчних рівнянь. Дізналися як працює GPU та в чому її переваги та недоліки порівняно з CPU. Також дослідили, які саме методи розв’язання СЛАР з розрідженими матрицями доцільно використовувати при обчислюванні на графічному процесорі.

Результатом же цієї роботи стала програмна реалізація двох алгоритмів для розв’язування СЛАР з розрідженими матрицями, та аналіз роботи цих методів на різних графічних процесорах та самому центральному процесорі.

Дана робота може бути цікава для тих, хто часто стикається з диференціальними рівняннями та вирішенням СЛАР з розрідженими матрицями, а також для тих хто хоче дізнатися як працює графічний процесор, з прикладом застосування його як GPGPU, використовуючи C++ з OpenCL.

Загальний обсях роботи: 92 сторінки, 11 рисунків, 12 таблиць, 15 посилань.

Ключові слова: СЛАР, графічний процесор, центральний процесор, GPGPU, C++, OpenCL, розріджена матриця, метод градієнтного спуску, метод спряжених градієнтів, Matrix-Market format, паралельні обчислення.

ABSTRACT

for the bachelor's thesis of Roman Durda Yevhenovych on the topic:
«SLAE solving procedures with sparse matrices using both GPU and CPU computing»

In this thesis we discussed methods of solving SLAE with sparse matrices, using GPU and CPU, using tools such as programming language C++ and the framework for computing GPGPU – OpenCL.

We considered what are the general methods for solving system of linear algebraic equations. Learned how the GPU works and what its advantages and disadvantages are compared to the CPU. Also we explored which methods of solving SLAE with sparse matrices should be used for computing on a graphics processor.

The result of this work became the software implementation of two methods for solving SLAE with sparse matrices, and an analysis of the operation of these methods on different GPUs and the CPU itself.

This work may be of interest to those who are often confronted with differential equations and solving SLAE with sparse matrices, as well as to those who want to know how a GPU works with an example of using it as a GPGPU using C++ with OpenCL.

Total workload: 92 pages, 11 pictures, 12 tables, 15 references.

Keywords: SLAE, GPU, CPU, GPGPU, C++, OpenCL, sparse matrix, steepest descent method, conjugate gradient method, Matrix-Market format, parallel computing.

ЗМІСТ

ВСТУП

Для чого потрібно вирішувати систему лінійних алгебраїчних рівнянь? Закладаюся ви ніколи не задавалися цим питанням (та і я в принципі раніше теж). Виявляється, що це дуже важливо, особливо коли потрібно на практиці вирахувати всі ризики при сторені якихось фізичних моделів. І це може стосуватися будь-чого, від моделювання ядерного реактору і хімічної інженерії, в яких може бути купа зв’язків між всіма елементами, до будівництва ЛЕП з комплексними зв’язками ліній передач.

Всі ці моделі можна записати як диференціальні рівняння, які в свою чергу можна перетворити в СЛАР, які можна розв’язати (і такі СЛАР в таких моделях часто мають саме розріджену матрицю). А якщо є попит на вирішення систему лінійних рівнянь, значить потрібно знайти способи, як це можна зробити найефективніше. Тому в цій дипломній роботі я зроблю дослідження, які саме алгоритми можна застосувати для вирішення СЛАР з розрідженими матрицями, і чи доцільно застосовувати для цього графічний процесор.

Чому саме графічний процесор? Теорію по архітетурі графічного процесора я опишу в другому розділі, але якщо коротко, основна його перевага – велика ступінь паралелізму порівняно з центральним процесором. Саме завдяки цієї можливості його використовують для відображення графіки, бо обраховування кольору пікселя, а саме це робить GPU під час відображення кадру, в основному виконується незалежно від інших пікселів, що дозволяє їх обчилювати паралельно. Знаючи цю властивість можна скласти формулу: якщо є багато даних, які треба обчислювати, і вони обчислюються незалежно один від одного, то в такій системі доцільно застосувати GPU.

Завдяки такій властивості графічний процесор уже застосовують в різноманітних системах: майнінг криптовалют, машинне навчання і штучний інтелект, аналітика і наука про дані.

Задачу про вирішення СЛАР також можна добавити до цієї групи, але з деякою умовою. Так, в таких задачах часто потрбіно обчислювати купу даних (особливо коли розмірність матриці величезна, що часто буває у громіздких системах), але в найпопулярніших методах вирішення СЛАР вони не обраховуються незалежно, так як попередні обчислення часто вимагаються для обрахування даних в наступному кроці, тому тут не обійтися без синхронізації потоків, що може сповільнити обчислення.

Враховуючи ці моменти, ми спробуємо дізнатися на скільки доцільно застосовувати GPU при вирішенні систем лінійних алгебраїчних рівнянь з розідженими матрицями.

РОЗДІЛ 1. ОГЛЯД ГІБРИДНИХ МЕТОДІВ ДЛЯ ВИРІШЕННЯ СИСТЕМ ЛІНІЙНИХ АЛГЕБРАЇЧНИХ РІВНЯНЬ. ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ.

1.1. Короткі теоретичні відомості.

У цьому розділі будуть досліджені методи вирішення систем лінійних алгебраїчних рівнянь, та оглянемо літературні джерела щодо прикладів реалізації гібридних обчислень для вирішення СЛАР. Але перед цим потрібно вияснити, що таке СЛАР і для чого їх вирішувати.

Система лінійних алгебраїчних рівнянь (СЛАР) – це така система лінійних рівнянь в лінійній алгебрі, яка має вигляд:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad\vdots \qquad\qquad \vdots \qquad\quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \tag{1.1}$$

Цей вираз є системою m лінійних рівнянь з n невідомими, де

$x_1, x_2, \dots, x_n$ – невідомі,

$a_{11}, a_{12}, \dots, a_{1m}$ – коефіцієнти системи,

$b_1, b_2, \dots, b_m$ – вільні члени.

1.1.1. Векторний запис.

СЛАР можна записати у виді, де кожна невідома є ваговим коефіцієнтом в лінійній комбінації вектор-стовпців.

$$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{1.2}$$

Це дозволяє переформулювати задачу в термінах векторного простору, що рівняння має розв'язок тоді, коли лінійна комбінація векторів лівої частини включає вектор правої частини.

1.1.2. Матричний запис.

Векторна форма еквівалентна матричній формі запису

$$Ax = b \tag{1.3}$$

де A – матриця $m \times n$,

x – вектор з n компонент,

b — вектор з m компонент.

$$A = \begin{bmatrix} a_{11} & a_{12} \cdots a_{1n} \\ a_{21} & a_{22} \cdots a_{2n} \\ \vdots & \vdots \ddots \vdots \\ a_{m1} & a_{m2} \cdots a_{mn} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{1.4}$$

У математиці теорія лінійних систем є основою і фундаментальною частиною лінійної алгебри, предметом, який використовується в більшості розділів сучасної математики. Обчислювальні алгоритми для пошуку розв’язків є важливою частиною чисельної лінійної алгебри і відіграють визначну роль в інженерії, фізиці, хімії, інформатиці та економіці. Систему нелінійних рівнянь часто можна апроксимувати лінійною системою (див. лінеаризацію), що є корисним методом під час створення математичної моделі або комп’ютерного моделювання відносно складної системи [1].

Отже, ми визначили що таке СЛАР, звідки вони беруться і для чого їх вирішувати. Тепер визначимо ряд основних джерел звідки буде братися інформація, а також оглянемо коротко основні методи для розв’язку системи лінійних алгебраїчних рівнянь.

1.2. Огляд літературних джерел.

Для перегляду коротких теоретичних відомостей по алгоритмах чудово підходить Вікіпедія, але вона зовсім недоречна, коли детально розглянути алгоритм і написати реалізацію. Для таких задач підходять інші джерела, наприклад відома студентам нашої кафедри методичка під назвою *«Чисельні методи в інформатиці»*. Також будуть застосовуватися різні інтернет ресурси, і інші посібники знайдені в інтернеті, а також роботи виконані різними дослідниками.

Мабуть основним джерелом звідки я почерпну основну інформацію – це книга *«OpenCL in Action. How to accelerate graphics and computations»*. Саме тут буде вся інформація про алгоритми, які я зможу застосувати для обчислення на GPU використовуючи кроссплатформений фреймворк OpenCL.

1.3. Методи розв’язування СЛАР.

Для розв’язування СЛАР існують прямі і ітераційні методи.

Прямий метод означає, що після скінченного числа арифметичних операцій, алгоритм надає точний розв’язок задачі. До прямих методів належать:

- матричний метод (за допомогою оберненої матриці);
- метод виключення Гаусса (і його модифікації)
- метод відбиття (Хаусхолдера);
- метод обертань (Гівенса);
- метод Холеського;
- метод квадратного кореня тощо.

Ітераційний метод – це такий метод розв‘язування СЛАР, що базується на послідовному наближенні до розв‘язку шляхом багатократного застосування

деякого обчислювального процесу, і при цьому вихідними даними для кожної наступної процедури є результатом застосування попередніх процедур. Як результа такого ітераційного процесу є послідовність, що при виконанні деяких умов збігається до розв‘язку задачі. До ітераційних методів належать:

- метод простої ітерації (метод Якобі);
- метод Гауса-Зейделя;
- метод релаксації;
- метод градієнтного спуску (steepest descent);
- метод спряженого градієнту (conjugate gradient);

Для матриць середньої розмірності часто більш привабливими є прямі методи. Ітераційні методи застосовуються в основному для розв‘язування складних задач великої вимірності. Для них, внаслідок обмежень об‘єму робочої пам‘яті та числу арифметичних операцій, використання прямих методів виявляється достатньо неефективним і важким. Наприклад, ітераційні методи в основному застосовуються для розв‘язання:

- задач з трьома просторовими змінними;
- задач, що включають системи нелінійних рівнянь;
- задач, що виникають при дискретизації систем рівнянь в частинних похідних;
- нестаціонарних задач з більш ніж однією просторовою змінною.

Формулювання і застосування ітераційних методів потребують спеціальних знань і деякого досвіду.

Для ефективного використання ітераційних методів потрібно пам’ятати про три обставини:

1). Невідомо, який метод потрібно застосовувати і яким чином його можна реалізувати;

2). Невідомо, як обирати ітераційні параметри, що необхідні для роботи конкретних методів, наприклад, коефіцієнт релаксації для методу послідовної верхньої релаксації.

3). Не дуже ясний вибір моменту закінчення ітераційного процесу.

Внаслідок такого великого різноманіття задач та ітераційних процедур, повністю позбавитися цих невизначеностей неможливо. Ефективність ітераційного методу розв‘язування конкретної задачі залежить від її характерних
властивостей та від архітектури обчислювального пристрою, на якій буде розв’язуватися задача. Також потрібно враховувати, що основна вимога до алгоритму – це можливість її добре розпаралелити, щоб можна було його ефективно розрахувати на GPU.

Дивлячись на ці всі умови, жодних загальних правил вибору найкращого методу розв‘язування не існує. Проте, знання порівняльних характеристик ряду ітераційних процедур загального вигляду може значно спростити проблему. Отже, підхід до вибору методу має базуватися на аналізі теоретичних та обчислювальних принципів загальних ітераційних методів. Ці принципи потім можна дійсно використовувати при виборі ефективного ітераційного методу [2].

Для прикладу розглянемо кілька методів розв'язування СЛАР.

1.3.1. Матричний метод.

Це один з найочевидніших методів вирішення СЛАР. Якщо матриця A є квадратною та невиродженою, то для неї існує обернена матриця.

Для вирішення СЛАР таким методом, на вираз (1.3) ми множимо ліву і праву частину рівняння на A^{-1}, тоді знаючи, що $A \cdot A^{-1} = I$, то отримаємо наступне:

$$x = A^{-1}b \tag{1.5}$$

Такий метод працює, але знаходження оберненої матриці – це дуже трудомістка задача. І навіть застосовуючи найшвидші алгоритми пошуку оберненої матриці, вони не зможуть швидко справитися з матрицями з розмірністю більше 100.

1.3.2. Метод виключення Гаусса.

Розв'язання системи лінійних алгебраїчних рівнянь методом Гаусса полягає в послідовному виключенні невідомих $x_1, x_2, \dots, x_n$ з цієї системи. Поділивши перше рівняння системи на a_{11} (при $a_{11} \neq 0$), отримаємо:

$$x_1 + c_{12}x_2 + \cdots + c_{1n}x_n = y_1, \text{ де } c_{1j} = \frac{a_{1j}}{a_{11}}, j = 2,3, \dots, n, y_1 = \frac{b_1}{a_{11}} \tag{1.5}$$

З решти рівнянь системи

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i, \quad i = 2,3, \dots, n \tag{1.6}$$

у кожному з них виключимо невідому x_1, помноживши (1.5) на a_{i1} і віднімемо отримане рівняння від i-го рівняння системи (1.6). У результаті отримаємо наступну систему рівнянь:

$$\begin{cases} x_1 + c_{12}x_2 + \cdots + c_{1j}x_j + \cdots + c_{1n}x_n = y_1 \\ a_{22}^{(1)}x_2 + \cdots + a_{2j}^{(1)}x_j + \cdots + a_{2n}^{(1)} = b_2^{(1)} \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ a_{n2}^{(1)}x_2 + \cdots + a_{nj}^{(1)}x_j + \cdots + a_{nn}^{(1)}x_n = b_n^{(1)} \end{cases}, \tag{1.7}$$

де $a_{ij}^{(1)} = a_{ij} - c_{1j}a_{i1}, b_i^{(1)} = b_i - y_1a_{i1}, \ \ i,j = 2,3, \dots m$

У даній системі невідома x_1 є тільки в першому рівнянні, тому надалі достатньо мати справу зі скороченою системою рівнянь:

$$\sum_{j=2}^{n} a_{ij}^{(1)} x_j = b_i^{(i)}, \quad i = 2,3,\dots,m \tag{1.8}$$

У такий спосіб здійснено перший крок методу Гаусса. Якщо $a_{22}^{(1)} \neq 0$, то з системи (1.8) аналогічно можна виключити x_2 і перейти до системи, яка еквівалентна першопочатковій (1.1). При цьому перше рівняння системи (1.7) залишиться без змін.

Виключаючи послідовно в такий спосіб невідомі $x_3, x_4, \dots, x_n$, прийдемо остаточно до системи рівнянь, яка має такий вигляд:

$$\begin{cases} x_1 + c_{12}x_2 + \cdots + c_{1j}x_j + \cdots + c_{1n}x_n = y_1 \\ x_2 + c_{23}x_3 + \cdots + c_{2n}x_n = y_2 \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ x_{n-1} + c_{n-1,n}x_n = y_{n-1} \\ x_n = y_n \end{cases} \tag{1.9}$$

Матриця цієї системи є верхньою трикутної, в якій всі елементи нижче головної діагоналі дорівнюють нулю, а вище – ні. Даний етап називається прямий хід методу Гаусса [4].

Для знаходження самих невідомих застосовується обернений хід Гаусса. Знаходимо спочатку останнє значення x_n:

$$x_n = y_n = \frac{b_n}{a_{nn}} \tag{1.10}$$

Потім підраховується $(n-1)$ крок, підставивши обраховану величину x_n до наступного рівняння:

$$y_{n-1} = d_{n-1} - c_{n-1,n}x_n = \frac{b_{n-1} - a_{n-1,n}x_n}{a_{n-1,n-1}} \tag{1.11}$$

Ну і виконуємо все те ж саме на наступних ітераціях. Тоді загальна формула для оберненого ходу наступна:

$$x_k = \frac{1}{a_{kk}}\left(b_k - \sum_{j=k+1}^{n} a_{kj}x_j\right) \tag{1.12}$$

1.3.3. Метод простої ітерації

Для великих розріджених систем рівнянь досить хороші результати можна отримати з використанням методу визначальних величин. Для рівнянь із заповненими матрицями коефіцієнтів застосовують ітераційні методи, які покращують початково обраний вектор розв'язку без обробки матриці коефіцієнтів (тобто зведення її до трикутної форми чи розкладання на добуток двох трикутних матриць), у результаті чого зберігається ненульова структура матриці і не з'являються нові ненульові елементи.

Для розв'язання системи рівнянь $Ax = b$ обирають деяке початкове наближення $x^{(0)}$, що використовується для обчислення наступних наближень $x^{(k)}$ за деякою рекурентною формулою обчислень:

$x^{(k+1)} = Mx^{(k)} + C^{(k)}$, де k – номер ітерації, M, $C^{(k)}$ – ітераційні оператори.

У цьому виразі під час обчислення $x^{(k+1)}$ використовується значення наближення тільки на одній попередній ітерації $x^{(k)}$. Тому цей метод називають однокроковим чи одношаровим. Але існують і більш складні рекурентні формули наближень, в яких, наприклад для обчислення $x^{(k+1)}$, використовують значення на двох попередніх ітераціях $x^{(k)}$ і $x^{(k-1)}$, тоді їх називають двокроковими чи двошаровими.

Найпростіша рекурентна формула наближень такого типу, яка відповідає методу простої ітерації, будується на основі використання нев'язки системи лінійних рівнянь, що розв'язуються:

$$r^{(k)} = Ax^{(k)} - b \tag{1.13}$$

Для цього методу формула наближень набуває вигляду

$$x^{(k+1)} = x^{(k)} - r^{(k)} = x^{(k)} - Ax^{(k)} + b = (E - A)x^{(k)} + b, \tag{1.14}$$

де $M = E - A, C^{(k)} = b$

Початкове наближення $x^{(0)}$ у цьому методі можна вважати вектор правої частини системи рівнянь b. І починаємо ітерувати допоки $x^{(k+1)} - x^{(k)} > \xi$, де ξ – деяка похибка.

Якщо ввести деякий коефіцієнт демпфірування ω до даної формули, то тоді отримаємо *метод верхньої релаксації*, що застосовується для прискорення збіжності:

$$x^{(k+1)} = x^{(k)} - \omega r^{(k)} \tag{1.15}$$

1.4. Висновки.

У першому розділі ми ознайомилися з тим, що це таке СЛАР, вияснили для чого вони потрібні і де застосовуються. Визначили які бувають методи для вирішення систем лінійних алгебраїчних рівнянь (прямі і ітераційні) і написали короткий опис декількох найпопулярніших з них. Також ми визначили літературу звідки буде братися інформація як для теоретичних знань, так і для практичного застосування.

РОЗДІЛ 2. ОПИС АРХІТЕКТУРИ ГРАФІЧНОГО ПРОЦЕСОРА. АЛГОРИТМИ ВИРІШЕННЯ СЛАР, ЩО ДОБРЕ ПІДХОДЯТЬ ДЛЯ ВИКОРИСТАННЯ GPU.

Перед тим як почати дослідження того, які алгоритми добре підходять для використання GPU, потрібно вияснити як саме працює GPU, в чому її різниця з роботу CPU і які переваги та недоліки існують в даних процесорах.

Потім ми дізнаємося з чого складається відеокарта і чому саме вона підходить для розв’язування СЛАР.

Ну і під кінець цього розділу ми розглянемо дослідження різних авторів, які застосовували GPU для розв’язання систем лінійних алгебраїчних рівнянь, і що з цього вийшло.

2.1. Чим відрізняється CPU від GPU.

У сучасному світі зараз майже в кожного є комп’ютер, а то й кілька. Це потужна машина, яка виконує мільйони, якщо не мільярди, операцій в секунду.

Але як ми зазвичай працюємо за ним? Ми відкриваємо кришку ноутбука чи вмикаємо комп’ютер, в нас завантажується операційна система і ми починаємо займатися своїми справами: дивимося фотографії, скролимо інтернет, працюємо.

Але що ж використовує комп’ютер при цьому? Звичайний процесор, який є невід’ємною частиною кожної машини.

Чи потрібно нам щось більше? Для описаних задач вище точно ні.

Так а що нам тоді треба крім процесора? Для чого нам потрібен якийсь графічний процесор? Ми знаємо тільки, що він використовується в іграх, але для чого, чому саме він потрібен?

Відповідь одна – паралелізм. У GPU можуть знаходитися сотні і навіть тисячі ядер, які можуть працювати паралельно. І коли ми граємо в гру, основна

робота програми – це замалювати потрібним кольором піксель на екрані. А так як пікселів мільйони, а нам дуже важлива швидкість роботи програми (особливо коли індустрія вимагає стабільні 30 чи 60 кадрів в секунду), то ми й використовуємо відеокарту для цього. А так як колір пікселя визначається в основному незалежно від інших, то ми можемо використати максимальну вигоду від GPU.

Ну добре, раз GPU такий крутий, то чому б не використовувати його замість CPU? Не все так просто. Якщо коротко, то в CPU хоч і менше ядер, але його ядра є набагато потужнішими і багатозадачнішими ніж ядра GPU. Тому для повсякденних задач процесор підходить набагато більше.

Так, почекайте, якщо говориться зараз за якісь ігри, як це взагалі стосується до вирішення СЛАР з розрідженими матрицями? Розрахунок СЛАР є досить об’ємною задачею. Час розрахунку може бути дуже довгою, і дійсно досягати кількох днів, особливо для великих матриць. Але розумні люди виявили, що багато інструкцій при розв’язку СЛАР деякими алгоритмами виконуються незалежно, а це дає нам можливість виконати ці дії паралельно. І як ми знаємо GPU дуже хороший в задачах, де потрібна паралельність, що може прискорити вирішення СЛАР в кілька разів!

2.2. Архітектура графічного процесора.

Графічний процесор – це окремий пристрій комп'ютера або ігрової приставки, що виконує графічний рендеринг. Сучасні графічні процесори дуже потужні, і ефективно обробляють і зображують комп'ютерну графіку. Використовуючи спеціалізовану конвеєрну архітектуру, вони набагато ефективніші в обробці графічної інформації, ніж звичайний центральний процесор.

Графічний процесор в сучасних відеокартах використовується як прискорювач трьохвимірної графіки, але в деяких випадках його можна використовувати і для обрахунків (GPGPU – саме такі обрахунки використовуються для вирішення СЛАР) [4].

Для настільних ПК існує два основних виробники графічних процесорів:

- *NVIDIA*: Nvidia – провідний виробник графічних процесорів на сьогоднішній день. Продукти NVIDIA широко відомі як карти N, які представляють такі продукти, як серії GeForce, GTX та RTX.
- *AMD*: Це виробник як і процесорів, так і графічних процесорів. Його графічна карта зазвичай відома як карта A. Типові продукти включають серію Radeon.

Звичайно, і NVIDIA, і AMD також виробляють графічні процесори для мобільних та графічних робочих станцій. Крім того, компанії, що виробляють мобільні відеокарти, включають ARM, Imagination Technology та Qualcomm.

2.2.1. Загальні відомості.

Сучасна відеокарта складається з таких частин:

- графічний процесор – займається розрахунками виведеного зображення, звільняючи з цього обов'язки центральний процесор, робить розрахунки для обробки команд тривимірної графіки.
- відеоконтролер - відповідає за формування зображення у відеопам'яті, дає команди RAMDAC на формування сигналів розгортки для монітора та здійснює обробку запитів центрального процесора. Крім цього, зазвичай присутні контролер зовнішньої шини даних (наприклад, PCI або AGP), контролер внутрішньої шини даних та контролер відеопам'яті.
- відеопам'ять - виконує роль кадрового буфера, в якому зберігається зображення, що генерується і постійно змінюється графічним процесором

і відображається на екрані монітора (або декількох моніторів).

- цифро-аналоговий перетворювач – служить для перетворення зображення, що формується відеоконтролером, на рівні інтенсивності кольору, що подаються на аналоговий монітор.
- відео-ЗСД (Video ROM) – постійний пристрій, в якому записані відео-BIOS, екранні шрифти, службові таблиці тощо. ROM не використовується відеоконтролером безпосередньо – до нього звертається тільки центральний процесор.
- система охолодження – призначена для збереження температурного режиму відеопроцесора та відеопам'яті у допустимих межах. Сучасні GPU мають високу швидкість доступу до своєї власної оперативної пам'яті, яка зазвичай називається текстурною пам'яттю, і мають високу обчислювальну потужність.

Сучасні GPU мають високу швидкість доступу до своєї власної оперативної пам'яті, яка зазвичай називається текстурною пам'яттю, і мають високу обчислювальну потужність.

2.2.2. Особливості архітектури GPU.

2.2.2.1. Конвеєр GPU.

Архітектура GPU визначилася завдяки специфічним потребам комп'ютерної графіки. Саме це визначило високий паралелізм архітектури та її специфіку. Завдяки специфічній архітектурі GPU виконують поставлені перед ними завдання набагато ефективніше, ніж класичні процесори.

Структура проведення операцій на всіх сучасних GPU може бути представлена конвеєром (graphics pipeline), який влаштований таким чином, щоб забезпечити максимальну ефективність виконання завдань комп'ютерної графіки за умови потокової паралельної архітектури.

На вхід подається набір вершин, що надходить із програми. Потім вершини обробляються у вершинних процесорах, що класифікуються як MIMD. Програма для вершинних процесорів називається вершинним шейдером. Після цього результат роботи вершинного шейдера надходить на складання примітивів, таких як полігони або лінії. За цим слідують тести видимості відсікання та інші стандартні операції комп'ютерної графіки, після них дані надходять на растеризацію. Тут об'ємне зображення проектується на плоский екран, і отримана картинка масштабується відповідно до параметрів вікна програми. Результатом цієї операції є картинка, текстура якої представляє собою прямокутний масив так званих пікселів. Піксель може бути представлений 1-4 числами, наприклад, у форматі RGBA, red, green, blue, alpha (коефіцент прозорості). Ця текстура обробляється в піксельних процесорах, які за класифікацією багатопроцесорних систем можна класифікувати як SIMD. Потім слідує ще кілька операцій, і дані надходять у буфер кадру, який і є тією картинкою, яку користувач бачить на екрані. Нижче наведена схема цього конвеєра. Звичайно, цей опис не претендує на повноту, але для цілей GPGPU цього цілком достатньо.

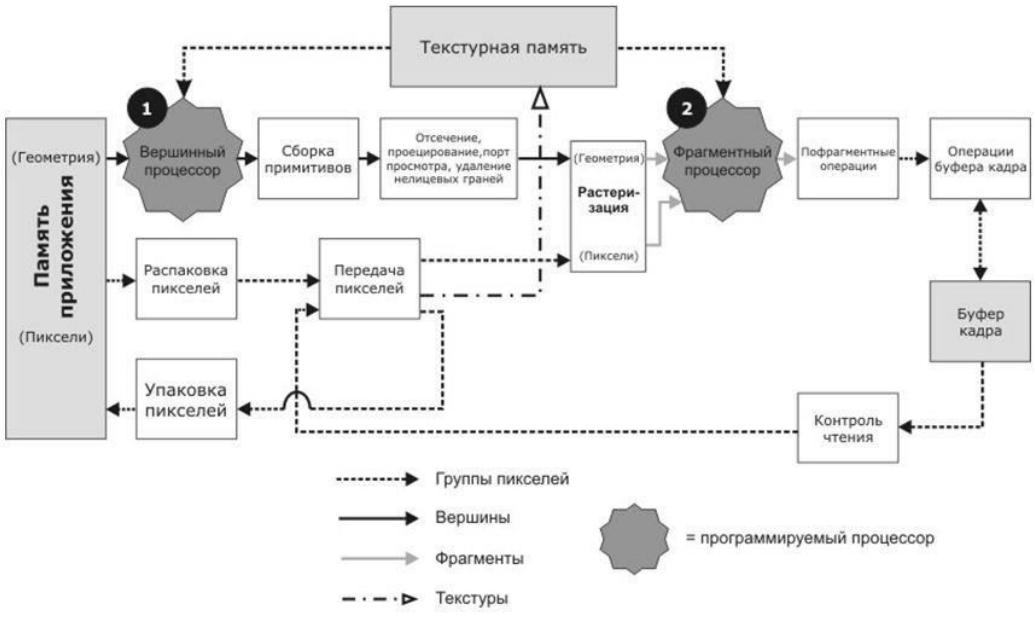

Рис. 2.1. Схематичне зображення роботи GPU-конвеєра.

І вершинні та фрагментні процесори мають спільну пам'ять, яка на відеокарті називається текстурною пам'яттю. У шейдері розробник має доступ читання текстурної пам'яті, але не на безпосередній запис.

2.2.2.2. Введення у модель програмування на GPU.

Програми, що працюють з тривимірною графікою та відео (ігри, GIS, CAD, CAM та ін.), використовують шейдери для визначення параметрів геометричних об'єктів, для зміни зображення (для створення ефектів зсуву, відображення, заломлення, затемнення з урахуванням заданих параметрів поглинання) та розсіювання світла, для накладання текстур на геометричні об'єкти та ін.).

Для вирішення проблеми у відеокарти стали додавати (апаратно) алгоритми, які потребують розробники. Незабаром стало ясно, що реалізувати всі алгоритми неможливо та недоцільно; вирішили дати розробникам доступ до відеокарти – дозволити збирати блоки графічного процесора в довільні конвеєри, що реалізують різні алгоритми. Програми, призначені до виконання на процесорах відеокарти, отримали назву «шейдери». Було розроблено спеціальні мови для складання шейдерів. Тепер у відеокарти завантажувалися не лише дані про геометричні об'єкти, текстури та інші дані, необхідні для відображення рисунку, а й інструкції для GPU.

Поділяють два види пристроїв – те, що управляє загальною логікою – host, і те, що вміє швидко виконати деякий набір інструкцій над великим обсягом даних – device. У ролі хоста зазвичай виступає центральний процесор. У ролі обчислювального пристрою – відеокарта. Відеокарта містить Compute Units – процесорні ядра.

Процесорні ядра можуть виконувати кілька потоків за рахунок того, що в кожному міститься кілька потокових процесорів (8-16) (Stream Cores або Stream Processor). Для карт NVidia – обчислення проводяться безпосередньо на потокових процесорах, але ATI ввели ще один рівень абстракції – кожен потоковий процесор, що складається з processing elements – PE (іноді званих ALU – arithmetic and logic unit) – та обчислення відбуваються на них.

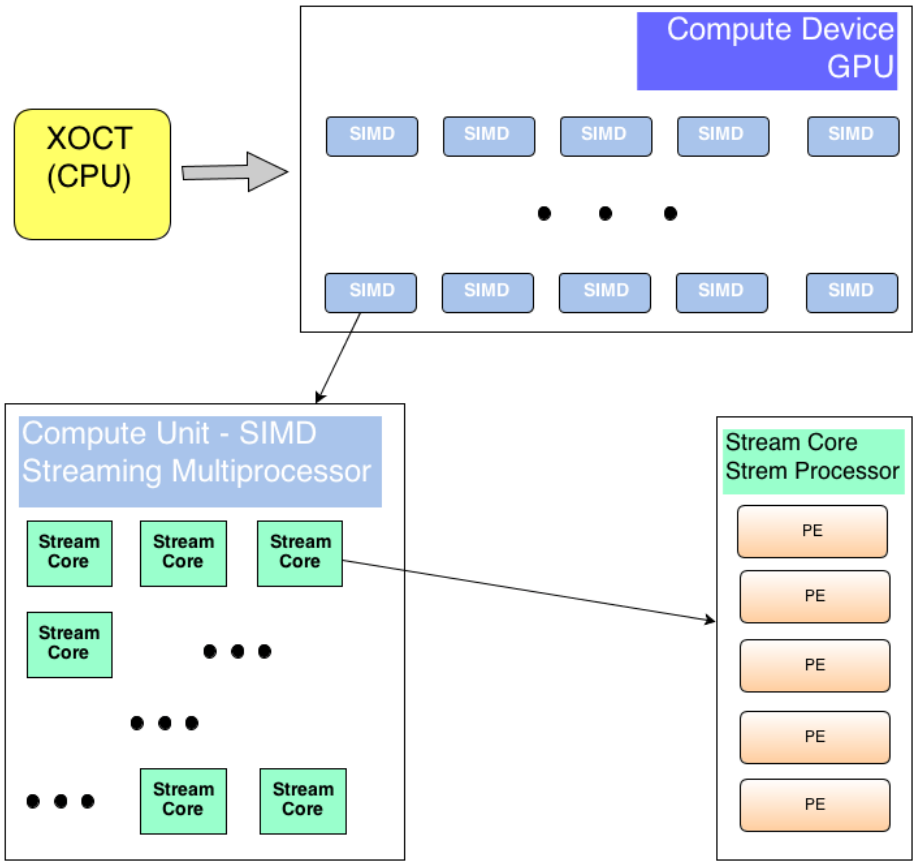

Рис. 2.2. Архітектура моделі GPU.

2.2.2.3. Модель пам'яті GPU.

Графічні процесори мають власну ієрархію пам'яті, багато в чому схожу на звичайний процесор. Проте він відрізняється, так як архітектура пам'яті GPU створювалася з розрахунку на максимальне прискорення роботи алгоритмів комп'ютерної графіки. На GPU/CUDA виділяють 6 видів пам'яті: реєстрова, локальна, глобальна, константна, текстурна.

GPU так само як CPU має власні регістри і кеш для прискорення доступу до даних під час обчислень. Крім цього, GPU має власну основну пам'ять (графічна пам'ять), тому для того, щоб програміст міг виконувати обчислення на GPU, він повинен попередньо передати дані з оперативної пам'яті CPU в пам'ять GPU. Ця операція традиційно є досить дорогою з точки зору продуктивності через відносно невисоку швидкість передачі даних між пам'яттю програми та пам'яттю відеокарти, хоча сучасні шини PCI Express та спеціальні чіпи на материнській платі (такі як NV3 та NV4) помітно прискорили цей процес.

На відміну від пам'яті CPU, пам'ять GPU має деякі серйозні обмеження, і доступ до неї може бути здійснений лише за допомогою деяких абстракцій графічного програмного інтерфейсу. Кожна така абстракція може вважатися типом потоку зі своїм власним набором правил доступу. Таких потоків безпосередньо доступних для програміста може бути виділено 4 – потоки вершин, потоки фрагментів, потоки буфера кадрів, потоки текстур [5].

2.3. Алгоритми вирішувачів СЛАР з використанням GPU.

Зараз багато вирішувачів СЛАР прискорюються за допомогою взаємодії CPU-GPU. Багато практичних завдань можна було б звести до розв’язування систем лінійних рівнянь, що формулюються як $Ax = b$.

Як ми вже знаємо для вирішення СЛАР існує два методи: прямий та ітераційний метод. Розв’язування величезної системи лінійних алгебраїчних рівнянь не є простим завданням і часто вимагає великої обчислювальної потужності. Використовуючи переваги графічного процесора, ми можемо прискорити обчислення багатьох програм. Нижче наведені статті в яких описується застосування алгоритмів для вирішення на GPU

2.3.1. LU-декомпозиція.

Основне застосування декомпозиції LU – вирішення щільних систем лінійних алгебраїчних рівнянь. Розглянемо знайому нам систему (1.3).

Роблячи LU розкладання A, можна переписати це як

$$LUx = b, \tag{2.1}$$

де L – нижня трикутна матриця, а U – верхня трикутна матриця.

Така система може бути вирішена в 2 кроки:

$$1).\ Ly = b \tag{2.2}$$

$$2).\ Ux = y \tag{2.3}$$

Це основні кроки в LU розкладання.

Для прискорення продуктивності LU-декомпозиції, розрахунки можна перенести на GPU. Ось деякі дослідження, в яких використовуються підходи декомпозиції LU на основі GPU.

2.3.1.1. LU-декомопзиція на основі GPU для великих задач методу моментів (GPU-based LU decomposition for large method of moments problems [6]).

У даному досліджені був розглянутий метод LU-декомпозиції на основі GPU для великих задач методу моментів. У цій пропонованій системі вони робили метод моментного аналізу (method of moments - MOM) електромагнітних явищ. Вони прискорили обчислення великої матриці, використовуючи LU-розкладання на GPU.

Для подолання обмежень пам’яті, які притаманні обчисленню GPU, в якості відправної точки використовується LU-декомпозиція поза ядром. Для паралельної реалізації LU-розкладання використовувалася бібліотека ScaLAPACK. Завдяки лінійному вирішувачу на основі графічного процесора

вони досягли більш високої продуктивності та прискорення вимірювання приблизно в 15 разів.

2.3.1.2. LU-факторизація для паралельної симуляції електричних кіл (LU Factorization for Parallel Circuit Simulation [7]).

У даній роботі було розглянуто розріджену LU-факторизацію для моделювання паралельної симуляції електричних кіл на GPU. У цій роботу були наступні кроки при взаємодії ЦП і ГП:

1). Попередня обробка виконується на ЦП лише один раз.

2). Після завершення попередньої обробки всі дані передаються на графічний процесор.

3). Вся робота з LU-факторизацією виконується на GPU.

4). Потім кінцевим результатом буде копіювання назад в ЦП.

Була досліджена оптимізація робочого розбиття, кількість активних груп потоків і шаблони доступу до пам’яті на основі архітектури GPU. На матрицях, факторизація яких включає в себе багато операцій з плаваючою комою, розріджена LU-факторизація на основі GPU досягла 7.90-кратного прискорення над 1-ядерним ЦП і 1.49-кратного прискорення над 8-ядерним ЦП.

2.3.2. Метод Гаусса-Жордана.

Метод виключення Гаусса-Жордана – метод, який використовується для вирішення квадратних систем лінійних рівнянь алгебри, знаходження зворотної матриці, знаходження координат вектора в заданому базисі або знаходження рангу матриці.

Метод Гаусса-Жордана складається з двох частин:

- Виключення вперед (forward elimination)
- Зворотна заміна (back substitution)

Нижче ми побачимо різні підходи, в яких використовувався цей вирішувач:

2.3.2.1. Алгоритм виключення Гаусса-Жордана для інверсії матриці (Gauss-Jordan Elimination Algorithm for Matrix Inversion [8]).

У даній статті був розглянутий метод Гаусса-Жордана для інверсії. Ми вже знаємо, що знаходити обернену матрицю досить затратно, особливо великі матриці. Тому для вирішення цих проблем нам потрібні високопродуктивні обчислення. З цією метою було запропоновано оцінку високопродуктивних кодів для інверсії матриці на основі усунення Гаусса-Джордана з частковим поворотом, що розвантажує основні обчислювальні ядра на один або кілька графічних процесорів під час виконання дрібних операцій на процесорі загального призначення. При цьому було розглянуто багатоядерний процесор, підключений до кількох графічних процесорів.

Основними перевагами GJE перед традиційним підходом до інверсії матриці за допомогою виключення Гаусса є те, що більшість обчислень у GJE можна відтворити в термінах матрично-матричних добутків. Множення матриць – це дуже паралельна операція, яка може використовувати всі можливості графічних процесорів. Оновлення блоків матриці в позиціях, відмінних від панелі, розрахованих під час поточної ітерації, можна виконувати паралельно. Далі цей підхід GJE було розглянуто для платформи з багатьма GPU.

2.3.2.2. Метод виключення Гаусса-Жордана для систем лінійних схематичних рівнянь з CUDA (Gauss-Jordan elimination method with CUDA for linear circuit equation systems [10]).

У даній статті було використано метод Гаусса-Жордана для розв’язування схематичних рівнянь з CUDA. За дослідженням ефективність методу виключення Гаусса-Джордана визначається з точки зору часу виконання за допомогою методів паралельного програмування на графічній карті. Додаток,

розроблений на графічному процесорі з використанням CUDA, є швидшим, ніж додаток, розроблений на ЦП.

2.3.3. Метод Якобі.

Це ітеративний метод розв'язування лінійної системи рівнянь. У ній, у системі лінійних рівнянь $Ax = b$, матрицю A можна розкласти на два доданки: D і R, де D – діагональна матриця:

$$A = D + R, D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix} \quad (2.4)$$

Тоді СЛАР можна переписати у виді:

$$Dx = b - Rx \quad (2.5)$$

Ітераційний метод Якобі виражається формулою:

$$x^{(k+1)} = D^{-1}\left(b - Rx^{(k)}\right) \text{ чи } x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right), i = 1,2,\ldots,n. \quad (2.6)$$

2.3.3.1. Алгоритм Якобі для знаходження власних значень симетричних матриць за допомогою CUDA (Jacobi Algorithm for Solving Eigenvalues of Symmetric Matrices with CUDA [11]).

У даній роботі було запропоновано реалізацію алгоритму Якобі для знаходження власних значень симетричних матриць за допомогою CUDA. Внесок цієї статті узагальнено таким чином:

1). У їхньому експериментальному середовищі використовується чотирьохядерний процесор Intel Core i5-760, карта NVIDIA GeForce GTX460.

2). Експериментальні результати цієї статті показують, що метод Якобі на GPU може заощадити більше часу роботи, ніж звичайний послідовний алгоритм.

3). Коефіцієнт прискорення становить 3,02 ~ 13,71.

4). Теоретичний аналіз показує, що часова складність паралельного алгоритму O(n).

Як ми можемо побачити, розв’язувати СЛАР реально на GPU. І за дослідженнями зверху дійсно видно пришвидшення при використовуванні графічного процесора в рази. Але ми тут розглянули тільки загальні алгоритми, які застосовують для щільних матриць. Для розріджених же матриць ці алгоритми не дуже підходять і застосовуються зовсім інші методи, які ми розглянемо пізніше.

2.4. Висновки.

Під час роботи над розділом 2 ми вияснили в чому різниця між процесором і графічним процесором. Дізналися яка архітектура GPU і в чому її перевага при вирішенні СЛАР (і як було вияснено основна причина – велика ступінь паралелізму). Також розглянули дослідження в яких застосовувався GPU для різних методів вирішення системи лінійних алгебраїчних рівнянь, в яких експериментатори змогли досягти пришвидшення в кілька разів, порівняно з CPU.

РОЗДІЛ 3. СПЕЦИФІКА РОЗРІДЖЕНИХ МАТРИЦЬ ПРИ ЇХ РОЗВ'ЯЗУВАННІ ТА ЗБЕРІГАННІ. АЛГОРИТМИ ВИРІШЕННЯ СЛАР, ЩО ПІДХОДЯТЬ САМЕ ЇМ, ВИКОРИСТОВУЮЧИ GPU.

3.1. Короткі теоретичні відомості.

Розріджена матриця – це матриця, в якій більшість елементів дорівнює нулю. Немає строгого визначення щодо частки елементів з нульовим значенням для того, щоб матриця кваліфікувалася як розріджена, але загальним критерієм є те, що кількість ненульових елементів приблизно дорівнює кількості рядків або стовпців [12].

Розріджені матриці з'являються, коли вченим та інженерам потрібно розв'язувати складні системи, змодельовані за допомогою диференціальних рівнянь. Ці рівняння відіграють важливу роль в аналізі реальних динамічних систем. Інженери NASA вирішують диференціальні рівняння, щоб запустити ракети в космос, а фінансові трейдери вирішують їх, щоб оцінити волатильність цін на акції. Якщо величина змінюється в часі або просторі, велика ймовірність, що прикладний математик вивів диференціальні рівняння для моделювання зміни. В особливості такі матриці є дуже важливими для аналізу складних систем, і існує ряд методів розв'язування рівнянь за допомогою розріджених матриць.

Загалом перетворення диференційного рівняння в систему лінійних рівнянь відбувається за допомогою *методу скінченних елементів*, і цю процедуру можна розбити на наступні кроки:

1. Складна система розкладається на прості елементи, які легко аналізувати.
2. Отримується диференціальне рівняння або рівняння, що відповідає кожному елементу.
3. Диференціальні рівняння перетворюються в лінійні за допомогою методу апроксимації, наприклад методу Петрова-Гальоркіна або методу Рітца-Гальоркіна.
4. Об’єднуються лінійні рівняння в матрицю. Часто це буде розріджена матриця.
5. Розв’язується матриця, щоб отримати наближене рішення для всієї системи.

Крок 3 особливо важливий. Метод скінченних елементів апроксимує диференціальні рівняння, які важко розв’язати, використовуючи лінійні рівняння, які легко розв’язувати.

Як приклад розглянемо рисунок 3.1. На ньому зображена матриця, яка відповідає структурній моделі реальної опори ЛЕП. Розмір матриці становить 153 на 153, але лише 10 відсотків (2423) елементів матриці не дорівнюють нулю. Значення представлені точками, і оскільки структурна модель симетрична, то і сама матриця симетрична [13].

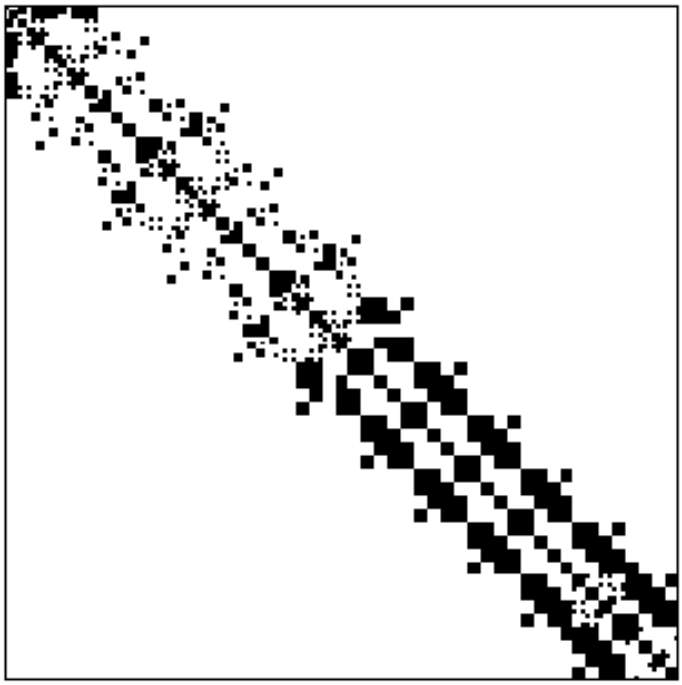

Рис. 3.1. Приклад розрідженої матриці.

3.1.1. Види розріджених матриць.

Загалом за структурою даних розріджені матриці бувають наступних типів:

- *Смугова матриця*

  Це така розріджена матриця, ненульові елементи якої обмежені діагональною смугою, що включає головну діагональ і яка може зберігати ще діагоналі з обох сторін.

- *Симетрична матриця*

  Це така матриця у яких елементи у верхньотрикутній частині матриці мають такі ж значення, як і у нижньотрикутній частині відносно діагоналі. Така матриця зображена на рисунку 1. Також одна з умов використання методів розв’язання СЛАР, які буде описано нижче, саме симетричність розрідженої матриці.

- *Блок-діагональна матриця*

Блок-діагональна матриця складається з підматриць уздовж її діагональних блоків. Блок-діагональна матриця A має вигляд

$A = \begin{bmatrix} A_1 & 0 & \cdots & 0 \\ 0 & A_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_n \end{bmatrix}$, де A_k – квадратна матриця для всіх $k = 1, 2, \ldots, n$.

3.1.2. Способи зберігання матриць.

Як ми вже знаємо, в розріджених матрицях ненульових значень набагато менше ніж нулів, тому зберігати всі значення матриці є не дуже правильним, так як додатково витрачається багато місця на зберігання нулів. Для таких випадків існують 2 формати збереження даних: MatrixMarket format і Harwell-Boeing format.

3.1.2.1. MatrixMarket format.

Для наглядності відкриємо файл в якому зберігається матриця, що на рисунку 1. Даний файл має розширення .mtx:

```
%%MatrixMarket matrix coordinate real symmetric
153 153 1288
1   1   3.1431392791300e+05
4   1  -8.6857870528200e+04
5   1   5.6340240342600e+04
7   1  -3.5149546714100e+04
8   1   9.3732124570800e+04
…   …   …
```

В першому рядку описана деяка метаінформація про цей файл. Розглянемо по порядку:

- %%MatrixMarket – вказує формат даного файлу;
- matrix – показує, що в даному файлі зберігається матриця;
- coordinate – показує, що матриця написана в координатному форматі;

- real – вказує, що матриця складається з дійсних чисел (також тут може бути значення complex для позначення комплексних чисел);
- symmetric – вказує, що матриця симетрична, тобто дані знаходять тільки в нижній чи верхній частині матриці (може бути також значення general для позначення несиметричної матриці).

У другому рядку у нас знаходяться послідовно наступні дані: кількість рядків, кількість стовпчиків і кількість ненульових елементів.

А в наступних рядках у нас уже зберігаються самі дані ненульових значень матриці, де в кожному рядку зберігається: індекс рядка матриці, індекс стовпчика матриці та саме ненульове значення, що знаходиться за цими індексами в матриці.

Саме такий тип матриць ми будемо розв’язувати в практичній частині. Сам парсер такого формату я вирішив написати самостійно, і також буде в практичні частині.

3.1.2.2. Harwell-Boeing format.

Даний формат також застосовується для розріджених матриць для збереження пам’яті. Він використовує розширення .rsa. Розглянемо вміст цього файлу матриці, що ми розглядали попередньо:

```
1SYMMETRIC STIFFNESS MATRIX, TRANSMISSION TOWER, LUMPED MASSES          BCSSTK05
          413            10            81           322             0
RSA                      153           153          1288             0
(16I5)          (16I5)          (4E20.12)
    1   18   36   49   55   56   61   67   74   78   87   94  102  106  107  110
  116  123  126  131  138  141  149  159  165  173  183  189  197  207  213  221
   …    …    …    …    …    …    …    …    …    …    …    …    …    …    …    …
   .314313927913E+06  -.868578705282E+05   .563402403426E+05  -.351495467141E+05
   .937321245708E+05  -.351495467141E+05  -.351495467141E+05  -.937321245708E+05
```

… … … …

Цей формат використовуватися не буде, і він є більш складним, тому детально я його розглядати не буду.

3.2. Методи вирішення СЛАР з розрідженими матрицями.

У даному пункті я розгляну 2 ітераційних методи для вирішення СЛАР з розрідженими матрицями: метод градієнтного спуску і метод спряженого грядієнта. Саме їх я буду реалізовувати в практичній частині.

3.2.1. Метод градієнтного спуску (steepest descent method).

Перед розглядом методу градієнтного спуску потрібно зрозуміти властивності позитивно визначеної матриці.

3.2.1.1. Позитивно визначена матриця.

Вектори зазвичай зображуються стрілками, і якщо $Ax = b$, то A можна розглянути як трансформацію, що перетворює стрілку x у стрілку b. Математики часто класифікують матриці за тим, як вони перетворюють вектори, і якщо дві стрілки мають абсолютно однакову довжину та напрямок, A називають одиничною матрицею.

Тепер припустимо, що A перетворює x так, що b вказує в протилежному напрямку, або має ненульовий компонент, який вказує в протилежному напрямку. Якщо це так, то добуток $x \bullet b$ буде від’ємним. Аналогічно, якщо скалярний добуток дорівнює нулю, то A повертає x так, що отриманий вектор вказує в ортогональному напрямку.

Часто задачі лінійної алгебри вимагають матриці, які ніколи не змінюють напрямок вектора або не створюють вектор, який вказує в ортогональному напрямку. Тобто $x \bullet (Ax)$ має бути додатним для всіх x. Якщо

матриця відповідає цій вимозі, ми називаємо її позитивно визначеною. Ця властивість відіграватиме важливу роль у подальшому обговоренні.

3.2.1.2.Метод градієнтного спуску.

Пригадаємо, наша мета – знайти x у рівнянні $Ax = b$. Давайте зробимо x_0 першим припущенням вирішення системи. Можна перевірити це припущення, обчисливши Ax_0 і віднявши його від Ax. Якщо різниця більше, ніж деяке допустиме значення, то прийдеться зробити більше припущень. Але ми не можемо просто вгадати навмання. Потрібен метод, який гарантуватиме, що наступне припущення x_1 буде ближче до x, ніж x_0. Але як?

Щоб відповісти на це питання, потрібний математичний аналіз. Все починається з того, що ми маємо функцію f, нахил якої в кожній точці z дорівнює $Az - b$. Виведення цієї функції вихosдить за рамки цього обговорення, але кінцевий результат такий:

$$f(z) = \frac{1}{2} z \cdot (Az) - z \cdot b + c \tag{3.1}$$

$$f'(z) = Az - b \tag{3.2}$$

Тут $b = Ax$ і c – довільна константа. Крім нахилу, ця функція має дуже важливу властивість. Щоб зрозуміти це, потрібно встановити z рівним двом векторам $x + y$. Якщо ви припустите, що A є симетричним, ви зможете замінити $x + y$ замість z і отримати такий результат:

$$f(z) = f(x) + \frac{1}{2} y \cdot Ay \tag{3.3}$$

Якщо A є позитивно визначеним, другий доданок має бути більшим за нуль для всіх y. Отже, $f(z)$ приймає своє мінімальне значення, коли $z = x$. Це показано на рисунку 2, де зображено графік $f(z)$. На рисунку 3.2 входять як $f(x)$, так і $f(x_0)$, де x_0 – це перше припущення щодо x.

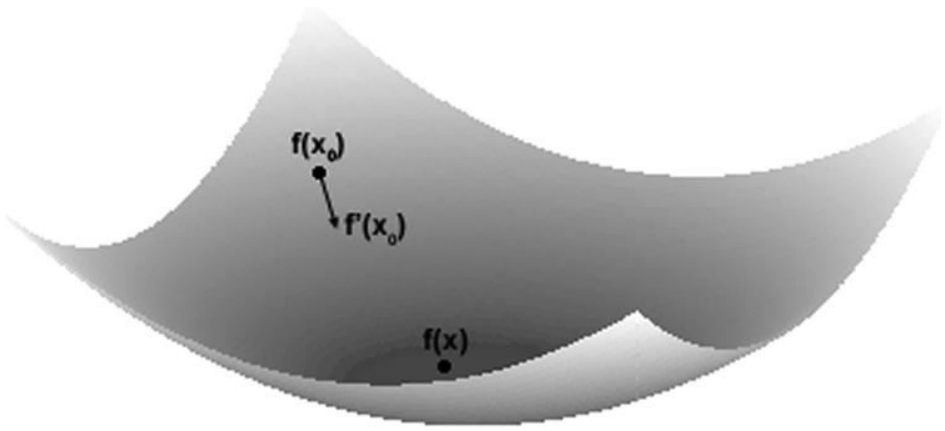

Рис. 3.2. Графік поверхні $f(z)$.

Нахил $f(z)$ дорівнює $Az - b$, тому $f'(x_0)$ дорівнює $Ax_0 - b$. $f'(x_0)$ визначає напрямок найбільшого підйому функції при $z = x_0$, тому напрямок найбільшого спаду при $z = x_0$ задається через $-f'(x_0)$, або $b - Ax_0$. Цей напрямок дуже важливий, і він є залишком r_0. З цим залишком можна зробити наступне припущення:

$$x_1 = x_0 + \alpha r_0 \tag{3.4}$$

Вектор залишку r_0 вказує напрямок, який повинен рухатися від x_0 до x_1. Скаляр α – це параметр, який змінюється по лінії від x_0 до x_1. Оскільки наша мета – опускатися до мінімуму, потрібно вибрати x_1 так, щоб $f(x_0 + \alpha r_0)$ було менше $f(z)$ у кожній іншій точці на прямій. Це мінімальне значення можна знайти, встановивши $f'(z)$ рівним нулю. Це задано в наступному рівнянні:

$$\frac{d\big(f(x_0 + \alpha r_0)\big)}{d\alpha} = f'(x_0 + \alpha_0 r_0) \cdot r_0 = 0 \tag{3.5}$$

У цьому рівнянні α_0 використовується для позначення відстані від x_0 до x_1. Це також показує, що для того, щоб $f(x_1)$ був мінімальним, $f'(x1)$ має бути ортогональним до r_0. Ми знаємо, що $r_1 = -f'(x_1)$, тому можемо

стверджувати, що r_0 має бути ортогональним до r_1. Це показано на рисунку 3.3.

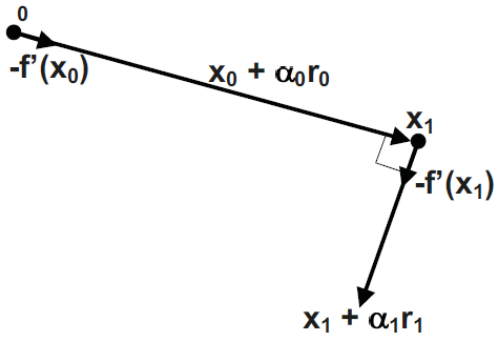

Рис. 3.3. Використання залишку для подальших припущень.

Ми знаємо, що r_1 дорівнює $b - Ax_1$, і що воно ортогональне до r_0. Маючи цю інформацію, можна визначити α_0 таким чином:

$$r_1 \cdot r_0 = 0$$
$$(b - Ax_1) \cdot r_0 = 0$$
$$\big(b - A(x_0 + \alpha_0 r_0)\big) \cdot r_0 = 0$$
$$(b - Ax_0 - \alpha_0 Ar_0) \cdot r_0 = 0$$
$$\alpha_0 Ar_0 \cdot r_0 = (b - Ax_0) \cdot r_0$$
$$\alpha_0 = \frac{r_0 \cdot r_0}{Ar_0 \cdot r_0} \tag{3.6}$$

За допомогою цієї формули для α_0 можна прийти до другого припущення x_1. Так можна продовжувати використовувати цей процес, щоб зробити подальші припущення (x_2, x_3, x_4 і так далі), і перевірити кожне, порівнявши довжину відповідного залишку $|r_i|$ з допустимим відхиленням.

Нам не потрібно обчислювати $r_i = b - Ax_i$ для кожного припущення. Замість цього можна підрахувати кожен r_i на основі попереднього r_i. Це показано в таких рівняннях:

$$r_i = b - Ax_i$$

$$r_i = b - A(x_{i-1} + \alpha_{i-1} r_{r-1}) = b - Ax_{i-1} + \alpha_{i-1} Ar_{i-1}$$
$$r_i = r_{i-1} + \alpha_{i-1} Ar_{i-1} \quad (3.7)$$

Перевага обчислення r_i таким чином полягає в тому, що добуток матриці-вектора Ar_{i-1} вже був обчислений у процесі знаходження α_{i-1}. Таким чином, потрібно виконати лише одну матрично-векторну операцію за ітерацію. Якщо $|r_i|$ спускається нижче допустимого значення, можна припинити обчислення: x_i – наша відповідь.

3.2.2. Метод спряженого градієнта (conjugate gradient method).

Метод спряженого градієнта робить серію здогадок для наближення x в $Ax = b$. Перед початком алгоритму важливо обговорити концепції векторної ортогоналізації та спряженості.

3.2.2.1. Векторна ортогоналізація.

У векторній ортогоналізації ви починаєте з групи векторів, порівнюєте їх по парах і вносите зміни, поки всі вони абсолютно не будуть відрізнятися один від одного. Два вектори абсолютно різні, або ортогональні, якщо їх точковий добуток дорівнює нулю. Метод Грама-Шмідта ортогоналізує набір векторів за допомогою векторних проекцій. Проекція вектора b на вектор a є складовою b, яка вказує в напрямку a. Це позначається $proj_a b$, і рівняння виглядає так:

$$proj_a b = \frac{a \cdot b}{|a|^2} a \quad (3.8)$$

Оскільки $proj_a b$ має той самий напрямок, що і $a, b - proj_a b$ повинен бути ортогональним a. Отже, ми можемо ортогоналізувати два вектори a і b, обчисливши проекцію b на a і віднявши її від b. Можна продовжити цей процес для трьох векторів. Якщо в просторі є також вектор c, три ортогональні вектори (v_1, v_2, v_3), що відповідають (a, b, c), можна обчислити наступним чином:

$$v_1 = a_1$$
$$v_2 = b - proj_{v_1} b \tag{3.9}$$
$$v_3 = c - proj_{v1} c - proj_{v2} c$$

Тоді якщо в нас є n векторів $(a_1, \dots, a_n)$, то обчислити ортогональні вектори можна наступною загальною формулою:

$$v_n = a_n - \sum_{i=1}^{n-1} proj_{v_1} a_n$$
$$v_n = a_n - \sum_{i=1}^{n-1} \frac{v_i \cdot a_n}{|v_i|^2} v_1 = a_n - \sum_{i=1}^{n-1} \frac{v_i \cdot a_n}{v_1 \cdot v_1} v_i \tag{3.10}$$

Метод Грама-Шмідта може створювати ортогональні вектори для будь-якої кількості неортогональних вхідних векторів за умови, що вектори є лінійно незалежними. Тобто жоден вектор не може бути виражений як зважена сума інших векторів. Якщо будь-який вектор лінійно залежний від інших, один з ортогональних векторів дорівнюватиме нулю.

3.2.2.2. Векторна спряженість.

Два вектори, p і q, є спряженими щодо матриці A, якщо $p \bullet Aq = 0$. Тобто два вектори є спряженими до матриці, якщо перший вектор ортогональний добутку матриці і другого вектор. Якщо А симетрична і позитивно визначена, то $p \bullet Aq = q \bullet Ap = 0$. Ми можемо зробити набір векторів, спряжених один до одного, використовуючи процес, подібний до методу Грама-Шмідта. Це показано в наступному рівнянні:

$$v_n = a_n - \sum_{i=1}^{n-1} \frac{v_i \cdot Aa_n}{v_i \cdot Av_i} v_i \tag{3.11}$$

Цей взаємозв'язок між векторами відіграє важливу роль у методі спряжених градієнтів у розв'язуванні розріджених матричних систем.

3.2.2.3. Метод спряженого градієнта.

У цьому методі ми маємо робити припущення, що x_i буде прямувати з x_0 до x. Щоб визначати чи ми йдемо в правильному напрямку, ми застосовуємо залишковий вектор r_i, який має напрямок з x_i до x_{i+1}, і чим менше це значення – тим ближче ми до правильного результату. Також крім цього вектору ми визначаємо інший вектор – p_i. Початковий напрям цього вектору дорівнює r_0, але кожний наступний напрямок буде спряжений до попереднього напрямку. Це виглядає так:

$$p_0 = r_0 \tag{3.12}$$

$$p_i = r_i - \sum_{j=0}^{i-1} \frac{p_j \cdot Ar_j}{p_j \cdot Ap_j} p_j \tag{3.13}$$

Маючи вибраний напрямок ми можемо зробити наступне припущення, яким буде *x* за допомогою ітераційного методу:

$$x_{i+1} = x_i + \alpha_i p_i, \text{де } \alpha_i = \frac{r_i \cdot r_i}{p_i \cdot Ap_i} \tag{3.14}$$

На отриманих знаннях ми можемо написати алгоритм роботи методу:

1. Зробити перше припущення x_0 (можна взяти 0.0);
2. Підрахувати початкове значення залишку і напрямку r_0 та p_0 (їх можна прирівняти до b);
3. Підрахувати довжину з x_i до x_{i+1}, $\alpha_i = r_i \bullet r_i / p_i \bullet Ap_i$;
4. Визначити наступне припущення $x_{i+1} = x_i + \alpha_i pi$;
5. Підрахувати наступний залишок $r_{i+1} = r_i - \alpha_i Ap_i$;
6. Підрахувати наступний напрям $p_{i+1} = r_{i+1} + (r_{i+1} \bullet r_{i+1} / r_i \bullet r_i) p_i$;

7. Знайти $|r_{i+1}|$. Якщо отримане значення менше ніж задана точність (0.001), то ми знайшли остаточний вектор x;
8. Повторити кроки 3 – 7 [13].

3.3. Висновки.

В третьому розділі ми визначили що таке розріджена матриця, які вони бувають і чому вона важлива при розв'язку диференційних рівнянь. Також вияснили в якому форматі зазвичай зберігаються розріджені матриці (MatrixMarket format і Harwell-Boeing format) для збереження пам'яті. Також ми детально описали 2 алгоритми для розв'язку СЛАР з розрідженими матрицями: метод градієнтного спуску (steepest descent method) та метод спряженого градієнта (conjugate gradient method).

РОЗДІЛ 4. ПРОГРАМНА РЕАЛІЗАЦІЯ ВИРІШУВАЧА СИСТЕМ ЛІНІЙНИХ АЛГЕБРАЇЧНИХ РІВНЯНЬ З РОЗРІДЖЕНИМИ МАТРИЦЯМИ. АНАЛІЗ РЕЗУЛЬТАТУ ВИКОНАНИХ ДОСЛІДЖЕНЬ.

4.1. Джерело даних розріджених матриць.

Існує чудовий інтернет ресурс: https://math.nist.gov/. Цей сервер надає інформацію про діяльність та послуги NIST (National Institute of Standards and Tecknology), пов'язані з прикладною математикою, статистикою та обчислювальною технікою.

Тут також можна знайти дані розріджених матриць, якщо перейти за посиланням https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/. Тут зберігаються найрізноманітніші тестові матриці, що виникають із задач лінійних систем, найменших квадратів та обчисленні власних значень із

широкого спектру наукових та інженерних дисциплін. Проблеми варіюються від малих матриць, які використовуються як протилежні приклади до гіпотез у дослідженні розріджених матриць, до великих тестових випадків, що виникають у додатках. Дану колекцію розробили Іен Дафф, Роджер Граймс та Джон Льюїс.

Одна з вимог до розріджених матриць, для застосування даними алгоритмами, – це симетричність. Саме такі матриці знаходяться списках BCSSTRUC1-BCSSTRUC6, які відповідають динамічному аналізу в інженерії конструкцій. Саме їх ми будемо тестувати для наших алгоритмів.

4.2. Інструменти для розробки вирішувача СЛАР з використанням GPU.

Взагалі першопочатково графічний процесор створювався в основному для того, щоб відображати пікселі на екрані. Ми вже розглядали з чого складається відеокарта, і причина чому саме її застосовують для відображення пікселів – це паралельні обчислення, дуже багато паралельних обчислень.

Є кілька графічних API для взаємодії з GPU: OpenGL, DirectX, Vulkan і т.д. Вони постійно розвиваються та створюються нові API для нового апаратного забезпечення. Вони успішно використовуються в іграх і в будь-яких програмах, де потрібно відображати хоч щось пов’язане з графікою за допомогою шейдерів, які також унікальні для кожного API.

Але люди вияснили, що паралельно можна обчислювати не тільки пікселі на екрані, а й для інших цілей. Так з’явився GPGPU - General-purpose computing on graphics processing units - техніка використання графічного процесора відеокарти, призначеного для комп'ютерної графіки, для виробництва математичних обчислень, які зазвичай проводить центральний процесор. Це стало можливим завдяки додаванню програмованих шейдерних блоків і вищої арифметичної точності растрових конвеєрів, що дозволяє

розробникам програмного забезпечення використовувати потокові процесори відеокарт для виконання неграфічних обчислень [14].

Розглянемо кілька найпопулярніших інструментів для програмування GPGPU.

4.2.1. CUDA.

CUDA (Compute Unified Device Architecture) – це програмний рівень, який надає прямий доступ до віртуального набору інструкцій GPU та паралельних обчислювальних елементів для виконання обчислювальних ядер [15].

CUDA розроблена для роботи з такими мовами програмування, як C, C++ і Fortran. Ця доступність полегшує фахівцям із паралельного програмування використовувати ресурси графічного процесора, на відміну від попередніх API, таких як Direct3D і OpenGL, які вимагали розширених навичок програмування графіки.

CUDA була створена компанією Nvidia. Коли вона була вперше представлена, назва була абревіатурою від Compute Unified Device Architecture, але пізніше Nvidia відмовилася від загального використання цієї абревіатури.

CUDA – це потужний інструмент, але його величезний мінус у тому, що він працює лише на відеокартах NVidia, що його дуже обмежує. Особисто у мене немає потрібного апаратного забезпечення, тому я використовую інший інструмент, який детальніше опишу в наступному пункті.

4.2.2. OpenCL.

OpenCL (Open Computing Language) – відкритий стандарт паралельного програмування для гетерогенних платформ (зокрема, для GPGPU), що

включають центральні, графічні процесори та інші дискретні обчислювальні пристрої. OpenCL дозволив використовувати потужності GPU на різних програмних та апаратних платформах.

Модель платформи (platform model) надає високорівневий опис гетерогенної системи. Центральним елементом даної моделі виступає поняття хоста (host) - первинного пристрою, яке управляє OpenCL-обчисленнями та здійснює всі взаємодії з користувачем. Хост завжди представлений в єдиному екземплярі, в той час як OpenCL-пристрої (devices), на яких виконуються OpenCL-інструкції, можуть бути представлені у множині. OpenCL-пристроєм може бути CPU, GPU, DSP або будь-який інший процесор у системі, що підтримується встановленими у системі OpenCL-драйверами. OpenCL-пристрої логічно діляться моделлю на обчислювальні модулі (compute units), які у свою чергу поділяються на обробні елементи (processing elements). Обчислення на OpenCL-пристроях насправді відбуваються на обробних елементах. На рисунку 4.1 схематично зображена архітектура OpenCL.

OpenCL спочатку було розроблено в компанії Apple Inc. Apple внесла пропозиції щодо розробки специфікації до комітету Khronos. Незабаром AMD вирішила підтримати розробку OpenCL (і DirectX 11), який повинен замінити фреймворк Close to Metal.

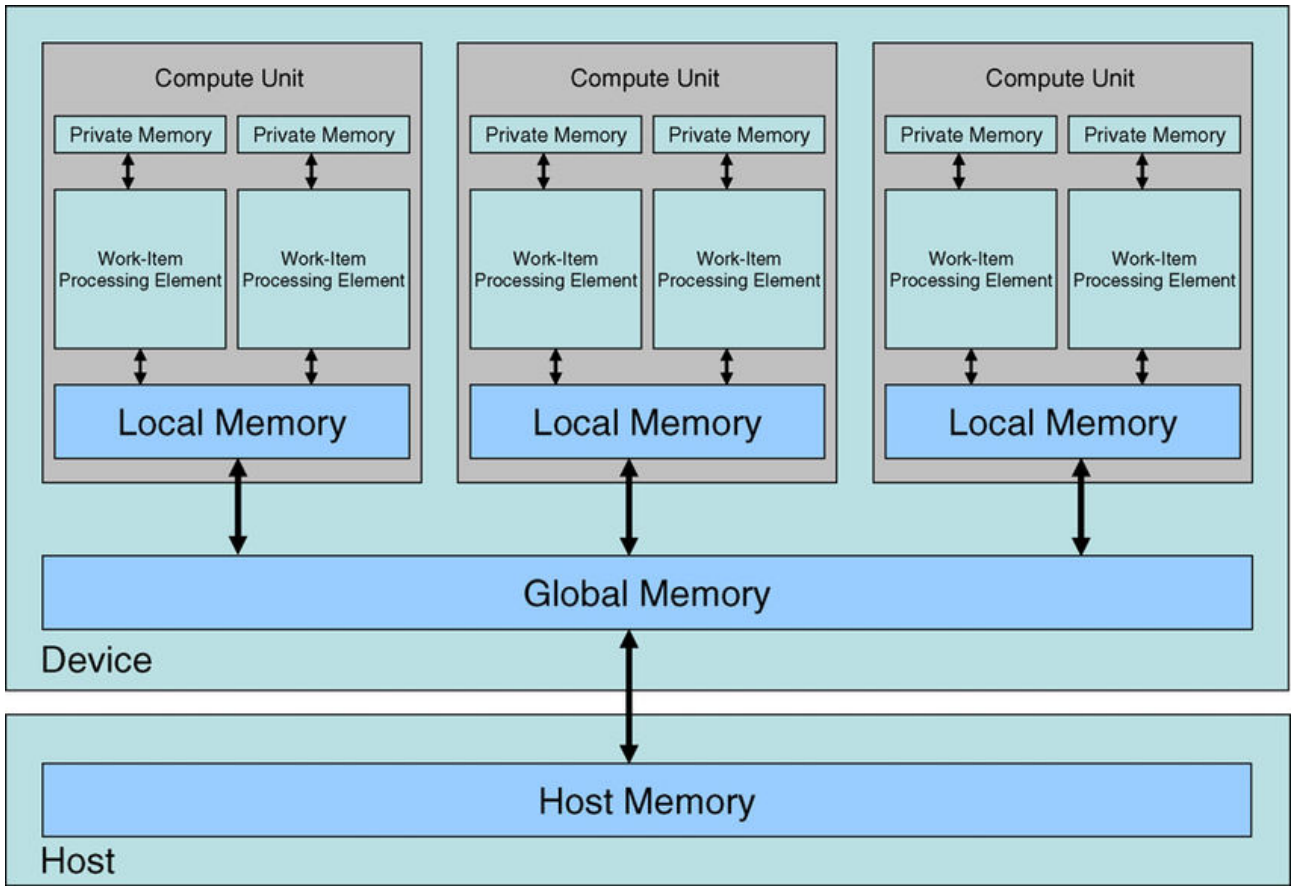

Рис. 4.1. Схематична архітектура OpenCL.

4.2.2.1. Програмна модель OpenCL.

Ядро (kernel) – функція, що виконується пристроєм. Має в описі специфікацію __kernel.

Програма (program) – набір ядер, а також, можливо, допоміжних функцій, які вони викликають, і константних даних.

Аплікація (application) – комбінація програм, що працюють на керуючому вузлі та обчислювальних пристроях.

Команда (command) – операція OpenCL, призначена для виконання (виконання ядра на пристрої, маніпуляції з пам'яттю тощо)

4.2.2.2. Об'єкти OpenCL.

Об'єкт (object) – абстрактне представлення ресурсу, керованого OpenCL API (об'єкт ядра, пам'яті тощо)

Дескриптор (handle) – непрозорий тип, що посилається на об'єкт, що виділяється OpenCL. Будь-яка операція з об'єктом виконується через дескриптор.

Черга команд (command-queue) – об'єкт, що містить команди виконання на пристрої.

Об'єкт ядра (kernel object) – зберігає окрему функцію ядра програми разом із значеннями аргументів.

Об'єкт події (event object) – зберігає стан команди. Призначений для синхронізації.

Об'єкт буфера (buffer object) – послідовний набір байтів. Доступний з ядра через вказівник і керуючого вузла за допомогою викликів API.

Об'єкт пам'яті (memory object) – посилається область глобальної пам'яті.

4.2.2.3. Пристрої OpenCL.

Робочий елемент (work-item) – набір ядер, що паралельно виконуються, на пристрої, викликаних за допомогою команди.

Робоча група (work-group) – набір взаємодіючих робочих елементів, що виконуються одному пристрої.

Обробний елемент (processing element) – віртуальний скалярний процесор. Робочий елемент може виконуватись на одному або кількох обробних елементах.

Обчислювальний вузол (compute unit) – виконує одну робочу групу. Пристрій може складатися з одного або кількох обчислювальних елементів.

4.2.2.4. Контекст OpenCL.

Контекст (context) – середовище, у якому виконуються ядра, і навіть область визначення синхронізації та управління пам'яттю. Включає:

- набір пристроїв;
- пам'ять, доступну пристроям;
- властивості пам'яті;
- одну чи кілька черг команд.

Об'єкт програми (program object) включає:

- посилання на зв'язаний контекст;
- вихідний текст або двійкове представлення;
- останній вдало зібраний код, список пристроїв, для яких він зібраний, налаштування та журнал збірки;
- набір поточних зв'язаних ядер [5].

4.3. Практична частина.

Реалізуємо метод спряжених градієнтів, який ми описували раніше, за допомогою ядра OpenCL:

```
__kernel void conjugateGradient(int dim, int num_vals,
                    __local double *r, __local double *A_times_p, __local double *p,
                    __constant int *rows, __constant int *cols, __constant double *A,
                    __constant double *b, __global double *x, __global double *result)
{
    int id = get_global_id(0);

    local double alpha, r_length, old_r_dot_r, new_r_dot_r;
    local int iteration;

    int start_index = -1;
    int end_index = -1;
    double Ap_dot_p;
```

```
for (int i = id; i < num_vals; i++)
{
    if ((rows[i] == id) && (start_index == -1))
    {
        start_index = i;
    }
    else if ((rows[i] == id + 1) && (end_index == -1))
    {
        end_index = i - 1;
        break;
    }
    else if ((i == num_vals - 1) && (end_index == -1))
    {
        end_index = i;
    }
}

x[id] = 0.0;
r[id] = b[id];
p[id] = b[id];

barrier(CLK_LOCAL_MEM_FENCE);

if (id == 0)
{
    old_r_dot_r = 0.0;
    for (int i = 0; i < dim; i++)
    {
        old_r_dot_r += r[i] * r[i];
    }
    r_length = sqrt(old_r_dot_r);
}

barrier(CLK_LOCAL_MEM_FENCE);

iteration = 0;
while ((iteration < 5000) && (r_length >= 0.01))
{
    A_times_p[id] = 0.0;
    for (int i = start_index; i <= end_index; i++)
    {
        A_times_p[id] += A[i] * p[cols[i]];
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    if (id == 0)
    {
        Ap_dot_p = 0.0;
        for (int i = 0; i < dim; i++)
        {
            Ap_dot_p += A_times_p[i] * p[i];
        }
        alpha = old_r_dot_r/Ap_dot_p;
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    x[id] += alpha * p[id];
    r[id] -= alpha * A_times_p[id];
```

```
        barrier(CLK_LOCAL_MEM_FENCE);

        if (id == 0)
        {
            new_r_dot_r = 0.0;
            for (int i = 0; i < dim; i++)
            {
                new_r_dot_r += r[i] * r[i];
            }
            r_length = sqrt(new_r_dot_r);
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        p[id] = r[id] + (new_r_dot_r/old_r_dot_r) * p[id];

        barrier(CLK_LOCAL_MEM_FENCE);

        old_r_dot_r = new_r_dot_r;

        if (id == 0) iteration++;

        barrier(CLK_LOCAL_MEM_FENCE);
    }

    result[0] = (double)iteration;
    result[1] = r_length;
}
```

Даний код був взятий зі згаданого в попередніх розділах книзі «OpenCL in Action. How to accelerate graphics and computations».

Тут є все, що ми описували в теорії по OpenCL: __kernel – ядро програми, яке запускається в одній work-group і виконується всіма доступними work-item’ами в цій робочій групі. Також аргументи, які знаходяться в своїй пам’яті: __global, __constant (як __global тільки значення не змінюються), __local (описано на рис. 4.1).

get_global_id – це індетифікатор work-item’у. По суті ми пишемо код як для однопоточної програми, тільки в ній враховуються унікальні ID – по суті індетифікатор потоку в GPU, що нумеруються послідовно від нуля, до кількості потоків, що використовуватимуться в ядрі (навіть якщо це той же потік, але він обробляє іншу область даних, то він матиме унікальний

ідентифікатор; щоб отримати істинне ID кожного потоку в своїй work group, потрібно використати get_local_id).

Варто також сказати, що повністю розпаралелити таку задачу як вирішення СЛАР, щоб всі обчислювання виконувалися незалежно – неможливо, так як значення всіх попередніх обрахунків потрібні для обрахування значень в наступному кроці, тому для таких ситуацій потрібна синхронізація. Синхронізацію нам надає функція *barrier(CLK_LOCAL_MEM_FENCE)*. Вона дозволяє синхронізувати потоки, які звертаються до локальної пам'яті. Щоб синхронізувати доступ до глобальної пам'ті, потрібно використати *barrier(CLK_GLOBAL_MEM_FENCE)*.

З сихронізацією є лише одна проблема: вона може синхронізувати work item'и лише однієї work group'и. Це значить, що алгоритм обмежений максимальною кількістю work item'ів в одній робочій групі. Це дуже обмежує розмірність матриці яку ми можемо порахувати (в даному випадку розмірність матриці не перевищує кількості робочих одиниць в робочій групі). Знаючи це обмеження я зміг масштабувати алгоритм на будь-яку розмірність матриці, але про це потім.

Щоб запустити нашу OpenCL програму, потрібно написати код хоста, який все це викликатиме, в даному випадку це мова C++17. Відобразимо Код хоста немасштабованого методу спряженого градієнта:

```
Result conjugateGradientGpu(const SparseMatrix& aSparseMatrix)
{
    const int dimension = aSparseMatrix.getDimension();
    const int numValues = aSparseMatrix.getValuesNum();

    std::vector<double> x(dimension);
    std::vector<double> result(2);

    const auto [context, program, device] = initOpenCL("kernels/conjugateGradient.cl");

    cl::Kernel kernel(program, "conjugateGradient");
```

```
    int deviceMaxWorkGroupSize = device.getInfo<CL_DEVICE_MAX_WORK_GROUP_SIZE>();
    if (dimension > deviceMaxWorkGroupSize)
    {
        std::cerr << "\nDimension of matrix is bigger than max work group size of the device. Use
scaledConjugateGradientGpu() algorithm instead" << std::endl;
        exit(-3);
    }

    cl::Buffer rowsBuf(context, CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS | CL_MEM_COPY_HOST_PTR,
numValues * sizeof(int), const_cast<int*>(aSparseMatrix.getRowIds()));
    cl::Buffer colsBuf(context, CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS | CL_MEM_COPY_HOST_PTR,
numValues * sizeof(int), const_cast<int*>(aSparseMatrix.getColIds()));
    cl::Buffer valuesBuf(context, CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS | CL_MEM_COPY_HOST_PTR,
numValues * sizeof(double), const_cast<double*>(aSparseMatrix.getValues()));
    cl::Buffer bBuf(context, CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS | CL_MEM_COPY_HOST_PTR,
dimension * sizeof(double), const_cast<double*>(aSparseMatrix.getVectorB()));
    cl::Buffer xBuf(context, CL_MEM_READ_WRITE, dimension * sizeof(double));
    cl::Buffer resultBuf(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY, 2 * sizeof(double));

    kernel.setArg(0, sizeof(int), &dimension);
    kernel.setArg(1, sizeof(int), &numValues);
    kernel.setArg(2, cl::Local(dimension * sizeof(double)));
    kernel.setArg(3, cl::Local(dimension * sizeof(double)));
    kernel.setArg(4, cl::Local(dimension * sizeof(double)));
    kernel.setArg(5, rowsBuf);
    kernel.setArg(6, colsBuf);
    kernel.setArg(7, valuesBuf);
    kernel.setArg(8, bBuf);
    kernel.setArg(9, xBuf);
    kernel.setArg(10, resultBuf);

    cl::CommandQueue queue(context, device, cl::QueueProperties::Profiling);


    cl_ulong kernel_compute_time = 0;
    cl_ulong read_buffer_time = 0;

    auto computeLinearSystem = [&]()
    {
        cl::Event event;
        queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(dimension), cl::NDRange(dimension),
nullptr, &event);
        kernel_compute_time += getComputeTime(event);

        queue.enqueueReadBuffer(xBuf, CL_TRUE, 0, x.size() * sizeof(double), x.data(), nullptr, &event);
        read_buffer_time += getComputeTime(event);
        queue.enqueueReadBuffer(resultBuf, CL_TRUE, 0, result.size() * sizeof(double), result.data(), nullptr,
&event);
        read_buffer_time += getComputeTime(event);
    };

    auto timeInfo = std::make_unique<NonScaledTimeInfo>();
    timeInfo->total_compute_time  = Timer::computeTime(computeLinearSystem);
    timeInfo->total_kernel_time   = Timer::toAppropriateMeasure(kernel_compute_time + read_buffer_time);
    timeInfo->kernel_compute_time = Timer::toAppropriateMeasure(kernel_compute_time);
    timeInfo->read_buffer_time    = Timer::toAppropriateMeasure(read_buffer_time);

    return { x, static_cast<int>(result[0]), result[1], std::move(timeInfo) };
}
```

В дану функцію ми передаємо створений мною клас SparseMatrix, який зберігає в собі всі необхідні дані для визначення матриці. Цей клас коротко виглядає якось так:

```
class SparseMatrix
{
public:
    SparseMatrix(const std::string& aPathToMatrix);

    void open(const std::string& aPathToMatrix);
    void clear();
    void print(std::ostream& aStream);

    int getDimension() const { return mDimension; }
    int getValuesNum() const { return mNumValues; }
    const int* getRowIds() const { return mRowIds.data(); }
    const int* getColIds() const { return mColIds.data(); }
    const double* getValues() const { return mValues.data(); }
    const double* getVectorB() const { return mB.data(); }

    void fillVectorBWithRandomValues(double aMinValue, double aMaxValue);
    void fillVectorBWithValue(double aValue);

private:
    std::vector<int> mRowIds;
    std::vector<int> mColIds;
    std::vector<double> mValues;
    std::vector<double> mB;
};
```

Повертаючись до нашої функції хоста, після ініціалізації OpenCL ми отримуємо її контекст, програму і девайс, які були згадані вище (я вирішив не вдаватися в подробиці як їх ініціалізовувати, тому в коді це показано як виклик функції initOpenCL). Потім створюємо ядро, в якому зберігається наш алгоритм і передаємо туди наші аргументи. Якщо аргумент ядра приймає якийсь масив даних, то ми маємо ініціалізувати для кожного з них спеціальний буфер. Далі ми створюємо чергу команд і кажемо цій черзі, щоб ми спочатку запустили нашу паралельну OpenCL програму, а потім зчитали результат, виміривши при цьому час виконання ядра і зчитування буферу.

І вкінці вимірюємо весь час роботи алгоримту та передаємо виміри в кастомний об'єкт NonScaledTimeInfo з абстрактним класом TimeInfo. Для

вимірів, крім доступних функцій самого фреймворку OpenCL, був також написаний інструмент Timer. В даному випадку заміри тут такі:

- Повністю витрачений час на роботу алгоритму;
- Час виконання всіх задач в черзі;
- Час витрачений тільки на роботу обчислюючого ядра, не враховуючи час на зчитування і записування даних (буферів) у відеопам'ять.
- Час витрачений на зчитування буферів.

Поглянемо результат обчислення СЛАР з розрідженою матрицею, яку ми розглядали в третьому розділі:

```
Choose an available matrix to solve:
1. bcsstk05.mtx
2. bcsstk06.txt
3. bcsstk08.txt
4. bcsstk15.mtx
5. bcsstk17.mtx
6. bcsstk24.mtx
7. bcsstk25.mtx
8. bcsstm12.mtx
9. pll_post_short_A.txt
10. sparse_matrix_112.txt
11. sparse_matrix_132.txt
12. sparse_matrix_153.txt
13. sparse_matrix_420.txt

Select option: 1

Dimention of matrix : 153x153
Number of non-zero values: 2423


Fill vector b:
1. With random values
2. With concrete value

Select option: 2

Enter value: 200


Choose method that solves SLAE:
1. Conjugate gradient method on GPU (matrix dimension must be less than work
group size of GPU)
2. Conjugate gradient method on GPU
3. Conjugate gradient method on CPU
4. Conjugate gradient method on GPU and CPU
5. Conjugate gradient method using ViennaCL library (no preconditioners)
```

```
6. Conjugate gradient method using ViennaCL library (Jacobi preconditioner)
7. Steepest descent method on GPU
8. Steepest descent method on CPU

Select option: 1


Choose an available platform:
1. Intel(R) OpenCL HD Graphics
        Devices:
        Intel(R) UHD Graphics 620
                MAX_WORK_GROUP_SIZE: 256
                MAX_COMPUTE_UNITS: 24
2. AMD Accelerated Parallel Processing
        Devices:
        Iceland
                MAX_WORK_GROUP_SIZE: 256
                MAX_COMPUTE_UNITS: 6

Select option: 1

Using platform: Intel(R) OpenCL HD Graphics
Using device: Intel(R) UHD Graphics 620


Solving of SLAE...


Results:

x = [ 0.675869, 0.0178154, 0.681222, 0.59276, -0.104837, 0.593641, 0.593099, -
0.0454896,  0.593961,  0.593145,  0.0167591,  0.597255,  0.590842,  0.154371,
0.607733,  0.589069,  0.0800806,  0.597508,  0.590196,  0.0176824,  0.594331,
0.534042,  -0.0451019,  0.536061,  0.536409,  0.0163401,  0.537671,  0.533901,
0.0789002,  0.539263,  0.529238,  0.0173334,  0.534734,  0.477197,  -0.126519,
0.476252,  0.477044,  -0.0436846,  0.475734,  0.476743,  0.0157192,  0.478962,
0.473917,  0.194448,  0.494126,  0.472329,  0.0762763,  0.478883,  0.473946,
0.016797,  0.476255,  0.422163,  -0.0401398,  0.421682,  0.420653,  0.014878,
0.418825,  0.421744,  0.0710664,  0.425789,  0.413333,  0.016087,  0.417302,
0.369549,  -0.0897448,  0.364249,  0.369656,  -0.0343436,  0.366108,  0.369954,
0.0138527,  0.369895,  0.367351,  0.140645,  0.379064,  0.364995,  0.0631742,
0.368732,  0.366937,  0.0152388,  0.366811,  0.310291,  -0.0424398,  0.302763,
0.302823,  0.0125875,  0.299229,  0.308553,  0.0682704,  0.310962,  0.296214,
0.0140518, 0.298162, 0.2491, -0.047847, 0.243336, 0.24908, 0.0117116, 0.247094,
0.24431,  0.07242,  0.246571,  0.245219,  0.0142522,  0.237145,  0.202901,  -
0.0504449,  0.186632,  0.196735,  0.0102526,  0.192286,  0.200081,  0.0708725,
0.201466,  0.190731,  0.0121676,  0.189824,  0.151049,  -0.0478442,  0.142771,
0.146479,  0.008592,  0.141715,  0.147802,  0.0661214,  0.148485,  0.140941,
0.0124083,  0.125039,  0.109056,  -0.0426112,  0.0973334,  0.105892,  0.00682139,
0.101126,  0.104985,  0.0570745,  0.105143,  0.100846,  0.00850996,  0.0965802,
0.0444287, -0.0249693, 0.0454527, 0.0397917, 0.00346434, 0.0365137, 0.0431992,
0.0319719, 0.0431648, 0.0365142, 0.003549, 0.0387328, ]


Ax = [ 200, 200.001, 200, 200, 200, 200, 200.002, 200, 200, 199.999, 200, 200,
200.001,  200,  200,  199.999,  200,  200,  200.002,  200,  200,  200,  200,  200.001,
199.999, 200, 200, 200, 200, 200.001, 200, 200.001, 199.999, 200.001, 200.002,
```

```
199.999, 200, 200, 199.999, 200, 200, 200, 200, 199.999, 199.999, 200.001, 200,
200, 199.999, 200, 200.001, 200, 200.001, 200, 200, 199.999, 199.999, 200, 200,
200.001, 200, 200.001, 200, 199.999, 200, 200, 200.001, 200, 200, 199.999,
200.001, 200, 200.001, 200, 200, 199.999, 200.001, 200.001, 200.001, 199.999,
200.001, 200, 200.001, 199.999, 199.999, 199.998, 200, 200, 199.999, 200.001,
199.999, 200.001, 200, 200, 200, 200, 200, 200.001, 200.001, 200, 200.001,
199.999, 200, 199.999, 199.999, 199.999, 200, 199.999, 200, 199.999, 200, 200,
199.999, 200, 199.999, 200.002, 200.001, 200, 200, 200.001, 199.999, 200, 200,
200, 200, 200, 200, 200.001, 199.999, 199.999, 200, 200.001, 200, 199.999, 200,
200, 200, 200, 200, 200.001, 200, 200, 200, 200, 200, 199.999, 199.999, 200,
200, 200, 200, 200, ]

Iterations: 251
Residual length: 0.00850857

Total compute time:  19.6259    milliseconds
Total kernel time:   14.9233    milliseconds

Kernel compute time: 14.9223    milliseconds    99.9926%
Read buffer time:    1.099      microseconds    0.0073643%
```

Спробуймо описати результат:

1. Спочатку ми вибираємо файл в якому зберігається розріджена матриця. Вибираємо варіант `1.bcsstk05.mtx`.
2. Потім нам надається вибір як заповнити вектор b у виразі $Ax = b$. Ми вибираємо, що цей вектор весь заповнюється значенням 200.
3. Далі ми маємо вибрати метод як буде розв’язуватися СЛАР. Всього в нас є 8 таких варіантів. Ми вибираємо немасштабований метод спряженого градієнта, що використовує GPU.
4. Потім потрібно вибрати платформу на якій будуть виконуватися обчислення. Всього на моєму комп’ютері 2 такі платформи: інтегрована відеокарта від Inter та дискретна відеокарта від AMD. У кожній з них є інформація, яка максимальна кількість work unit’ів може бути в work group і скільки всього work group’ів (compute units) є в девайсі.
5. На останок у нас виконуються самі обрахунки. Розмірність матриці не перевищує значення MAX_WORK_GROUP_SIZE, тому ми можемо розв’язати СЛАР. В результаті ми отримуємо обчислений

вектор x. Для перевірки ми також виконуємо векторно-матричний добуток Ax, щоб перевірити, чи значення вектора b співпадає зі значеннями, які ми йому передали. Ми отримали на всіх значеннях число 200 з деякою похибкою, як ми й заповнювали. Також нам надається додаткова інформація, а саме: кількість ітерацій, залишок, та час витрачений на роботу алгоритму.

Крім методу спражених градієнтів на GPU був реалізований цей же масштабований метод на GPU, який міг обчислювати матриці з будь-якою розмірністю. Також метод на CPU, і об'єднаний метод, що застосовує обчислення і на GPU, і на CPU. Ще я віднайшов бібліотеку ViennaCL, яка дозволяє також вирішувати СЛАР використовуючи CUDA, OpenCL, OpenMP, тому я добавив бібліотечну реалізуцію алгоритму спряжених градієнтів для порівняння зі своїми методами. Також я вирішив добавити метод градієнтного спуску на GPU та CPU, але він майже на всіх матрицях розбігався, тому його використання є недоцільним.

Метод, що ми розглянули, був уже написаний іншою людиною, а тому що це за наукова робота без мого особистого вкладу? Знаючи про обмеження в розмірності матриці в цьому алгоритмі я вирішив масштабувати вирішувач СЛАР з розрідженими матрицями, що використовує метод спряженого градієнту для матриць будь-якої розмірності. Головне, щоб вона була симетричною і позитивно визначеною, але і тоді не факт, що алгоритм буде збігатися.

Головною ідеєю було розбити алгоритм на окремі ядра, які точно виконуються незалежно (ці частини коду знаходяться між фукнціями barrier) і виконувати синхронізацію в коді хоста.

Ось як виглядає код з окремими ядрами:

```
__kernel void init(int num_vals, __global int *start_index, __global int *end_index,
```

```
                __global double *x, __global double *r, __global double *p,
                __constant int *rows, __constant double *b)
{
    int id = get_global_id(0);

    start_index[id] = -1;
    end_index[id] = -1;

    for (int i = id; i < num_vals; i++)
    {
        if ((rows[i] == id) && (start_index[id] == -1))
        {
            start_index[id] = i;
        }
        else if ((rows[i] == id + 1) && (end_index[id] == -1))
        {
            end_index[id] = i - 1;
            break;
        }
        else if ((i == num_vals - 1) && (end_index[id] == -1))
        {
            end_index[id] = i;
        }
    }

    x[id] = 0.0;
    r[id] = b[id];
    p[id] = b[id];
}

__kernel void update_r_length(int dim, __constant double *r, __global double *r_dot_r,
__global double *r_length)
{
    r_dot_r[0] = 0.0;
    for (int i = 0; i < dim; i++)
    {
        r_dot_r[0] += r[i] * r[i];
    }
    r_length[0] = sqrt(r_dot_r[0]);
}

__kernel void update_A_times_p(__global double *A_times_p, __constant int *start_index,
__constant int *end_index, __constant double *A, __constant double *p, __constant int *cols)
{
    int id = get_global_id(0);

    A_times_p[id] = 0.0;
    for (int i = start_index[id]; i <= end_index[id]; i++)
    {
        A_times_p[id] += A[i] * p[cols[i]];
    }
}

__kernel void calculate_alpha(int dim, __constant double *old_r_dot_r, __constant double
*A_times_p, __constant double *p, __global double *alpha)
{
    double Ap_dot_p = 0.0;
    for (int i = 0; i < dim; i++)
    {
        Ap_dot_p += A_times_p[i] * p[i];
```

```
    }
    alpha[0] = old_r_dot_r[0]/Ap_dot_p;
}

__kernel void update_guess(__global double *x, __global double *r, __constant double *alpha,
__constant double *p, __constant double *A_times_p)
{
    int id = get_global_id(0);

    x[id] += alpha[0] * p[id];
    r[id] -= alpha[0] * A_times_p[id];
}

__kernel void update_direction(__constant double *old_r_dot_r, __constant double
*new_r_dot_r, __constant double *r, __global double *p)
{
    int id = get_global_id(0);

    p[id] = r[id] + (new_r_dot_r[0]/old_r_dot_r[0]) * p[id];
}

__kernel void sync_r_dot_r(__global double *old_r_dot_r, __constant double *new_r_dot_r)
{
    old_r_dot_r[0] = new_r_dot_r[0];
}
```

Код хоста масштабованого алгоритму дуже схожий на код хоста немасштабованого, може тільки трохи більше змінних, об’єктів буферів і ядер. Тому я тут покажу тільки те, як ми передаємо в чергу задач OpenCL наші процеси:

```
auto computeLinearSystem = [&]()
{
    queue.enqueueNDRangeKernel(init_kernel, cl::NullRange, cl::NDRange(global),
cl::NDRange(local));
    queue.enqueueNDRangeKernel(update_r_length_old_kernel, cl::NullRange, cl::NDRange(1),
cl::NDRange(1));

    queue.enqueueReadBuffer(r_length_Buf, CL_TRUE, 0, sizeof(double), &r_length);

    while (iterations < 50000 && r_length >= 0.01)
    {
        queue.enqueueNDRangeKernel(update_A_times_p_kernel, cl::NullRange,
cl::NDRange(global), cl::NDRange(local));
        queue.enqueueNDRangeKernel(calculate_alpha_kernel, cl::NullRange, cl::NDRange(1),
cl::NDRange(1));
        queue.enqueueNDRangeKernel(update_guess_kernel, cl::NullRange, cl::NDRange(global),
cl::NDRange(local));
        queue.enqueueNDRangeKernel(update_r_length_new_kernel, cl::NullRange,
cl::NDRange(1), cl::NDRange(1));
        queue.enqueueNDRangeKernel(update_direction_kernel, cl::NullRange,
cl::NDRange(global), cl::NDRange(local));
        queue.enqueueNDRangeKernel(sync_r_dot_r_kernel, cl::NullRange, cl::NDRange(1),
cl::NDRange(1));

        queue.enqueueReadBuffer(r_length_Buf, CL_TRUE, 0, sizeof(double), &r_length);
```

```
        iterations++;
    }

    queue.enqueueReadBuffer(xBuf, CL_TRUE, 0, x.size() * sizeof(double), x.data());
};
```

Я тут вирішив залишити тільки основний код алгоритму. По суті ми зробили то й же алгоритм спряженого градієнта, але тепер ми можемо обчислити його на GPU з будь-якою розмірністю!

Спробуймо розглянути результат вимірів мастабованого алгоритму, обчисливши матрицю розмірністю 3948x3948 та кількістю ненульових елементів: 117816.

```
Iterations: 22043
Residual length: 0.00938629

Total compute time:          60.4151    seconds
Total kernel time:           32.3333    seconds

Init time:                   55.9919    milliseconds    0.173171%
Init residual length time:   684.333    microseconds    0.00211649%
Calculate A*p time:          3.08074    seconds         9.52807%
Calculate alpha time:        14.2676    seconds         44.1266%
Update guess time:           176.983    milliseconds    0.547371%
Update residual length time: 14.5519    seconds         45.0058%
Sync r*r time:               56.443     milliseconds    0.174566%
Read buffers time:           7.3384     milliseconds    0.0226961%
```

З даних результатів, ми можемо побачити, що час витрачений на все обчислення алгоритму більший за час корисного обчислення в самих ядрах майже в 2 рази. Тому можемо зробити висновок, що в хості багато часу витрачається на виклик функцій OpenCL і на саму синхронізацію роботи ядер.

Також видно, що на обчислення значення альфи і оновлення довжини залишку витрачається майже 90% всього часу обчислень. Як виявляється, то саме ці частини навіть в GPU використовують один потік для обрахунків, через неможливість розпаралелювання. Як спосіб оптимізації я вирішив добавити метод, що однопоточні частини рахує на CPU, і як результат виходить наступне:

```
Iterations: 23078
```

```
Residual length: 0.0098407

Total compute time:          28.6407    seconds
Total kernel time:           11.3866    seconds

Init time:                   393.99     milliseconds    3.46011%
Init residual length time:   31.1       microseconds    0.000273128%
Calculate A*p time:          9.14537    seconds         80.3169%
Calculate alpha time:        384.021    milliseconds    3.37257%
Update guess time:           317.242    milliseconds    2.7861%
Update residual length time: 431.974    milliseconds    3.7937%
Sync r*r time:               60.3596    milliseconds    0.530093%
Read buffers time:           435.641    milliseconds    3.8259%
```

Як бачимо, завдякій маленькій зміні ми змогли пришвидшити сам алгоритм в 2 рази, а час корисного обчислення в 3!

Об'єднаємо всі обрахунки і з'єднаємо все в одну табличку:

Таблиця 4.1. Таблиця вимірювань часу матриці 153x153.

	153x153 (2423)						
	conjugateGradient (GPU, non scaled)		conjugateGradient (GPU, scaled)		conjugateGradient (GPU+CPU, scaled)		conjugateGradient (CPU, scaled)
	Intel	AMD	Intel	AMD	Intel	AMD	
Total compute time	19.38 ms	26.73 ms	226.9 ms	211.4 ms	136.3 ms	267.8 ms	851 µs
Total kernel time	14.9 ms	21.06 ms	24.4 ms	27.07 ms	8.51 ms	10.27 ms	-
Init time	14.9 ms	20.67 ms	876 µs	1.114 ms	821.7 µs	1.11 ms	5.3 µs
Init residual length time			27.2 µs	36.4 µs	200 ns	300 ns	100 ns
Calculate A*p time			4.21 ms	5.14 ms	3.91 ms	5.13 ms	642 µs
Calculate alpha time			7.67 ms	8.87 ms	73 µs	68.5 µs	63 µs
Update guess time			1.75 ms	849 µs	1.66 ms	835 µs	40 µs
Update residual length time			7.68 ms	9 ms	56 µs	66 µs	43 µs
Sync r*r time			621 µs	453 µs	528 µs	483 µs	2.9 µs
Read buffers time	400 ns	386.22 µs	83 µs	647 µs	175 µs	1.52 ms	-
Tolerance	10^{-2}						
Iterations	~251						

Таблиця 4.2. Таблиця вимірювань часу матриці 1473x1473.

	1473x1473 (19659)				
	conjugateGradient (GPU, scaled)		conjugateGradient (GPU+CPU, scaled)		conjugateGradient (CPU, scaled)
	Intel	AMD	Intel	AMD	
Total compute time	4.2 s	35.2 s	1.54 s	33.5 s	69.82 ms
Total kernel time	1.553 s	27.63 s	139.2 ms	26.09 s	-
Init time	6.6 ms	8.33 ms	6.9 ms	8.35 ms	18.6 µs
Init residual length time	253 µs	297 µs	1.7 µs	2 µs	1.5 µs
Calculate A*p time	73.2 ms	26.99 s	69.5 ms	26 s	54.12 ms

Calculate alpha time	706.7 ms	800.3 ms	5.57 ms	5.8 ms	4.4 ms
Update guess time	20.8 ms	11.5 ms	19.9 ms	11.4 ms	4.12 ms
Update residual length time	721.7 ms	799.7 ms	6.06 ms	5.2 ms	4.42 ms
Sync r*r time	6.95 ms	4.97 ms	7.33 ms	5.5 ms	48.9 μs
Read buffers time	892 μs	1.97 ms	8.25 ms	36.16 ms	-
Tolerance	10^{-2}				
Iterations	~6390				

Таблиця 4.3. Таблиця вимірювань часу матриці 10974x10974.

	10974x10974 (428650)				
	conjugateGradient (GPU, scaled)		conjugateGradient (GPU+CPU, scaled)		conjugateGradient (CPU, scaled)
	Intel	AMD	Intel	AMD	
Total compute time	132.3 s	∞	29.68 s	∞	13.32 s
Total kernel time	95.32 s	∞	11.71 s	∞	-
Init time	391.6 ms	∞	396.2 ms	∞	184 μs
Init residual length time	1.83 ms	∞	11.5 ms	∞	13 μs
Calculate A*p time	9.88 s	∞	9.45 s	∞	12.28 s
Calculate alpha time	42.04 s	∞	366.8 ms	∞	290 ms
Update guess time	367 ms	∞	321 ms	∞	328 ms
Update residual length time	42.34 s	∞	429 ms	∞	275 ms
Sync r*r time	57.8 ms	∞	51 ms	∞	436 μs
Read buffers time	11 ms	∞	484 ms	∞	-
Tolerance	10^{-2}				
Iterations	~23060				

У даному дослідженні я використовую GPU девайси:

- Intel(R) UHD Graphics 620 (скорочено Intel)
- AMD Radeon R7 M460 (скорочено AMD)

  Та CPU:

- Intel(R) Core(TM) i7-8550U CPU

В таблиці 4.1 порівнюється швидкість опрацювання матриць виду *n* x *n (k)*, де *n* – розмірність матриці, а *k* – кількість ненульових значень, використовуючи різні методи і девайси.

Можна побачити, що найшвидше працює однопоточний алгоритм (!) на звичайному CPU в Release режимі, а найдовше – паралельний алгоритм на GPU AMD. Це дуже дивний результат, я не можу точно пояснити чому так стається.

Одна моя здогадка була в тому, що можливо багато часу займає передача даних з оперативної пам'яті у відеопам'ять, тому може якщо вимірювати час тільки на виконання ядра, результат може бути швидшим.

Я вирішив перевірити це, і зробити заміри часу окремо кожного ядра. Як виявилося час на зчитування буфера відносно всіх обчислень виявився нехтовно малим. Зато вияснилися інші складні місця.

Можна побачити, що найбільш складним місцем в алгоритмі є обчислення $A \cdot p$, у всіх він триває найдовше (якщо не враховувати алгоритми, де навіть однопоточні процеси відбуваються на GPU, в тих ситуаціях найдовшими виходять однопоточні етапи в алгоритмі). І в матриці з розмірністю 10 974 видно, що GPU реалізація в цій частині є швидшою ніж в CPU, і загалом, якщо додати весь час на обрахунки ядер, то результат є навіть швидшим за реалізацію на центральному процесорі. Все тільки псує величезна затримка в хост коді, яка теж не дуже зрозуміло звідки береться.

Мене ці результати дуже здивували, бо здавалося би, у мене є GPU, який може паралельно обраховувати алгоритм сотнями потоками, але той же однопоточний алгоритм на центральному процесорі працює в кілька разів швидше… Можливо, дійсно, те що між методами є багато синхронізації на стільки сповільнює алгоритм, що він стає повільнішим порівняно з обчислюванням на CPU, а можливо сама технологія OpenCL, ще не дуже розвинута і не дуже оптимізована. Бо якби я запустив якусь гру, як приклад, під інтегрованою відеокартою, то вона б працювала набагато повільніше ніж під дискретною.

Отримавши такі цікаві результати, я вирішив пошукати, які є бібліотеки, що дозволяють розв’язувати СЛАР з розідженими матрицями, що застосовують GPU при цьому. Як виявилося такі бібліотеки є, одна з яких – це

ViennaCL. Трішки розібравшись з цим інструментом, я отримав наступні результати:

Таблиця 4.4. Таблиця вимірювань часу матриці 153x153 засобами бібліотеки.

	153x153 (2423)			
	conjugateGradient (ViennaCL library, no preconditioners)		conjugateGradient (ViennaCL library, Jacobi preconditioner)	
	Intel	AMD	Intel	AMD
Total compute time	870 ms	554 ms	1.84 s	510 ms
Tolerance	10^{-5}		10^{-9}	
Iterations	~246		~139	

Таблиця 4.5. Таблиця вимірювань часу матриці 1473x1473 засобами бібліотеки.

	1473x1473 (19659)			
	conjugateGradient (ViennaCL library, no preconditioners)		conjugateGradient (ViennaCL library, Jacobi preconditioner)	
	Intel	AMD	Intel	AMD
Total compute time	2.67 s	2.08 s	1.71 s	541 ms
Tolerance	10^{-5}		10^{-9}	
Iterations	~6505		~150	

Таблиця 4.6. Таблиця вимірювань часу матриці 10974x10974 засобами бібліотеки.

	10974x10974 (428650)			
	conjugateGradient (ViennaCL library, no preconditioners)		conjugateGradient (ViennaCL library, Jacobi preconditioner)	
	Intel	AMD	Intel	AMD
Total compute time	19.46 s	15.24	4.54	2.32
Tolerance	10^{-5}		10^{-9}	
Iterations	~21340		~2770	

Дані результати також виявилися дуже цікавими.

Одразу ж можна помітити, що в кожному випадку, застосування GPU AMD є більш швидшим, ніж це було при ручному написанню алгоритма. Виходить, що мої здогадки, що AMD повільно працює, бо застарілі драйвери чи сама технологія – хибні. Дані результати прекрасно це показують.

Можна помітити, що на маленьких матрицях, мої реалізації є швидшими ніж бібліотечні, але ситуація змінюється при збільшені розмірності матриці. І в результаті в матриці розмірністю 10974x10974 з застосування AMD GPU швидкість обчислювання є майже таким (а іноді й швидшим) за ручну реалізацію на CPU.

Також виявляється, що дана бібліотека надає вибірку з кількох передумовлювачів, які можуть пришвидшити збіжність вирішення СЛАР. Я вирішив застосувати передумовлювач Якобі, і в результаті кількість ітерацій зменшилася в рази, і дуже пришвидшилося виконання методу!

Але навіть якщо не використовувати передумовлювачі і застосовувати чистий метод спряжених градієнтів, то видно, що в обчисленнях на GPU є потенціал, і можливо з більш потужною відеокартою, я б зміг досягти більших швидкостей.

Весь код до звіту можна знайти за посиланням https://github.com/HauntTheHouse/DiplomaOpenCL.

4.4. Висновки.

У четвертому розділі ми дослідили, де можна знайти дані для розріджених матриць. Вияснили, які інструменти бувають завдяки яким можна розв’язати СЛАР використовуючи GPU – це GPGPU, з застосуванням CUDA чи OpenCL. Також були реалізовані 2 методи для обчислення СЛАР – метод градієнтного спуску і метод спряженого градієнта на GPU і CPU, а також використання бібліотеки ViennaCL для застосування цієї версії методу. І в результаті ми зробили невелике дослідження, які методи швидше працюють і на яких девайсах.

РОЗДІЛ 5. ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОДУКТУ.

У цьому розділі проводиться оцінка основних характеристик програмного продукту, розробленого для вирішення систему лінійних рівнянь з розрідженими матрицями на графічному процесорі.

Функціонально-вартісний аналіз – це технологія, що дозволяє оцінити реальну вартість послухи чи продукту незалежно від організаційної структури компанії. Даний аналіз проводиться з метою виявлення резервів зниження

витрат за рахунок ефективніших варіантів виробництва, кращого співвідношення між вартістю споживання виробу та витратами на його виготовлення. Для проведення цього аналізу експлуатується технічна, економічна та конструкторська інформація.

Алгоритм функціонально-вартісного аналізу включає в себе визначення послідовності етапів розробки продукту, визначення річних витрат та кількість робочих часів, а також визначення джерел витрат та кінцевий розрахунок вартості продукту.

5.1. Постановка задачі проектування.

У роботі застосовується метод ФВА для проведення техніко-екномічного аналізу розробки вирішувача СЛАР з розрідженими матрицями використовуючи при цьому GPU. Кожне рішення стосовно проектування та реалізації компонентів, що розробляються, впливають на всю систему, де кожна підсистема має її задовільняти. Тому виходить, що фактичний аналіз – це аналіз функцій програмного продукту, що призначений для обробки даних, отримані від файлів, що містять у собі розріджені матриці.

Технічні вимоги до системи:

- можливість запуску програми на персональних комп’ютерах зі мінімум залежностей;
- висока швидкість обробки даних та відповідь користувачеві у реальному часі;
- зручність та простота взаємодії з програмою;
- загальна доступність програми широкому колу користувачів;
- мінімальні витрати на впровадження програмного продукту.

5.2. Обгрунтування функцій програмного продукту.

Функція F_0 – є головною функцією для опису вирішувача СЛАР з розрідженими матрицями, використовуючи GPU. Для ФВА було складено наступні три функції ПП:

- F_1 – вибір методу вирішення СЛАР;
- F_2 – вибір технології для обчислення GPGPU;
- F_2 – вибір середовища розробки.

Кожна з цих функцій має декілька варіантів реалізації:

1) Функція F_1
   a) Steepest descent method;
   b) Congurate gradient method;
2) Функція F_2
   a) OpenCL API;
   b) CUDA API;
3) Функція F_3
   a) Середовище розробки Visual Studio 2022;
   b) Середовище розробки CLion;
   c) Середовище розробки VS Code.

Морфологічна карта відображає множину всіх можливих варіантів основних функцій. На основі цієї карти побудовано позитивно-негативну матрицю варіантів основних функцій (табл. 5.1).

Варіанти реалізації основних функцій наведено у морфологічній карті системи (рисунок 5.1):

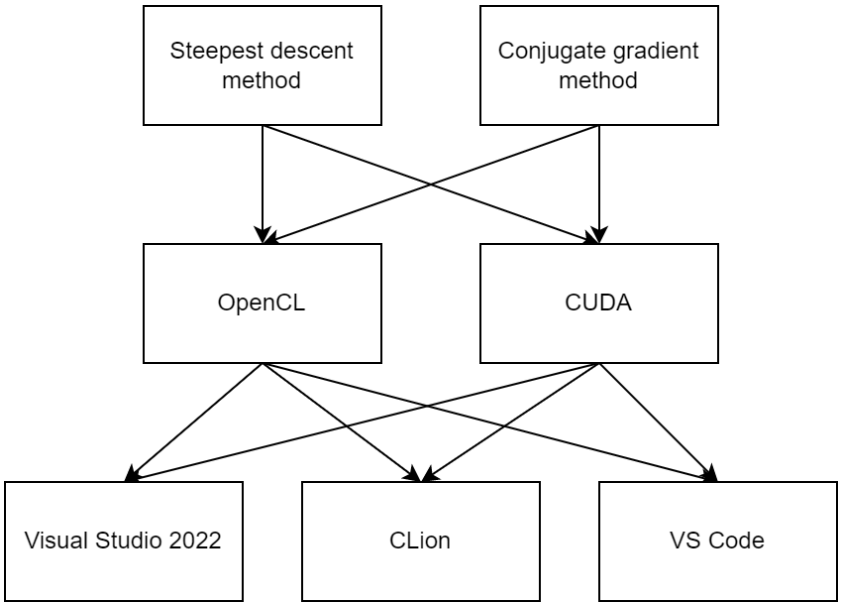

Рис. 5.1. Морфологічна карта

На основі морфологічної карти будуємо позитивно-негативну матрицю варіантів основних функцій (таблиця 5.1.).

Таблиця 5.1. Позитивно-негативна матриця.

Основна функція	**Варіант реалізації**	**Переваги**	**Недоліки**
F_1	a)	Метод простіший в реалізації. Мало ділянок алгоритму, які можна синхронізувати, завдяки чому задачу можна краще розпаралелити.	Дуже жорсткі умови для сходження ітераційного методу. Майже на всіх тестових матрицях метод розходиться. Працює тільки з симетричними матрицями з дійсними числами.
	b)	Метод сходиться до правильного розв’язку в більшості випадків.	Метод більш складніший і має багато блоків коду, які потрібно синхронізувати, що може сповільнити виконання. Працює тільки з симетричними матрицями з дійсними числами
F_2	a)	Працює працює майже на всіх платформах (Intel, AMD,	Така гнучість у варіаціях доступних платформ може зменшити

		NVIDIA, включаючи мобільні платформи та вбудовані системи). Також це open-source фреймворк.	продуктивність. Має меншу підтримку через те, що це open-source фреймворк. Менше ком'юніті та готових бібліотек.
	b)	Через розробку фреймворка тільки для однієї платформи, та через те, що це комерційний продукт, може мати краще підтримку і кращі результати у продуктивності. Також має багато якісних готових рішень для відомих задач як розв'язування СЛАР з розрідженими матрицями.	Працює тільки на відеокартах NVIDIA. Є комерційним продуктом
F_3	a)	Зручне середовище для користування завдяки підтримці Microsoft. Має власний компілятор, завдяки чому можна писати одразу програми без попереднього конфігурування. Дуже зручно дивитися debug інформацію, включаючи профайлинг. Нативна підтримка CUDA (але на OpenCL). Має підтримку CMake.	Займає досить багато місця. Працює тільки на Windows та має обмежену кількість мов для програмування (C++ та C#). Біблеотеки потрібно окремо шукати запущені під їхнім компілятор. Немає плагіна для підсвітки синтаксису OpenCL.
	b)	Схожий на інші продукти (інтерфейс і keywords) JetBrains. Зручний, кросплатформений та дозволяє вибирати компілятор самостійно. Нативна	Доступна тільки платна версія. Займає досить багато місця. Немає плагіна для підсвітки синтаксису OpenCL.

		підтримка CMake.	
	с)	Зручний і гнучкий, інтерфейс можна налаштувати як завгодно користувачеві. Кросплатформений. Має купу плагінів (включаючи підсвітку синтаксису OpenCL). Порівняно легковісний. Безкоштовний весь інструмент. Має пітримку CMake.	Через гнучкість приходиться всю конфігурацію брати у свої руки. Не такий зручний як попередні інструменти.

Базуючись на позитивно-негативній матриці, робимо висновок, що при розробці програмного продукту деякі варіанти функцій варто відкинути, і залишимо ті функії, які найбільше відповідають поставленим перед програмним продуктом задачам.

Функція F_1:

Звичайно ж перевагу надаємо методу спряжених градієнтів, так як те, що метод буде сходитися і розв’язувати СЛАР важливіше, ніж теоретично більш швидший алгоритм, але в більшості випадків непрацюючий. Тому залишаємо варіант б).

Функція F_2:

Тут звичайно можна було б оцінити в якій технології більше переваг, але тут очевидний варіант для мене – це фреймворк OpenCL, так як у мене немає відеокарти NVIDIA. Зато є інтегрована відеокарта від Intel і дискретна відеокарта від AMD, що дозволяє робити аналіз одразу на двох платформах. Залишаємо варіант а).

Функція F_3:

В різні моменти часу я використовував для одного й того ж програмного продукту одразу 3 середовища розробки. Спочатку CLion як і на Windows з бібліотекою OpenCL під MSYS2, так і на Linux (є підписка від КПІ), для написання OpenCL kernel'ів – VS Code, так як є підсвітка синтаксису. Ну і вкінці, коли розібрався як знаходити бібліотеку OpenCL через CMake, - Visual Studio 2022, так як для мене вона найзвичніша і найзручна для написання коду. Тому залишаємо всі 3 варіанти.

Отже, за нашими результатами, будемо розглядати наступні варіанти реалізації програмного продукту:

- $F_1(b) - F_2(a) - F_3(a)$
- $F_1(b) - F_2(a) - F_3(b)$
- $F_1(b) - F_2(a) - F_3(c)$

5.3. Обргрунтування системи параметрів ПП.

На основі даних, розглянутих вище, ми визначимо основні параметри вибору, які будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- X_1 – швидкість роботи програми.;
- X_2 – кількість доступних GPU для аналізу;
- X_3 – кількість методів вирішення СЛАР;
- X_4 – кількість тестових розріджених матриць.

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію ПП як показано у таблиці 5.2

Таблиця 5.2. Основні параметри ПП.

Параметр	Умовне позначення	Одиниця виміру	Значення параметра		
			Гірші	Середні	Кращі
Швидкість роботи програми	X_1	Кількість ненульових елементів на 1 мс	10	200	2500
Кількість доступних GPU для аналізу	X_2	Штук	1	3	6
Кількість методів вирішення СЛАР	X_3	Штук	1	2	4
Кількість тестових розріджених матриць	X_4	Штук	1	8	20

За даними таблиці 5.2. побудуємо графічні характеристики параметрів ПП: рис. 5.2. – рис. 5.5.

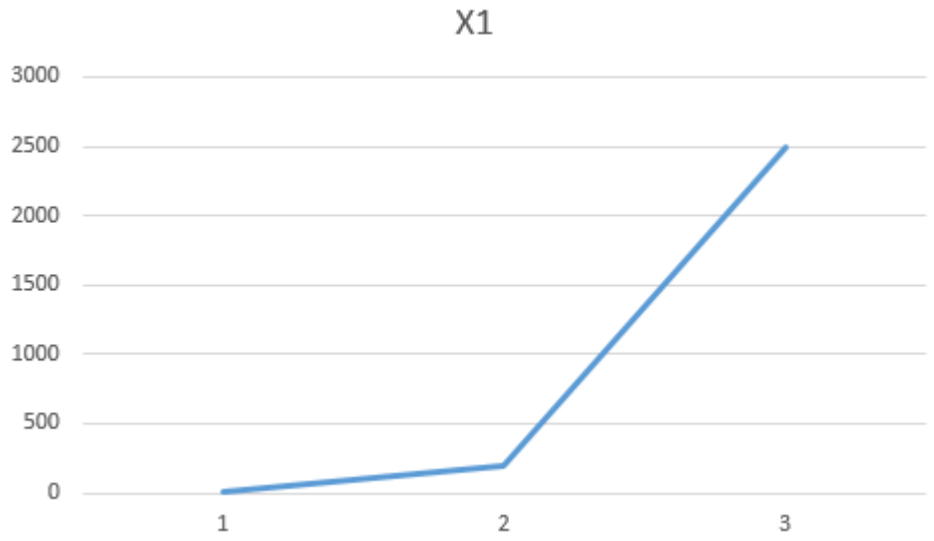

Рис. 5.2. X_1, швидкість роботи програми.

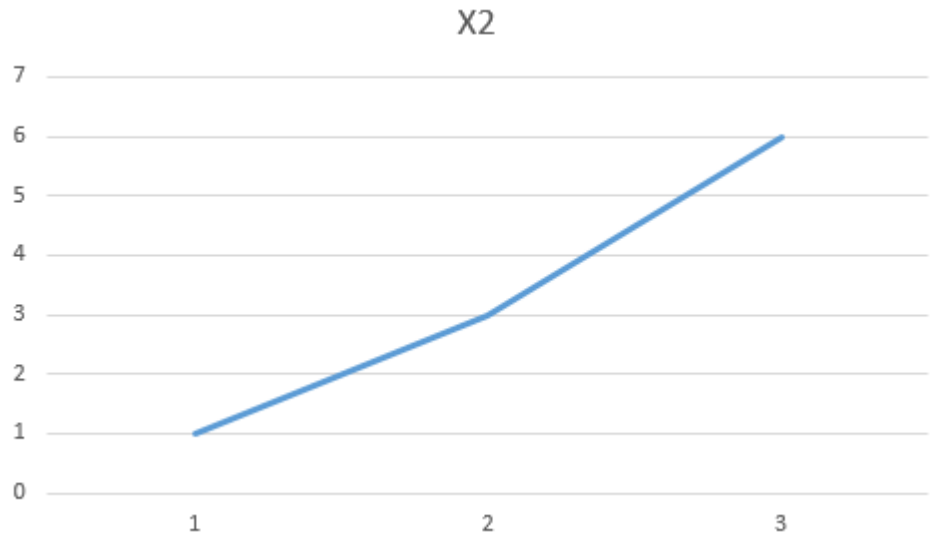

Рис. 5.3. X_2, кількість доступних GPU для аналізу.

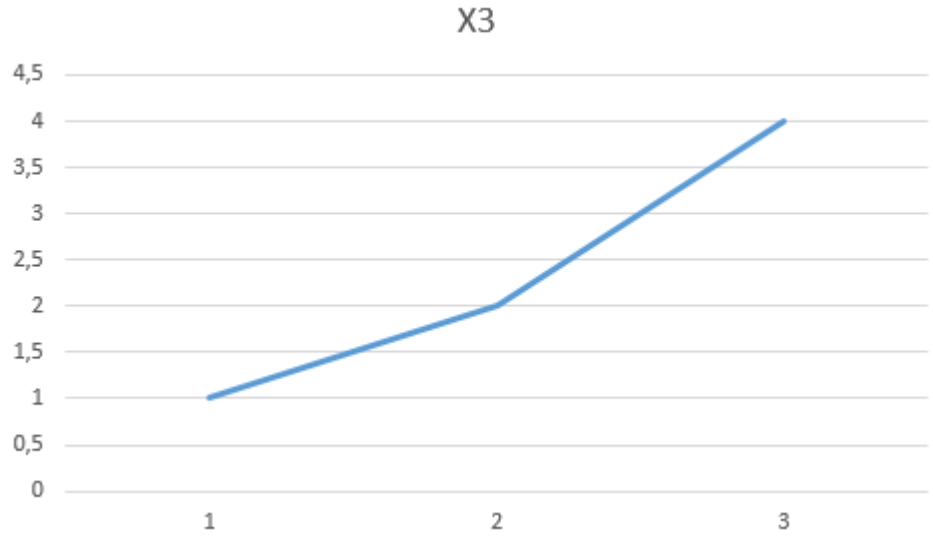

Рис. 5.4. X_3, кількість методів вирішення СЛАР.

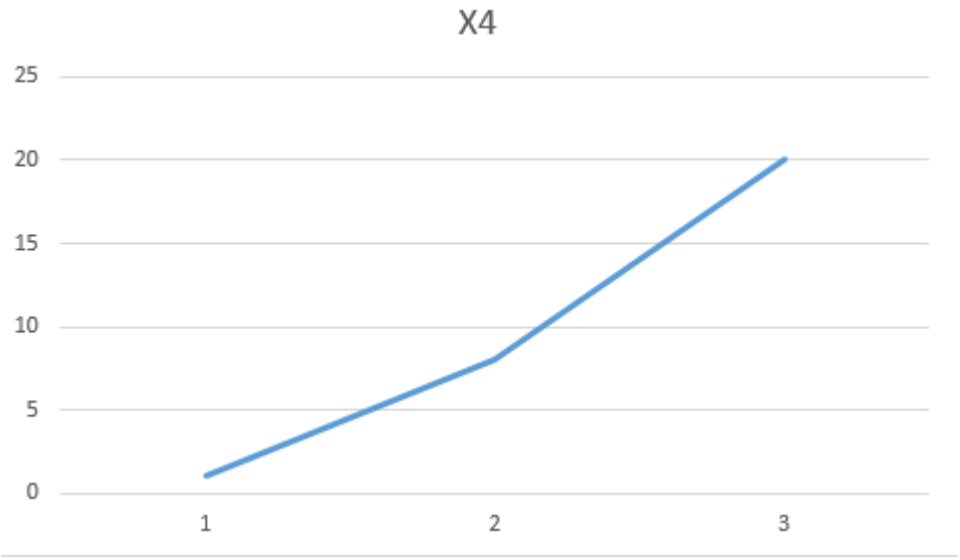

Рис. 5.5. X_4, кількість тестових розріджених матриць

5.4. Аналіз експертного оцінювання параметрів.

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні п араметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 5 людей. Визначення коефіцієнтів значимості передбачає:

Таблиця 5.3. Результат експерного ранжування.

Параметр	**Ранг параметра за оцінкою експерта**					**Сума рангів** R_i	**Відхилення** Δ_i	Δ_i^2
	1	**2**	**3**	**4**	**5**			
X_1	2	2	2	1	2	9	-5.5	30.25
X_2	2	3	3	2	2	12	-2.5	6.25

X_3	3	3	3	3	3	15	0.5	0.25
X_4	4	5	4	4	5	22	7.5	56.25
Разом	11	13	12	10	12	58	0	93

Для перевірки ступеню достовірності експертних оцінок, визначимо наступні параметри:

1. Сума рангів кожного з параметрів і загальна сума рангів

$$R_i = \sum_{j=1}^{N} r_{ij} = 9 + 12 + 15 + 22 = 58, \quad (5.1)$$

де N – кількість експертів, n – кількість оцінюваних параметрів.

2. Середня сума рангів T:

$$T = \frac{R_i}{n} = \frac{58}{4} = 14.5 \quad (5.2)$$

3. Відхилення суми рангів кожного параметра від середньої суми рангів за наступною формулою:

$$\Delta_i = R_i - T \quad (5.3)$$

4. Загальна сума квадратів відхилення за наступною формулою:

$$S = \sum_{i=1}^{n} \Delta_i^2 = 30.25 + 6.25 + 0.25 + 56.25 = 93 \quad (5.4)$$

5. Коефіцієнт узгодженості (конкордації)

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 93}{5^2(4^3 - 4)} = \frac{1116}{1500} = 0.744 > W_k = 0{,}67 \quad (5.5)$$

Знайдений коефіцієнт узгодженості перевищує нормативному (0.67), тому ранжування можна вважати достовірним.

Скористаємось результатами ранжування для проведення попарного порівняння всіх параметрів:

Таблиця 5.4. Попарне порівняння параметрів.

Параметри	**Експерти**					**Кінцева оцінка**	**Числове значення**
	1	**2**	**3**	**4**	**5**		
X_1 *і* X_2	=	<	<	<	=	<	0.5
X_1 *і* X_3	<	<	<	<	<	<	0.5
X_1 *і* X_4	<	<	<	<	<	<	0.5
X_2 *і* X_3	<	=	=	<	<	<	0.5
X_2 *і* X_4	<	<	<	<	<	<	0.5
X_3 *і* X_4	<	<	<	<	<	<	0.5

Числове значення, що визначає ступінь переваги i–го параметра над j–тим, a_{ij} визначається по формулі:

$$a_{ij} = \{1.5 \text{ при } X_i > X_j; 1.0 \text{ при } X_i = X_j; 0.5 \text{ при } X_i < X_j \qquad (5.6)$$

З отриманих числових оцінок переваги складемо матрицю $A = \left|\left|a_{ij}\right|\right|$. Для кожного параметра потрібно зробити розрахунок вагомості K_{Bi} за формулою:

$$K_{Bi} = \frac{b_i}{\sum_{i=1}^{n} b_i}, \qquad (5.7)$$

де $b_i = \sum_{j=1}^{N} a_{ij}$ – вагомість i-го параметра за результатами оцінок всіх експертів; a_{ij} – коефіцієнт переваги i-го параметра над j-тим.

Відносні оцінки будуть розраховуютися до того моменту, коли наступні значення не будуть мало відрізнятися від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за формулою 5.7.

$$K_{Bi} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}, \quad (5.8)$$

де $b_i' = \sum_{j=1}^{N} a_{ij} b_j$.

Розразуємо значення коефіцієнтів вагомості параметрів.

Таблиця 5.5. Розрахунок вагомості параметрів.

Параметри X_i	Параметри X_j				Перша ітерація		Друга ітерація		Третя ітерація	
	X_1	X_2	X_3	X_4	b_i	K_{Bi}	b_i^1	K_{Bi}^1	b_i^2	K_{Bi}^2
X_1	1	0.5	0.5	0.5	2.5	0.156	6.25	0.09	15.625	0.05
X_2	1.5	1	0.5	0.5	3.5	0.219	12.25	0.178	42.875	0.136
X_3	1.5	1.5	1	0.5	4.5	0.281	20.25	0.294	91.125	0.288
X_4	1.5	1.5	1.5	1	5.5	0.344	30.25	0.438	166.375	0.527
Разом					16	1	69	1	316	1

У таблиці 5.5 наведено розрахунок вагомості параметрів. Як видно, різниця значень коефіцієнтів вагомості після третьої ітерації не перевищує 2%, тому додаткові ітерації не потрібні.

5.5. Аналіз рівня якості варіантів реалізації функцій.

Потрібно визначити рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів X_1 обрано не найгіршим, так як швидкість роботи ядер це важливо, тому обрані варіанти а). 2500, б). 200.

Абсолютне значення параметрів X_2 прийнято і вважається достатнім значенню б). 3. Так як 1 девайс – мало для аналізу, а 5 – добре, тільки це надлишково, можна просто мати по одній платформі з найпопулярніших відеоадаптерів: Intel, AMD, NVIDIA.

Абсолютне значення параметрів X_3 може набувати ненайкращого варіанту, так як пари методів достатньо для аналізу. Тому підходять варіанти б). 2, в). 1.

Абсолютне значення параметрів X_4 має бути не найгіршим, бо однієї матриці недостатньо для аналізу. Тому обрано варіанти а). 20, б). 8.

Коефіцієнт технічого рівня якості для кожного варіанту реалізації ПП розраховується за наступною формулою:

$$K_{TP} = \sum_{i=1}^{n} K_{Bi} B_i, \tag{5.9}$$

де, n – кількість параметрів,

K_{Bi} – коефіцієнт вагомості i-го параметра,

B_i – оцінка i-го параметра в балах.

Таблиця 5.6. Розрахунок показників якості

Основна функція	**Варіант реалізації**	**Абсолютне значення параметру**	**Бальна оцінка параметру**	**Коефіцієнт вагомості параметру**	**Коефіцієнт якості**
$F_1(X_1)$	A	2500	9	0.35	3.15
	B	200	5	0.35	1.75
$F_2(X_2, X_3)$	A	5	7	0.46	3.22
	B	4	6	0.46	2.76
$F_4(X_4)$	A	20	7	0.19	1.33
	B	8	6	0.19	1.14

Застосовуючи результати з таблиці 5.6., спробуймо визначити рівень якості кожного з варіантів:

$$\boldsymbol{F_1(A) - F_2(A) - F_3(A) = 3.15 + 3.22 + 1.33 = 7.7};$$

$\boldsymbol{F_1(A) - F_2(A) - F_3(B) = 3.15 + 3.22 + 1.14; = 7.51;}$

$\boldsymbol{F_1(A) - F_2(B) - F_3(A) = 3.15 + 2.76 + 1.33 = 7.24;}$

$F_1(A) - F_2(B) - F_3(B) = 3.15 + 2.76 + 1.14 = 7.05;$

$F_1(B) - F_2(A) - F_3(A) = 1.75 + 3.22 + 1.33 = 6.3;$

$F_1(B) - F_2(A) - F_3(B) = 1.75 + 3.22 + 1.14; = 6.11;$

$F_1(B) - F_2(B) - F_3(A) = 1.75 + 2.76 + 1.33 = 5.84;$

$F_1(B) - F_2(B) - F_3(B) = 1.75 + 2.76 + 1.14 = 5.65;$

Отже виходить, що найкращим варіантом є перші варіанти конфігурації, для якого значення коефіцієнту технічного рівня набуває найбільше значення.

5.6. Економічний аналіз варіантів розробки ПП.

Для визначення вартості розробки програмного продукту, потрібно спочатку провести розрахунок трудомісткості.

Усі варіанти включають в себе два конкретних завдання:

1. Розробка проекту програмного продукту;
2. Розробка програмної оболонки;

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію даних. Потрібно провести розрахунок норм часу на розробку та програмування для кожного з завдань. Загальна трудомісткість обчислюється за наступною формулою:

$$\mathrm{Т_О} = \mathrm{Т_Р} \cdot \mathrm{К_П} \cdot \mathrm{К_{СК}} \cdot \mathrm{К_М} \cdot \mathrm{К_{СТ}} \cdot \mathrm{К_{СТ.М}}, \qquad (5.10)$$

де $\mathrm{Т_Р}$ – трудомісткість розробки ПП;

$К_П$ – поправочний коефіцієнт;

$К_{СК}$ – коефіцієнт на складність вхідної інформації;

$К_М$ – коефіцієнт рівня мови програмування;

$К_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$К_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення.

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $Т_Р = 45$ людино-днів. Поправочний коефіцієнт, який враховує вид вхідної інформації для першого завдання: $К_П = 1.7$. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації рівний $К_{СК} = 1$. Оскільки при розробці першого завдання можна опиратись на стандартні архітектури, врахуємо це за допомогою коефіцієнта $К_{СТ} = 0.92$. Тоді загальна трудомісткість розробки першого завдання дорівнює:

$$Т_1 = 45 \cdot 1.7 \cdot 0.92 = 70.38 \text{ людино-днів}$$

Проведемо аналогічні розрахунки для другого завдання, в якому використовується алгоритм третьої групи складності зі ступенем новизни Б, тобто $Т_Р = 90$ людино-днів, $К_П = 0{,}9$, $К_{СК} = 1$, $К_{СТ} = 0.92$:

$$Т_2 = 90 \cdot 0.9 \cdot 0.92 = 74.52 \text{ людино-днів}$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$$Т_I = (70.38 + 74.52 + 4.8 + 74.52) \cdot 8 = 1\,793.76 \text{ людино-годин;}$$

$$Т_{II} = (70.38 + 74.52 + 6.91 + 74.52) \cdot 8 = 1\,810.64 \text{ людино-годин.}$$

Найбільшу трудомісткість має результат під варіантом II.

В розробці беруть участь два інженери графіки з окладом 29000 грн. та один архітектор обчислень на GPU з окладом 80000. Визначимо середню зарплату за годину за наступною формулою:

$$С_{ч} = \frac{M}{T_m \cdot t} \text{ грн.}, \quad (5.11)$$

де, M – місячний оклад працівників;

T_m – кількість робочих днів на місяць;

t – кількість робочих годин в день.

$$С_{ч} = \frac{29\,000 \cdot 2 + 80\,000}{3 \cdot 21 \cdot 8} = 273.81 \text{ грн.}$$

Тоді, розрахуємо заробітну плату за наступною формулою:

$$С_{ЗП} = С_{ч} \cdot Т_i \cdot К_{Д}, \quad (5.12)$$

де, $С_{ч}$ – величина погодинної оплати праці програміста;

$Т_i$ – трудомісткість відповідного завдання;

$К_{Д}$ – норматив, який враховує додаткову заробітну плату.

Заробітна плата за варіантамитоді становить:

1. $С_{ЗП1} = 273.81 \cdot 1\,793.76 \cdot 1.2 = 589\,379.31$ грн;

2. $С_{ЗП2} = 273.81 \cdot 1\,810.64 \cdot 1.2 = 594\,925.61$ грн.

Відрахування за єдиний соціальний внесок становить 22%:

$$С_{СВ} = С_{ЗП} \cdot 0{,}22 \quad (5.13)$$

Тоді за формулою отримаємо:

1. $С_{ВІД1} = С_{ЗП} \cdot 0.22 = 589\,379.31 \cdot 0.22 = 129\,663.45$;

2. $С_{ВІД2} = С_{ЗП} \cdot 0.22 = 594\,925.61 \cdot 0.22 = 130\,883.63$.

Тепер визначимо витрати на оплату однієї машино-години. (См)

Так як одна ЕОМ обслуговує одного програміста з окладом 29 000 грн., з коефіцієнтом зайнятості $К_З = 0.25$, то для однієї машини отримаємо:

$$С_{Г} = 12 \cdot М \cdot К_З \quad (5.14)$$

$$С_{Г} = 12 \cdot 29\,000 \cdot 0.25 = 87\,000 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г(1 + К_З) \quad (5.15)$$

$$С_{ЗП} = 87\,000 \cdot (1 + 0.25) = 108\,750 \text{ грн.}$$

Відрахування на соціальний внесок становить 22%:

$$С_{ВІД} = С_{ЗП} \cdot 0.22 = 108\,750 \cdot 0.22 = 23\,925 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 38 000 грн.

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР}, \quad (5.16)$$

де $К_{ТМ}$ – коефіцієнт, що враховує витрати на транспортування та монтаж приладу в користувача;

$К_А$- річна норма амортизації;

$Ц_{ПР}$ – договірна ціна прикладу.

Нехай коефіцієнт витратів на траспортування та монтаж приладу буде 1.15, тоді:

$$С_А = 1.15 \cdot 0.25 \cdot 38\,000 = 10\,925 \text{ грн.}$$

Витрати на ремонт тп профілактику розраховуємо за наступною формулою:

$$С_Р = К_{ТМ} \cdot К_Р \cdot Ц_{ПР}, \quad (5.17)$$

де $К_Р$ – відсоток витрат на поточні ремонти.

Нехай відсоток витрат на поточні ремонти буде 5%. Тоді маємо:

$$С_Р = 1.15 \cdot 0.05 \cdot 38\,000 = 2\,185 \text{ грн.}$$

Ефективний годинний онд часу комп’ютера за рік розраховуємо за наступною формулою:

$$Т_{ЕФ} = (Д_К - Д_В - Д_С - Д_Р) \cdot t \cdot К_В, \quad (5.18)$$

де $Д_К$ – календарна кількість днів у році;

$Д_В, Д_С$ – кількість вихідних та святкових днів відповідно;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t – кількість робочих годин в день;

$К_В$ – коефіцієнт використання приладу у часі протягом зміни.

Тоді отримаємо наступний вираз:

$$Т_{ЕФ} = (365 - 104 - 12 - 16) \cdot 8 \cdot 0.9 = 1\,667.6 \text{ год.}$$

Витрати на полату електроенергії розраховується за наступною формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_C \cdot К_З \cdot Ц_{ЕЛ}, \quad (5.19)$$

де N_C – середньо-споживча потужність приладу;

$К_З$ – коефіцієнт зайнятості приладу;

$Ц_{ЕЛ}$ – тариф за 1 КВт-годин електроенергії.

Нехай середньо-споживча потужність приладу матиме значення 0.85, коефіцієнт зайнятості приладу – 0.6, тариф за 1 КВт-годин електроенергії – 3.8 грн.. Тоді отримаємо наступне:

$$С_{ЕЛ} = 1\,667.6 \cdot 0.85 \cdot 0.6 \cdot 3.8 = 3\,231.81 \text{ грн.}$$

Накладні витрати розраховуються за наступною формулою:

$$С_Н = Ц_{ПР} \cdot 0.67 \quad (5.20)$$

$$С_Н = 38\,000 \cdot 0.67 = 25\,460 \text{ грн.}$$

Річні експлуатаційні витрати вираховуються за наступною формулою:

$$С_{ЕК} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н \quad (5.21)$$

$$С_{ЕК} = 108\,750 + 23\,925 + 10\,925 + 2\,185 + 3\,231.81 + 25\,460 = 174\,476.81 \text{ грн}$$

Собівартість однієї машино-години ЕОМ вираховується за наступною формулою:

$$С_{МГ} = \frac{С_{ЕК}}{Т_{ЕФ}} \quad (5.22)$$

$$С_{МГ} = \frac{174\,476.81}{1\,677.6} = 104 \text{ грн./год.}$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу вираховується за наступною формулою:

$$C_М = C_{МГ} \cdot T \quad (5.23)$$

1. $C_{М1} = 104 \cdot 1\,793.76 = 186\,551.04$ грн.

2. $C_{М2} = 104 \cdot 1\,810.64 = 188\,306.56$ грн.

Накладні витрати складають 67% від заробітньої плати вираховуються за формулою 5.20, тоді отримаємо:

1. $C_{Н1} = 589\,379.31 \cdot 0.67 = 394\,884.14$ грн.

2. $C_{Н2} = 594\,925.61 \cdot 0.67 = 398\,600.16$ грн.

В результаті вартість розробки ПП за варіантами вираховуються за наступною формулою:

$$C_{ПП} = C_{ЗП} + C_{ВІД} + C_М + C_Н \quad (5.24)$$

1. $C_{ПП1} = 589\,379.31 + 23\,925 + 186\,551.04 + 394\,884.14 = 1\,194\,739.49$ грн.

2. $C_{ПП2} = 594\,925.61 + 23\,925 + 188\,306.56 + 398\,600.16 = 1\,205\,757.33$ грн.

5.7. Вибір кращого варіанту ПП техніко-економічного рівня.

Спочатку розрахуємо коефіцієнти техніко-економічного рівня за наступною формулою:

$$К_{ТЕР_j} = \frac{К_{К_j}}{C_{ПП_j}} \quad (5.25)$$

$$К_{ТЕР1} = \frac{7.51}{1\,194\,739.49} = 6.2859 \cdot 10^{-6}$$

$$К_{ТЕР1} = \frac{7.7}{1\,205\,757.33} = 6.386 \cdot 10^{-6}$$

Як можна побачити, найбільш ефективним є варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР} = 6.386 \cdot 10^{-6}$.

У даному варіантів ми маємо такі параметри:

- Алгоритм вирішення СЛАР - метод спряжених градієнтів.
- Вибір GPGPU фреймвор – OpenCL. Так як кросплатформеність і кількість платформ, на яких можна робити аналіз є важливішим ніж теоретичне збільшення швидкості (особливо, коли в мене немає відповідного апаратного запезпечення).
- Середовище розробки – Visual Studio 2022. Так як зручність у використанні і продуктивність при написанні коду є більш важлим ніж кросплатформеність (плюс я в даний час більшу частину працюю за Windows).

Я вважаю, що отриманий варіант виконання програмного комплексу є чудовим, і дає користувачу зручний інтерфейс, непоганий функціонал і швидкодію.

5.8. Висновки.

У даному розділі ми провели функціонально-вартісний аналіз програмного продкту. Було проведено оцінку основних функцій ПП. А також визначили параметри, що характеризують програмний продукт, і провели експертне оцінювання параметрів та аналіз якості варіантів реалізації функцій.

Наостано ми провели економічний аналіз варіантів розробки, а саме: витрати на трудомісткість, витрати на заробітну плату і т.д. Також на основі нашого аналізу, ми обрали найкращий варіант реалізації ПП.

ЗАГАЛЬНІ ВИСНОВКИ

У даній, як на мене цікавій, дипломній роботі ми дізналися, які існують загальні методи розв’язку СЛАР, а також спеціалізовані методи, що підходять тільки для розріджених матриць, які можна обчислити на GPU. Загалом вийшла дійсно дослідницька робота, в якій ми по суті відповідали на питання, чи доцільно застосовувати GPU для ров’язку СЛАР з розрідженими матрицями? Відповідь не так однозначна, спробуймо написати висновок з кожного розділу послідовно.

В першому розділі ми вияснили, що загалом існують дві варіації методів для вирішення СЛАР: прямі та ітераційні, а також розглянули детальніше пару методів з кожної варіації. Як виявилося ітераційні методи є більш застосовними, та вони більш підходять для великих матриць, і саме алгоритми такого типу ми пізніше реалізували в практичній частині. Також були розглянуті основні джерела, звідки можна почерпнути інформацію і готові рішення.

У другому ж розділі ми розглянули те, як працює графічний процесор, яка його архітектура, і яка його різниця з центральним процесором. Відповідь виявилася нескладною – у GPU набагато більше ядер і потоків порівняно з CPU, хоча вони не такі багатозатачні як в центральному процесорі. Завдяки такій властивості відеокарту застосовують в різних напрямках, де треба робити багато незалежних обчислень, бо саме в таких задачах GPU себе найкраще проявляє (наприкад відображення графіки, майнинг криптовалют, машинне навчання тощо). Також були коротко описані дослідження в яких застосовували GPU в якості вирішувача СЛАР і інших матричних операцій, де він себе чудово проявив, прискоривши обчислення в кілька разів порівняно з розв’язанням на центральному процесорі. Завдяки таким успішним застосуванням й було вирішено спробувати вирішити СЛАР з розрідженими марицями на GPU і повторити цей успіх.

У третьому розділі ми детальніше розглянули те, що таке розріджена матриця, і чому взагалі потрібно вирішувати СЛАР з ними. Як виявилося, в такі матриці часто трансформуються вирази диференціальних рівнянь, які використовуються для опису справжніх фізичних процесів в таких моделях як ядерний реактор, хімічна інженерія, будівництво тощо. Тому розв'язок СЛАР з розрідженими матрицями часто має на увазі практичне застосування. Також ми детально розглянули 2 методи з усіма кроками реалізації, які ми написали в практичній частині: метод градієнтного спуску і метод спряженого градієнта.

У четвертій частині було більше розглянуто специфікацію під час розробки алгоритму. Вияснили, які бувають формати для зберігання розріджених матриць, і як їх розшифровувати і парсити: Matrix-Market format і Harwell-Boeing format. Було вирішено використовувати перший формат, бо він є простішим.

Розглядали також, які є існуючі інструменти для обчислень на GPU (такі обчислення називаються GPGPU), і вияснилося, що їх основних 2: CUDA і OpenCL. Було вирішено використати OpenCL, так як банально в мене немає відеокарти NVIDIA (а саме тільки на таких платформах працює CUDA). Ну і як результат, використовуючи мову програмування C++17, було реалізовано 2 методи, що ми обговорювали раніше, окремо на GPU і CPU. І як висновок, застосовним був тільки метод спряжених градієнтів, бо метод градієнтного спуску в більшості випадках розбігався. Крім реалізації методу, також було вирішено використати open-source бібліотеку для матричних операцій на GPU – ViennaCL.

Якщо говорити за результати, то вони мене здивували, бо як виявилося, в моїй реалізації, однопоточний алгоритм на CPU був в кілька разів швидше, ніж той же метод на GPU. Точно чому так сталося я сказати не можу, бо інтуїтивно здається так, що сотні потоків GPU краще будуть справлятися з задачею, ніж один потік CPU.

Як припущення, може бути, що алгоритм в GPU суттєво сповільнюється, бо потрібно робити синхронізацію між потоками, бо операції не є повністю незалежними. Також, можливо, сповільнення дає те, що в кожній ітерації потрібно перекидувати буферні дані з оперативної пам’яті до відеопам’яті. Ну і звичайно ж самі обмеження апаратного забезпечення, бо відеокарта в мене не найсучасніша, а Intel(R) Core(TM) i7-8550U є на даний час досить потужним процесором.

Використовуючи ж реалізацію методу спряженого градієнту бібліотеки ViennaCL можна помітити, що даний алгоритм може бути швидким, і буває навіть швидшим за реалізацію на CPU (особливо якщо використовувати передумовлювачі). Можливо в цих методах автори змогли прибрати недоліки чистого алгоритму, як в мене, і якось більш вигідно використовувати можливості GPU. І зважаючи на цей приклад можна сказати, що у вирішеннях СЛАР з розрідженими матрицями на GPU є потенціал. І використовуючи більші потужності і більш ефективно застосовуючи багатопоточність графічного процесора, можна дійсно пришвидшити пришвидшити обчислювання в рази.

Ну і в п’ятій частині було проведено функціонально-вартісний аналіз програмного продкту. Ми зробили оцінку основних функцій ПП, а також визначили параметри, що характеризують програмний продукт. Провели експертне оцінювання параметрів та аналіз якості варіантів реалізації функцій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.

1. Wikipedia. System of linear equations [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/System_of_linear_equations.
2. Лекція 9. Ітераційні та прямі методи [Електронний ресурс] – Режим доступу: http://om.univ.kiev.ua/users_upload/15/upload/file/cm_lecture_09.pdf
3. Л. П. Фельдман, А. І. Петренко, О. А. Дмитрієвна, Чисельні методи в інформатиці, ISBN 966-552-155-1
4. Вікіпедія. Графічний процесор [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Графічний_процесор
5. RSDN. Использование технологии OpenCL для разработки высоконагруженных приложений [Електронний ресурс] – Режим доступу: https://rsdn.org/article/OpenCL/opencl.xml
6. E. Lezar, D.b. Davidson, "GPU-based LU Decomposition for Large Method of Moments Problems", Electron. Lett. Electronics Letters 46.17 (2010),pp. 1194-1196.
7. Xiaoming Chen, Ling Ren, Yu Wang, Huazhong Yang, "GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling", IEEE Trans. Parallel Distrib. Syst. IEEE Transactions on Parallel and Distributed Systems 26.3 (2015), pp. 786-795.
8. Pablo Ezzatti, Enrique Quintana-Orti, Alfredo Remon, "High Performance Matrix Inversion on a Multi-core Platform with Several GPUs", 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing (2011), pp. 87-93.
9. Tsybin M. The art of use Microsoft Word. Kyiv: Ranok, 2018. 1024 p.
10. Atasoy, Baha Sen, Burhan Selcuk, "Using Gauss - Jordan Elimination Method with CUDA for Linear Circuit Equation Systems", Procedia Technology 1

(2012),pp. 31-35.

11. Tao Wang, Longjiang Guo, Guilin Li, Jinbao Li, Renda Wang, Meirui Ren, Jing He, "Implementing the Jacobi Algorithm for Solving Eigenvalues of Symmetric Matrices with CUDA", 2012 IEEE Seventh International Conference on Networking, Architecture, and Storage (2012), pp. 69-78.
12. Wikipedia. Sparse matrix [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Sparse_matrix
13. Matthew Scarpino (2012), OpenCL in Action. How to accelerate graphics and computations, ISBN 9781617290176.
14. "Using Multiple Graphics Cards as a General Purpose Parallel Computer: Applications to Computer Vision", Proceedings of the 17th International Conference on Pattern Recognition (ICPR2004) 18 July 2011 at the Wayback Machine, Cambridge, United Kingdom, 23–26 August 2004, volume 1, pages 805–808.
15. Tom’s Hardware. Nvidia's CUDA: The End of the CPU? [Електронний ресурс] – Режим доступу: https://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954.html