Міністерство освіти і науки України

Тернопільський національний технічний університет імені Івана Пулюя

(повне найменування вищого навчального закладу)

Факультет комп'ютерно-інформаційних систем і програмної інженерії

(назва факультету)

Кафедра комп'ютерних наук

(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

Магістр

(назва освітнього ступеня)

на тему: Застосування паралельного програмування на основі технології CUDA в задачах багатовимірної оптимізації

Виконав: студент 6 курсу, групи СНнм-61

спеціальності

122 Комп’ютерні науки

(шифр і назва спеціальності)

		Воронка А.О.
	(підпис)	(прізвище та ініціали)
Керівник		проф. Литвиненко Я.В.
	(підпис)	(прізвище та ініціали)
Нормоконтроль		доц. Мацюк О.В.
	(підпис)	(прізвище та ініціали)
Завідувач кафедри		доц. Боднарчук І.О.
	(підпис)	(прізвище та ініціали)
Рецензент		доц. Тиш Є.В.
	(підпис)	(прізвище та ініціали)

м. Тернопіль – 2022

Міністерство освіти і науки України

Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
______________ доц. Боднарчук І.О.
(підпис) (прізвище та ініціали)
« » 20___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістр
(назва освітнього ступеня)

за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)

студенту Воронці Андріяну Олеговичу

1. Тема роботи Застосування паралельного програмування на основі технології CUDA в задачах багатовимірної оптимізації

Керівник роботи Литвиненко Ярослав Володимирович, д.т.н., професор
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом по університету від «__28__» _жовтня_ 2021 року №__4/7-909_

2. Термін подання студентом роботи 25.05.2022

3. Вихідні дані до роботи наукові літературні джерела

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1 Аналіз предметної області. 2 Теоретична частина.
3. Практична частина. 4. Охорона праці та безпека в надзвичайних ситуаціях

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Тема роботи. 2. Актуальність. 3. Мета, задачі дослідження. 4. Об'єкт, предмет дослідження наукова новизна, практичне значення роботи. 5. Техніка загальних обчислень на графічному процесорі. 6. Переваги технології CUDA. 7. Архітектура CUDA, ієрархія пам'яті.
8. Мультипроцесор. 9. Модель програмування технології CUDA. 10. Розв'язування задач багатовимірної оптимізації. 11. Алгоритми випадкового пошуку. 12. Системні вимоги.
13. Технічні характеристики ПК, на якому проводилося дослідження.
14. ПЗ, яке використовувалося в роботі. 15, 16. Структура програми.
17. Результат розробки додатку. 18 - 21. Результати проведених чисельних експериментів.
22.Основні результати дослідження

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Дмитроца Л.П., к.т.н., доцент	01.05.22	17.05.22
Безпека в НС	Клепчик В.М., ст. викладач	01.05.22	19.05.22

7. Дата видачі завдання ____________ 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Затвердження теми кваліфікаційної роботи	28.10.21	Виконано
2	Аналіз літературних джерел	29.10.21-18.12.21	Виконано
3	Обґрунтування актуальності дослідження	18.12-29.12.21	Виконано
4	Аналіз предмету дослідження та предметної області	02.01-26.01.22	Виконано
5	Проведення дослідження методів та засобів аналітичного опрацювання даних	27.01-28.02.22	Виконано
6	Оформлення розділу «Аналіз предметної області»	01.03-26.03.22	Виконано
7	Оформлення розділу «Теоретична частина»	27.03-15.04.22	Виконано
8	Оформлення розділу «Практична частина»	16.04-30.04.22	Виконано
9	Оформлення розділу «Охорона праці та безпека в надзвичайних ситуаціях»	01.05-18.05.22	Виконано
10	Нормоконтроль	11.05-15.05.22	Виконано
11	Перевірка на плагіат	09.05.22	Виконано
12	Попередній захист роботи	16.05.22	Виконано
13	Захист кваліфікаційної роботи	26.05.22	

Студент ____________ Воронка А.О.

(підпис) (прізвище та ініціали)

Керівник роботи ____________ Литвиненко Я.В.

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Застосування паралельного програмування на основі технології CUDA в задачах багатовимірної оптимізації // Кваліфікаційна робота освітнього рівня «Магістр» // Воронка Андріян Олегович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем та програмної інженерії, кафедра комп'ютерних наук, група СНнм–61 // Тернопіль, 2022 // С. – 81, рис. – 23 , табл.– 10, слайдів – 22, додат. – 1, бібліогр. – 43.

Ключові слова: GRAPHICS PROCESSING UNIT, COMPUTE UNIFIED DEVICE ARCHITECTURE, БАГАТОВИМІРНА БАГАТОЕКСТРЕМАЛЬНА ОПТИМІЗАЦІЯ, АЛГОРИТМ ВИПАДКОВОГО ПОШУКУ

Кваліфікаційна робота присвячена використанню паралельних обчислень для максимального пришвидшення процесу отримання екстремуму в задачах багатовимірної оптимізації.

Показано переваги використання технології CUDA в порівнянні з технологією послідовних обчислень. Описані моделі мультипроцесора CUDA, програмування технології CUDA в цілому та з використанням стандартного і графічного процесорів. Описані їх характеристики та особливості використання Проаналізовані спеціальні функції для розв'язування задач багатовимірної оптимізації. Досліджено алгоритми випадкового пошуку

В рамках виконання роботи створено програмний комплекс, котрий реалізує методи багатовимірної оптимізації, розроблені алгоритми та дозволяє знаходити мінімум функції шляхом паралельної обробки даних.

ANNOTATION

Applying of parallel development on the base of CUDA technology in multidimensional optimization problems // Master thesis // Voronka Andriian // Ternopil Ivan Pul'uj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer Science // Ternopil, 2022 // P. - 81, Fig. - 23, Table – 10, Slide - 22, References - 43.

Keywords: GRAPHICS PROCESSING UNIT, COMPUTE UNIFIED DEVICE ARCHITECTURE, MULTI-DIMENSIONAL MULTI-EXTREME OPTIMIZATION, RANDOM SEARCH ALGORITHM

Thesis deals with the use of parallel calculations to maximize the acceleration of the process of obtaining the extremum in multidimensional optimization problems.

The advantages of using CUDA technology in comparison with sequential computing technology are shown. Describes models of CUDA multiprocessor, programming CUDA technology in general and using standard and graphics processors. Their characteristics and features of use are described. Special functions for solving multidimensional optimization problems are analyzed. Random search algorithms are investigated

As part of the work, a software package was created that implements multidimensional optimization methods, developed algorithms and allows to find a minimum of functions through parallel data processing.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ СКОРОЧЕНЬ І ТЕРМІНІВ

CPU (Central processing unit) – центральний процесор.

CUDA (Compute Unified Device Architecture) – програмно-апаратна архітектура, розроблена компанією NVIDIA, що дозволяє проводити обчислення з використанням графічних процесорів NVIDIA, що підтримують технологію GPGPU (довільних обчислень на відеокартах).

GPGPU (General-purpose graphics processing units) – техніка використання графічного процесора відеокарти для загальних обчислень, які, зазвичай, проводить центральний процесор.

GPU (Graphics Processing Unit) – графічний процесор.

Rmax – досягнута максимальна продуктивність.

Rpeak – теоретична максимальна продуктивність.

SIMD (Single Instruction stream Multiple Data stream) – метод обчислень, при якому ядра GPU виконують один і той самий набір інструкцій для кожного екземпляра даних.

SIMT (Single-Instruction, Multiple-Thread) - модель виконання, що використовується в паралельних обчисленнях, де одна інструкція, кілька даних поєднуються з багатопотоковістю.

Багатоекстремальне завдання – нелінійне завдання математичного програмування, цільова функція якого може мати як глобальні, так і локальні оптимуми.

Мультимодальна функція – функція, що має більше одного екстремуму.

ОС – операційна система.

ПЗ – програмне забезпечення.

Функція Розенброка – невипукла функція для оцінки продуктивність алгоритмів оптимізації.

Функція Растригіна - невипукла функція, що використовується для тестування ефективності алгоритмів оптимізації, типовий приклад нелінійної мультимодальної функції.

Функція Хіммельблау – мультимодальна функція двох змінних, що використовується для перевірки ефективності алгоритмів оптимізації.

ЦФ – цільова функція.

ЗМІСТ

Вступ 10
1 Аналіз предметної області 12
1.1 Аналітичний огляд 12
1.2 Техніка загальних обчислень на графічному процесорі 14
1.2.1 GPGPU 14
1.2.2 Програмна архітектура NVIDIA CUDA 18
1.2.3 AMD ATI Stream Technology 18
1.2.4 Переваги технології CUDA 19
1.3 Архітектура NVIDIA CUDA 19
1.3.1 Програмно – апаратна платформа CUDA 19
1.3.2 Модель пам'яті технології CUDA 21
1.4 Висновки до першого розділу 26
2 Теоретична частина 27
2.1 Мультипроцесори 27
2.2 Модель програмування технології CUDA 29
2.2.1 Модель програмування CPU на CUDA 32
2.2.2 Модель програмування GPU на CUDA 33
2.3 Розв’язування задач багатовимірної оптимізації 36
2.3.1 Функції багатовимірної оптимізації 36
2.3.2 Алгоритми випадкового пошуку 39
2.4 Висновки до другого розділу 43
3 Практична частина 44
3.1 Процес встановлення CUDA для Microsoft Windows 44
3.2 Структура програми 53
3.3 Результат розробки додатку 55
3.3.1 Результати чисельних експериментів на функції Растригіна 56
3.3.2 Результати чисельних експериментів на функції Розенброка 59

3.4 Висновки до третього розділу 61
4 Охорона праці та безпека в надзвичайних ситуаціях..................................... 63
4.1 Закордонний досвід організації охорони праці в ІТ-компаніях............. 63
4.2 Оцінка дії електромагнітного імпульсу (ЕМІ) на елементи комп’ютерної системи. 67
4.3 Висновки до четвертого розділу... 71
Висновки.. 72
Перелік джерел.. 74
Додатки

ВСТУП

Актуальність теми. За останні десятиліття у всіх галузях науки значно зросла потреба у обробці величезної кількості даних. Для цього необхідні потужні обчислювальні системи. Через те, що стало неможливо збільшувати тактову частоту процесора, з'явились багатоядерні архітектури, які вимагали навичок паралельного програмування. У зв'язку з нестачею висококваліфікованих фахівців постала проблема створення ефективних високорівневих технологій програмування. В результаті відбувся бурхливий розвиток суперкомп'ютерних технологій у всьому світі. Проектуванням та розробкою високопродуктивних систем, які базуються на алгоритмах паралельної розробки вже давно займаються підрозділи та лабораторії всіх у провідних IT компаніях світу, зокрема Intel, Microsoft, Google, AMD, NVIDIA тощо. Паралельні обчислення привернули таку увагу завдяки стрімкому зростанню обсягів даних, котрі потребують обробки.

Вирішення багатьох затребуваних завдань, таких як комп'ютерне моделювання, обробка відео, візуалізація, розпізнавання образів, обчислювальна біологія та хімія, сейсмічний аналіз, прогнозування часових рядів, фінансовий аналіз, варіаційне обчислення, чисельні методи тощо вимагає значних витрат процесорного часу виконання обчислень. Використання математичних методів оптимізації вимагає величезної обчислювальної роботи, особливо великі труднощі виникають під час вирішення завдань оптимізації через велику кількість параметрів. Вирішити цю проблему і прискорити час обчислень можна шляхом паралельної обробки даних.

Мета дослідження: за допомогою паралельних обчислень максимально прискорити процес отримання екстремуму у завданнях багатовимірної оптимізації і тим самим показати переваги використання вищезгаданої технології порівняно з технологією послідовних обчислень.

Для досягнення вказаної мети, в роботі поставлено та розв’язано **наступні задачі:**

– вивчення техніки використання GPU;

– проаналізувати особливості застосування паралельних обчислень для покращення продуктивності ПЗ;

– аналіз та порівняння продуктивності CPU та GPU на прикладі випадкового пошуку;

– провести аналіз функцій, котрі застосовуються для розв’язування задач багатовимірної оптимізації;

– розробка програмного засобу для реалізації методів випадкового пошуку.

Об’єкт дослідження: процес застосування технології CUDA для проведення паралельних обчислень в задачах багатовимірної оптимізації.

Предмет дослідження: функції та алгоритми для здійснення багатовимірної оптимізації.

Методи дослідження: для виконання дослідження застосовувалися теорія паралельних обчислень, теорія багатоекстремальної оптимізації, теорія алгоритмів випадкового пошуку

Наукова новизна роботи:

– побудовано моделі ЦФ для алгоритмів випадкового пошуку;

– побудовано порівняльні характеристики продуктивності CPU та GPU на основі даних моделей;

– створено програмний засіб, котрий реалізує методи багатовимірної оптимізації та розроблені алгоритми.

Практичне значення одержаних результатів. Запропоновані алгоритми та створений на їх основі програмний додаток можуть бути використані для вирішення задач багатоекстремальної оптимізації, в теорії прийняття рішень, економіці та ін.

Апробація. Окремі результати роботи були представлені на IX науково-технічній конференції «Інформаційні моделі, системи та технології» Тернопільського національного технічного університету імені Івана Пулюя (08-09 грудня 2021 р.) у вигляді опублікованих тез [12].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналітичний огляд

Пікова продуктивність суперкомп'ютерів TOP500 [1] зростає з великою швидкістю. Перша десятка рейтингу TOP500 на листопад 2021 року представлена в табл. 1.1.

Таблиця 1.1 – Характеристики сучасних суперкомп’ютерів

№ з/п	Країна	Система	Ядра	Rmax (PFlop/s)	Rpeak (PFlop/s)	Потужність (MW)
1.	Японія	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu	7,630,848	442,010	537,212	29,899
2.	США	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM	2,414,592	148,600	200,794	10,096
3.	США	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox	1,572,480	94,640	125,712	7,438
4.	Китай	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC	10,649,600	93,014	125,435	15,371
5.	США	Perlmutter - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE	761,856	70,870	93,750	2,589
6.	США	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia	555,520	63,460	79,215	2,646
7.	Китай	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT	4,981,760	61,444	100,678	18,482
8.	ФРН	JUWELS Booster Module - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos	449,280	44,120	70,980	1,764
9.	Італія	HPC5 - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, DELL EMC	669,760	35,450	51,720	2,252
10.	США	Voyager-EUS2 - ND96amsr_A100_v4, AMD EPYC 7V12 48C 2.45GHz, NVIDIA A100 80GB, Mellanox HDR Infiniband, Microsoft Azure	253,440	30,050	39,531	2,148

Варто коротко зупинитися на кількох перших суперкомп'ютерах. Перше місце займає японське чудо техніки Fugaku з продуктивністю більше 440 петафлопс на с. Ця монструозна система включає більше 7,5 мільйонів обчислювальних ядер, а її споживана потужність сягає близько 30 МВт. Потім йде Summit (США) з продуктивністю 148,6 петафлопс на с. За енергоефективністю вона істотно поступається лідеру. При кількості ядер біля 2,5 млн має втричі меншу продуктивність Summit споживає 10 МВт електроенергії. Третю позицію зайняв суперкомп'ютер Sierra із США з продуктивністю майже 95 петафлопс. Також системи США завоювали п'яту, шосту і десяту позиції рейтингу - Perlmutter , Selene, Voyager.

Треба зауважити, що в рейтингу присутні розробки зі Німеччини (JUWELS Booster Module, 8 місце) та Італії (HPC5, 9 місце).

Графічні прискорювачі (GPU) стали використовуватися для неграфічних обчислень, що дозволило досягти прискорення роботи додатків у десятки, а то й у сотні разів. Для написання програм, які призначені для виконання на графічному процесорі, були розроблені спеціальні мови програмування: C-CUDA, Fortran-CUDA [2-5], OpenCL [6], OpenACC [7] та інші.

Існує багато успішних прикладів використання технології паралельних обчислень для збільшення продуктивності програмних продуктів, і, як наслідок, збільшення числа їхніх користувачів. За даними NVIDIA[8], на фінансовому ринку компанія Numerix використала технологію паралельних обчислень у новому додатку аналізу ризику контрагентів і досягла приросту у швидкості роботи майже у 20 разів. Понад 500 фінансових інститутів використовують у своїй діяльності програмне забезпечення Numerix. Більшість програм для опрацювання відео вже використовують CUDA-технологію для прискорення обчислень, в т.ч. продукти від Elemental Technologies, MotionDSP та LoiLo. Наразі CUDA прискорює спеціалізовану AMBER – програму для симуляції молекулярної динаміки, яка використовується понад 70 000 академічними

дослідниками та різноманітними фармацевтичними компаніями у всьому світі для зменшення періоду створення лікарських засобів.

Показником широкого застосування GPGPU технології є постійне зростання використання графічних прискорювачів у компаніях з списку Fortune 500, зокрема Schlumberger та Chevron в секторі енергетики та BNP Paribas у банківському секторі. Грунтуючись на багаторічному досвіді у розробці та використанні технології паралельних обчислень, компанія Simmakers пропонує послуги з впровадження цієї технології для компаній із різних галузей. Це дозволить підвищити обчислювальну продуктивність програмних продуктів, що збільшить їх конкурентоспроможність у сучасних умовах розвитку ринку ПЗ.

1.2 Техніка загальних обчислень на графічному процесорі

1.2.1 GPGPU

Фактично є набором апаратних та програмних технологій, котрі дають змогу застосовувати графічні процесори з метою пришвидшення багатьох не графічних додатків [1]. GPU є високоефективними багатоядерними процесорами, здатними до складних обчислень і дуже високої пропускної спроможності даних. GPGPU є результатом розвитку шейдерних програм та спеціалізованих для них мов програмування високого рівня, таких як Cg, GLSL і HLSL.

Початкове призначення графічних процесорів спрямоване на вирішення вузького кола завдань, що полягає в обробці графічних даних. Виходячи з цього, архітектури центрального та графічного процесорів істотно різняться (будова CPU і GPU наведена на рис. 1.1). Наприклад, основою відеочіпів NVIDIA є вісімдесятиядерний мультипроцесор, що має кілька тисяч регістрів.

Рисунок 1.1 - Будова GPU та CPU. Зліва – CPU, праворуч GPU

Графічний процесор підтримує метод обчислень SIMD, який означає, що ядра GPU виконують той самий набір інструкцій для кожного екземпляра даних. Більшість графічних алгоритмів використовують цей підхід, як найбільш ефективний засіб для задач графічної візуалізації. Такий модуль, що перетворює потік вхідних даних у вихідні, називається kernel - ядро (рис. 1.2) [2].

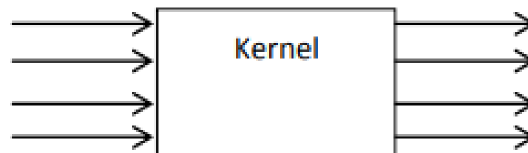

Рисунок 1.2 - Робота SIMD-архітектури

Для узагальнення основних відмінностей між будовами центрального та графічного процесорів варто зазначити, що основними завданням CPU є послідовне виконання одного потоку вказівок з найбільшою продуктивністю, тоді як конструкція GPU передбачає виконання найбільшого числа паралельних потоків одночасно.

З метою досягнення максимальної продуктивності центрального процесора розробники підвищують кількість завдань, які виконуються за один

такт. Однак, потік виконуваних CPU команд незмінно є послідовним, що виключає будь-яку можливість при поточній архітектурі збільшити швидкість виконання алгоритмів пропорційним збільшенням кількості ядер.

На кожному етапі графічного конвеєра дані один від одного не залежать і можуть оброблятися паралельно завдяки тому, що GPU займається обробкою графічних примітивів. На противагу послідовному потоку вказівок для центрального Kernel-процесора, GPU використовує виконавчі блоки, котрі є простими для завантаження.

Принцип організації доступу до пам'яті у обох видів процесорів теж є відмінним. На противагу CPU, саме GPU здійснює доступ послідовно, що означає для кожного моменту часу, якщо було звернення до осередку пам'яті, далі будуть задіяні дані, що йдуть слідом. Закономірно, що запис даних здійснюється за тим самим принципом. Крім іншого, колосальний обсяг даних характеризує більшість завдань, коті розв'язуються на графічному процесорі. Проблема затримки доступу до даних на CPU вирішується із застосуванням технології кешування та передбачення розгалуження коду. Вирішення того ж завдання засобами GPU виглядає зовсім інакше - якщо в очікуванні доступу до пам'яті один із паралельно виконуваних процесів призупинив виконання, то відеочіп перемикається на інший процес, котрий вже має всі необхідні для подальшого виконання дані.

Пропускна здатність пам'яті відеокарт, до речі, має суттєво більшу пропускну здатність, ніж оперативна. Механізм кешування для зменшення затримки доступу до пам'яті використовується і в GPU. Однак якщо в центральному процесорі кеш займає значну частку чіпа, то GPU відводить під потреби кешу всього 128-256 кілобайт, що в своє чергу виконується швидше для підвищення пропускної можливості.

Багатопотоковість у графічних процесорах також реалізована і на апаратному рівні. На CPU її задіяти не доцільно, оскільки кожне перемикання між потоками неминуче веде до значних тимчасовим затримок тривалістю

кілька сотень тактів. До того ж, ядро CPU здатне виконувати лише кілька одночасних. потоків (1-2). GPU, у свою чергу, здатний миттєво перемикатися між потоками, а кожному ядру під силу обробка до 1024 потоків.

CPU, будучи універсальним обчислювальним пристроєм, ефективно справляється з цілим спектром різних завдань, тоді як призначення графічних процесорів набагато більш вузькоспрямоване. У задачах з множинними розгалуженнями та переходами графічний процесор менш ефективний як центральний. Тому слід пам'ятати, що технологія GPGPU актуальна при обробці великих обсягів даних На ранній стадії розвитку цієї технології, продуктивність GPU перевищувала в 10 разів CPU. Виробники відеокарт побачили велику перспективу у GPGPU, тому стрімко почали розвиватися у цьому напрямі. Використання технології GPGPU як ніколи актуальне для розробників. У зв'язку з цим, виробники відеокарт ще 2003 року почали активно розвиватися у сфері неграфічних обчислень. Таким чином, результатом роботи компанії NVIDIA, показники продуктивності якої найвищі, стала NVIDIA CUDA.

Мова обрахунків OpenCL є відкритим фреймворком для створення спеціалізованих програм, котрі зв'язані з паралельними обчисленнями для різних видів процесорів [6]. Важливо, що OpenCL є повністю відкритою технологіяю, він розробляється та підтримується некомерційним консорціумом Khronos Group, до якого входить багато великих компаній, включаючи Apple, AMD, Intel, nVidia, Sun Microsystems, Sony Computer Entertainment та інші.

Але OpenCL має ряд істотних недоліків для виконання поставленої мети. По-перше, для використання необхідно зв'язатися з розробниками. По-друге, OpenCL має низку обмежень, наприклад, відсутність підтримки покажчиків на функції, рекурсії, бітових полів та ін.

На сьогоднішній день OpenCL за оцінками фахівців nVidia є «сирим» та недоопрацьованим. Але виробники відеокарт активно розвиваються у загальній стандартизації GPGPU. Безсумнівно, OpenCL полегшить програмістам роботу, і дасть новий поштовх розвитку обчислень на GPU.

1.2.2 Програмна архітектура NVIDIA CUDA

В архітектуру CUDA включено уніфікований шейдерний конвеєр, який виконує загальні обчислення та залучає будь-які арифметично-логічні пристрої, що входить до мікросхеми. Метою CUDA є створити GPU, який безперебійно справляється з загальними обчисленнями, а не лише з традиційними завданнями комп'ютерної графіки, тому виконуючим пристроям GPU дозволено довільний доступ до пам'яті для читання та запису, а також доступ до програмно-керованого кешу, що отримав назву пам'ять, котра розділяється.

Щоб охопити максимальну кількість розробників, NVIDIA взяла стандартну мову C та доповнила її кількома новими ключовими словами, що дозволяють задіяти спеціальні засоби, властиві архітектурі CUDA. За кілька місяців після випуску GeForce 8800GTX відкрила доступ до компілятора нової мови CUDAC. Вона стала першою мовою, розробленою компанією з виробництва GPU з метою спростити програмування GPU для обчислень загального призначення.

Окрім створення мови для програмування GPU, NVIDIA пропонує спеціалізований драйвер, що дозволяє використовувати можливості потужно-паралельних обчислень в архітектурі CUDA. Тепер користувачам не потрібно вивчати програмні інтерфейси OpenGL або DirectX або представляти свої завдання у вигляді задач комп'ютерної графіки.

1.2.3 AMD ATI Stream Technology

Технологія AMD STREAM є рядом апаратних і програмних оптимізацій, призначених для високопродуктивних обчислювальних робочих потоків та ресурсомістких додатків з використанням технології OpenCL™ - відкритого міжплатформного стандарту програмування, який використовується для стандартних обчислень. Серед функцій можна виділити підтримку з корекцією пам'яті помилок (ECC), швидкі операції з плаваючою комою та безпосередній

доступ до пам'яті, який забезпечує обмін даними між кількома графічними процесорами з низькою затримкою.

У поєднанні з бібліотеками, оптимізованими для роботи з графічними процесорами та стороннім проміжним ПЗ технологія AMD STREAM дозволяє реалізувати обчислювальну продуктивність відеокарт AMD FirePro [9].

1.2.4 Переваги технології CUDA

Це єдине середовище розробки на мові C, яке дає змогу фахівцям писати ПЗ для розв’язування складних обчислювальних проблем за коротший час, через багатоядерну обчислювальну потужність GPU. Гнучкість, простота (розширена мова програмування C) та безкоштовність (SDK вільно скачується з developer.nvidia.com) є основними перевагами CUDA.

Вивчивши ринок відеокарт, можна сказати, що NVIDIA CUDA найбільше поширена. За цією технологією є більша кількість навчальних матеріалів, ніж інших аналогічних. NVIDIA веде активну підтримку розробників та безкоштовні відеоуроки з роботи з CUDA. А також на офіційному сайті виділено спеціальний розділ CUDAZONE для спілкування програмістів, обміну матеріалами, перекладів статей та описів технології, можливості задати запитання творцям CUDA.

1.3 Архітектура NVIDIA CUDA

1.3.1 Програмно – апаратна платформа CUDA

Перший реліз вийшов 15 лютого 2007 року. Основне призначення – дати програмісту можливість використовувати GPU (надалі «пристрій») як співпроцесор для задач, що вимагають паралельних обчислень, абстрагуючись при цьому від термінології та не використовуючи бібліотеки, специфічні для обробки 3D графіки. Однак, не все так просто, хоча б тому, що відеокарти були спочатку орієнтовані та багато років розвивалися як пристрої обробки графіки. І

саме особливості архітектури GPU викликають найбільші складності при першому знайомстві із платформою.

До складу платформи, крім самої відеокарти (починаючи з GeForce 8-тисячної серії) входить, так званий драйвер CUDA, CUDA Toolkit і CUDA SDK. CUDA Toolkit включає бібліотеки, необхідні для роботи з платформою, і компілятор nvcc, що транслює вихідний код програм у проміжний асемблерний код, а також додаткові бібліотеки.

Драйвер CUDA в ранніх версіях включав лише компілятор, що здійснює перетворення проміжного асемблера, генерованого компілятором nvcc мікрокод, виконуваний на GPU. Зараз драйвер CUDA та драйвер пристрою об'єднані в один пакет. CUDA SDK містить набір вихідних кодів простих програм, що ілюструють методи та можливості роботи з платформою CUDA.

CUDA підтримується лише процесорами відеоприскорювачів GeForce восьмого покоління і старше (GeForce 8, GeForce 9, GeForce 200), а також Qaadro і Tesla, на це варто звернути особливу увагу. Основними атрибутами CUDA є взаємодія з графічними API OpenGL та DirectX.25, наявність низькорівневої розробки, забезпечення доступу з швидкої пам'яті, що розділяється, підтримка 32- і 64-бітних ОС, також CUDA використовує розширений варіант мови C.

Розглянемо логічну архітектуру, для розуміння роботи з платформою (рис. 1.3):

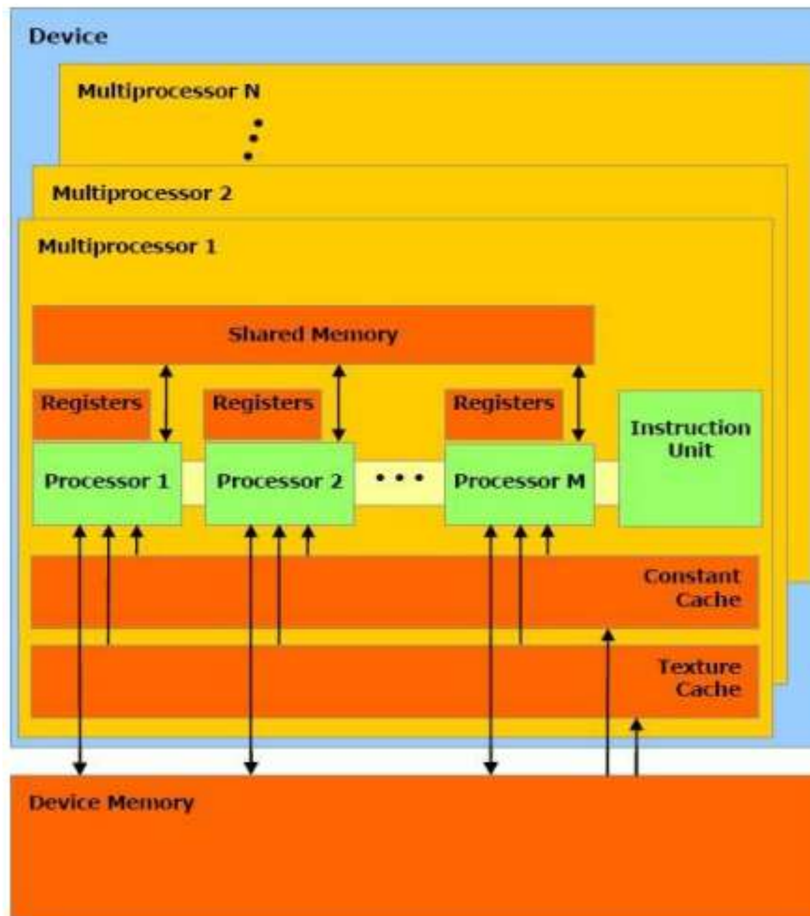

Рисунок 1.3 – Архітектура CUDA

1.3.2 Модель пам'яті технології CUDA

Можливість вільного доступу до пам'яті з побайтовою адресацією, є надзвичайно важливим моментом. Самі потоки CUDA можуть адресуватися до даних з різних просторів пам'яті в один і той самий час, як показано малюнку 1.4.

Будь-який потік має приватну локальну пам'ять. Кожен блок потоку володіє загальною пам'яттю, котра видима всім потокам блоку і з тим самим часом життя, як і блок. У всіх потоків є доступ до однієї й тієї ж глобальної пам'яті.

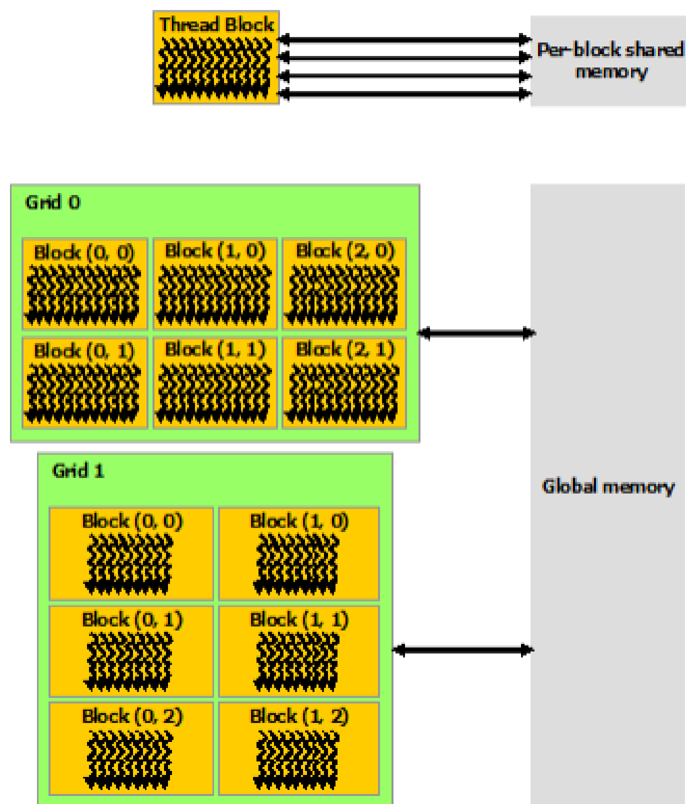

Рисунок 1.4 – Ієрархія пам'яті

Є також два простори для зчитування на додачу, котрі доступні для всіх потоків, а саме постійної та текстурної пам'яті. Пам'ять текстур також пропонує ряд варіантів звертань та фільтрацію даних для окремих форматів. Зокрема, команди, котрі звертаються до адресованої пам'яті (мається на увазі глобальна, локальна, загальна, постійна або текстурна), ймовірно матимуть потребу повторитися декілька разів власне залежно від розподілу адрес пам'яті за потоками. Здебільшого для глобальної пам'яті чим більше розкидані адреси, тим більше скорочено пропускну здатність [12].

Глобальна пам'ять. Міститься у пам'яті пристрою, а вона - через 32-, 64- або 128-байтові транзакції. Вони повинні бути вирівняні: тільки 32-, 64- або 128-байтові сегменти пам'яті пристрою, які вирівняні за тим, чия власне перша адреса кратна їх розміру, можуть бути прочитані чи записані в пам’ять угоди.

Скільки транзакцій необхідно і скільки пропускної здатності зрештою потрібно, залежить від обчислювальної здатності пристрої. Compute Capability

2.x, 3.x, 5.x та 6.x дають більш детальну інформацію про те, як обробляються глобальні звернення до пам'яті для різних обчислювальних можливостей.

Інструкції з глобальної пам'яті підтримують читання або запис слів розміром 1, 2, 4, 8 чи 16 байт. Будь-який доступ (через змінну або показник) до даних, котрі знаходяться в глобальній пам'яті, компілюється в одну команду глобальної пам'яті тоді і лише тоді, коли розмір типу даних дорівнює 1, 2, 4, 8 або 16 байтам, і дані, природно, (тобто його адреса кратна цьому розміру).

Якщо цей розмір та вимога вирівнювання не виконуються, доступ компілюється в кілька інструкцій з шаблонами доступу, які чергуються та дозволяють цим інструкціям повністю об'єднатися. Тому рекомендується використовувати типи, що відповідають цьому на вимогу, для даних, котрі містяться в глобальній пам'яті. Вимога вирівнювання автоматично виконується для вбудованих типів char, short, int, long, longlong, float, double.

Локальна пам'ять. Доступ до неї відбувається лише для деяких автоматичних змінних. Автоматичними змінними, які компілятор може розмістити в такій пам'яті, є: масиви, для котрих він не може визначити, що їх індексують зі сталими одиницями, великі структури чи масиви, які споживатимуть занадто багато простору для реєстрації, будь-яка змінна, якщо власне ядро бере більше регістрів, ніж доступно (розлиття регістрів) [12].

Локальний простір пам'яті є в чіпі. Завдяки цьому доступ до нього має таку ж високу часову затримку та властиво низьку пропускну здатність, так само як і доступ до власне глобальної пам'яті, і для них однакові вимоги до коалесценції пам'яті. Проте локальна пам'ять організована так, що послідовні 32-бітові слова матимуть доступ до послідовного ідентифікатора потоку. Отже, доступ до них повністю об'єднаний, поки всі нитки в warp отримують одну і ту саму відносну адресу (наприклад, один і той же індекс у змінному масиві, той самий член у структурній змінній).

Загальна пам'ять. Так як вона вмонтована в чіп, тому володіє значно більш вищою пропускною здатністю та меншою затримкою, ніж локальна чи

глобальна пам'ять. Для одержання високої пропускної здатності пам'ять поділяється на однакові за розміром модулі однакового розміру (банки), до яких можна одночасно звертатися. Таким чином, будь-який запит на читання або запис в пам’ять, що складається з n адрес, які потрапляють у n різних банків пам'яті, може обслуговуватися одночасно, що дає загальну пропускну здатність, яка в n разів перевищує пропускну здатність одного модуля [12].

Щоб одержати максимальну продуктивність, треба знати, як адреси пам'яті співставляються з банками пам'яті, щоб запланувати запити пам'яті, щоб мінімізувати конфлікти у банках. Це описано в Compute Capability 2.x, 3.x, 5.x та 6.x для обчислювальних пристроїв можливості 2.x, 3.x, 5.x і 6.x відповідно.

Постійна пам'ять. Є в пам'яті пристрою та постійно кешується. Потім запит розбивається на стільки окремих, що у вихідному запиті є різні адреси пам'яті, що зменшує пропускну здатність на коефіцієнт, що дорівнює числу окремих запитів. Потім одержані запити опрацьовуються за пропускної здатності постійного кешу у випадку попадання в кеш або за пропускної здатності власне пам'яті пристрою [12].

Текстурна пам'ять. Пам'ять текстури та поверхні є у пам'яті пристрої та кешуються в кеші текстури, тому при їх зчитуванні стоїть одна пам'ять, що зчитується з пам'яті пристрою тільки при пропуску кеша, інакше це просто коштує одного читання з кешу текстур. Кеш текстури оптимізовано для 2D просторової локальності, тому потоки одного і того ж детектора, які читають текстурні або поверхневі адреси, що знаходяться близько один до одного в 2D, досягнуть найкращої продуктивності. Крім того, він призначений для потокового завантаження із постійною затримкою. Кеш- кеш зменшує потребу в пропускній здатності, але не покриває затримку [12].

В табл. 1.2 наведені технічні характеристики різних видів пам’яті.

Таблиця 1.2 - Технічні характеристики різних видів пам’яті

Назва виду пам’яті	Розміщення	Кешованість	Зчитування / Запис	Доступність	Тип пам’яті
Регістрова	Чіп	Ні	Так / Так	Ядро	Dedicated HW
Локальна	За межами чіпу	Ні	Так / Так	Ядро	DRAM
Загальна	Чіп	Ні	Так / Так	Всі ядра в групі	Dedicated HW
Глобальна	За межами чіпу	Ні	Так / Так	GPU і CPU	DRAM
Константна	За межами чіпу	Так	Так / Ні	GPU і CPU	DRAM
Текстурна	За межами чіпу	Так	Так / Ні	GPU і CPU	DRAM

На рис. 1.5 показана програмна модель пам'яті відеокарти.

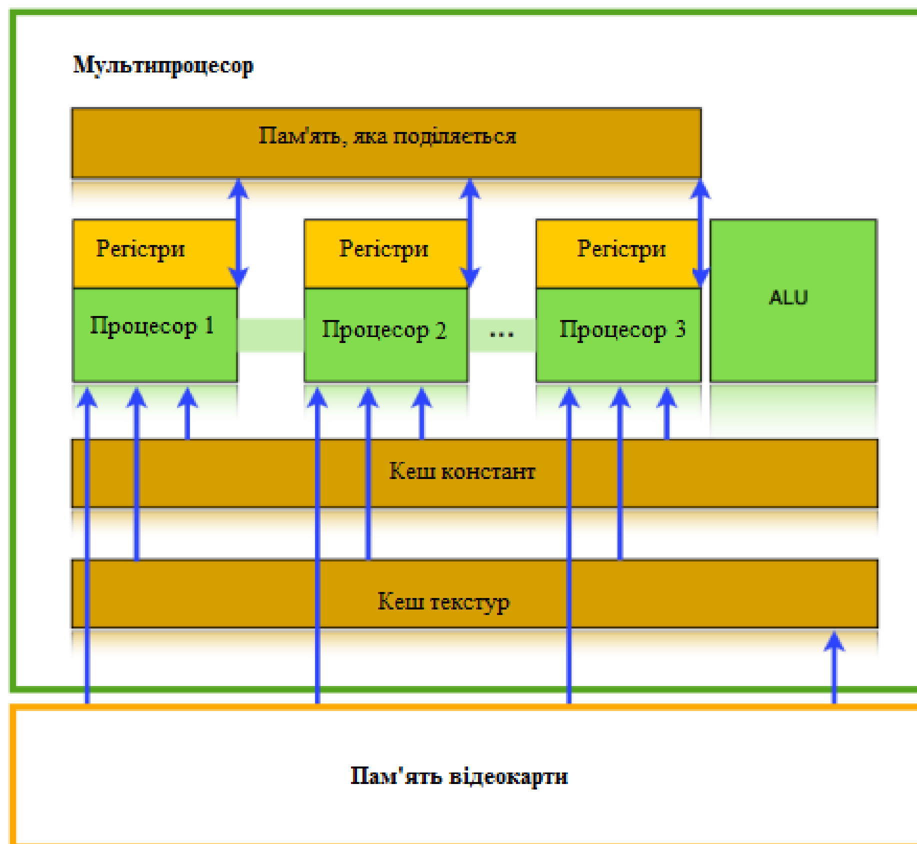

Рисунок 1.5 – Програмна модель пам'яті мультипроцесора

1.4 Висновки до першого розділу

В цьому розділі проаналізовано параметри сучасних суперкомп'ютерів. Також представлено окремі приклади вживання механізму паралельних обчислень для збільшення продуктивності програмних продуктів.

Описана техніка загальних обчислень на графічному процесорі. Проаналізовано специфіку архітектури NVIDIA CUDA та технічні параметри різних видів пам’яті.

2 ТЕОРЕТИЧНА ЧАСТИНА

2.1 Мультипроцесори

Поява багатоядерних процесорів та багатопроцесорних графічних процесорів означає, що основні процесори тепер є паралельними системами. Більше того, їхній паралелізм продовжує діяти відповідно до закону Мура. Завдання полягає в тому, щоб розробити прикладне ПЗ, яке прозоро масштабує свій паралелізм, щоб використовувати дедалі більше процесорних ядер, так само, як програми 3D-графіки прозоро масштабують їхній паралелізм для багатьох чистих графічних процесорів з різною кількістю ядер.

Модель паралельного програмування CUDA призначена для подолання цієї проблеми, зберігаючи при цьому низьку криву навчання для програмістів, знайомих зі стандартними мовами програмування, такими як C.

По суті, це три ключові абстракції – ієрархія груп потоків, розділених спогадів та бар'єрної синхронізації, які просто надаються програмісту як мінімальний набір мовних розширень. Ці абстракції забезпечують дрібнозернистий паралелізм даних та паралелізм потоків, вкладені в грубий паралелізм даних та паралелізм завдань. Вони направляють програміста на те, щоб поділити проблему на грубі підзадачі, які можуть вирішуватись незалежно паралельно блоками потоків, а кожна підзадача - у дрібніші частини, які можуть спільно вирішуватись спільно всіма потоками всередині блоку.

Це розкладання зберігає виразність мови, даючи змогу потокам взаємодіяти під час вирішення підзадачі й водночас забезпечує автоматичну масштабованість (рис. 2.1).

Архітектура побудована навколо масштабованого масиву SMs [4]. Мультипроцесор призначений для одночасного виконання сотень потоків. Для керування такою власне великою кількістю потоків він використовує унікальну архітектуру SIMT. На відміну від процесорних ядер, інструкції видаються по

порядку, але немає прогнозу розгалуження і немає спекулятивного виконання. Архітектура SIMT та апаратна багатопотоковість описують характеристики архітектури потокового багатопроцесора, які є спільними для всіх пристроїв.

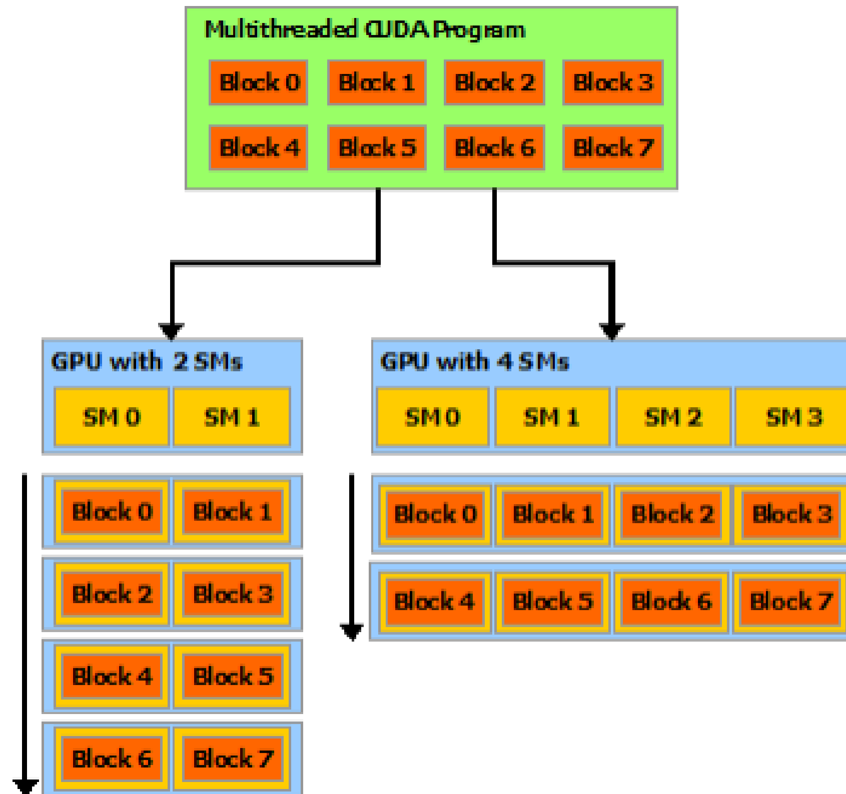

Рисунок 2.1 - Модель мультипроцесора CUDA

Коли мультипроцесору надається один або кілька блоків потоків для виконання, він розбиває їх на перекоси, і кожен warp отримує запланований планувальник warp для виконання. Спосіб, яким блок розбивається на перекоси, завжди той самий. Ієрархія потоків описує, як ідентифікатори потоків відносяться до індексів потоків у блоці.

Архітектура SIMT схожа на організацію векторів SIMD, оскільки одна команда управляє кількома елементами обробки. Ключовою відмінністю є те, що організації векторів SIMD надають SIMD-ширину ПЗ, тоді як інструкції SIMT визначають поведінку виконання та розгалуження одного потоку [9].

2.2 Модель програмування технології CUDA

Модель виконання CUDA заснована на примітивах потоків, блоків потоків та сіток з функціями ядра, що визначають програму, котра виконується окремими потоками в потоковому блоці та сітці. Коли викликається функція ядра, властивості сітки описуються конфігурацією виконання, яка має спеціальний синтаксис у CUDA. Підтримка динамічного паралелізму в CUDA розширює можливості налаштування, запуску та синхронізації на нових мереж для потоків, що працюють на пристрої (див. рис. 2.2).

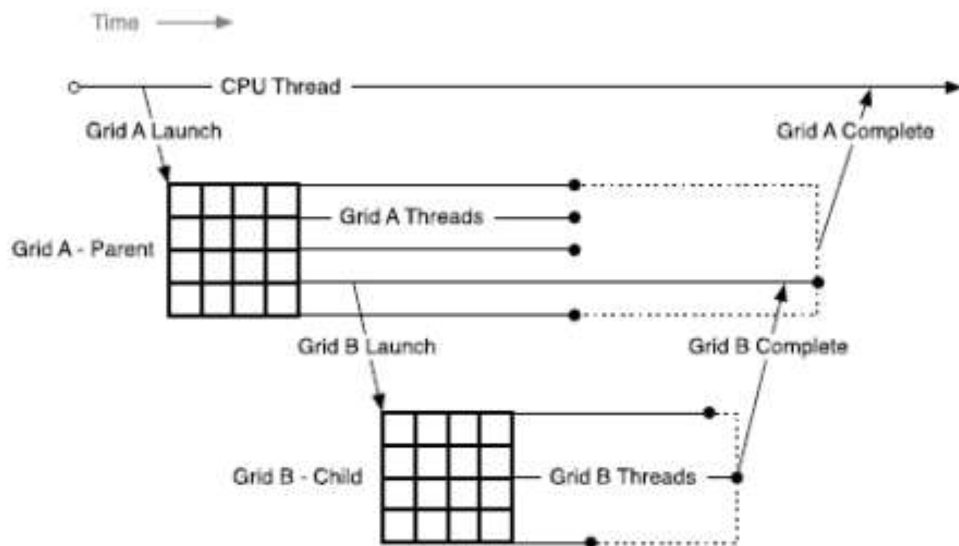

Рисунок 2.2 – Модель програмування CUDA

Потік пристрою, який налаштовує та запускає нову сітку, належить батьківській сітці, а сітка, створена викликом, представляє собою дочірню сітку.

Виклик та завершення дочірніх сіток правильно складений, що означає, що батьківська сітка не вважається завершеною доти, доки всі дочірні сітки, створені її потоками, не будуть завершені. Навіть якщо потоки, які викликаються, явно не синхронізуються на запущених дочірніх сітках, середовище виконання гарантує неявну синхронізацію між батьківським та дочірнім.

На хості та пристрої середовище виконання CUDA пропонує API для запуску ядер, для очікування завершення роботи та для відстеження залежностей між запусками через потоки та події. У хост-системі стан запусків та примітиви CUDA, що посилаються на потоки та події, поділяються всіма потоками всередині процесу; Проте процеси виконуються незалежно та не можуть спільно використовувати об'єкти CUDA (рис. 2.3).

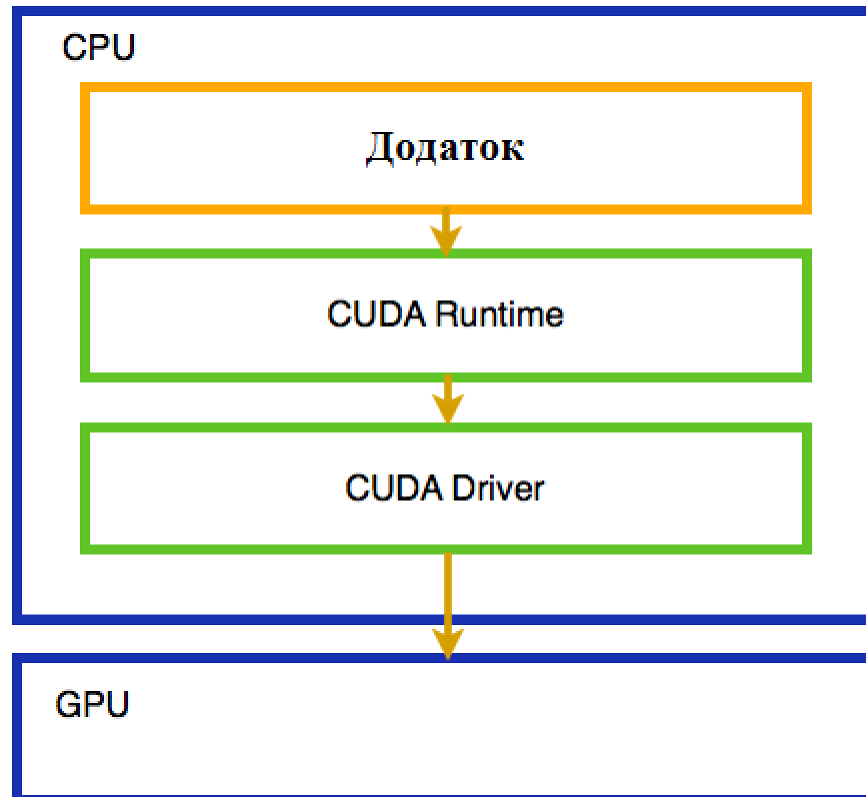

Рисунок 2.3 – Схема CUDA API

На пристрої існує аналогічна ієрархія: запущені ядра та об'єкти CUDA видно для всіх потоків у блоці потоків, але незалежні між блоками потоків Це означає, наприклад, що потік може бути створений одним потоком і використовуватись будь-яким іншим потоком в одному блоці потоку, але не може бути розділений нитками у будь-якому іншому блоці потоку.

Операції виконання CUDA з будь-якого потоку, включаючи запуск ядра, видно через блок потоків. Це означає, що потік, котрий викликається, у

батьківській сітці може виконувати синхронізацію на сітках, запущених цим потоком, іншими потоками в потоковому блоці або потоками, створеними в одному блоці потоків. Виконання блоку потоку не вважається завершеним доки всі запуски всіх потоків у блоці не будуть завершені. Якщо всі потоки в блоковому завершенні до запуску всіх дочірніх запусків завершено, тоді операцію синхронізації буде автоматично запущено.

Потоки та події CUDA дозволяють керувати залежностями між запуском сітки: сітки, запущені в один потік, виконуються в порядку, а події можуть використовуватись для створення залежностей між потоками.

Потоки та події, створені на пристрої, служать цій же меті. Потоки та події, створені в сітці, існують в області потоку, але мають невизначену поведінку при використанні поза блоком потоку, де їх було створено. Як описано вище, вся робота, запущена блоком потоку, що неявно синхронізується при виході з блоку. Робота, запущена в потоки, включена до цього, причому всі залежності вирішуються відповідним чином. Поведінка операцій над потоком, який був змінено поза потоком, не визначено.

Потоки та події, створені на хості, мають невизначену поведінку при використанні в будь-якому ядрі, так само як потоки та події, створені батьківською сіткою, мають невизначену поведінку, якщо вони використовуються у дочірній сітці.

Порядок запуску ядра з середовища виконання пристрою йде за семантикою впорядкування потоку CUDA. У блоці потоку всі ядра запускаються в один потік, що виконуються в порядку. З кількома потоками в одному потоковому блоці, запущеному в той самий потік, упорядкування всередині потоку залежить від планування потоку всередині блоку, яке може керуватися за допомогою примітивів синхронізації, таких як __syncthreads ().

Оскільки потоки поділяються всіма потоками в поточному блоці, неявний потік NULL також використовують спільно. Якщо кілька потоків у потоковому блоці запускаються в неявний потік, то ці запуски будуть виконуватись у

порядку. Якщо потрібно паралелізм, слід використовувати явні іменовані потоки.

Динамічний паралелізм дозволяє спростити паралелізм усередині програми. Проте час виконання пристрою не вносить жодних нових гарантій паралелізму у модель виконання CUDA. Немає гарантії одночасного виконання між будь-якою кількістю блоків потоків на пристрої.

Відсутність гарантії паралелізму поширюється на блоки батьківських потоків та їх дочірні сітки. Коли блок батьківського потоку запускає дочірню сітку, дочірньому пристрою не гарантується початок виконання, поки блок батьківського потоку не досягне явної точки синхронізації (наприклад, cudaDeviceSynchronize ()).

Хоча паралелізм часто може бути легко досягнутий, він може варіюватися в залежності від deviceconfiguration, робочого навантаження додатку та планування часу виконання. Тому небезпечно залежати від будь-якого паралелізму між різними блоками потоків.

2.2.1 Модель програмування CPU на CUDA

Так як в GPU доступ до ОЗП відсутній, то програміст повинен наперед дбати, щоб ресурси, котрі потрібні для запуску ядра програми, були у пам'яті відеокарти. З цією метою із CUDA SDK беруться три основні функції: cudaMalloc, cudaMemcpy та cudaFree. Вони мають те саме призначення, що і стандартні malloc, memcpy та free, але, звичайно, всі операції виконуються у відеопам'яті.

Процес налаштування сітки та блоків полягає у налаштуванні їх розміру. Основне завдання кодера на цьому етапі – знайти оптимальний баланс між розміром та кількістю блоків. Збільшуючи у блоці число потоків, можна зменшити число викликів у глобальній пам'яті під дією зростання інтенсивності обміну даними власне між потоками із застосуванням швидкої пам'яті. Проте число регістрів, виділених блоку, фіксовано, і, якщо число потоків набагато

більше, то збільшиться час виконання ядра, тому що GPU буде поміщати дані у локальну пам'ять. При застосуванні операції виклику ядра передаються всі розмірності сітки та блоку, визначені раніше.

Властиво виклик ядра проходить як регулярна функція мовою С. Одна значна відмінність у тому, що з виклику ядра потрібно перенести раніше визначені розміри сітки та блоку.

Після виконання ядра слід скопіювати результати програми назад в пам'ять з використанням функції cudaMemcpy, вказавши напрямок зворотної копії (від GPU до CPU).

2.2.2 Модель програмування GPU на CUDA

Кваліфікатори типу функції визначають, чи виконуватиметься функція на хості або на пристрої і чи він може бути викликаний з хоста або з пристрою.

__device__ оголошує функцію, яка буде виконуватись на пристрої та викликатися лише з пристрою.

__global__ оголошує функцію як ядро. Така функція виконується на пристрої та викликається з хоста. Викликається з пристрою для пристроїв з обчислювальною здатністю 3,2 або вищою. Виклик функції __global__ є асинхронним, тобто він повертається до того, як пристрій завершить виконання.

__host__ оголошує функцію, яка виконується на хості та викликається лише з хоста.

__global__ та __host__ не можуть використовуватися разом. Однак кваліфікатори __device__ та __host__ можуть використовуватися разом, і в цьому випадку функція компілюється як для хоста, так і для пристрою.

__noinline__ і __forceinline__ Специфікатор функції __noinline__ може використовуватись як підказка для компілятора, щоб не включати функцію, якщо це можливо. Тіло функції все одно має бути в тому ж файлі, де він викликається. Позначення функції __forceinline__ можна використовувати, щоб змусити компілятор вбудувати функцію.

Визначники типу змінної визначають розташування пам'яті на пристрої змінної.

Автоматична змінна, оголошена в коді пристрою без будь-яких або специфікаторів __device__, __shared__ і __constant__, зазвичай знаходиться у регістрі. Однак у деяких випадках компілятор може захотіти помістити його у локальну пам'ять, що може мати несприятливі наслідки для продуктивності. Розглянемо набір специфікаторів, що визначають тип пам'яті для розміщення змінних:

__device__ оголошує змінну, яка знаходиться на пристрої. Не більше одного з кваліфікаторів іншого типу можуть застововуватися разом з __device__ для подальшого визначення того, який простір пам'яті належить змінній. Якщо жоден із них не присутній, то змінна зберігається у глобальній пам'яті. Доступний з усіх потоків сітці і від хоста через бібліотеку часу виконання (cudaGetSymbolAddress() / cudaGetSymbolSize() / cudaMemcpyToSymbol() / cudaMemcpyFromSymbol()).

__constant__ оголошує змінну, яка залишається у постійному просторі пам'яті. Доступна з усіх потоків у сітці та від хоста через бібліотеку часу виконання (cudaGetSymbolAddress() / cudaGetSymbolSize() / cudaMemcpyToSymbol() /cudaMemcpyFromSymbol()).

__shared__оголошує змінну, яка залишається в області загальної пам'яті блоку. Доступна для всіх потоків усередині блоку, при оголошенні змінної у спільній пам'яті як зовнішній масив, такий як Extern __shared__ float shared [];

Розмір масиву визначається під час запуску. Усі змінні оголошені таким чином, починаються з однієї й тієї ж адреси в пам'яті, що макет змінних у масиві має явно керуватися за допомогою зміщень. Наприклад, якщо потрібний еквівалент

```
short array0 [128];
float array1 [64];
int array2 [256];
```

то у динамічно розподіленій спільній пам'яті можна оголосити і ініціалізувати масиви наступним чином:

```
Extern __shared__ float array [];
__device__ void func() // Функція __device__ або __global__
{
Short * array0 = (short *) array;
Float * array1 = (float *) & array0 [128];
Int * array2 = (int *) & array1 [64];
}
```

Покажчики мають бути прив'язані до типу, на який вони вказують.

__managed__оголошує змінну, яка може посилатися як на пристрій, так і на хост-код, наприклад його адресу можна взяти або його можна прочитати або записати безпосередньо з пристрою чи функції хоста.

Nvcc підтримує обмежені покажчики за допомогою ключового слова __restrict__. Обмежені покажчики були введені на C99, щоб полегшити проблему згладжування, яка існує в мові C, і яка забороняє всі види оптимізації від переупорядкування коду до загального виключення суб-експресії.

У мові C покажчики a, b і c можуть бути згладжені, тому будь-який запис через c може змінювати елементи a чи b. Це означає, що для забезпечення функціональної коректності компілятор не може завантажувати [0] і b[0] у регістри, множити їх і зберігати результат як з [0], так і з [1], оскільки результати відрізнятимуться від результатів Абстрактна модель виконання, якщо, скажімо, [0] - це справді те саме місце, що і c[0]. Тому компілятор не може скористатися загальним виразом. Аналогічно компілятор не може просто переупорядкувати обчислення c[4] у безпосередній близькості від обчислення c[0] та c[1], оскільки попередній запис c[3] може змінити входи на обчислення c[4].

2.3 Розв’язування задач багатовимірної оптимізації

2.3.1 Функції багатовимірної оптимізації

Для тестування алгоритмів багатовимірної оптимізації, заснованих на як на методах еволюційного моделювання, так і на будь-яких інших слід використовувати спеціальні тестові функції.

Нижче наведено перелік деяких з них, вказані рекомендовані інтервали зміни змінних та глобальні оптимальні значення.

Функція Розенброка. Є невипуклою функцією, котра використовується з метою оцінки продуктивності алгоритмів оптимізації (рис. 2.4). Мінімум у точці (1, 1) [31].

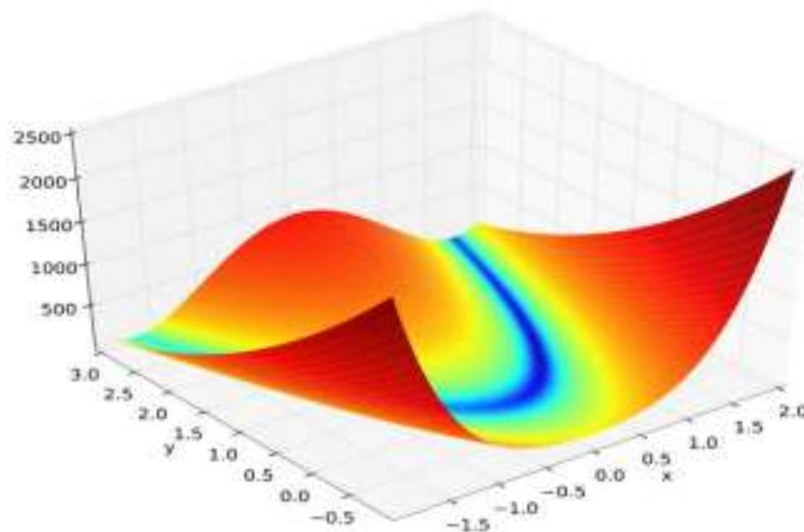

Рисунок 2.4 – Графік функції Розенброка

Функцію Розенброка можна визначити так:

$$f(x,y) = (1-x^2) + 100(y - x^2)^2 \quad (2.1)$$

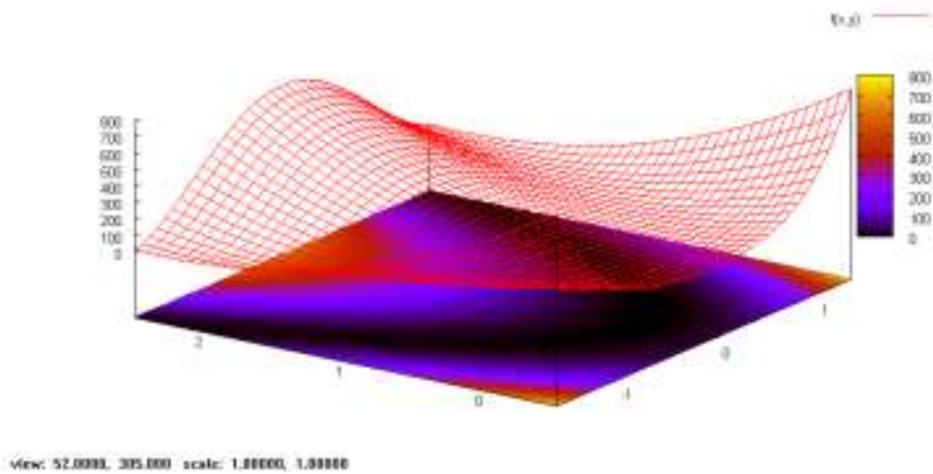

Рисунок 2.5- Значення функції в околі точки (0,0).

Рекомендований інтервал пошуку оптимального рішення щодо кожної змінної (-5; 5). Глобальний мінімум функції знаходиться у точці (x, y) =(1,1), де f(x, y)=0 [31].

Функція Хіммельблау. Є мультимодальною функцією двох змінних, котра застосовується з метою перевірки ефективності оптимізаційних алгоритмів [26] (рис. 2.6) і описується формулою:

$$f(x, y) = (x^2+y-11)^2 + (x+y^2-7)^2 \tag{2.2}$$

Має локальний максимум x = -0.270845, y = -0.923039 зі значенням f(x,y) = 181.617 і чотири рівнозначні локальні мінімуми:

f(3.0, 2.0) = 0.0,

f(-2.805118, 3.131312) = 0.0,

f(-3.779310, -3.283186) = 0.0;

f(3.584428, -1.848126) = 0.0.

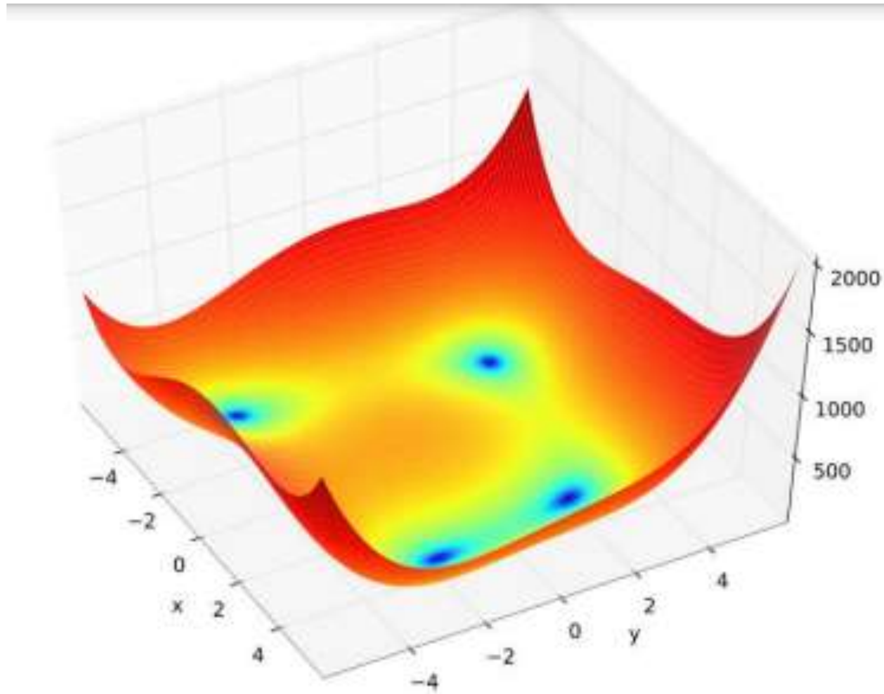

Рисунок 2.6 - Графік функції Хіммельблау для двох змінних

Функція Растригіна. Є невипуклою функцією для тестування ефективності оптимізаційних алгоритмів. Функцію можна назвати типовим прикладом нелінійної мультимодальної функції [10, 11].

Визначення функції:

$$f(x) = An + \sum_{i=1}^{n}[x_i^2 - A\cos(2\pi x_i)], \tag{2.3}$$

де A = 10 і $x_i \in [-5{,}12; 5{,}12]$.

Глобальний мінімум у точці x=0 де f (x)=0.

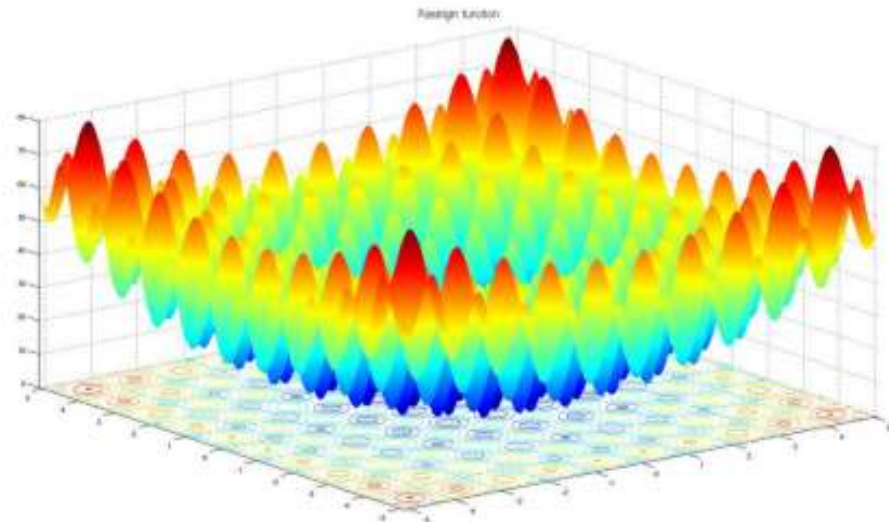

Рисунок 2.7 - Графік функції Растригіна для двох змінних

Для дослідження використано функції Розенброка та Растригіна. Вибір даних функцій обумовлений можливістю узагальнення для великих розмірностей, а також наявністю множини локальних мінімумів, що робить завдання знаходження глобального мінімуму відносно важким. Ще одним фактом є те, що обрані функції є найбільш поширеними для тестування різних алгоритмів оптимізації [13, 15].

2.3.2 Алгоритми випадкового пошуку

Існує направлений та ненаправлений випадковий пошук [10, 26].

У першому випадку випробування між собою пов'язані. Використовувані методи ведуть тільки до локальних екстремумів. Збіжність методів висока. Одержані дані використовуються для побудови подальших випробувань.

При ненаправленому пошуку одержані результати не залежать від попередніх. Сходимість такого пошуку дуже мала. Методи мають перевагу, таку як вирішення багатоекстремальних завдань. Для більш цікавого результату досліджуємо поставлене завдання на направленому та ненаправленому випадковому пошуку.

Простий випадковий пошук є прикладом ненаправленого пошуку. Передбачається, що необхідний мінімум лежить у якомусь n-вимірному паралелепіпеді. У цій фігурі згідно з єдиним законом N випадкових вибірок вибираються випадково і в них розраховується ЦФ. Як вирішення проблеми береться точка, в котрій функція має мінімум. Імовірність мала, що хоча б одна точка потрапить у власне невеликій окіл локального мінімуму. Дійсно, хай N = 106, а діаметр басейну складає 10% поблизу мінімуму. Тоді обсяг цієї депресії становить 0,1 н. частини обсягу n-вимірного паралелепіпеда. Навіть із числом змінних n> 6 практично немає точки в басейні.

Ось чому використовують мале число точок N і кожну таку точку вважають за нульове наближення. Потім з кожної такої точки проводять спуск, досить швидко потрапляючи в найближчу виїмку чи улоговину; якщо кроки такого спуску стають коротшими, він зупиняється без досягнення високої точності. Власне цього вже досить, щоб розуміти величину функції у самому ближньому локальному мінімумі із приємлемою точністю.

Якщо порівняти фінальні величини функції на усіх здійснених спусках, тоді вдасться вивчити розміщення всіх локальних мінімумів та співставити їх. Властиво потім можна визначити необхідні за змістом задачі мінімуми та здійснити в них ще спуски для одержання за вищої точності координат точок мінімуму [15].

Дану область треба вписати в n-вимірний гіперпаралепіпед і залишити лише ті точки, що потрапляють у допустиму область. Графічне представлення алгоритму наведене на рис. 2.8.

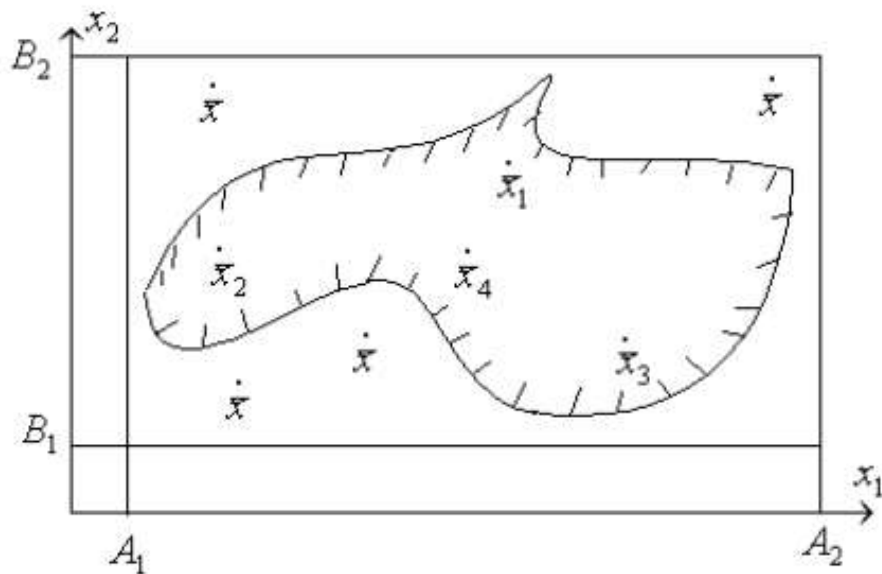

Рисунок 2.8 - Побудова обмежувального гіперпаралелепіпеда (A, B - межі паралелепіпеда)

Алгоритм найкращої проби з напрямним гіперквадратом. Відноситься до ненаправленого пошуку. Нехай є допустима область, в якій власне формується гіперквадрат. Значення функції обчислюються у випадково розкиданих точках x_1, ..., x_m, далі вибирається найкраща серед них. Потім треба сформувати новий гіперквадрат, на цій точці. Точка, в котрій досягнуто мінімум функції на якомусь k-му етапі, визначається як центр нового гіперквадрату вже на (k+1)-му етапі. Варто зауважити, що координати вершин гіперквадрату на (k+1)-му етапі можна розрахувати із формул (2.4) [15] :

$$a_i^{k+1} = x_i^{k+1} - \frac{b_i^k - a_i^k}{2}, \quad b_i^{k+1} = x_i^{k+1} + \frac{b_i^k - a_i^k}{2}, \tag{2.4}$$

На рис. 2.9 зображено пояснення до алгоритму.

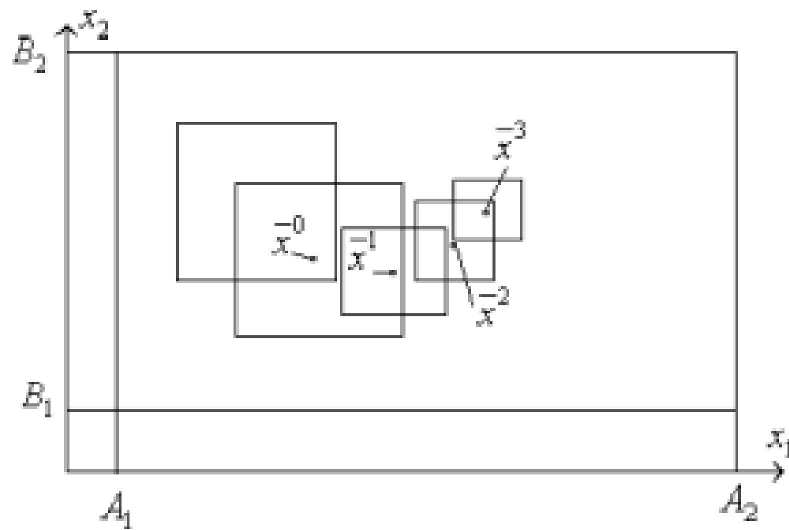

Рисунок 2.9 - До алгоритму з напрямним гіперквадратом

У подальшому вже у новому гіперквадраті виконуємо ту саму послідовність дій з випадковим розміщенням m точок, і т.д.

Властиво на 1-му етапі координати таких випадкових точок повинні задовольняти співвідношення $a_i^1 \le x_i \le b_i^1$, i=1,…,n та $x^1 = \arg\min_{j=1,m}\{f(x_j)\}$ - точка, котра має мінімальне значенням ЦФ.

При застосуванні алгоритму з навчанням сторони гіперквадрату можуть піддаватися решглюванню згідно до зміни параметра α за визначеним законом. Тоді координати вершин гіперквадрату на власне (k+1)-му етапі модна визначати із співвідношень (2.5) [15]:

$$a_i^{k+1} = x_i^{k+1} - \frac{b_i^k - a_i^k}{2\alpha}, \quad b_i^{k+1} = x_i^{k+1} + \frac{b_i^k - a_i^k}{2\alpha}. \tag{2.5}$$

Можна використовувати також алгоритми випадкового пошуку з напрямним гіперконусом для вирішення подібних завдань.

2.4 Висновки до другого розділу

У цьому розділі описано різного роду моделі, зокрема мультипроцесора CUDA, програмування технології CUDA в цілому та з використанням CPU і GPU. Проаналізовано їх характеричтики та особливості використання.

Для розв’язування завдань багатовимірної оптимізації проаналізовані функції Розенброка, Хіммельблау та Растригіна. Досліджено алгоритми випадкового пошуку (випадки направленого та ненаправленого пошуку).

3 ПРАКТИЧНА ЧАСТИНА

3.1 Процес встановлення CUDA для Microsoft Windows

Для застосування CUDA система потрібна мати:

– GPU із підтримкою CUDA;

– підтримувана версія Microsoft Windows;

– підтримувана версія Microsoft Visual Studio;

– NVIDIA CUDA Toolkit (доступний за адресою http://developer.nvidia.com/ cuda-downloads).

У наступних двох таблицях перераховані поточні підтримувані ОС та компілятори Windows.

Таблиця 3.1 – Підтримка ОС Windows у CUDA 8.0

Operating System	Native x86_64	Cross (x86_32 on x86_64)
Windows 10	YES	YES
Windows 8.1	YES	YES
Windows 7	YES	YES
Windows Server 2016	YES	NO
Windows Server 2012 R2	YES	NO
Windows Server 2008 R2 DEPRECATED	YES	YES

Таблиця 3.2 – Підтримка компілятора Windows у CUDA 8.0

Compiler	IDE	Native x86_64	Cross (x86_32 on x86_64)
Visual C++ 14.0	Visual Studio 2015	YES	NO
	Visual Studio Community 2015	YES	NO
Visual C++ 12.0	Visual Studio 2013	YES	YES
Visual C++ 11.0	Visual Studio 2012	YES	YES
Visual C++ 10.0 DEPRECATED	Visual Studio 2010	YES	YES

Підтримка x86_32 обмежена. Рідна розробка з використанням CUDA Toolkit на x86_32 не підтримується. Розгортання та виконання додатків CUDA на x86_32 як і раніше підтримується, але обмежено використанням графічних процесорів GeForce Щоб створити 32-бітові програми CUDA, використовуйте можливості крос-розробки CUDA Toolkit на x86_64 [16].

Підтримка розробки та запуску 32-розрядних програм x86 на x86_64 Windows обмежена використанням:

– графічні процесори GeForce;
– драйвер CUDA;
– час виконання CUDA (cudart);
– CUDA Math Library (math.h);
– компілятор CUDA C++ (nvcc);
– інструменти розробки CUDA.

Встановлення засобів розробки CUDA. Налаштування інструментів розробки CUDA у системі під Windows складається з кількох простих кроків:

– переконайтеся, що в системі встановлений GPU з підтримкою CUDA;
– завантажте NVIDIA CUDA Toolkit;
– встановіть NVIDIA CUDA Toolkit;
– перевірте, чи встановлене ПЗ працює правильно та взаємодіє з обладнанням.

З’ясувати, чи є GPU з підтримкою CUDA через розділ «Дисплеї» у Диспетчері пристроїв Windows. Саме тут відображається ім'я та модель відеокарти. Якщо карта NVIDIA, вказана у http://developer.nvidia.com/cuda-gpus, цей графічний процесор є CUDA-сумісним. Примітки до випуску інструментарію CUDA також містять список продуктів, що підтримуються.

Інструмент NVIDIA CUDA Toolkit доступний за адресою http://developer.nvidia.com/cuda-downloads. Виберіть платформу, яка використовується, і один із наступних форматів встановлення:

– мережеве встановлення: мінімальний варіант, який потім завантажує пакети, потрібні для встановлення. Завантажуються лише пакети, вибрані на етапі вибору встановлення. Цей варіант корисний для користувачів, які бажають мінімізувати час завантаження;

– повне встановлення: варінат, який містить усі компоненти CUDA Toolkit і не потребує подальшого завантаження. Корисний для систем, яким не вистачає доступу до мережі та для розгортання на підприємстві.

CUDA Toolkit встановлює драйвер CUDA та інструменти, необхідні для створення, складання та запуску програми CUDA, а також бібліотек, файлів заголовків, вихідного коду CUDA та інших ресурсів

Підтвердження завантаження може бути перевірено шляхом порівняння контрольної суми MD5, опублікованої за адресою http://developer.nvidia.com/ cuda-downloads/checksums, з інформацією про завантажений файл. Якщо якась із контрольних сум відрізняється, файл пошкоджений та потребує повторного завантаження [16].

Щоб обчислити контрольну суму MD5 завантаженого файлу, дотримуйтесь інструкцій на сторінці http://support.microsoft.com/kb/889768.

Встановлення ПЗ CUDA. Перед встановленням інструментарію ви повинні прочитати примітки до випуску, оскільки вони містять докладну інформацію про встановлення та функціональність ПЗ.

Примітка. Для роботи CUDA необхідно встановити драйвер та інструментарій. Якщо ви не встановили автономний драйвер, установіть драйвер із набору інструментів NVIDIA CUDA.

Примітка. Інсталяція може завершитися невдачею, якщо Windows Update розпочнеться після початку інсталяції. Дочекайтеся завершення оновлення Windows Update і повторіть спробу інсталяції.

Графічне встановлення. Встановіть ПЗ CUDA, виконавши встановлення CUDA і дотримуючись інструкцій на екрані. Наприклад, для встановлення лише

компонентів компілятора та драйвера: <PackageName>.exe -s compiler_8.0 Display.Driver

Вилучення та перевірка файлів вручну. Іноді буває доцільно отримувати чи інспектувати файли, що встановлюються безпосередньо, наприклад, при розгортанні або переглядати файли перед встановленням. Повний пакет встановлення можна витягти, використовуючи інструмент декомпресії, який підтримує метод стиснення LZMA, такий як 7-zip чи WinZip.

Після вилучення файли CUDA Toolkit будуть у папці CUDAToolkit та аналогічно для CUDA Samples та CUDA Visual Studio Integration. Усередині кожного каталогу є файли .dll та .nvi, котрі можна ігнорувати, оскільки вони не є частиною інсталяційних файлів.

Примітка. Доступ до файлів таким чином не встановлює жодних параметрів середовища, таких як змінні або інтеграція з Visual Studio. Це для розгортання лише на рівні підприємства.

Використовуйте відповідну модель драйвера. У Windows 7 та пізніших версіях ОС надає дві моделі драйверів, під якими може працювати драйвер NVIDIA:

– модель драйвера WDDM використовується для відображення пристроїв;

– режим Tesla Compute Cluster (TCC) Драйвер NVIDIA доступний для пристроїв без дисплея, таких як графічні процесори NVIDIA Tesla та графічні процесори GeForce GTX Titan. Він використовує модель драйвера Windows WDM.

Режим драйвера TCC надає ряд переваг для програм CUDA на графічних процесорах, які підтримують цей режим. Наприклад:

– усуває таймаути, які можуть виникати під час роботи під WDDM через механізм виявлення та відновлення часу очікування пристроїв відображення;

– надає змогу застосовувати CUDA з Windows Remote Desktop, що неможливе для WDDM-пристроїв;

– дозволяє використовувати CUDA з процесів, що працюють під керуванням Windows, що неможливо для WDDM-пристроїв;

– зменшує затримку запуску ядра CUDA;

– ввімкнений за замовчуванням на сучасних GPU NVIDIA Tesla. Щоб перевірити, який режим драйвера використовується та / або перемикати режими драйвера, використовуйте інструмент nvidia-smi, який входить до комплекту встановлення драйвера NVIDIA.

Примітка. Майте на увазі, що, коли режим TCC увімкнено для конкретного графічного процесора, цей GPU не можна використовувати як пристрій відображення.

Примітка. Графічні процесори NVIDIA GeForce (за винятком GPU GeForce (GTX Titan) не підтримують режим TCC.

Перевірити встановлення. Перш, ніж продовжити, важливо переконатися, що набір інструментів CUDA може правильно знайти та встановити зв'язок з обладнанням, котре підтримує CUDA. Для цього потрібно скомпілювати та запустити деякі з включених стандартних програм.

Виконання скомпільованих прикладів. Версію CUDA Toolkit можна перевірити, запустивши nvcc-V у вікні командного рядка. Зразки CUDA включають приклади програм у вихідній та скомпільованій формі. Щоб перевірити правильну конфігурацію апаратного забезпечення та ПЗ, настійно рекомендується запустити програму deviceQuery, розташовану в C:\ProgramData\NVIDIACorporation\CUDASamples\v8.0\bin\win64\Release

Передбачається, що використовувалася структура каталогів встановлення за замовчуванням. Якщо CUDA встановлено та налаштовано правильно, вихід повинен виглядати так, як показано на рис. 3.1.

Правильний зовнішній вигляд та вихідні рядки можуть відрізнятися у вашій системі. Важливими результатами є виявлення пристрою, що пристрій відповідає тому, що встановлений у вашій системі, і що тест пройшов.

```
deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 680"
  CUDA Driver Version / Runtime Version          6.0 / 6.0
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Clock rate:                                1059 MHz (1.06 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA Runtime Version = 6.0, NumDevs = 1, Device0 = GeForce GTX 680
Result = PASS
```

Рисунок 3.1 - Правильні результати прикладу DeviceQuery CUDA

Якщо встановлено пристрій з підтримкою CUDA та драйвер CUDA, але DeviceQuery повідомляє, що немає пристроїв з підтримкою CUDA, переконайтеся, що драйвери встановлено правильно.

Запуск програми bandwidthTest, розташованої в тому ж каталозі, що і DeviceQuery вище, гарантує правильний зв'язок системи та пристрою з підтримкою CUDA. Результат має виглядати на рис. 3.2.

```
C:\Windows\system32\cmd.exe
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Quadro K5000
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     5751.2

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     6342.7

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     131928.7

Result = PASS
```

Рисунок 3.2 – Правильні результати з використанням тесту bandwidthTest CUDA

Ім'я пристрою (другий рядок) та номери смуги пропускання варіюються від системи до системи. Важливим пунктом є рядок, який підтверджує, що пристрій CUDA знайдено, і рядок, який каже, що всі необхідні тести пройдено.

Якщо тести не пройдуть, переконайтеся, що у вашій системі є графічний процесор NVIDIA за допомогою CUDA і переконайтеся, що він правильно встановлений.

Щоб побачити графічне представлення того, що може зробити CUDA, запустіть зразок Particles, що виконується в C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\bin\win64\Release

Компіляція програм CUDA. Файли проектів у CUDA Samples були розроблені для створення простих однорядкових програм, що включають весь вихідний код. Для створення проектів Windows (для режиму випуску або налагодження) використовуйте надані файли рішень * .sln для Microsoft Visual Studio 2010, 2012 або 2013. Ви можете використовувати файли рішень, розміщені в кожному з каталогів прикладів у C:\ProgramData\ NVIDIACorporation\CUDASamples\v8.0\<category>\<sample_name>.

Або глобальні файли рішень. Samples * .sln, розташовані в C:\ProgramData \NVIDIA Corporation\CUDA Samples\v8.0.

Зразки CUDA організовані відповідно до <категорії>. Кожен зразок організований в одну з наступних папок: (0_Simple, 1_Utilities, 2_Graphics, 3_Imaging, 4_Finance, 5_Simulations, 6_Advanced, 7_CUDALibraries).

Компіляція прикладів проектів. Проект bandwidthTest - хороший зразковий проект для створення та запуску. Він розташований у каталозі NVIDIA Corporation \CUDA Samples\v8.0\1_Utilities\bandwidthTest.

Якщо вирішили використовувати місце встановлення за замовчуванням, висновок буде поміщений в CUDA Samples \ v8.0 \ bin \ win64 \ Release. Створіть програму, використовуючи відповідний файл рішення та запустіть виконуваний файл. Якщо все працює правильно, вихід має бути схожим на рисунок 3.2.

Створення налаштувань для нових проектів. При створенні нової програми CUDA файл проекту Visual Studio повинен бути налаштований для увімкнення налаштувань збірки CUDA. Для цього натисніть "Файл"> "Створити | Project ... NVIDIA->CUDA->, потім виберіть шаблон для версії CUDA Toolkit. Наприклад, вибір шаблону «CUDA 8.0 Runtime» дозволить налаштувати ваш проект для використання з CUDA 8.0 Toolkit. Новий проект є проектом C++ (.vcxproj), який попередньо налаштований для використання налаштувань збирання NVIDIA. Усі стандартні можливості проектів Visual Studio C++ будуть доступні.

Щоб вказати налаштоване розташування інструмента CUDA Toolkit, у розділі CUDA C/C++ виберіть «Загальний» та налаштуйте поле Custom Dir Toolkit CUDA за бажанням. Зверніть увагу, що вибраний набір інструментів повинен відповідати версії налаштувань збирання.

Створення налаштувань для існуючих проектів. При додаванні прискорення CUDA до існуючих програм відповідні файли проекту Visual Studio повинні бути оновлені для увімкнення налаштувань складання CUDA. Це можна зробити одним із наступних двох способів:

– відкрийте проект Visual Studio, клацніть правою кнопкою миші ім'я проекту та виберіть «Побудувати налаштування» ... потім виберіть версію інструментарію CUDA Toolkit, на яку ви хочете налаштувати націлення;

– крім того, ви можете налаштувати свій проект завжди на складання з самої останньою встановленою версією CUDA Toolkit. Спочатку додайте налаштування складання CUDA у свій проект, як зазначено вище. Потім клацніть правою кнопкою миші ім'я проекту та виберіть «Властивості». У розділі CUDA C/C++ виберіть «Загальний» та встановіть для параметра CUDA Toolkit Custom Dir значення $(CUDA_PATH). Зверніть увагу, що змінна середовища $ (CUDA_PATH) встановлюється інсталятором.

В той час, як Варіант 2 дозволить вашому проекту автоматично використовувати будь-яку нову версію CUDA Toolkit, яку ви можете встановити

в майбутньому вибір версії інструментарію явно, як і в Варіанті 1, часто краще на практиці, тому що, якщо в налаштуванні збирання додані нові параметри конфігурації CUDA

Якщо використовується змінна середовища $ (CUDA_PATH) для цільової версії CUDA Toolkit для побудови, і виконується інсталяція або видалення будь-якої версії CUDA Toolkit, ви повинні перевірити, що змінна середовища $ (CUDA_PATH) вказує на Правильний каталог інсталяції CUDA Toolkit для ваших цілей. Файли, що містять код CUDA, повинні бути позначені як CUDA C / C++. Це можна зробити, додавши файл, клацнувши правою кнопкою миші на проекті, до якого хочете додати файл, вибравши «Додати \ Новий елемент», вибравши «NVIDIA CUDA 8.0 \ Code \ CUDA C/C++», а потім вибрати файл, який бажаєте додати.

Технічні характеристики ПК, де проводилося дослідження на рис. 3.3.

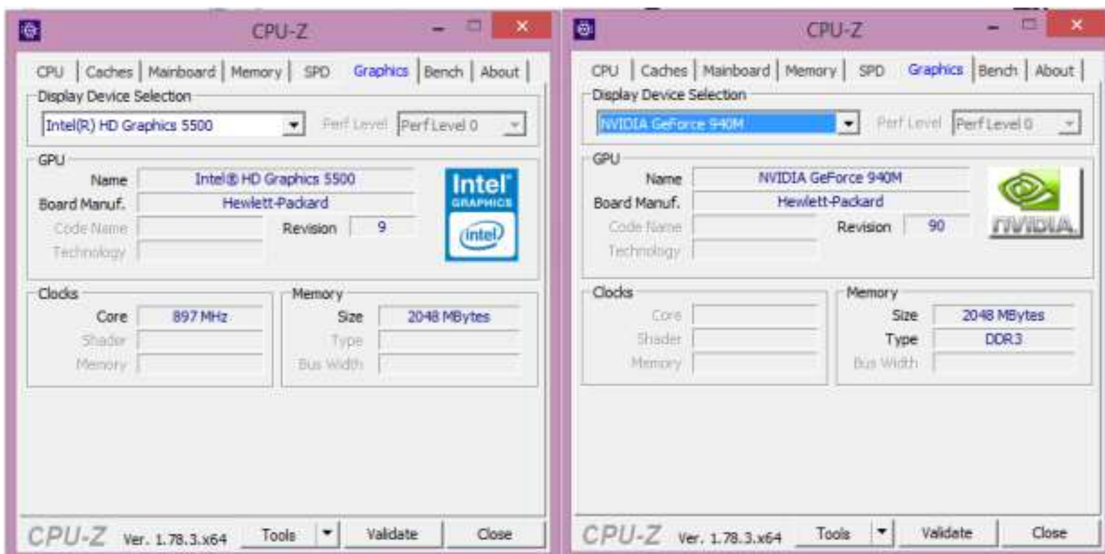

Рисунок 3.3 – Характеристика GPU

Для кращого розуміння ситуації варто навести опис відеокарти, карти застосовувалася для провення описаних в роботі досліджень. NVIDIA GeForce 940M - середньорівнева відеокарта з підтримкою DirectX 11, яка була представлена березні 2015 року. В основі вона має архітектуру Maxwell,

отримала 384 шейдерних блоків, виготовлена за нормами 28-нм техпроцесу і працює з 2 ГБ відеопам'яті DDR3. Енергоспоживання відеокарти становить близько 30 Вт, що забезпечує її використання в 13-14-дюймових лептопах та вище. Решту технічних характеристик відеокарти такі: архітектура – Maxwell; потоки – 384-unified; частота ядра – 1072-1176 (Boost) МГц; частота пам'яті – 2000 МГц; розрядність шини пам'яті – 64 Біт; тип пам'яті – DDR3; максимум пам'яті 4096 МБ; загальна пам'ять – ні; DirectX 11, Shader 5.0; технологія – 28 нм; додатково – GPU Boost 2.0, Optimus, PhysX, CUDA, GeForce Experience.

3.2 Структура програми

Основна частина коду міститься у файлі search.cu та представлений у вигляді набору функцій, що виконуються на пристрої (графічному прискорювачі). У лістингах 3.1, 3.2, 3.3, 3.4 наведено коди реалізації ядер.

Лістинг 3.1 – Ядро ініціалізації генератора випадкових чисел

```
__global__ void init(unsigned int seed, curandState_t* states)
{
 curand_init(seed blockIdx.x,
 0,&states[blockIdx.x]);
}
```

Лістинг 3.2 – Ядро рівномірного розподілу точок усередині поточних меж

```
	__device__
	void	distributionC(square*	borders,	coord*	point,
curandState_t* states)
	{
	// CurandState_t state;
	//curand_init(0, 0, 0, &state);
	int inst = threadIdx.x + blockIdx.x * blockDim.x;
	point[inst]->x1 = borders->X1.x1 + (borders->X1.x2 - borders-
>X1.x1) *
	curand(&states[blockIdx.x]);
	point[inst]->x2 = borders->X2.x1 + (borders->X2.x2 - borders-
>X2.x1) *
	curand(&states[blockIdx.x]);
	}
```

Лістинг 3.3 – Ядро обчислення алгоритму випадкового пошуку

```
__device__ void Ordinary_Search(square current_square, float *
F, int func_type, coord crds)
{
InitSyncWholeDevice(threadIdx.x + blockIdx.x*blockDim.x);
/* I */
SyncWholeDevice();
int inst = threadIdx.x + blockIdx.x * blockDim.x;
F[inst] = find_ordinary(func_type, crds);
/* II */
}
```

Лістинг 3.4 – Функція ініціалізації синхронізації виконання ядер по всьому пристрої

```
__device__ void InitSyncWholeDevice(const int index)
{
if (index == 0) count = 0;
if (threadIdx.x == 0) while (count! = 0);
__syncthreads();
}
```

У файлі rand_search.h міститься код функцій, що виконуються на CPU. Програмні коди наведено в додатку Б.

Основні етапи CUDA-програми.

1. Хост виділяє необхідну кількість пам'яті на пристрої: cudaMalloc(); Хост копіює дані зі своєї пам'яті в пам'ять пристрою: cudaMemcpy();
2. Хост стартує виконання певних ядер на пристрої. Параметри запуску ядер dim3 blockSize; dim3 gridSize; int threadNum; відповідно blockSize – розмір блоку, що задається; gridSize – розмір сітки; gridSize = dim3(instances, 1, 1); instances – кількість екземплярів обчислень функції, що запускаються blockSize = dim3(threadNum, 1, 1); - задання розмірності блоку threadNum = 1024; - максимально можлива кількість тредів на пристрої.

3. Пристрій виконує ядра.

__device__ void Ordinary_Search<<<gridSize, blockSize>>>(current_square, F, func_type, crds);

__device__ void Hypersquare_Search<<<gridSize, blockSize>>>(current_square, F, func_type, crds);

4. Хост копіює результати з пам'яті пристрою у пам'ять. Для найбільшої ефективності застосування GPU потрібно, щоб відношення часу, витраченого на роботу ядер, до часу, витраченого на виділення пам'яті та переміщення даних, було якнайбільше.

3.3 Результат розробки додатку

Для проведення чисельних експериментів була написана програма, в якій реалізовано описані раніше алгоритми оптимізації - випадковий пошук та алгоритм найкращої проби з напрямним гіперквадратом. У реалізації передбачено ПЗ коду на пристрої з підтримкою CUDA технології, CPU, та CPU з використанням технології OpenMP [23, 24, 27, 28].

Робочий вигляд головного вікна програми представлений рис 3.4.

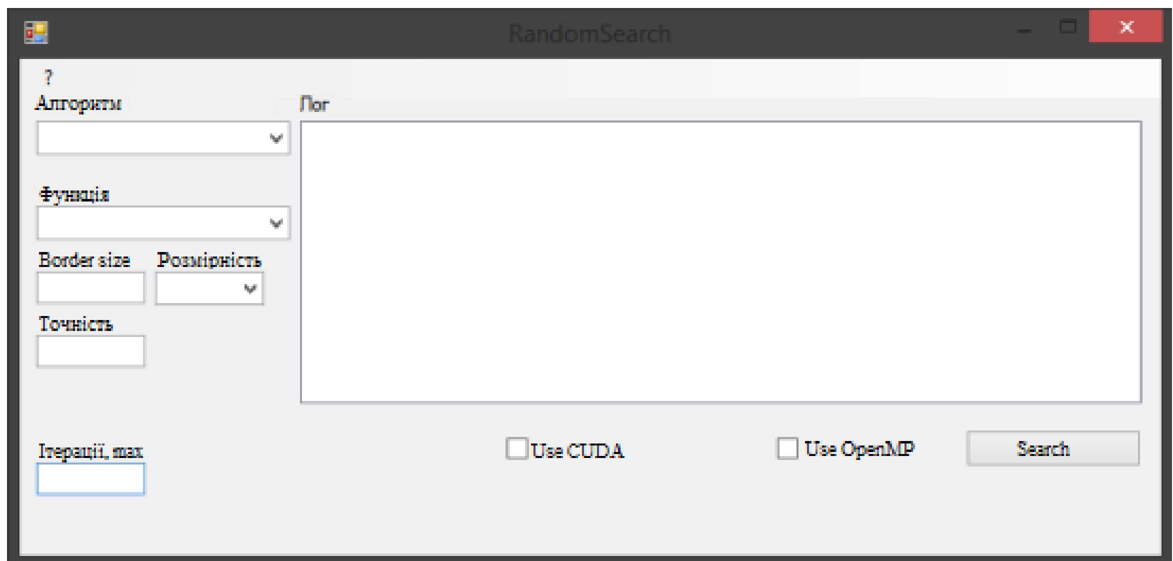

Рисунок 3.4 – Інтерфейс програми

Для роботи з програмою треба зі списку вибрати потрібний алгоритм та функцію, також задати початкові змінні, точність, кількість ітерацій та межі. Після цього вибираємо потрібне виконання програмного коду з підтримкою CUDA-технології та натискаємо кнопку Search.

Розглянемо роботу програми з допомогою функції Розенброка (рис. 3.5).

Виберемо алгоритм простого випадкового пошуку. Далі поставимо точність і число ітерацій. Задаємо технологію OpenMP. Після цього натискаємо кнопку Search та отримуємо результат.

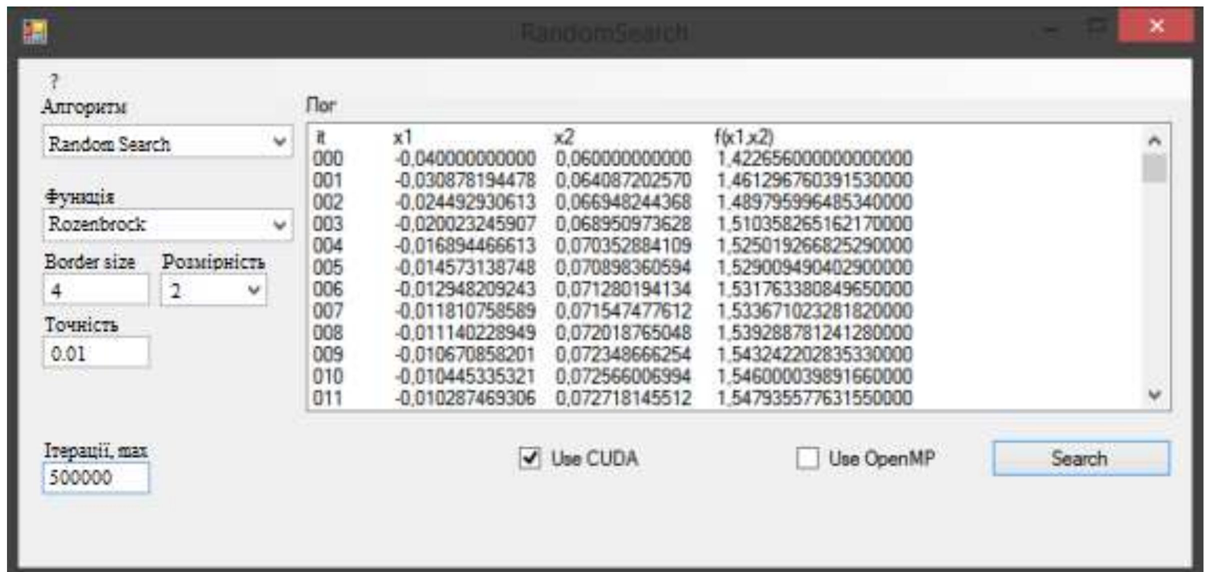

Рисунок 3.5 - Результат роботи програми для простого випадкового пошуку на функції Розенброка

В результаті отримали час знаходження екстремуму для функції Розенброка за методом простого випадкового пошуку при заданих параметрах: сторін квадрата = 4, у двовимірному просторі з точністю 0,01. Також задали кількість ітерацій 500 000.

3.3.1 Результати чисельних експериментів на функції Растригіна

Параметри: точність Eps = 0.01; обмеження кількості ітерацій 500 000.

Таблиця 3.3 – Розмірність = 2

Алгоритм	CPU, час в сек	CPU, OpenMP, час в сек	GPU, CUDA, час в сек
Випадковий пошук	43.003	38.540	12.090
Найкращої проби з напрямним гіперквадратом	24.112	29.010	11.370

Таблиця 3.4 – Розмірність = 3

Алгоритм	CPU, час в сек	CPU, OpenMP, час в сек	GPU, CUDA, час в сек
Випадковий пошук	57.124	49.000	14.010
Найкращої проби з напрямним гіперквадратом	34.178	32.149	13.730

Таблиця 3.5 – Розмірність = 4

Алгоритм	CPU, час в сек	CPU, OpenMP, час в сек	GPU, CUDA, час в сек
Випадковий пошук	84.0	62.000	30.007
Найкращої проби з напрямним гіперквадратом	52.0	49.001	24.031

Експеримент на функції Растригіна показав, що зі збільшенням розмірності продуктивність падає. В результаті чисельних експериментів, що проводилися на різному наборі функцій з розмірностями 2,3,4 можна зробити висновок про ефективність використання GPU для оптимізації задач. Однак варто відзначити, що спостерігався приріст продуктивності лише у випадку, коли співвідношення часу, витраченого на виділення та переміщення/роботу з даними до часу виконання ядер було якнайменше. Для більш чіткого розуміння представимо результати у графічному вигляді. На рис. 3.6 представлений результат функції Растригіна.

Рисунок 3.6 – Результат експерименту на функції Растригіна

З графіка видно, що при підвищенні розмірності, продуктивність усіх алгоритмів падає.

Також порівняємо на рис. 3.7 продуктивність виконуваної реалізації на графічному та центральному процесах.

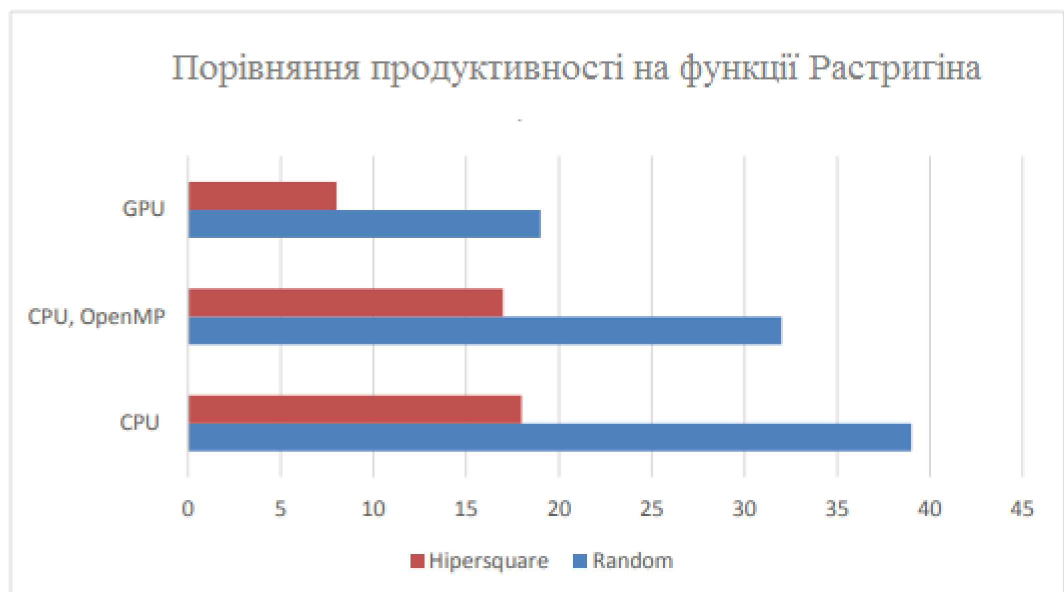

Рисунок 3.7 – Порівняння продуктивності на функції Растригіна

Як видно з графіка реалізація у вигляді виконуваного на графічному процесорі коду дозволяє отримати приріст у продуктивності – до 6 разів.

3.3.2 Результати чисельних експериментів на функції Розенброка

Параметри: точність Eps = 0.005; обмеження кількості ітерацій 500 000.

Таблиця 3.6 – Розмірність = 2

Алгоритм	CPU, час в сек	CPU, OpenMP, час в сек	GPU, CUDA, час в сек
Випадковий пошук	39.043	32.432	10.026
Найкращої проби з напрямним гіперквадратом	18.003	17.026	8.211

Таблиця 3.7 – Розмірність = 3

Алгоритм	CPU, час в сек	CPU, OpenMP, час в сек	GPU, CUDA, час в сек
Випадковий пошук	62.315	55.003	12.006
Найкращої проби з напрямним гіперквадратом	38.702	31.210	10.214

Таблиця 3.8 – Розмірність = 4

Алгоритм	CPU, час в сек	CPU, OpenMP, час в сек	GPU, CUDA, час в сек
Випадковий пошук	69.140	55.049	19.040
Найкращої проби з напрямним гіперквадратом	49.098	42.306	11.148

Експеримент на функції Розенброка показав, що зі збільшенням розмірності продуктивність падає. В результаті чисельних експериментів, що проводилися на різному наборі функцій з розмірностями 2,3,4 можна зробити висновок про ефективність використання GPU для оптимізації задач. Однак варто відзначити, що спостерігати приріст продуктивності лише у випадку, коли співвідношення часу, витраченого на виділення та переміщення/роботу з даними до часу виконання ядер було якнайменше. Для більш чіткого розуміння

представимо результати у графічному вигляді. На рис. 3.8 представлений результат функції Розенброка.

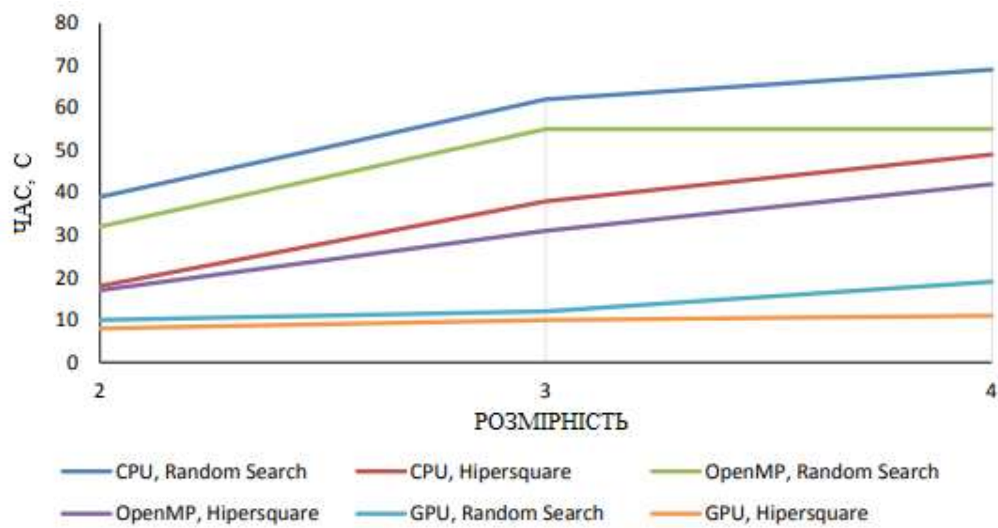

Рисунок 3.8 - Результат експерименту на функції Розенброка

З графіка видно, що при підвищенні розмірності, продуктивність усіх алгоритмів падає.

Також порівняємо на рис. 3.9 продуктивність виконуваної реалізації на графічному та центральному процесах.

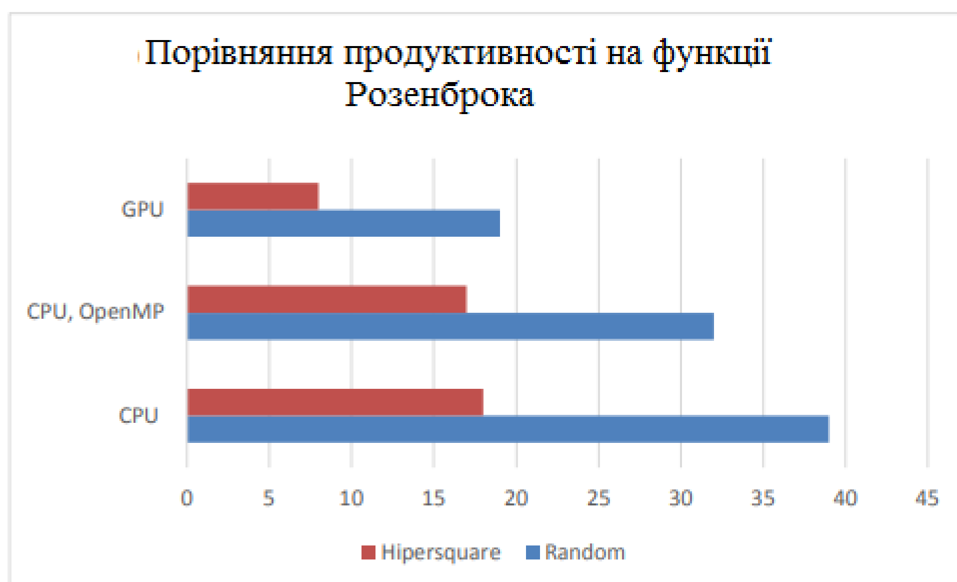

Рисунок 3.9 – Порівняння продуктивності на функції Розенброка

Як видно з графіка реалізація у вигляді виконуваного на графічному процесорі коду дозволяє отримати приріст у продуктивності, також як і функції Растригіна– до 6 разів.

3.4 Висновки до третього розділу

У вигляді посібника зі встановлення CUDA для Microsoft Windows описано особливості інсталяції спеціалізованого ПЗ. Навдено структуру розробленого програмного засобу для проведення досліджень в задачах багатовимірної оптимізації.

Результати проведених чисельних експериментів з використання для розв’яхування функцій Растригіна та Розенброка дозволяють стверджувати, що для обох випадків отримано приріст у продуктивності роботи до 6 разів.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Закордонний досвід організації охорони праці в IT-компаніях

Стан справ з охороною праці у світі стає все більш актуальною проблемою як для профспілок, так і для міждержавних структур, насамперед Міжнародної організації праці (МОП). МОП розглядає цю тему як частину своєї Програми гідної праці. Підвищена увага до проблем безпеки праці пояснюється в першу чергу тим, що з кожним роком, незважаючи на заходи, що вживаються, у різних країнах зростає рівень виробничого травматизму, у тому числі зі смертельними наслідками, і кількість профзахворювань.

На перший погляд, робота за комп'ютером при застосуванні паралельного програмування на основі технології CUDA в задачах багатовимірної оптимізації здається безпечною, але саме легковажність до неї може призвести до певних проблем у здоров'ї людини. Професія програміста та інших фахівців IT-технологій пов'язана з колосальним розумовим напруженням. Розробники – настільки захоплені люди, що навіть відволікаючись від роботи над проектом, продовжують думати про роботу. Нерідко відпочинком вони вважають паралельну заміну основної діяльності, наприклад, читання профільної літератури, верстку сайтів, вивчення нових мов програмування. Однак мозок не може до безкінечності приймати виключно корисну інформацію, яку розробник прагне направляти в русло особистісного та професійного зростання [35].

Зарубіжний досвід охорони праці при використанні новітніх інформаційних технологій та сучасного комп’ютерного обладнання передбачає з метою попередження наслідків монотонної праці, підвищення рівня рухової активності і покращення розумової працездатності фахівців IT-індустрії під час технологічних перерв участь у спеціальних облаштованих приміщеннях необхідним спортивним інвентарем та різними тренажерами відповідних фізичних вправ, індивідуальних тренінгових завдань відповідно до віку, статі та

категорії зорової роботи. Такий підхід дозволяє зняти надлишкове психофізіологічне перевантаження, підвищити працездатність центральної нервової системи, попередити перевтому зорового аналізатора. Показана ефективність проведення різноманітних за своєю спрямованістю вправ робітників цієї галузі (приблизно на 5-30%) [35].

Зараз багато IT-компаній обладнують свої офіси кімнатами відпочинку та лаунж-зонами, які забезпечують психофізіологічне розвантаження працівників. Адже окремим робочим столом з ноутбуком вже давно нікого не здивуєш. Тому, бажаючи підвищити продуктивність працівників, міжнародні компанії змагаються, перетворюючи нудні одноманітні офіси в креативні простори, де нові ідеї народжуються без титанічних зусиль. Наприклад, винахідники компанії Inventionland трудяться в казкових декораціях. Тут і гігантський гоночний трек, і пряниковий будиночок, і дуже реалістичний піратський корабель, на палубі якого розташувалися комп'ютерні столи, ніжками яких служать винні бочки. Робочі місця співробітників компанії Google в Цюріху нагадують гігантські вулики, а офіс шведського інтернет-провайдера Bahnhof розташувався в бомбосховищі часів холодної війни і походить на підземний притулок землян після глобальної катастрофи. А щоб співробітників не тягнуло додому, роботодавці створюють і можливість релаксувати, не відходячи від робочого місця, обладнавши басейни, ігрові кімнати та спортзали [35].

Корпорація Google піклується і про санітарно-гігієнічні умови праці та використовує систему очищення повітря, яка видаляє з атмосфери всі токсини та тяжкі домішки [36]. Також незвичайними на території офісу є спеціальні ізолюючі світло й звук капсули, де втомлений співробітник може відпочити, так само як і в спеціалізованих кімнатах у зоні відпочинку з приглушеним світлом та заспокійливою музикою, масажних кабінетах, ігрових кімнатах. Культивації активного відпочинку та спорту в Google відводиться особлива увага, оскільки одні з найпоширеніших хвороб програмістів пов’язані зі спиною. Саме тому в головному офісі міститься спортзал, в якому можуть одночасно займатися

велика частка співробітників, до того ж на території є плавальний басейн з черговим рятувальником, тому, фактично, людина може працювати прямо звідти (пам'ятаючи про дотримання правил техніки безпеки) [37]. Для переміщення по великій території Гуглмістечка «гуглівці» (так себе називають співробітники цієї компанії) користуються самокатами, які мають моторчики. Звичайно, усі ці зручності безкоштовні.

Намагається не відставати від Googleі корпорація Facebook. Так, дана фірма пропонує своїм співробітникам чудові умови праці: якісне різноманітне харчування, зручне апаратне забезпечення, важливою складовою якого є широкі монітори ПКз високою роздільною здатністю [38]. Особливістю Facebook є те, що співробітники не мають ані своїх кабінетів, ані офісних комірок – усі сидять разом, оскільки це стимулює комунікацію та обмін ідеями, підвищує продуктивність роботи. Загалом у компанії царює атмосфера позитиву. Усі співробітники привітні та посміхаються. На стінах можна побачити карикатури на топменеджерів, а робітники одягають футболки зі смішними написами [38]. 16 Співробітники тут, на відміну від Гугл, пересуваються по офісу не на самокатах, а на скейтах. У Facebook також наявна велика кількість тематичних та ігрових зон, де кожен може розслабитися та відпочити, і це все також абсолютно безкоштовно [38].

Поява та впровадження нових інформаційно-комунікаційних технологій зумовлює необхідність подальшого вдосконалення охорони праці фахівців ІТ-індустрії. З метою належного правового забезпечення необхідно розширити ти доповнити перелік основних професій комп'ютерної галузі у національному класифікаторі ДК-003-2010 [39], а також підготувати відповідний випуск у кваліфікаційному довіднику посад фахівців ІТ-індустрії, що сприятиме вирішенню питань їх соціального захисту, пенсійного забезпечення, атестації робочих місць основних професій за умовами праці на предмет подальших певних видів пільг та компенсацій за важкі шкідливі і небезпечні умови праці. Важливим напрямом стосовно визначення професійної придатності фахівців з

інформаційних технологій є проведення психофізіологічної експертизи відповідно до 5 статті [40].

ІТ-фахівці, як і будь-які інші працівники, повинні проходити навчання і перевірку знань з охорони праці або в навчальному центрі, або в самій організації. Якщо в ній є комісія з перевірки знань з охорони праці, атестованих в спеціалізованому навчальному центрі. Навчання охорони праці в організації проводять по самостійно розробленими програмами. Їх складають, спираючись на типові програми, а також з огляду на особливості галузі, в якій працює організація.

Робота з комп’ютерами нового покоління характеризується певним психофізіологічними перенавантаженнями, втомою зорового аналізатора, гіпокінезією, відсутність диференційованих норм праці при роботі з новою комп’ютерною технікою в залежності від віку, статі, категорії зорової роботи, режимів праці і відпочинку (протягом робочого дня, тижня, щорічного режиму відпусток). Все це потребує розробки нових нормативно-правових актів з регламентації праці та відпочинку фахівців ІТ-індустрії і стандартів підприємств, центрів комп’ютерної техніки, центрів інформаційних технологій, сучасних комп’ютерних класів. Особлива роль з точки зору збереження та відновлення здоров’я працюючих в комп’ютерні галузі належить попереднім та періодичним наглядам з подальшої психофізіологічної експертизи і встановленням професійної придатності при роботі з комп’ютерами нового покоління, який супроводжується виникненням певних факторів професійного ризику електротравматизму при їх ремонті та обслуговуванні. В цьому зв’язку необхідне запровадження експертизи на предмет безпечної експлуатації ПЕОМ, тобто офіційне підтвердження фактичних параметрів електробезпеки, їх відповідності вимогам нормативної документації фахівців, які проводять таку експертизу повинні пройти навчання і перевірку знань відповідно до вимог [41]. За результатами експертизи повинні прийматися рішення про відповідність ПЕОМ нормам безпеки, терміни чергової експертизи, оформлюються протоколи

вимірювань і випробувань, проведені у разі потреби розрахунки та експертний висновок. Для підвищення розумової працездатності то зорової роботи повинна здійснюватися ергономічна оптимізація в рамках системи «оператор-термінал», яка сприятиме результативній фізичній та інтелектуальний працездатності і відновленню психосоматичного здоров'я фахівців ІТ-індустрії.

Заслуговує на увагу зарубіжний досвід створення у приміщеннях та в зоні їх розміщення на територіях підприємств спеціальних візуальних комфортних умов та забезпечення вимог виробничої естетики, дотримання норм рівнів виробничого шуму та акустичної тиші за межами офісу. Також дуже важливим є використання в офісних приміщеннях та кабінетах психофізіологічного розвантаження функціональної музики, яка сприяє попередженню перевтоми і підтриманню необхідного рівня розумової працездатності фахівців комп'ютерної галузі. В цьому напрямі заслуговує на увагу створення при великих центрах інформаційних технологій кімнат (кабінетів) психофізіологічного розвантаження працівників галузі (на 5 місць) [35].

Всі наведені заходи щодо вдосконалення охорони праці фахівців ІТ-індустрії повинні контролюватися службою охорони праці та комісією з охорони праці підприємства. Особливе значення у соціальному захисті цієї категорії працівників належить прийняття комплексного договору, який може забезпечити фахівців додатковими пільгами та компенсаціями.

Таким чином, використання новітніх технологій вимагає від фахівців ІТ-індустрії додержання певних правил та вимог з точки зору безпеки праці, її нормування з урахуванням віку працюючих та загального інформаційного навантаження, розробки та впровадження індивідуальних, щотижневих та щорічних режимів праці та відпочинку, які сприятимуть профілактиці перевтомлення і підвищенню розумової працездатності працюючих. Особливу роль в цьому напрямі повинні відігравати ергономічні заходи стосовно створення робочих місць, оптимізації взаємодії людини в рамках системи «оператор-термінал». Всі ці вимоги повинні бути втілені у відповідних

нормативно-правових актах (стандартах підприємств), що регламентують різноманітні питання охорони та психології безпеки праці фахівців ІТ-індустрії.

4.2 Оцінка дії електромагнітного імпульсу (ЕМІ) на елементи комп'ютерної системи

У воєнний час при застосуванні ядерної зброї проти України на електронно-обчислювальне обладнання в першу чергу буде впливати ЕМІ ядерного вибуху у вигляді короткого імпульсу, який вражає головним чином електричну та електронну апаратуру [42].

ЕМІ виникають в основному в результаті взаємодії гамма-випромінювання з атомами навколишнього середовища. На утворення ЕМІ йде невелика кількість ядерної енергії, але він здатен викликати високі імпульси струмів та напруг в кабелях повітряних і підземних ліній зв'язку, сигналізації, управління, електропередачі, в антенах радіостанцій. Вплив ЕМІ може привести до згортання чутливих електронних та електричних елементів, зв'язаних з великими антенами чи відкритими дротами, а також до порушень в обчислювальних пристроях. Вплив ЕМІ необхідно враховувати для всіх електричних та електронних систем. Для найбільш важливих приладів треба використовувати засоби захисту і підвищувати їх стійкість до ЕМІ.

Особливістю ЕМІ, як вражаючого фактору є його здатність розповсюджуватись на десятки і сотні кілометрів в оточуючому середовищі. Тому ЕМІ може вплинути своєю дією на об'єкти, там де вибухова хвиля, світлове випромінювання, проникаюча радіація втрачають своє значення, як вражаючі фактори. При наземних та низьких повітряних вибухах в лініях зв'язку та електрозабезпечення виникають напруги, які можуть викликати пробій ізоляції провідників та кабелів відносно землі, пробій ізоляції елементів приладів підключених до повітряних і підземних ліній. Степінь враження залежить від наведеного імпульсу напруги чи струму і також електричної міцності обладнання.

Найбільш піддані впливу ЕМІ системи зв'язку, сигналізації, управління. Використані в цих системах кабелі та апаратура мають обмежену електричну міцність не більше 10кВ імпульсної напруги, тоді як наведені імпульси напруги від ЕМІ можуть перевищувати ці значення. Найбільш піддана впливу ЕМІ апаратура виконана на напівпровідниках та інтегральних схемах, працюючих на малих струмах і напругах, і значить відчутних до впливу зовнішніх електричних і магнітних кіл, в тому числі і елементи програмного засобу для управління процесом міграції віртуальних машин в обчислювальній хмарі. ЕМІ пробиває ізоляцію, спалює елементи електричних схем радіоапаратури, викликає коротке замикання в радіопристроях, іонізацію діелектриків, змінює або повністю стирає магнітний запис. Встановлено, що при дії ЕМІ на апаратуру найбільша напруга наводиться на вході. В транзисторах відбувається така залежність: чим більший коефіцієнт підсилення транзистора, тим менша його електрична міцність.

ЕМІ пошкоджує також резистори, викликає іскріння в їх міжконтактних з'єднаннях і деяких областях провідної поверхні. Найбільшу небезпеку ЕМІ представляє для апаратури, яка встановлена в особливо міцних спорудах, які витримують великі тиски ударної хвилі. В цих спорудах апаратура не виходить з ладу від механічних пошкоджень, але ЕМІ може вивести з ладу всю незахищену апаратуру системи зв'язку, сигналізації і керування. Найбільших значень досягають напруги, які наводяться між кабелем і землею. Напруженість електромагнітного поля всередині споруди в деяких випадках недостатня для того, щоб вивести з ладу апаратуру, але такі поля в змозі викликати короткочасний збій роботи радіотехнічних пристроїв.

Розглянемо можливі шляхи рішення задачі захисту від ЕМІ сервісу для адміністрування і обліку роботи автомобільної пар. Ідеальним захистом від ЕМІ виявилося б повне укриття приміщення, в якому розміщена радіоелектронна апаратура, металевим екраном. Водночас зрозуміло, що практично забезпечити такий захист у ряді випадків неможливо, тому що для роботи апаратури часто потрібно забезпечити її електричний зв'язок із

зовнішніми пристроями. Тому використовуються менш надійні засоби захисту, такі, як струмопровідні сітки, або плівкові покриття для вікон, щільникові металеві конструкції для повітрезабірників і вентиляційних отворів і контактні пружинні прокладки, розміщувані по периметру дверей і люків.

Більш складною технічною проблемою рахується захист від проникнення ЕМІ в апаратуру через різноманітні кабельні входи. Радикальним рішенням даної проблеми міг би стати перехід від електричних мереж зв'язку до практично не схильних до впливу ЕМІ волоконно-оптичних. Проте заміна напівпровідникових приладів у всьому спектрі виконуваних ними функцій електронно-оптичними пристроями можлива тільки у віддаленому майбутньому. Тому в даний час в якості засобів захисту кабельних входів найбільші широко використовуються фільтри, у тому числі волоконні, а також іскрові розрядники, металлоокисні варистори та ін. [43].

Всі ці засоби мають як переваги, так і недоліки. Так, ємнісно-індуктивні фільтри достатньо ефективні для захисту від ЕМІ малої інтенсивності, волоконні фільтри захищають у відносно вузькому діапазоні надвисоких частот. Іскрові розрядники мають значну інерційність й в основному придатні для захисту від перевантажень, що виникають під впливом напруг і струмів, що наводяться в обшивці літака, кожусі апаратури й оплетенні кабеля.

Металоокисні варистори є напівпровідниковими приладами, що різко підвищують свою провідність при високій напрузі. Проте, при застосуванні цих приладів у якості засобів захисту від ЕМІ варто враховувати їх недостатньо високу швидкодію і погіршення характеристик при кількаразовому впливі навантажень. Ці недоліки відсутні у високошвидкісних зенеровських діодах, дія яких заснована на різкій лавиноподібній зміні опору від високого значення практично до нуля, при перевищенні прикладеної до них напруги граничного розміру. Крім того на відміну від варисторів характеристики зенеровських діодів після багатократних впливів високих напруг і переключень режимів не погіршуються.

Найбільш раціональним підходом до проектування засобів захисту від ЕМІ кабельних входів є створення таких роз'ємів у конструкції яких передбачені спеціальні заходи, що забезпечують формування елементів фільтрів і установку вмонтованих зенеровських діодів. Подібне рішення сприяє одержанню дуже малих значень ємності й індуктивності, що необхідно для забезпечення захисту від імпульсів, що мають незначну тривалість і, отже, потужну високочастотну складову.

Складність рішення задачі захисту від ЕМІ і висока вартість розроблених для цих цілей засобів і методів змушують піти по шляху їхнього вибіркового застосування в особливо важливих системах зброї і військової техніки. Такий же шлях обраний і для захисту систем, що мають велику протяжність, керування і зв'язку. Проте основним методом рішення даної проблеми спеціалісти вважають створення так званих розподілених мереж зв'язку.

4.3 Висновки до четвертого розділу

В цьому розділі розглянуто важливі питання охорони праці та безпеки в надзвичайних ситуаціях, зокрема закордонний досвід організації охорони праці в ІТ-компаніях. Також проведено оцінку дії ЕМІ на елементи комп’ютерної системи.

ВИСНОВКИ

В кваліфікаційній роботі за допомогою застосування паралельних обчислень максимально пришвидшено процес отримання екстремуму в завданнях багатовимірної оптимізації. Показано переваги використання вищезгаданої технології в порівнянні з технологією послідовних обчислень.

Основні результати дослідження:

– проаналізовано основні характеристики новітніх суперкомп'ютерів;

– описано особливості використання технології паралельних обчислень з метою покращення продуктивності ПЗ;

– проведено порівняльний аналіз технологій послідовних та паралельних обчислень;

– проаналізовані спеціальні функції, які використовуються для розв’язування задач багатовимірної оптимізації;

– здійснено дослідження алгоритми випадкового пошуку;

– розроблено додаток для знаходження мінімуму функцій Розенброка і Растригіна за алгоритмом простого випадкового пошуку та алгоритмом найкращої проби з направляючим гіперквадратом.

При тестуванні програмного програмного засобу не було виявлено проблем із зовнішнім виглядом, відображенням його елементів, а також роботи загалом.

Розроблений програмний продукт:

– має зручний інтерфейс для роботи з даними;

– скоротить час обробки даних значних розмірностей;

– має максимальну ефективність у завданнях, які не потребують інтенсивного звернення до пам'яті.

Отримані результати виконаних чисельних експериментів із застосуванням функцій Растригіна та Розенброка показують, що у обидвох випадках отримано збільшення продуктивності роботи практично у 6 разів.

ПЕРЕЛІК ДЖЕРЕЛ

1. Рейтинг суперкомп'ютерів TOP50 [Електронний ресурс] – Режим доступу: http://www.top500.org/ (дата звертання 20.01.2022).
2. Архітектура Nvidia Kepler [Електронний ресурс] – Режим доступу: http://www.nvidia.com/content/ PDF/kepler/NVIDIAKepler-GK110- Architecture-Whitepaper.pdf/ (дата звертання: 13.02.2022)
3. Что такое CUDA? [Електронний ресурс] – Режим доступу: www.nvidia.ru/object/what_is_cuda_new_ru.html/(дата звертання: 19.02. 2022)
4. Знакомство с NVIDIA CUDA. [Електронний ресурс] – Режим доступу: www.render.ru/books/show_book.php?book_id=840/(дата звертання: 03.01.2022)
5. Информация о CUDA. [Електронний ресурс] – Режим доступу: ru.wikipedia.org/wiki/CUDA/(дата звертання: 11.01.2022)
6. Боресков А.В. , Харламов А.А. Основы работы с технологией CUDA, Изд. ДМК Пресс. – Москва, 2010г. – 234 стр.
7. Стандарт OpenCL. [Електронний ресурс] – Режим доступу: http://www.khronos.org/opencl/ (дата звертання: 17.02.2022).
8. Параллельные вычисления с CUDA. [Електронний ресурс] – Режим доступу: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html/ (дата звертання: 23.01.2022)
9. Технология AMD STREAM. [Електронний ресурс] – Режим доступу: http://www.amd.com/ruru/innovations/software-technologies/firepro-graphics/stream (дата звертання: 26.01.2022).
10. Растригин Л.А. Стохастические методы поиска. – М.: Наука, 1968. – 376 с.
11. Растригин Л.А. Адаптация сложных систем. – Рига, Зинатне, 1981. – 376 с.

12. Воронка А.О. Модель пам'яті технології CUDA // Інформаційні моделі, системи та технології: Праці IX наук.-техн. конф. (08-09 грудня 2021 р.), Тернопіль, 2021. – С. 36.

13. Теория и применение случайного поиска / под ред. Л.А. Растригина. – Рига: Зинатне, 1969. – 305 с.

14. Ермаков С.М., Михайлов Г.А. Курс статистического моделирования. – М.: Наука, 1976. – 320 с.

15. Попов Ю.Д., Тюптя В.І., Шевченко В.І. Методи оптимізації. − Київ: КНУ ім. Тараса Шевченка, 2003.−215 с.

16. Установка на Windows - Режим доступу: http://edu.chpc.ru/cuda/mainse5.html/ (дата звертання: 14.03.2022).

17. Сандерс Д., Кэндрот Эю. NVIDIA CUDA в примерах: введение в программирование графических процессоров – ДМК Пресс -2011.

18. Параллельные вычисления на GPU. Архитектура и программная модель CUDA : учеб. пособ. / А. В. Боресков и др. ; предисл. В. А. Садовничий. – Москва : Изд-во Московского ун-та, 2012. – 336 с. – (Серия «Суперкомпьютерное образование»).

19. Семеренко В. П. Технології паралельних обчислень : навчальний посібник. – Вінниця : ВНТУ, 2018. – 104 с.

20. Эхтер Ш., Робертс Дж. Многоядерное программирование. – СПб. : Питер, 2010. – 536 с.

21. Боресков А.В., Марковский Н.Д. Параллельные вычисления на GPU. Архитектура и программная модель CUDA. – М. : Изд-во МГУ, 2012. – 336 с.

22. Антонов А. С. Параллельное программирование с использованием технологии OpenMP : учебное пособие. – М. : Изд-во МГУ, 2009. – 77 с.

23. Хьюз К., Хьюз Т. Параллельное и распределенное программирование на C++. – М. : Вильямс, 2004. – 672 с.

24. Антонов А. С. Параллельное программирование с использованием технологии MPI : учебное пособие. – М. : Изд-во МГУ, 2004. – 71 с

25. Минайленко Р.М. Паралельні та розподілені обчислення: навч. посіб. — Кропивницький: Видавець Лисенко В. Ф., 2021. — 153 с.

26. Гергель В.П. Теория и практика параллельных вычислений: учебное пособие. – М.:ИНТУИТ, 2016. – 501 с.

27. Уильямс Э. Параллельное программирование на С++ в действии. Практика разработки многопоточных программ. – М.: ДМК Пресс, 2012. – 672 с:

28. Сайт Української команди розподілених обчислень. [Електронний ресурс] – Режим доступу: http://distributed.org.ua/ (дата звертання: 26.01.2022).

29. Шимчук Г.В., Маєвський О.В., Назаревич О.Б., Стадник М.А. Грід-системи та технології хмарних обчислень (конспект лекцій для студентів освітніх рівнів «бакалавр», «магістр»). - Тернопіль : Тернопільський національний технічний університет імені Івана Пулюя, 2016. – 340 с.

30. Rosenbrock, H. H. An automatic method for finding the greatest or least value of a function,The Computer Journal T. 3? 1960 – p. 175-184.

31. Luebke D., Harris M., Kruger J. GPGPU: General Purpose Computation On Graphics Hardware - SIGGRAPH, 2005. – 277 с.

32. Chang Dar-Jen Hierarchical Clustering with CUDA/GPU / Dar-Jen Chang, Mehmed M. Kantardzic, Ming Ouyang // ISCA PDCCS. – 2009. – P. 7–12.

33. Yang Chao-Tung Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters / Chao-Tung Yang, Chih-Lin Huang, Cheng-Fang Lin // Computer physics communications. – 2011. – No. 1. – P. 266–269.

34. Pllana Sabri. Programming multicore and many-core computing systems/ Sabri Pllana, Fatos Xhafa.Wiley, 2017. – 528 p.

35. Сьогодні UA [Електронний ресурс] – Режим доступу: https://www.segodnya.ua/lifestyle/fun/pochti-kak-u-googlechemudivlyayut-ofisy-ukrainskih-it-kompaniy--764025.html (дата звертання 07.05.2022).

36. Як працюють в Google? Умови, в яких хочеться трудитися. [Електронний ресурс] – Режим доступу: http://www.clevers.com.ua/articles-cleveradvertising-agency/success-stories/245-google2 (дата звертання 07.05.2022).

37. Офіс мрії: Робота в компанії Google. [Електронний ресурс] – Режим доступу:http://bigpicture.ru/?p=187580 (дата звертання 07.05.2022).

38. Офіс Facebook: Репортаж із RMA SiliconTrip. [Електронний ресурс] – Режим доступу:https://habrahabr.ru/company/rma/blog/103800/ (дата звертання 08.05.2022).

39. Класифікатор професій ДК 003:2010. [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/rada/show/va327609-10 - (дата звертання 07.05.2022).

40. Закон України «Про охорону праці». [Електронний ресурс] – Режим доступу: https://zakon.rada.gov.ua/laws/show/2694-12 - (дата звертання 06.05.2022).

41. ДНАОП 0.00-8.20-99. Порядок проведення експертизи електроустановок споживачів/ [Електронний ресурс] – Режим доступу: https://dnaop.com/html/43255/doc-%D0%94%D0%9D%D0%90%D0%9E%D0 %9F _ 0.00-8.20-99 - (дата звертання 07.05.2022).

42. Зеркалов Д.В. Безпека життєдіяльності та основи охорони праці. Навч. посібник. К.: «Основа». 2016. 267 с.

43. Сакевич В.Ф., Поліщук О.В. Цивільна оборона. Теоретичні основи. Навч. посібник. — Вінниця : ВНТУ, — 2009. — 136 с.

ДОДАТКИ

ДОДАТОК А
Тези конференції

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

МАТЕРІАЛИ

IX НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

8–9 грудня 2021 року

ТЕРНОПІЛЬ
2021

Р. Боднар, І. Кормило, О. Задолинний, Т. Маєвський
СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ОПРАЦЮВАННЯ ДАНИХ В УМОВАХ ПАНДЕМІЇ
R. Bodnar, I. Kormylo, O. Zadolynnyi, T. Maievskyi
ARTIFICIAL INTELLIGENCE SYSTEMS FOR DATA PROCESSING IN A PANDEMIC CONDITION 29

Р. Боднар, І. Кормило, О. Задолинний, Т. Маєвський
ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ ДЛЯ ОПРАЦЮВАННЯ ДАНИХ В УМОВАХ ПАНДЕМІЇ
R. Bodnar, I. Kormylo, O. Zadolynnyi, T. Maievskyi
CONVOLUTIONAL NEURAL NETWORKS FOR DATA PROCESSING IN A PANDEMIC CONDITION 31

А. Войтович
ДОСЛІДЖЕННЯ АКТУАЛЬНОСТІ УПРАВЛІННЯ ПРОЕКТАМИ У СФЕРІ ОБСЛУГОВУВАННЯ 33

Р. Волощак
РОЗРОБКА ІНФОМАЦІЙНОЇ СИСТЕМИ ДЛЯ ЗБОРУ ТА АНАЛІЗУ ПОКАЗНИКІВ ЛІЧИЛЬНИКА ЕЛЕКРОЕНЕРГІЇ З ВИКОРИСТАННЯМ ARDUINO
R. Voloshchak
DEVELOPMENT OF INFORMATION SYSTEM FOR COLLECTION AND ANALYSIS OF ELECTRICITY METER INDICATORS USING ARDUINO 34

О. Волинюк, Нвамене Крістофер Чізоба, С. Лупенко
ПРОТОТИП ІНФОРМАЦІЙНОЇ ОНТООРІЄНТОВАНОЇ ДОВІДКОВОЇ СИСТЕМИ ПРЕДМЕТНОЇ ОБЛАСТІ «МОДЕЛЮВАННЯ ТА ОПРАЦЮВАННЯ ЦИКЛІЧНИХ СИГНАЛІВ»
O. Volynyk, Nnamene Christopher Chizoba, S. Lupenko
PROTOTYPE OF ONTO-ORIENTED INFORMATION HELP SYSTEM IN SUBJECT AREA «MODELING AND PROCESSING CYCLIC SIGNALS» 35

А. Воронка
МОДЕЛЬ ПАМ'ЯТІ ТЕХНОЛОГІЇ CUDA
A. Voronka
CUDA TECHNOLOGY MEMORY MODEL 36

А. Гайдар, В. Готович
РОЗРОБКА ПЛАТФОРМИ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ШЛЯХОМ ТЕСТУВАННЯ
A. Haidar, V. Hotovych
DEVELOPMENT OF PLATFORMS FOR VERIFICATION OF KNOWLEDGE THROUGH TESTING 37

Ю. Горбуляк
ОГЛЯД МЕТОДІВ МАЙНІНГУ WEB-КОНТЕНТУ
Yu. Horbuliak
SURVEY OF THE METHODS OF WEB-CONTENT MINING 38

Є. Гоцко, Г. Козбур
ВИКОРИСТАННЯ ВЕЛИКИХ ДАНИХ В РОЗУМНОМУ МІСТІ
E. Hotsko, H. Kozbur
USING BIG DATA IN A SMART CITY 39

УДК 004.42
А.О. Воронка
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

МОДЕЛЬ ПАМ'ЯТІ ТЕХНОЛОГІЇ CUDA

UDC 004.42
A.O. Voronka

CUDA TECHNOLOGY MEMORY MODEL

Можливість вільного доступу до пам'яті з побайтовою адресацією, є надзвичайно важливим моментом. Самі потоки CUDA можуть адресуватися до даних з різних просторів пам'яті в один і той самий час. Будь-який потік має приватну локальну пам'ять. Кожен блок потоку володіє загальною пам'яттю, котра видима всім потокам блоку і з тим самим часом життя, як і блок. У всіх потоків є доступ до однієї й тієї ж глобальної пам'яті. Є також два додаткові простори для зчитування, котрі доступні для всіх потоків, а саме постійної та текстурної пам'яті. Пам'ять текстур також пропонує ряд варіантів звертань та фільтрацію даних для окремих форматів.

Глобальна пам'ять. Є у пам'яті пристрою, а вона – через 32-, 64- або 128-байтові транзакції. Вони повинні бути вирівнені: тільки 32-, 64- або 128-байтові сегменти пам'яті пристрою, які вирівнені за тим, чия власне перша адреса кратна їх розміру, можуть бути прочитані чи записані в пам'ять угоди. Скільки транзакцій та пропускної здатності зрештою потрібно, залежить від обчислювальної здатності пристрою. Compute Capability 2.x, 3.x, 5.x та 6.x дають більш детальну інформацію про те, як обробляються глобальні звернення до пам'яті для різних обчислювальних можливостей.

Локальна пам'ять. Доступ до неї відбувається лише для деяких автоматичних змінних. Автоматичними змінними, які компілятор може розмістити в такій пам'яті, є: масиви, для котрих він не може визначити, що їх індексують зі сталими одиницями, великі структури чи масиви, які споживатимуть занадто багато простору для реєстрації, будь-яка змінна, якщо власне ядро бере більше регістрів, чим доступно (це також відомо як розлиття регістрів).

Загальна пам'ять. Так як вона вмонтована в чіп, тому володіє вищою пропускною здатністю та значно меншою затримкою, ніж локальна чи глобальна пам'ять. Для одержання високої пропускної здатності пам'ять поділяється на однакові за розміром модулі однакового розміру (банки), до яких можна одночасно звертатися. Таким чином, будь-який запит на читання або запис в пам'ять, що складається з n адрес, які потрапляють у n різних банків пам'яті, може обслуговуватися одночасно, що дає загальну пропускну здатність, яка в n разів перевищує пропускну здатність одного модуля.

Постійна пам'ять. Є в пам'яті пристрою та кешується у постійному кеші, вказаному в Compute Capability 2.x. Потім запит розбивається на стільки окремих запитів, що у вихідному запиті є різні адреси пам'яті, що зменшує пропускну здатність на коефіцієнт, що дорівнює кількості окремих запитів. Потім одержані запити обслуговуються за пропускної здатності постійного кешу у випадку попадання в кеш або за пропускної здатності власне пам'яті пристрою в іншому разі.

Текстурна пам'ять. Пам'ять текстури та поверхні є у пам'яті пристрої та кешуються в кеші текстури, тому при їх зчитуванні стоїть одна пам'ять, що зчитується з пам'яті пристрою тільки при пропуску кешу, інакше це просто коштує одного читання з кешу текстур. Кеш текстури оптимізовано для 2D просторової локальності, тому потоки одного і того ж детектора, які читають текстурні або поверхневі адреси, що знаходяться близько один до одного в 2D, досягнуть найкращої продуктивності. Крім того, він призначений для потокового завантаження із постійною затримкою. Кеш- кеш зменшує потребу в пропускній здатності, але не покриває затримку.

ДОДАТОК Б

1. Програмний код функції випадкового пошуку

```
void compute_ordinary(square current_square, int func_type, int iterations)
{
float step;
const int instances = 1500;
float F[instances] = {0}, F_min, prev_F;
int it;
float eps = 0.05;
int instance;
coord point;
coord init_crds;
coord current_crds;
coord prev_crds;
size_t points_num;
clock_t time;
time = clock();
init_crds = distribution(current_square);
F[0] = function(func_type, init_crds);
printf("%.0d\t", it);
printf("%.0f (%f, %f)\t", F[0], current_crds.x1, current_crds.x2);
F_min = F [0];
prev_F = F[0];
current_crds = init_crds;
it = 0;
while (it<= iterations && std::abs(0.0 - F_min) >= eps)
{
prev_F = F[instiance];
prev_crds = current_crds;
for (size_t i = 0; i < instances; i++)
{
current_crds = distribution(current_square);
F[i] = function(func_type, current_crds);
if (F_min > F[i])
{  F_min = F[i];  }
}
printf("%.
log_("log_file.txt", F_min,it, current_crds);
it++;
}
time = clock() - time;
float temp = (float)time/CLOCKS_PER_SEC/60;
printf("\n%f\r", time);
//log_time("log_file.txt", temp);
//cout << "total time in cpu mode\t" << (double)time / CLOCKS_PER_SEC / 60 << "sec" << endl;
}
```

2. Програмний код функції з напрямним гіперквадратом

```
void compute_hypersquare(square current_square, int func_type, int iterations)
{
float init_X1, init_X2;, step;
const int instances = 1500;
float F[instances] = {0}, F_min, prev_F;
int it;
float eps = 0.05;
int instance;
float alpha = 1;
coord point;
coord init_crds;
coord current_crds;
coord prev_crds;
square new_square;
size_t points_num;
init_crds = distribution(current_square);
F[0] = function(func_type, init_crds);
F_min = F [0];    prev_F = F[0];
clock_t time;
time = clock();
current_crds = init_crds;
it = 0;
new_square = current_square;
while (it <= iterations && std::abs(0.0 - F_min) >= eps)
{
prev_F = F[i];
prev_crds = current_crds;
current_square = new_square;
for (size_t i = 0; i < instances; i++)
{
current_crds = distribution(current_square);
F[i] = function(func_type, current_crds);
if (F_min > F[i])
{ F_min = F[i]; }
}
printf("%.0d, %f (%f, %f) \r", it, F_min, current_crds.x1, current_crds.x2);
//("log_file.txt", F_min, it, current_crds);
get_new_hypersquare(current_square, new_square, current_crds, alpha);
it++;
}
time = clock() - time;
float temp = (float)time/CLOCKS_PER_SEC/60;
printf("\n%f\r", time);
//log_time("log_file.txt", temp);
}
```