Київський національний університет імені Тараса Шевченка
Факультет кібернетики та комп'ютерних наук
Кафедра обчислювальної математики

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА
на тему:
Гібридні ітераційні алгоритми розв'язання дискретних задач для еліптичних рівнянь

Виконав студент 2-го курсу магістратури

Оленченко Ілля Андрійович

Науковий керівник:
доктор фізико-математичних наук
професор, член-кореспондент НАН України
Хіміч Олександр Миколайович

Роботу заслухано на засіданні кафедри обчислювальної математики та рекомендовано до захисту. Протокол № 11 від 18 травня 2017 року.

Завідувач кафедри обчислювальної математики проф. Ляшко С. І. _____

Київ - 2017

ЗМІСТ

Список позначень, скорочень, термінів

- **GPGPU** - (***G**eneral **P**urpose computing for **G**raphics **P**rocessing **U**nits*) - техніка використування графічного процесора відеокартки для загальних обчислень, які зазвичай проводяться на центральному процесорі
- **GPU** - (***G**raphics **P**rocessing **U**nit*) - графічний процесор
- **CPU** - (***C**entral **P**rocessing **U**nit*) - центральний процесор
- **CUDA** - (***C**ompute **U**nified **D**evice **A**rchitecture*) - програмно - апаратна архітектура розроблена компанією NVIDIA. Дозволяє використовувати відеокартки для загальних обчислювальних потреб
- **ATI Stream Technology** - API з відкритим кодом, який дає можливості розробникам використовувати обчислювальні можливості GPU
- **SISD**, **SIMD**, **MISD**, **MIMD** - (***S**ingle / **M**ultiple **I**nstructions **S**ingle / **M**ultiple **D**ata*) - позначення загальної класифікації архітектур ЕОМ за ознакою присутності паралелізма в потоках команд та даних
- **ALU** - (***A**rithmetic and **L**ogic **U**nit*) - Арифметико - логічний пристрій - блок процесора під керуванням пристрою керування. Використовується для арифметичних або логічних перетворень
- **Flop/s** - (***Fl**oat **Op**erations per **S**econd*) - кількість операцій з плаваючею точкою за секунду

Вступ

В сучасний період розвитку обчислювальної техніки актуальність числових методів, що дозволяють розв’язувати широкий клас задач за допомогою ЕОМ, продовжує зростати. Зростаюча потужність персональних, серверних, кластерних навіть мобільних ЕОМ вже набула неймовірного рівня а з часом лише збільшується.

Зважаючи на те, що центральні процесори останнім часом не можуть збільшувати тактову частоту - потужності набувають інші опції оновлення, а саме оновлення архітектур, збільшення ядер, покращення співпроцесорів наприклад графічний процесор. Звичайно зі збільшенням кількості таких компонентів та їх якості особливо гостро постає питання оптимального використання усього доступного арсеналу з архітектури комп’ютера [3, 4].

За останні 10 років розвиток графічних процесорів сягнув значно більших висот ніж центральний процесор. Зважаючи на це було запропоновано використати потенціал “графічного” обчислення для загальних потреб.

Як і більшість систем найбільша потужність досягається при використанні у сукупності із центральним процесором, що з неймовірною швидкістю відкриває все нові можливості ЕОМ. З одного боку, як і раніше продовжується приріст продуктивності ЕОМ за рахунок збільшення кількості ядер процесору. З іншого боку, гібридні системи стають більш популярні, що зумовлено використанням елементів принципово нової архітектури [5].

Актуальність роботи зумовлена нестримним рухом технологій з плином часу. Таким чином на вже розв’язані задачі можна подивитися під іншим кутом, а саме використання гібридних комп’ютерів для розв’язання диференціальних рівнянь. Значна частина прикладних задач зводиться до математичних моделей, які описуються системами лінійних алгебраїчних рівнянь (СЛАР) [9].

Використання комп’ютера встановлює задачу раціонального використання ресурсів, адже навіть в наш час ресурсами є скінчені величини,

які потрібно використовува оптимально. А тому потрібно використовувати усі можливі архітектури для розв’язання задач.

Зараз ми маємо багатопроцесорні системи, які можуть виконувати декілька процесів одночасно, або майже одночасно. Проте архітектура CPU достатньо обмежена. Для збільшення кількості процесорів потрібно або збільшувати сам процесор фізично або зменшувати розміри процесору при цьму зберігаючи поставлену потужність. Обидва варіанти розробляються компаніямі гігантами (Intel, AMD) та мають границі у які мі вже починаємо впиратися. А найпростіший з першої точки зору варінт збільшення кількості обїєднаних CPU та побудови суперкомп’ютера буде коштувати значної суми. Другий за потужністю китайський суперкомп’ютер Tianhe-2 з 3,120,000 ядрами та обчислювальною потужністю у 54,902.4 TFlop/s коштує близько 390 мільйонів американських долларів.

З іншого боку архітектура GPU початково створена для великої кількості обчислювальних елементів для паралельного опрацювання даних. Звичайно уперше таке завдання перед ЕОМ поставив попит на графічний контент та необхідністю його генерування без затримок основного процесору. Принциповим є кількість процесорів та кількість потоків (ниток) які можуть виконувати операції одночасно.

Звичайно маючи можливість будувати гібридні архітектури і використовувати лише одну з них не є оптимальним, тож є сенс для розв’язання такої задачі використовувати обидва процесора при умові, що для задачі є можливість залучити GPU [5].

Протягом останніх десятиліть на основі розроблених алгоритмів було створено низку бібліотек, до яких увійшли програми для розв’язування СЛАР: SparseBLAS, SparsPak, SSP, BoeingLibrary, BellLaboratories, IMSL, NAG та інші. Серед програмних засобів, призначених для паралельних комп’ютерів, ефективні реалізації алгоритмів пропонують бібліотеки Aztec, BlockSolver95, Hypre, ILUS, MUMPS, PARMS, PSBLAS, PSPASES, PSparslib, SUPERLU, SPARSKIT.

Розвиває та покращує ринок GPU на сьогодні компанії NVidia та AMD. Їх розробки CUDA (Compute Unified Device Architecture) та ATI Stream Technology дають можливості використовувати потенціал GPU для запуску обчислювальних програм. Використання декількох відеокарт в одному комп'ютері, або великого числа графічних чіпів, дозволяє реалізувати паралельну обробки на додаток до паралельної обробки графіки. Крім того, навіть одна структура GPU-CPU за рахунок спеціалізації на кожному чіпі надає переваги, недоступні при застосуванні кількох звичайних процесорів.

Для паралельних розрахунків є можливість використовувати центральний процесор з використанням декількох процесорів та гібридні системи з розподіленням керуючої частини до CPU і паралельної частини до GPU. Різниця між використанням багатопроцесорного одного керуючого пристрою та гібридної архітектури полягає у наступному:

Перший потребує від алгоритмів більшої степені паралелізму на однотипних процесорних ядрах, які на програмному рівні вирішуються за допомогою спеціальних програмних систем такі як MPI.

Другий потребує від алгоритму більш складної багаторівневої паралельної моделі. Такі системи потребують відповідей на додаткові запитання до алгоритму та використовування пам’яті.

Розв’язання диференціальних рівнянь завжди було суттєвою проблемою багатьох задач з моменту існування таких задач. Розв’язуючи ту чи іншу реальну проблему за допомогою математики дослідники будують математичні моделі, які в свою чергу у багатьох випадках зводяться до розв’язання диференціального рівняння бо саме диференційні рівняння краще за будь які інші окреслюють суть процесу.

Було б чудово, якби ЕОМ мали засоби для розв’язання такого великого класу задач аналітично, проте саме лише використання машини для пошуку розв’язку змушує нас відмовитися від точних розв’язків на користь наближених в деякому наборі точок. Замість точної задачі можна використати наближення диференційного оператора у вигляді різницевої схеми, поставити

йому у відповідність граничні початкові умови і знайти наближений розв’язок за допомогою системи лінійних алгебраїчних рівнянь.

Використання таких систем і залежність від їх розв’язку має не лише клас диференційних рівнянь, а й багато областей науки й техніки. Незважаючи на велику увагу до створення програмного забезпечення з лінійної алгебри багато проблем ефективного його використання залишаються.

Такі машинні алгоритми здебільшого передбачають розв’язання задач із потрібною точністю та будь-якою кількістю потрібних точок, де ми можемо знайти наближений розв’язок. Простіше за інші, математичні моделі записуються та обчислюються на ЕОМ за допомогою лінійної алгебри та СЛАР. Список методів для розв’язання систем можемо перераховувати достатньо довго (метод Гауса, метод Гауса-Жордана, метод Гауса-Зейделя, матричний метод розв’язання систем лінійних алгебраїчних рівнянь, метод квадратного кореня, метод Крамера, метод прогонки, метод Якобі, метод релаксації, розвинення Холецького, проекційні методи, метод регуляризації Тихонова, ітераційні методи, метод Річардсона) [9].

Вибір того чи іншого методу залежить від вигляду отриманої системи, її властивостей, бажаної швидкості знаходження розв’язку. Для використання методу на ЕОМ, маючи різницеву залежність для знаходження наближеного розв’язку, ітеративні матимуть перевагу при розпаралелюванні. Кожна наступна ітерація знаходиться з використанням попередніх обчислень, доки процес не збіжиться в усіх точках. Гарантія збіжності забезпечується кроками методу та початковим наближенням. Такі методи, маючи коректну постановку збігаються при достатній кількості операцій. Тож головне питання для ітеративного процесу складається у швидкості знаходження наближення, бо в деяких умовах використання методів, як для швидкого обчислення у реальному часі, при надточних обчисленнях можуть залежати не тільки отримані розв’язки, прийняття рішення, а навіть життя людини.

Метою роботи є розробка та дослідження гібридних ітераційних алгоритмів для розв’язання різницевих рівнянь для еліптичних операторів. Можливість використання наступних гібридних архітектур:

- 1 CPU + 1 GPU
- 1 CPU + 2 GPU
- 2 CPU + 2 GPU

ЧАСТИНА 1. ОГЛЯД АРХІТЕКТУР ПАРАЛЕЛЬНИХ КОМП'ЮТЕРІВ ТА ЗАСОБІВ РОЗПАРАЛЕЛЮВАННЯ ПРОГРАМ

1.1 Існуючі архітектури, та їх розвиток

Зазвичай основним обчислювальним компонентом систем для високопродуктивних обчислень, включаючи кластери, є центральний процесор. Проте, вже починаючи з процесорів, які з'явилися в 1989 році у складі комп'ютерів з'явився такий елемент, як співпроцесор, що можна вважати гібридизацією на апаратному рівні.

Співпроцесор - спеціалізований процесор, що розширює можливості центрального процесора комп'ютерної системи, але оформлений як окремий функціональний модуль. Фізично співпроцесор може бути окремою мікросхемою або вбудований у центральний процесор, як це робиться у випадку математичного співпроцесора у процесорах для ПК починая з Intel486DX представлений 10 квітня 1989 року.

Можна виокремити наступні види співпроцесорів:

- математичні загального призначення (пришвидшують обчислення з плаваючою комою)
- співпроцесор вводу-виводу
- виконання вузькоспеціалізованих обчислень

Відеокарта (пристрій - device). – це набір незалежних мультипроцесорів (multiprocessors) на GPU, на яких реалізується розпаралелення окремих підзадач.

У середині 2000-х років після виходу у світ нової серії відеоадапторів GeForce8 від Nvidia стало можливим використання графічного процесора для обчислювальних цілей [5].

Виходячи зі специфіки архітектури GPU переведення обчислень з звичайного способу стає складною задачею. До того саме використання додаткового обчислювального модуля накладає відповідні обмеження. Архітектури GPU які підтримують парадигму GPGPU дають особливий інтерфейс для роботи з графічними картками використовуючи вже існуючі

способи програмування. Завдяки таким інтерфейсам можливо вдосконалити вже побудовані алгоритми та досягти більшої швидкості обчислення.

GPGPU є результатом розвитку шейдерних програм та спеціалізованих для них мов програмування високого рівня [7], таких як Cg, GLSL, HLSL. Початкове призначення графічних процесорів направлене на розв'язання вузького кругу задач для обробки графічних даних.

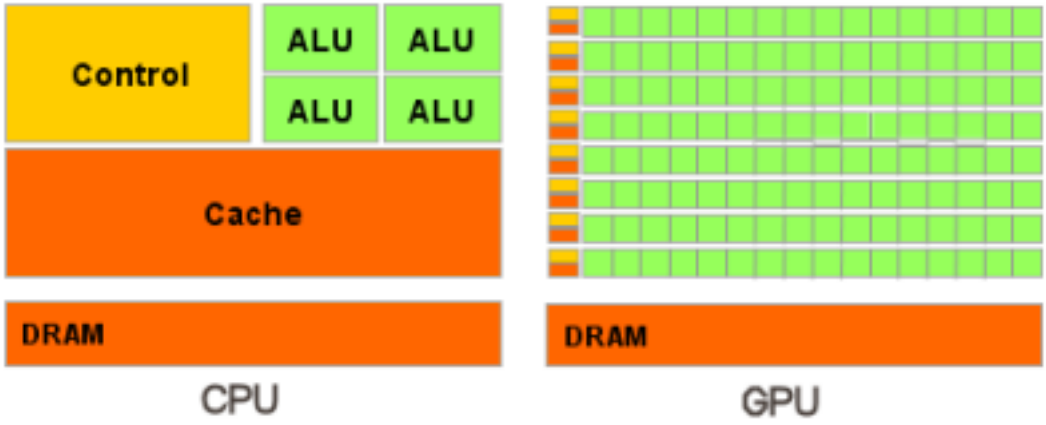

Рис. 1.1.1 Візуальне порівняння архітектури CPU та GPU

На Рис.1.1.1 можна побачити різницю у організації та структурі пристроїв центрального процесору та графічного.

Такі особливості GPU пояснюються особливостями архітектури. Сучасні CPU мають декілька ядер (2, 4, 8, 16) та не завжди кількість ядер відповідає фізичної кількості. Графічний процесор спочатку створювався як багатоядерна структура, у якій кількість ядер на порядок більше. Різниця в архітектурі обумовлює й різницю в принципах дії. Якщо архітектура CPU пропонує послідовну обробку інформації, то GPU історично пропонувався для обробки комп'ютерної графіки, тому розраховані на масивні паралельні обчислення.

Кожна з цих двох архітектур має свої переваги. CPU краще працює з послідовними задачами, виконує логічні операції. При великій кількості оброблюваної інформації та одноманітності операцій перевагу має GPU. Умова лише одна – в задачі повинен спостерігатися паралелізм.

Графічний процесор підтримує SIMD метод обчислень. Більшість графічних алгоритмів використовує саме такий підхід як найбільш

ефективний спосіб для задач графічної візуалізації. Модуль який перетворює потік вхідних даних до вихідного зветься kernel - ядро.

«GPU вже досягли тієї точки розвитку, коли багато додатків реального світу можуть з легкістю виконуватися на них, при чому швидше, за багатоядерні системи. Майбутні обчислювальні архітектури стануть гібридними системами з графічними процесорами, які будуть складатися з паралельних ядер працюючи у зв'язку з багатоядерними CPU»

За класифікацією по Фліну [3, 4] загальна архітектура може мати наступні варианти реалізації:

- SISD
- MISD
- SIMD
- MIMD

За цією класифікацією ЕОМ поділяються на 4 типа за кількістю потоків команд та даних. Зазвичай попередники сучасних комп'ютерів мали одну головну архітектуру SISD. Паралельно йшла розробка векторних та матричних архітектур MISD, SIMD та багатопроцесорні MIMD.

Кожна архітектура має свої особливості та свої як гарні так і погані сторони. Методи розпаралелювання можуть бути оптимізовані для одних чи інших типів архітектур.

Вигляди архітектур:

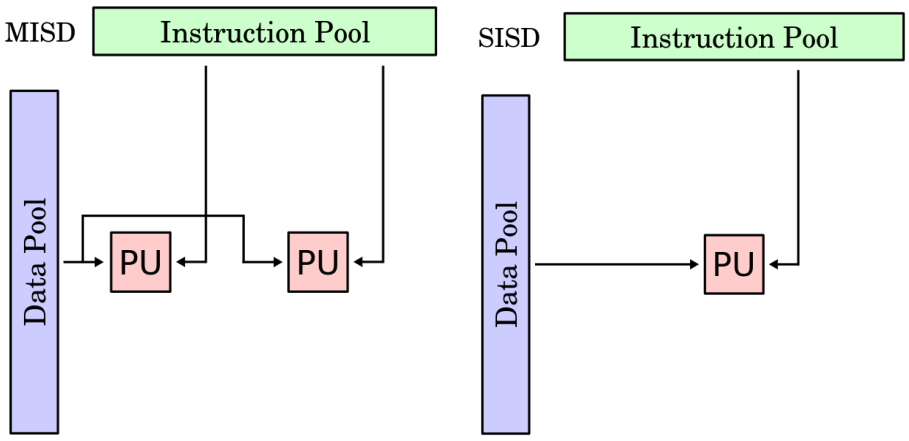

Рис. 1.1.2 Вигляд MISD та SISD архітектури

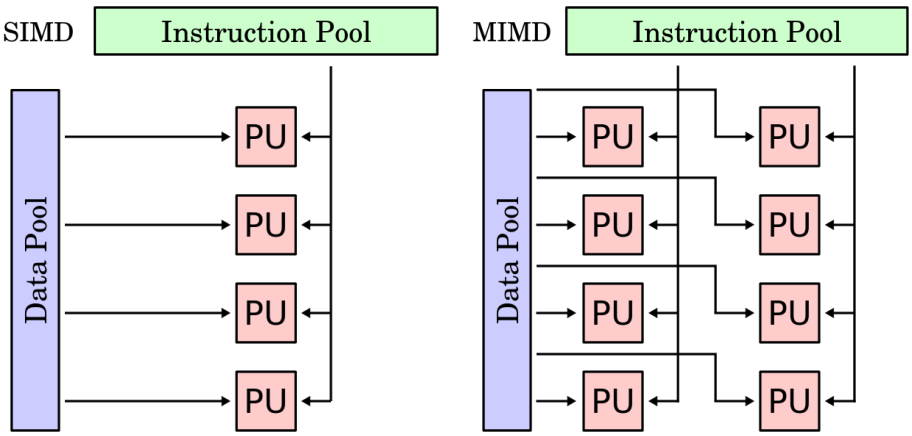

Рис. 1.1.3 Вигляд SIMD та MIMD архітектури

Саме MIMD (Multiple Instruction stream, Multiple Data stream) - концепція архітектури комп'ютера [2], що використовується для досягнення паралелізму обчислень.

1.2 Комп'ютери гібридної архітектури

Вважатимемо, що паралельний обчислювальний комплекс має такі складові:

1. Хост-комп'ютер для здійснення керування використанням багатопроцесного обчислювального ресурсу, проведення загальносистемного моніторингу, комунікації з термінальними мережами користувачів, візуалізації розв'язків задач та реалізації тієї частини обчислювального процесу, яка не розпаралелюється;
2. Обробляюча частина, що містить обчислювальні вузли для розв'язування задачі з паралельною організацією обчислень. Вона є однорідною масштабованою системою, яка складається з багатьох високопродуктивних процесів з власною оперативною та дисковою пам'яттю, об'єднаних комунікаційним середовищем міжпроцесної взаємодії;
3. Дискове сховище для зберігання програмних модулів, великорозмірних даних та результатів обчислень;
4. Комунікаційні середовища та комутаційне середовище, призначені для ефективної взаємодії обчислювальних вузлів при проведенні розрахунків.

Операційна система хост-комп’ютера повинна забезпечувати виконання ряду завдань, таких, як компіляція та запуск програми на хост-комп’ютері, формування завдання і запуск процесу розв’язування задачі на вибраній кількості процесів, моніторинг виконуваних завдань, збереження і візуалізація протоколів паралельних розрахунків, адміністрування доступних частин розподіленої файлової системи. Також має бути встановлено відповідне середовище міжпроцесної взаємодії та компілятор, що підтримує мову програмування, на якій написано виконувану програму.

Хоч якою чудовою не була одна чи інша система, саме комбінація переваг дає найбільш потужну перевагу над усіма недоліками. В результаті зараз в списку найпотужніших суперкомп’ютерів світу представленні системи як класичної MIMD-архітектури так і гібридної архітектури (MIMD+SIMD).

1.3 Програмні інтерфейси та технології для програмування на комп’ютерах гібридної архітектури

На сучасному ринку можна виокремити 2 конкурентні програмно-апаратні архітектури графічних процесорів, за допомогою яких можливо використати повну потужність гібридного комп’ютера.

- NVidia CUDA
- ATI Stream Technology

Це найпопулярніші та швидко зростаючі системи, які поєднують у собі всі вдалі напрацювання обох компаній та інших досліджень. До цієї двійки прагне долучитися і компанія Intel з їх технологією Intel Larrabee, проте на дану мить, ця пропозиція не має цінності.

Додаткові можливості паралелізації процесів надають бібліотеки загального призначення типу OpenMP, MPI. Їх APIв багатьох випадках доступні для загальних потреб та широкий клас існуючих задач можливо швидко реалізувати за допомогою цих бібліотек. Основна можливість таких пакетів полягає у швидкому директивному стилі додавання оптимізації, не вкладаючись у подробиці паралельних обчислень чи графічних девайсів.

Для обґрунтованого вибору бібліотеки для використання проведемо їх аналіз.

Технологія CUDA

CUDA - паралельна обчислювальна платформа і модель програмування, створена NVIDIA і виконувана на графічних процесорах (GPU), які вони виробляють. CUDA дає розробникам програм прямий доступ до великої кількості віртуальних команд і пам'яті на паралельних обчислювальних елементах у CUDA GPU [5].

Графічні процесори, які використовують CUDA, можуть використовуватися не тільки для обробки графіки; але і для обчислень загальних потреб. Такий підхід відомий як GPGPU. У порівнянні з традиційним підходом до організації обчислень загального призначення за допомогою можливостей графічних API, у архітектури CUDA відзначають наступні переваги в цій області:

- CUDA архітектура
    - Використання GPU обчислень для звичайних цілей
    - Збереження продуктивності
- CUDA C/C++ мова
    - Заснований на стандартизованому C/C++
    - Малий набір доповнень для включення можливостей гетерогенного програмування
    - Чітке API для управління пристроями, пам’яттю.

Найбільшу увагу надамо саме мові та її можливостям. У подальших викладках будемо використовувати наступні поняття:

- Host (хост) – CPU та його пам’ять
- Device (пристрій) – GPU та його пам’ять

Гетерогенне програмування

Гетерогенне програмування складається з двох частин коду котрі записані разом, але мають різний спосіб дії. Окремими функціями пишеться код для пристрою і хоста, після початку дії хоста в деякий момент часу ми викликаємо функції пристрою [7].

На рівні керуючого процесора виконуємо послідовні операції з використанням Сітки GPU. Сітка GPU - задана користувачем однорозмірна чи дворозмірна ієрархія блоків, кожен з яких також задається користувачем в розмірності 1-3 блоків ниток. Комунікація через спільну пам'ять та синхронізація можлива лише всередині блоку користувача [3, 4, 5].

Код головної програми викликає код девайсу. У цьому виклику задаються Сітка і блоки ниток наступним чином:

- *kernelName* <<<*gridDims*, *threadDims*>>>(*params*);

Блоки Сіток і ниток описуються заданою змінною типу 'dim3'. Разом з додатковим типом задаються необхідні параметри доступу до пам'яті:

- *gridDim* - розмір блоку Сітки
- *blockIdx* - індекс поточного блоку
- *blockDim* - розмір блоку ниток
- *threadIdx* - індекс поточної нитки

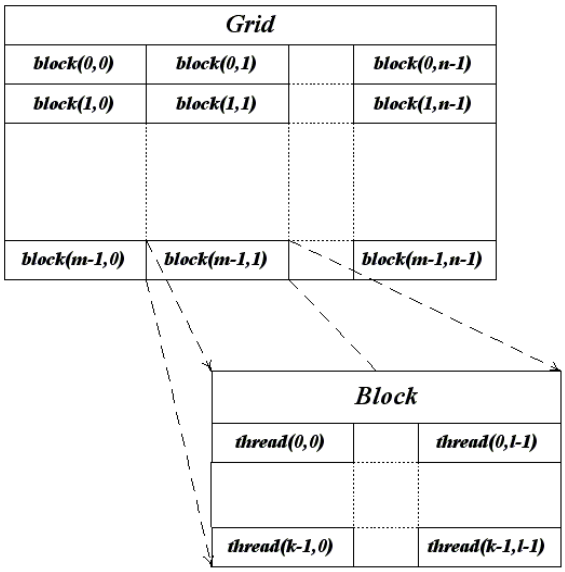

Рис. 1.2.1 - візуальне представлення блоків Сітки і ниток

Основною задачею такого підходу є правильне індексування елементів в залежності від розмірності блоку ниток, тож унікальни індекси потрібно описувати наступними формулами

- *id* = *threadIdx.x;*
- *id* = *threadIdx.x* + *threadIdx.y* * *blockDim.x;*

• *id = threadIdx.x + threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockDim.y;*

У блоці ниток створюється форма з 32 ниток - warp. В нього входять нитки з послідовними ідентифікаторами. Така форма об'єднана за принципом виконання - виконується одна інструкція з блоку, за нею виконується інструкція іншого блоку і так далі.

Для виконання такого блоку потрібно 32 / кількість ALU на GPU циклів, наприклад якщо в GPU маємо 8 ALU елементів - потрібно 4 циклів. При умовних операціях, які виконуються на пристрої починається послідовна обробка даних, тож рекомендовано використовувати менше логічних операцій на девайсах.

Виходячи з того, що GPU не має доступу до оперативної пам'яті - програмісту потрібно самому перенести усі необхідні ресурси, необхідні для використання ядра. Для цього зазначені основні функції:

- *cudaMalloc* - резервування пам'яті
- *cudaMemcpy* - копіювання блоків пам'яті
- *cudaFree* - звільнення пам'яті

В CUDA розрізняються 6 видів пам'яті GPU, кожна з яких має своє призначення і до яких можна звертатися в (див. табл. 1.2.1).

Тип пам'яті	Доступ	Рівень виділення	Швидкість виконання
регістри (registers)	R/W	per-thread	висока (on chip)
local	R/W	per-thread	низька (DRAM)
shared	R/W	per-block	висока (on-chip)
global	R/W	per-grid	низька(DRAM)
constant	R/O	per-grid	висока(on chip L1 cache)
texture	R/O	per-grid	висока(on chip L1 cache)

Табл. 1.2.1 Види пам'яті GPU

Глобальна пам’ять (global) необхідна в основному для відправлення даних задачі з хоста (CPU) на GPU та для збереження результатів роботи програми перед відправленням їх на хост. Глобальна пам’ять – єдиний вид пам’яті на GPU, куди можна щось записувати. Глобальна пам’ять не кешується. Вона працює дуже повільно, тому кількість звернень до глобальної пам’яті слід в будь-якому випадку мінімізувати. Дані з CPU (з хоста) можна відправляти на GPU викликом функції cudaMemcpy. Глобальну пам’ять можна виділити також динамічно, викликавши функцію cudaMalloc(void * mem, int size) на хості. З процесорів GPU цю функцію викликати не можна. Звідси випливає, що розподілом пам’яті повинна займатися програма-хост, що працює на CPU.

Локальна пам’ять (local) фізично є аналогом глобальної пам’яті, і працює з тією ж швидкістю. У випадках, коли локальні дані процедур займають занадто великий розмір, їх можна помістити в локальну пам’ять.

Константна пам’ять (constant) здебільшого використовується для збереження лише невеликої кількості даних, що часто використовуються всередині одного блоку. Константна пам’ять кешується. Кеш існує в єдиному екземплярі для одного мультипроцесора, а значить, вона є загальною для всіх задач всередині блоку. Її розмір становить всього 64 Кбайт (на всі пристрої).

Текстурна пам’ять (texture) використовується, як-правило, в задачах не обчислювального характеру.

Розділена пам’ять (shared memory) – це швидка пам’ять. Саме цю пам’ять рекомендується використовувати як керований кеш при розв’язуванні задач. Проте ця пам’ять дуже мала. На один мультипроцесор доступно всього 16 Кбайт розподіленої пам’яті. Оскільки shared-пам’ять невелика, то блоки, які виявилися найбільш придатними для кеш-пам’яті CPU, не помістяться в ній. Таким чином, при застосуванні GPU доцільніше спочатку розбити матриці СЛАР на смуги по 16 стовпчиків, скопіювати ці смуги на глобальну пам’ять GPU. Це достатньо велика пам’ять для збереження інформації, проте швидкодія обчислень в ній значно повільніша, ніж на shared-пам’яті. А потім, розбивши смуги матриць СЛАР на підматриці 16×16, організувати матрично-векторні операції цих підматриць на shared-пам’яті. При цьому одна нитка

буде завантажувати рівно по одному елементу для виконання операцій над матрицями або векторами.

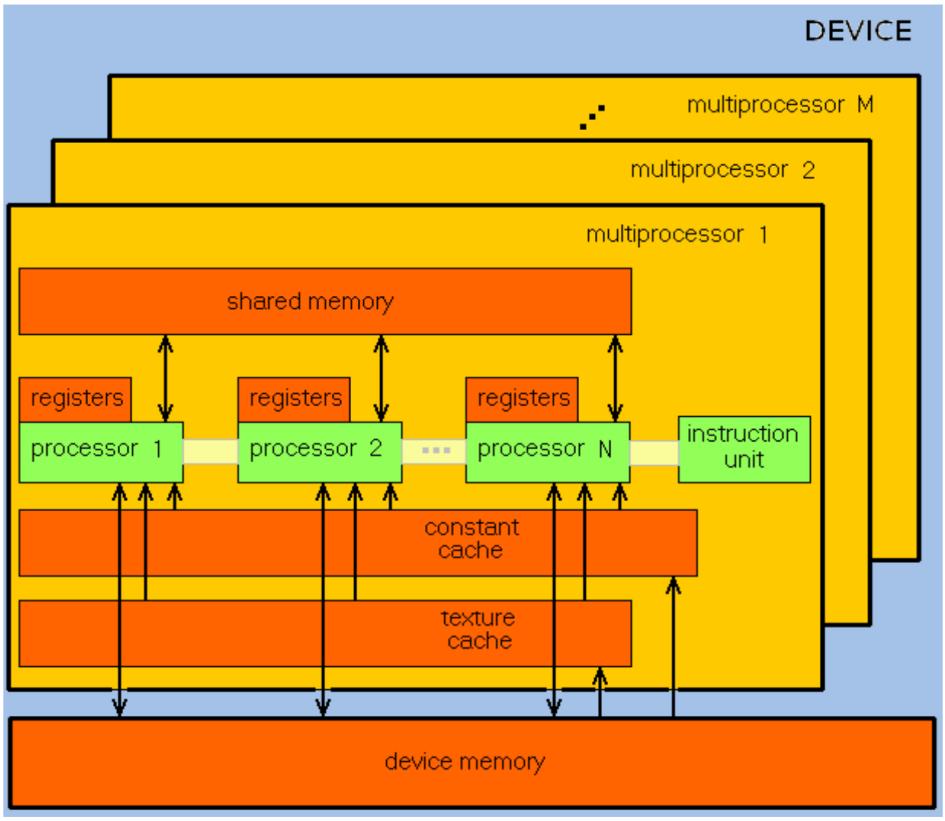

Рис. 1.2.2 Схема роботи мультипроцесорів GPU

Величезний розрив між високою продуктивністю графічних процесорів і повільними зв'язками між CPU і GPU значно ускладнює розробку масштабованого програмного забезпечення з використанням декількох графічних прискорювачів. Виникає необхідність у плануванні обчислень на обчислювальних ресурсах гібридної системи з метою якнайшвидшого отримання розв'язку задачі, в узгодженні розподілу обчислювальних ресурсів на ядрах CPU і GPU, а також оптимізації комунікаційних витрат між ядрами CPU і процесорами GPU.

Доцільно розбити задачу на деякі підзадачі з метою досягнення високої продуктивності за рахунок збільшення обчислювальної інтенсивності на GPU та зменшення кількості передач інформації між CPU і GPU.

Відмінностями від звичайного програмування мають наступний характер. Функції, змінні пристрою мають декілька специфікаторів які дозволяють бачити функцію чи змінні на хості та пристрою чи лише на пристрої. Щоб

передати аргументи потрібно скопіювати їх з пам’яті хоста у пам’ять пристрою та передати посилання на них у аргументах.

Особливість потоків в кожному блоці полягає у тому, що вони мають можливість синхронізуватися, тобто кожен потік може дочекатися іншого потоку.

MPI

Message Passing Interface (MPI, інтерфейс передачі повідомлень) - програмний інтерфейс(API) для передачі інформації, який дозволяє обмінюватися повідомленнями між процесами, що виконують одну задачу. Розроблено Вільямом Гроуппом, Евін Ласко та іншими.

MPI є найбільш поширеним стандартом інтерфейсу обміну даними в паралельному програмуванні, існують його реалізації для великого числа комп'ютерних платформ. Використовується при розробці програм для кластерів і суперкомп’ютерів. Основним засобом комунікації між процесами в MPI є передача повідомлень один одному. У стандарті MPI описаний інтерфейс передачі повідомлень, який повинен підтримуватися як на платформі, так і в додатках користувача [4].

У першу чергу MPI орієнтований на системи з розподіленою пам'яттю, тобто коли витрати на передачу даних великі, у той час як OpenMP орієнтований на системи з загальною пам'яттю (багатоядерні із загальним кешем). Обидві технології можуть використовуватися спільно, щоб оптимально використовувати в кластері багатоядерні системи.

OpenMP

OpenMP(Open Multi-Processing) — це набір директив компілятора, бібліотечних процедур та змінних середовища, які призначені для програмування багатониткових програм на багатопроцесорних системах із спільною пам'яттю на мовах C, C++ та Fortran.

Розробку специфікації OpenMP ведуть кілька великих виробників обчислювальної техніки та програмного забезпечення, робота яких регулюється некомерційною організацією, названою OpenMP Architecture

Review Board (ARB). Специфікації для мов Fortran і C/C++ з'явилися відповідно в жовтні 1997 року і жовтні 1998 року.

OpenMP можна розглядати як високорівневу надбудову над Pthreads (або аналогічними бібліотеками ниток). POSIX-інтерфейс для організації ниток Pthreads підтримується широко (практично на всіх UNIX-системах).

OpenMP реалізує паралельні обчислення за допомогою багатопоточності, [4] в якій «головна» (master) нитка створює набір підлеглих (slave) ниток і завдання розподіляється між ними. Передбачається, що нитки виконуються паралельно на машині з декількома процесорами (кількість процесорів не обов'язково має бути більше або дорівнювати кількості ниток).

Завдання, що виконуються нитками паралельно, так само як і дані, необхідні для виконання цих завдань, описуються за допомогою спеціальних директив препроцесора відповідної мови.

Кількість створюваних ниток може регулюватися як самою програмою за допомогою виклику бібліотечних процедур, так і ззовні, за допомогою змінних оточення.

OpenMP має такі переваги:

1. За рахунок ідеї «інкрементального розпаралелювання» OpenMP ідеально підходить для розробників, що прагнуть швидко розпаралелювати свої обчислювальні програми з великими паралельними циклами. Розробник не створює нову паралельну програму, а просто послідовно додає в текст послідовної програми OpenMP-директиви.
2. При цьому, OpenMP - досить гнучкий механізм, що надає розробникові великі можливості контролю над поведінкою паралельного додатку.
3. Передбачається, що OpenMP-програма на однопроцесорній платформі може бути використана як послідовна програма, тобто немає необхідності підтримувати послідовну та паралельну версії. Директиви OpenMP просто ігноруються послідовним компілятором, а для виклику процедур OpenMP можуть бути підставлені заглушки (stubs), текст яких приведений в специфікаціях.

4. Одним з переваг OpenMP його розробники вважають підтримку так званих «orphan» (відірваних) директив, тобто директиви синхронізації і розподілу роботи можуть не входити безпосередньо в лексичний контекст паралельної області.

1.4 Основні переваги та обмеження використання CUDA

В даній роботі була обрана CUDA як досліджувана технологія, з наступних причин:

- Інтерфейс програмування додатків CUDA (CUDA API) заснований на стандартній мові програмування C з деякими обмеженнями. На думку розробників це повинно спростити та пом’якшити процес вивчення архітектури CUDA.
- Підтримка розробників від авторів, безкоштовні відео лекції, відкриті форуми тощо.
- Поділена між потоками пам’ять (shared memory) розміром у 16 Кб може бути використана під організований користувачем кеш з більш ширшою полосою пропуску ніж при виборці зі звичайних текстур.
- Більш ефективні транзакції між пам’яттю центрального процесора та відеопам’яттю.
- Повна апаратна підтримка цілочисельних та бітових операцій.
- Підтримка компіляції GPU-коду засобами відкритого LLVM (низькорівнева віртуальна машина).
- Велика кількість підтримуваних графічних плат.

З обмежень маємо:

- Усі функції, виконані на пристрої не підтримують рекурсії.
- Збільшення розмірності задачі збільшує проблему індексування.
- Потреба до копіювання усіх необхідних даних у пам’ять девайсу.
- Бажане використання лише послідовних арифметичних операцій.

Такі висновки представлені при порівнянні CUDA з традиційним підходом до організації обчислень загального призначення за допомогою можливостей графічних API.

Серед основних вимог до програм, написаних для виконання на гібридних архітектурах, виділяють:

1) паралелізм – здатність виконання програми багатьма процесами одночасного;

2) масштабованість – забезпечення можливості виконання програми з використанням різної кількості процесів;

3) локальність – така організація обчислень, при якій звернення до локальних даних відбувається значно частіше, ніж до віддалених.

Додаткові складнощі, які виникають при розробці такого забезпечення мають такі характери:

- Програма повинна бути складена з коду для CPU (на звичайній мові програмування C / C++) та коду для графічного процесора написаного на спеціальній мові, CUDA.
- Друга проблема пов’язана з ефективним використанням обчислювальних ресурсів, з узгодженням розподілу обчислювальних ресурсів на ядрах (GPU та CPU).

Також на ефективність реалізації паралельного алгоритму впливає використання різних способів обміну даними між процесами. Процеси можуть взаємодіяти попарно за допомогою обмінів типу «точка-точка» або групою з використанням колективнихобмінів. До найбільш часто використовуваних належать:

1. Мультирозсилка, за якої один з процесів взаємодіючої групи розсилає дані всім процессам цієї групи.

2. Мультизбірка числа, при якій усі процеси взаємодіючої групи передають рівні порції даних одному з процесів цієї групи. При цьому над відповідними компонентами даних від різних процесів виконуються операції зведення, наприклад, додавання, вибір максимального або мінімального значення і тому подібні.

3. Мультизбірка вектора, під час якої всі процеси взаємодіючої групи передають порції даних одному з процесів групи, в якому з прийнятих даних формується новий вектор.

ЧАСТИНА 2. АНАЛІЗ АЛГОРИТМІВ ІТЕРАЦІЙНИХ МЕТОДІВ

2.1. Основні властивості та оцінки паралельних алгоритмів

Щоб дати відповідь на запитання чи має сенс використовувати той чи інший метод для даних архітектур важливо мати коефіцієнти, що характеризують ефективність паралельних алгоритмів. Розпаралелювання обчислень багатоваріантно, тобто для MIMD машин з однією й тою ж самою структурою між процесорних зв'язків можуть бути побудовані різноманітні варіанти алгоритмів про паралельну організацію обчислень.

Ефективність алгоритмів у значній мірі визначається схемою розподілу вихідних даних між процесами. Наприклад, найпростішим способом розподілу є блочний, при якому матриця розподіляється на декілька рівних блоків, кількість яких дорівнює кількості процесів [10, 11, 12].

Введемо такі позначення:

- t_C - середній час виконання однієї арифметичної операції на CPU
- t_G - середній час виконання однієї арифметичної операції на GPU
- n_0 - кількість арифметичних операцій, які можуть бути виконані одночасно на GPU
- t_1 - сумарний час, який потрібний для пересилки одного блоку від CPU до GPU і в зворотньому напрямі
- t_B - час обміну блоком між двома GPU

Для порівняння й оцінки якості алгоритмів паралельних обчислень будемо користуватися такими критеріями, як коефіцієнт прискорення та коефіцієнт ефективності:

$$К_п = \frac{T_1}{T_p} К_е = \frac{К_п}{p}$$

Де p – кількість процесорів на $MIMD$машині

T_p – час розв'язання на p – процесорах

T_1 – час розв'язання на 1 – процесорній ЕОМ

Деякі автори вводять й інші характеристики:

$$\text{Ціна алгоритма: } C_p = pT_p$$

$$\text{Якість: } F_p = \frac{К_п}{C_p} = \frac{К_е}{T_p} = \frac{T_1}{pT^2{}_p}$$

t_0 — час обміну одним машинним словом між ПП

t_c — час встановлення зв′язку між ПП

$$\tau_0 = \frac{t_0}{t}\tau_c = \frac{t_c}{t}$$

При побудові паралельних алгоритмів вважається, що необхідна для реалізації обчислювального алгоритму інформація зберігається і обробляється в оперативній пам’яті гіпотетичного послідовного комп’ютера або ж у сумарній пам’яті MIMD-комп’ютера, на якому виконуються *p* процесів, тобто обчислювальний процес здійснюється без використання зовнішньої пам’яті.

2.2. Постановка модельної задачі та впорядкування

Опис задачі:

У якості модельної розглядаємо задачу для самоспряжених рівнянь другого порядку в прямокутнику з заданими граничними умовами.

$$\Delta u = f(x)$$

$$u\,|_{\partial\Omega} = g(x)$$

На сітці поставимо у відповідність різницеву задачу, використовуючи звичайну схему «хрест».

Для запису системи лінійних алгебраїчних рівнянь в матрично-векторному вигляді необхідно спочатку встановити відповідність між впорядкуванням рівнянь та впорядкуванням невідомих. Роздивимось два впорядкування невідомих. Природне впорядкування та червоно-чорне.

Природне:

$y_{i^*j^*}$наступна за y_{ij}при $j^* > j$ або якщо $j^* = j$ та $i^* > i$.

Червоно-чорне:

Нехай червоні невідомі утворюють множину всіх таких y_{ij} для яких $i+j$ парне, і нехай чорні невідомі при не парній сумі.

Тоді червоно-чорним впорядкуванням може бути будь яке впорядкування, при якому будь яке чорне невідоме йде після червоної невідомої.

Приклади:

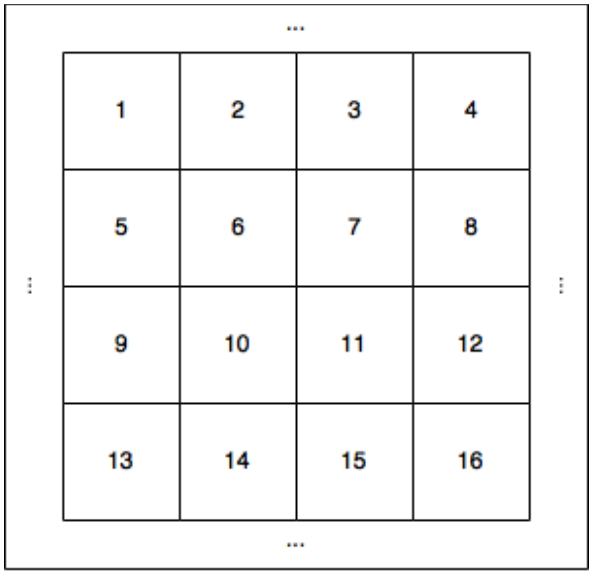

Рис. 2.2.1 Звичайне впорядкування.

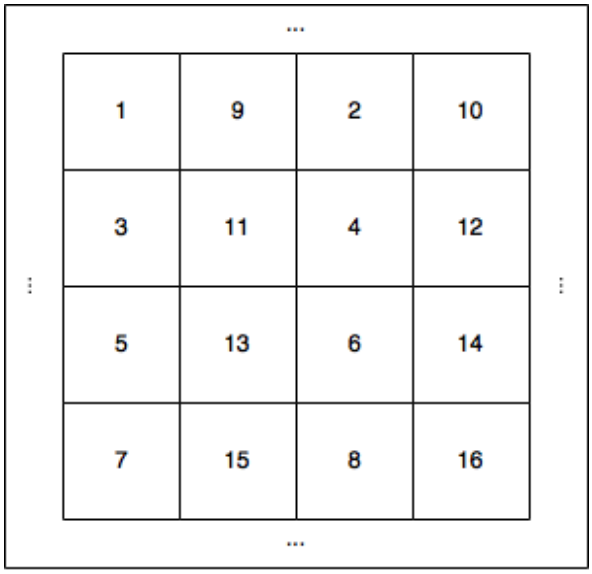

Рис. 2.2.2 Червоно-чорне впорядкування.

При розв’язування СЛАР з матрицями великих порядків реалізація прямих методів, навіть тих, що враховують розрідженість матриці, на комп’ютері може виявитися досить складною і неефективною. Проблеми машинної реалізації прямих методів пояснюються обмеженням об’єму

$$\lim_{i \to \infty} x_i = x$$

оперативної пам’яті та зниженням швидкодії при використанні дискової. Запис систем з матрицями, які мають певну специфіку, у вигляді процедури обчислення вектора Ax потребує у ряді випадків менше комп’ютерної пам’яті, ніж при реалізації прямих методів. У такому випадку для розв’язування СЛАР доцільно використовувати швидкозбіжні ітераційні методи.

Значного ефекту від використання ітераційних методів можна отримати у задачах з розрідженими матрицями, оскільки при використанні прямих методів відбувається збільшення кількості ненульових елементів, а відтак зростають вимоги до пам’яті та об’єму обчислень. Особливо відчутною може бути різниця у випадку паралельних обчислень, оскільки тоді час розв’язування задачі можна скоротити ще більше.

Ідея типового ітераційного методу полягає у виборі початкового наближення x_1 до розв’язку x і побудові послідовності x_2, x_3,…, такої, що.

Теоретично при використанні ітераційного методу необхідно виконати нескінченну кількість арифметичних операцій, щоб отримати x, але на практиці призупинення відбувається тоді, коли, на наш погляд, чергове наближення достатньо близьке до x. Такі методи є досить привабливими з точки зору вимог до машинної пам’яті, оскільки їх реалізація у загальному випадку вимагає зберігати лише матрицю A, вектори b, $x^{(i)}$ і ще, можливо, один або кілька векторів. До того ж, варіювання величини бажаної точності розв’язку дає змогу змінювати загальний час розв’язування системи.

2.3 Аналіз методів та вибір оптимальних для досліджень

Одна за найбільш вагомих частин дослідження випала на аналіз та обирання методу розв’язання системи рівнянь. На вимогу попередніх викладок, умови на обирання методу були наступні:

- Ітеративний
- Можливість розпаралелювання
- Можливість пошуку наближеного розв’язку з деякою точністю
- Збіжність методу
- Схильність до розріджених матриць

Серед розглянутих, були наступні методи:

- Метод Якобі
- Метод Гауса-Зейделя
- Метод верхньої релаксації
- Метод Річардсона (явний Чебишевський метод)

Для досліджень були обрані 2: Метод Річардсона та Метод верхньої релаксації. Метод Якобі був відхилений як спрощення Річардсона, а Гауса-Зейделя є модифікацією Якобі, та все ж не є кращим за Річардсона.

Метод релаксації має менше аналогів та спрощень (хоча при особливому додатковому параметрі можна перейти до методу Зейделя), та має 2 види верхні та нижні в залежності від обраного додаткового параметра. Цей метод є представником стаціонарних одно крокових ітераційних методів лінійної алгебри.

Розглядається саме 2 методи з причин різного підходу методів до розпаралелювання в залежності від моделі методу, на це впливає саме використання точок даної ітерації при розрахунку тієї ж самої ітерації.

Основні формули для обох методів виглядають наступним чином [6, 13]:

Метод верхньої релаксації.

$$\frac{(D+\omega L)(x^{(s+1)}-x^{(s)})}{\omega}+Ax^{(s)}=b$$

$$A=L+D+U$$

$$Dx=\left(\frac{2}{h_1^2}+\frac{2}{h_2^2}\right)x$$

$$Lx(i,j)=-\frac{1}{h_1^2}\dot{x}(i-1,j)-\frac{1}{h_2^2}\dot{x}(i,j-1)$$

$$Ux(i,j)=-\frac{1}{h_1^2}\dot{x}(i+1,j)-\frac{1}{h_2^2}\dot{x}(i,j+1)$$

Тут: $x^{(s)}$ - наближення, отримане на ітерації з номером s, $x^{(s+1)}$ - наступне наближення, ω - параметр метода. H - крок розбиття.

Необхідною умовою збіжності метода з будь якого початкового наближення до точного розв’язку задачі є виконання умови:

$$\omega \in (0, 2)$$

Якщо A відповідає наданим вище умовам це є і достатньою умовою. При різних значеннях параметру говорять про нижні чи верхні релаксації. При проміжному значенні 1 отримаємо Метод Зейделя.

У загальному випадку немає аналітичної формули для обчислення оптимального параметру методу. Наприклад, при розв’язанні системи рівнянь, отриманих при апроксимації диференційних рівнянь в частинних похідних, можна використати евристичну оцінку вигляду:

$$\omega_{opt} \approx 2 - O(h)$$

де h – крок сітки дискретизації.

В деяких випадках можна оцінити більш точно оптимальний параметр:

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - q^2\left(D^{-1}(U + L)\right)}}$$

де ρ - спектральний радіус матриці а U, D, L - верхня, діагональна та нижня матриця від похідної A.

Метод Річардсона.

Для методу Річардсона достатньо лише додатної визначеності.

$$\frac{\overline{x}_{k+1} - \overline{x}_k}{\tau_{k+1}} + A \cdot \overline{x}_k = \overline{b}, \quad k = 0, 1, \ldots, n-1$$

де τ – оптимальний чебишевський параметр, окремий для кожної ітерації. Записаний наступним чином для найменшої похибки (доведено), і детермінується наступним чином [6]:

$$\tau_k = \frac{\tau_0}{1 + \rho_0 \cdot \nu_k}, \; k = 1, 2, \ldots, n$$

$$\tau_0 = \frac{2}{\lambda_{\min}(A) + \lambda_{\max}(A)}, \quad \rho_0 = \frac{1 - \eta}{1 + \eta}, \quad \eta = \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)},$$

$$\nu_k = \cos\frac{(2 \cdot k - 1) \cdot \pi}{2 \cdot n}, \quad k = 1, 2, \ldots, n.$$

Такий метод з таким набором параметрів називається явним ітераційним методом з чебишевським набором параметрів, при використанні лише нульового елементу з вектора отримаємо метод Якобі.

Для програмування цього методу потрібно знати кількість ітерацій, тож за вхідними даними потрібно зробити оцінку, яку можна виписати наступною формулою:

$$q_n = \frac{2\cdot\rho_1^n}{1+\rho_1^{2n}}, \quad \rho_1 = \frac{1-\sqrt{\eta}}{1+\sqrt{\eta}}, \quad \eta = \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)}.$$

$$n \geq n_0(\varepsilon) = \frac{\ln(2/\varepsilon)}{\ln(1/\rho_1)}.$$

де ε – бажана похибка.

Для найбільш невдалого випадку, коли η дуже мале, отримаємо наступну нерівність:

$$n_0(\varepsilon) = \frac{\ln(2/\varepsilon)}{2\cdot\sqrt{\eta}}.$$

Одна з важливих задач тут правильне впорядкування параметрів, бо саме від них залежать збіжність методу. Розв'язок цієї задачі був запропонований Самарським і Фрязіновим і є достатньо громіздким, опустивши їх можемо виписати конкретну формулу

$$\tau_j = \left[\frac{L+l}{2} + \frac{L-l}{2}\cos\frac{\pi(2j-1)}{2i}\right]^{-1}, j = 1, 2, \ldots, i.$$

Де L та lнайбільше та найменше власне значення початкової матриці. Виходячи з таких обчислень отримаємо нову оцінку кількості кроків:

$$i \approx \left[\frac{\sqrt{\mu}}{2}\ln\cdot\varepsilon^{-1}\right] + 1.$$

Програмна реалізація використання повного оптимального набору має свої додаткові аспекти, збіжність методу залежить від послідовності оптимальних параметрів, використовуючи чебишевский перелік оптимальних параметрів метод збігається.

Таким чином обидва методи можуть бути використані, як досліджувані. Вони задовольняють нашим прописаним умовам, мають передумови для

паралелізму, тож можемо перейти до реалізації їх послідовних та паралельних алгоритмів.

Послідовні алгоритми повністю відповідають формулам алгоритмів та ніяк не відрізняються від них, більшої уваги слід надати паралельним алгоритмам.

ЧАСТИНА 3. ПАРАЛЕЛЬНІ АЛГОРИТМИ ТА ЧИСЕЛЬНІ ЕКСПЕРИМЕНТИ АРХІТЕКТУРИ 1 CPU, 1 CPU + 1 GPU

3.1. Загальні положення до паралелізації

Проводити розпаралелювання можна двома способами, за допомогою написання своїх функцій для графічних пристроїв, чи за допомогою програмних паралельних інтерфейсів. Розглянемо ці підходи.

За особливістю GPU ми маємо велику кількість процесорів, проте не нескінчену. Для кожного пристрою кількість оптимальних одночасних операцій не перевищує кількості процесорів, проте ламати систему заради оптимальної кількості потоків чи блоків не є суттєвим, бо GPU має змогу самостійно відредагувати поставлену задачу.

3.2. Метод Річардсона

Метод Річардсона шукає наступне наближення за допомогою попереднього наближення, оптимальних параметрів, правої частини, основної матриці. Тож для розрахунку наступної ітерації ми можемо паралельно обрахувати одразу всі внутрішні точки нашої сітки.З урахуванням цього запропонуємо наступний паралельний алгоритм:

Розіб’ємо область внутрішніх та пригранічних вузлів на p підобластей за кількістю процесорних ядер, виділених для розрахунків, прямими перпендикулярними до тієї вісі, за якою більше вузлів розбиття (у нашому випадку сітка однорідна, і не має переваги та чи інша вісь). Для визначеності у подальшому будемо вважати, що розбиття проводимо прямими, які паралельні до 1 вісі. Розбиття проводимо таким чином, щоб час на ітерацію у кожній підобласті був приблизно однаковий.

Нехай підобласті нумеруються знизу вгору. Розширимо всі підобласті окрім граничних на один додатковий рядок до гори та до низу, граничні розширимо на один рядок у сторону сітки. Таким чином додаткові сіткові прямі, відіграють роль тих сіткових прямих, які знаходяться у сусідніх процесорах. Таким чином можна встановити зв’язок перед кожною ітерацією у процесах.

При такому розбитті можна представити наступний паралельний алгоритм:

1. Задаємо початкове наближення у кожну підобласть. Вводимо в кожний процес величину бажаної похибки, яка детермінує закінчення ітеративного процесу. Обчислюємо і зберігаємо в усіх процесах вектор оптимальних параметрів.
2. У кожному процесі у вузлах сітки одночасно й незалежно знаходимо наступну ітерацію. На цьому ж кроці перевіряємо умову закінчення методу. Якщо вона не виконується то переходимо на наступний крок.
3. Послідовно встановлюємо два рази зв'язок між процесами таким чином, щоб у результаті цього процеси, які містить у собі сусідні підобласті, з'єдналися кожен раз попарно між собою. Після кожного такту встановлюємо зв'язок між процесами щоб вони обмінялися додатковими значеннями.

Для визначення коефіцієнтів, що характеризують явний чебишевский метод з паралельною організацією обчислень, достатньо розглянути одну ітерацію.

Час для реалізації однієї ітерації за основною формулою дорівнює

$$T_1 = MQt$$

де M – кількість арифметичних операцій, необхідних для обчислення наступної ітерації для будь якого вузла.

Після цього машині потрібно лише 2 рази послідовно встановити зв'язок між сусідніми процесами. Після встановлення зв'язку процеси обмінюються $2Q_1$ словами.

Отримаємо наступну формулу, що визначає час виконання алгоритм при паралельній організації обчислень:

$$T_p = \frac{MQt}{p} + 4Q_1 t_0 + 2t_c$$

Наступні формули показують коефіцієнти прискорення та ефективності для паралельного алгоритму

$$K_e = [1 + \frac{4pt_0}{MQ_1} + \frac{2pt_0}{MQ}]^{-1} \text{ К}_{\text{пр}} = 1 + \frac{4pt_0}{MQ_1} + \frac{2pt_c}{MQ}.$$

З чого виходить, що з приростом p ефективність розпаралелювання буде зменшуватися, якщо інші величини, що входять до формули, залишаються незмінними. Також бачимо, що при зростанні M та Q, тобто при збільшенні навантажень на процеси за числом арифметичних операцій збільшується й ефективність. Також отримаємо приріст у швидкості при збільшенні кроків сітки.

3.3 Метод верхньої релаксації

Для розв'язання задачі даним методом при умові звичайного впорядкування невідомих задають початкове наближення у вузлах сітки та величину похибки, яка детермінує закінчення ітераційного процесу. Важливо на кожному вузлі враховувати поточні розрахунки у сусідніх вузлах, які вже були розраховані для даної ітерації. Цей момент змінює нашу схему кардинально наступним чином. Ми вже не в змозі розпаралелити одразу всю сітку. На це ще накладається проблема синхронізації блоків, мі не можем розрахувати першу діагональ, потім послідовно другу і першу, и так далі до закінчення ітераційного процесу. Це питання пропонується розв'язати наступним чином.

Аби надати максимальну паралелізацію такому доволі послідовному алгоритму розв'язувати вузли діагоналями. Таке припущення можна отримати аналізуючи формулу обчислення кроку ітерації. Діагоналі проходять наступним чином:

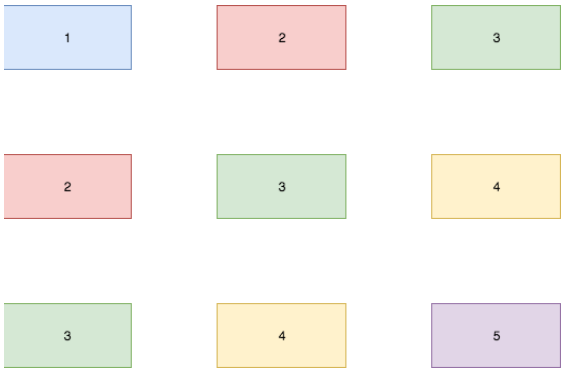

Рис. 3.3.1 Діагональне впорядкування.

Кожен однаковий колір позначає лінії процесу. Номер у клітинці позначає належність вузла до будь якої з ліній. Позначимо кожну діагональ як окремий блок. Тож для коректних розрахунків ми маємо розрахувати наступну ітерацію для першої діагоналі. Другий крок першої ітерації розрахує перше наближення для другої діагоналі використовуючи початкове наближення більших діагоналей (з більшим номером) та першу ітерацію менших діагоналей (вже розраховану). Проблема синхронізації постає саме тут, на другому кроці першої ітерації ми не в змозі обчислити перший крок другої ітерації для першої лінії бо ми не можемо дочекатися повного обчислення другого блоку. Ми зможемо обрахувати перший крок другої ітерації під час третього кроку першої ітерації. Таким чином отримаємо хвильовий ефект, що надає нам дуже гарну оптимізацію. Хвилі такого процесу будуть виглядати наступним чином:

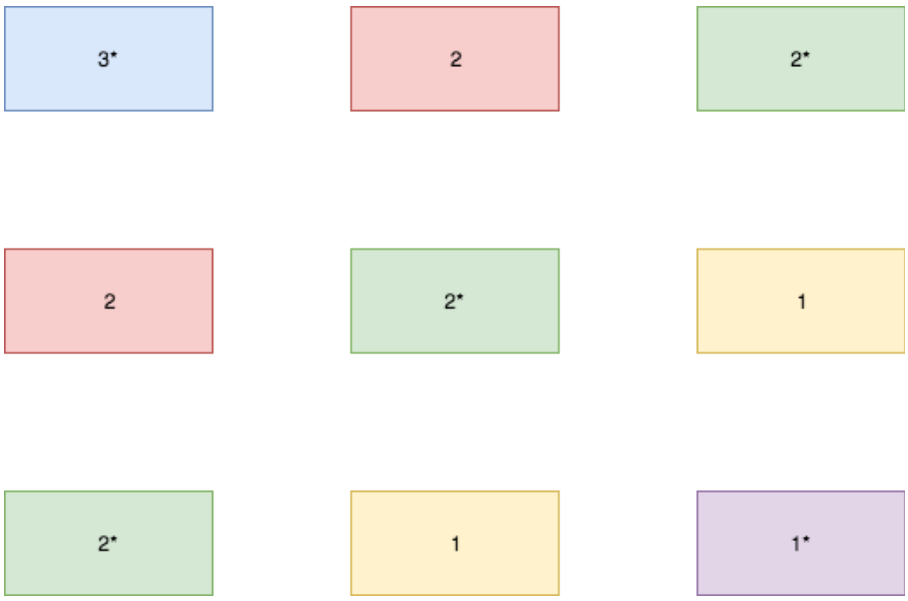

Рис. 3.3.2 Хвилі діагонального впорядкування.

Тут позначені ітерації які будуть паралельно обчислюватися, зірочкою позначені ті вузли що будуть обраховуватися на деякому кроці, відповідний номер позначає ітерацію які вони будуть обчислювати в тому чи іншому вузлі. Щоб досягти паралельного обчислення потрібно опрацювати перші декілька кроків. Таким алгоритмом ми скоротимо витрати на синхронізацію блоків, та лишимося можливості оперувати одразу всіма вузлами. Це питання

вирішимо наступним чином. Послідовно викличемо алгоритм для не парних блоків та для парних. Це трохи зменшить час виконання.

Приведемо характеристики такого алгоритму, для роботи під час навантаження по всіх вузлах:

$$T_p = \frac{MQt}{p}$$

Час виконання такого алгоритму можна зменшити наступним чином – зменшити кількість арифметичних операцій, чи підвищуючи кількість процесорів.

$$K_e = Q$$

Бачимо, що отриманий коефіцієнт ефективності лінійно пропорційний до кількості вузлів, чим більше отримаємо сітку для обчислень тим більш ефективніший буде паралельний алгоритм.

Вочевидь, що описана організація обчислень для розв’язання сіткових рівнянь методом верхньої релаксації чи Річардсона може бути ефективно реалізована на MIMD – машині зі всіма можливими структурами міжпроцесорних зв’язків.

3.4. Програмна реалізація та чисельні експерименти

Для реалізації використовуються стандартні обчислювальні процедури (множення матриці, розв’язування трикутних систем тощо), що реалізовані у відомих бібліотеках програм, наприклад ALGLIB, CUSPARSE, CUSP, Paralution. Для зберігання матриць застосовано звичайного вигляду вектори з стандартного набору шаблонів та звичайні вказівники.

Розглянемо більш детально програмну реалізацію алгоритму, а саме роботу з GPU. Основними блоками операцій, що виконуються з GPU є:

1. виділення пам'яті для змінних;
2. копіювання даних на GPU;
3. запуск обчислень;
4. копіювання результатів в оперативну пам'ять;
5. звільнення пам’яті GPU.

В роботі показано результати програмної реалізації для архітектур: 1CPU, 1 CPU + 1 GPU.

Розрахунки апробовано на вузлі суперкомп'ютера Inparcom-G, який має наступні характеристики:

- Процесори: 2 Xeon 5606 (4 ядра з частотою 2.13 ГГц);
- Графічні прискорювачі: 2 Tesla M2090 (6 Гб пам'яті);
- Об'єм оперативної пам'яті: 24 Гб;
- Комунікаційне середовище: InfiniBand 40 Гбіт/с (з підтримкою GPUDirect), Gigabit Ethernet.

Також на вузлах встановлена бібліотека MKL 10.2.6 та CUDA починаючи з версії 3.2.

Для модельної задачі:

$$\Delta u = 2\sin(y) - x^2\sin(y)$$

$$u = x^2\sin(y) + 1$$

Додаткові умови на границі збираються з реальних значень функції в тих точках.

В таблицях 1. - 3. Показано часи виконання методів на відповідних архітектурах, при кроках розбиття, що рівні 5, 10, 25. У таблицях рядок «Загальний час» - це час виконання всієї програми. Рядок «Головний цикл» - час виконання безпосередньо самого методу. Всі часи приводяться в секундах.

	Метод Річардсона		Метод верхньої релаксації	
	1 CPU	1 CPU + 1 GPU	1 CPU	1CPU + 1 GPU
Головний цикл	0,0015	0,0022	0,0174	0,018
Загальний цикл	0,0024	2,68285	0,085108	0,587721

Табл. 3.4.1.Часові характеристики роботи методів для розбиття з кроком 5

	Метод Річардсона		Метод верхньої релаксації	
	1 CPU	1 CPU + 1 GPU	1 CPU	1CPU + 1 GPU
Головний цикл	0,01355	0,0049	0,172671	0,03865
Загальний цикл	0,032	2,82	0,0846629	0,632961

Табл. 3.4.2. Часові характеристики роботи методів з кроком 10

	Метод Річардсона		Метод верхньої релаксації	
	1 CPU	1 CPU + 1 GPU	1 CPU	1CPU + 1 GPU
Головний цикл	0,734045	0,07132	0,87493	0,133061
Загальний цикл	6,6208	8,8451	33,807	33,6342

Табл. 3.4.3 Часові характеристики роботи методів для розбиття з кроком 25

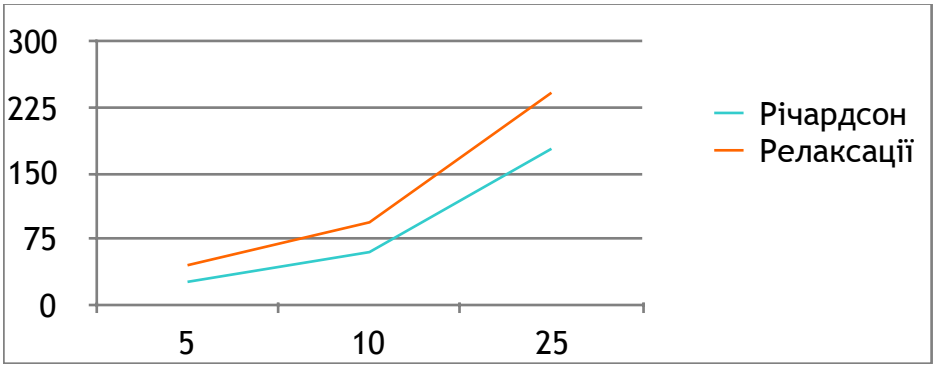

Рис. 3.4.1. Графік кількості ітерацій в методах залежно від кроку розбиття

На рис. 3.4.1 показано графік залежності виконуваних методом ітерацій в залежності від кроку розбиття сітки.

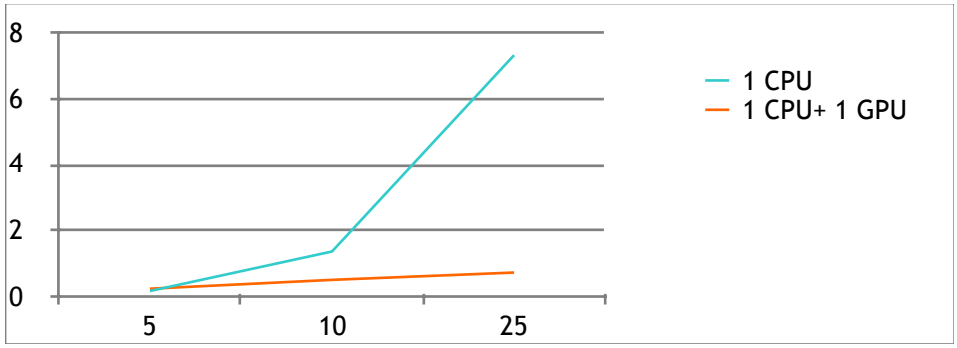

Рис. 3.4.2. Час виконання Річардсона на різних архітектурах.

На рис. 3.4.2. показано графік залежності часу виконання методу Річардсона на різних архітектурах від кроків в розбитті. На рис. 3. показано залежність прискорення від кількості кроків.

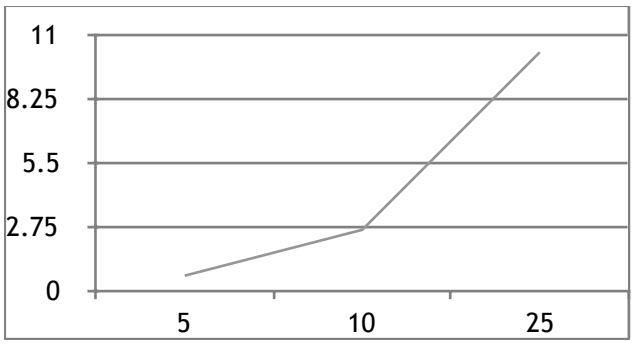

Рис. 3.4.3. Прискорення методу Річардсона в залежності від кроків.

Такий ефект методу Річардсона отримуємо за рахунок дуже гарної можливості до паралелізму в основних формулах.

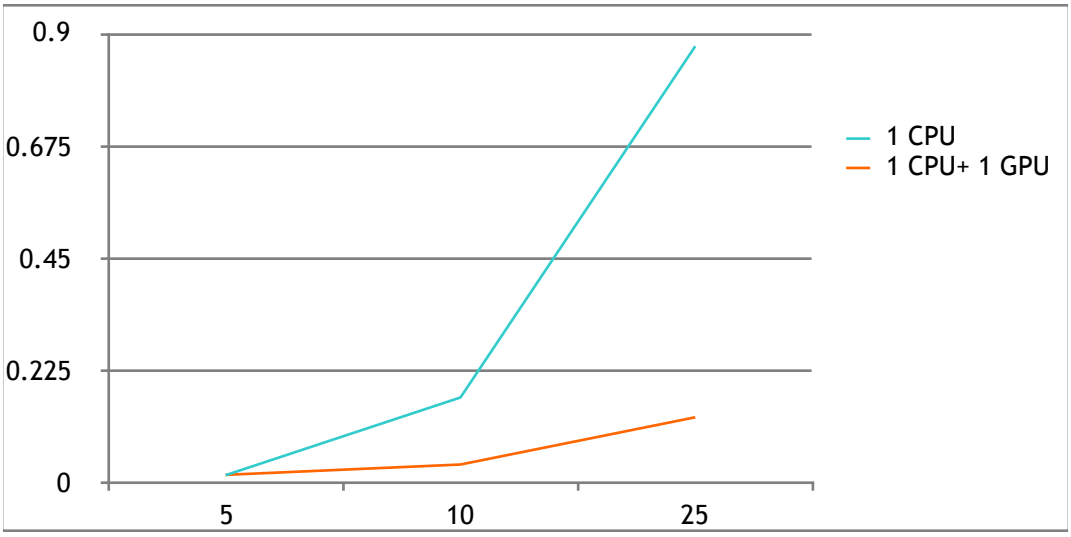

Рис. 3.4.4. Час виконання методу верхньої Релаксації на різних архітектурах

На рис. 3.4.4. показано графік залежності часу виконання методу верхньої релаксації на різних архітектурах від кроків в розбитті. На рис. 3.4.5. показано залежність прискорення від кількості кроків

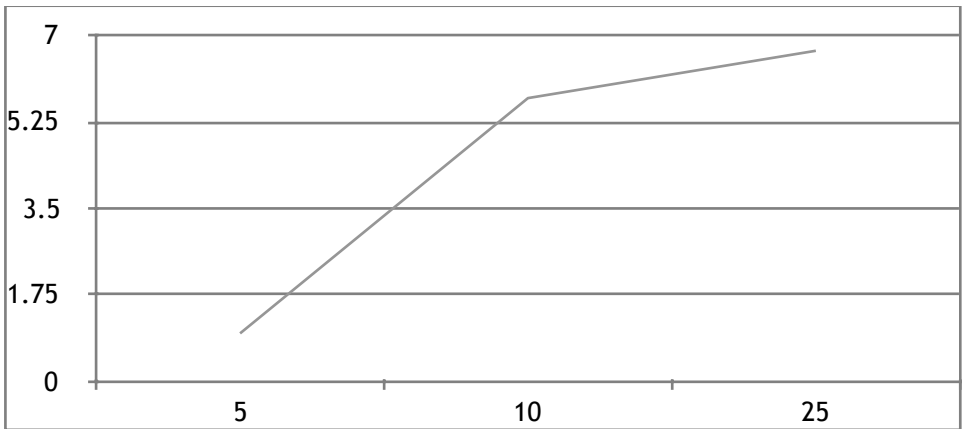

Рис. 3.4.5. Прискорення методу верхньої релаксації в залежності від кроків.

ЧАСТИНА 4. ПАРАЛЕЛЬНІ АЛГОРИТМИ ТА ЧИСЕЛЬНІ ЕКСПЕРИМЕНТИ АРХІТЕКТУРИ 1/2 CPU + 2 GPU

4.1 Основні положення до паралелізації до архітектури з 2 GPU

Для архітектури з 2 GPU паралелізація задачі змінюється у декількох напрямках. По перше задачу потрібно розбити на блоки які картки можуть опрацьовувати незалежно одна від одної. По друге збільшується кількість попередньої обробки та постобробки данних вдвічи. Також з'являється проблема синхронизації пам'яті між картками.

Виокремлюючи підзадачі для кожної картки ми розбиваємо логіку та дані, тобто нема сенсу займати пам'ять кожної відеокартки усіма необхідними даними для задачі. Кожній картці потрібна лише її частина даних. З розділенням даних постає задача синхронізації пам'яті карток.

4.2 Розбиття даних для архітектури 2-х GPU методу Річардсона

Розглянемо двовимірну сітку з архітектури для 1 GPU. У такій задачі на одну картку подається весь набір даних - параметри системи, коефіцієнти правої та лівої частини, пам'ять для зберігання отриманих результатів.

Усі ці значення можна представити у вигляді матриць N * N (у двовимірному випадку), а зважаючи на те, що у роботі ми використовуємо звичайну крестову різницеву схему то для обчислень кожної клітини нам потрібно лише сусідні елементи.

Розбити матрицю на 2 частини (для 2-х карток) можна 3-ма способами [13]:

- горизонтально
- вертикально
- діагонально

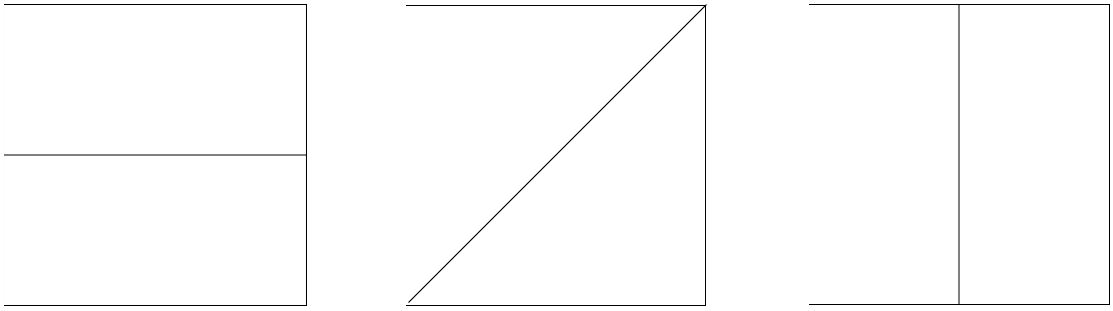

Рис. 4.2.1.Способи розбиття матриць на 2 частини

Горизонтальний та вертикальний розріз змінює спосіб індексування даних, проте можна їх розглядати як еквівалентні. Для можливості окремо обчислити весь поділений блок даних кожній частини потрібні додаткові N елементів з іншої частини по якій і проходить розріз.

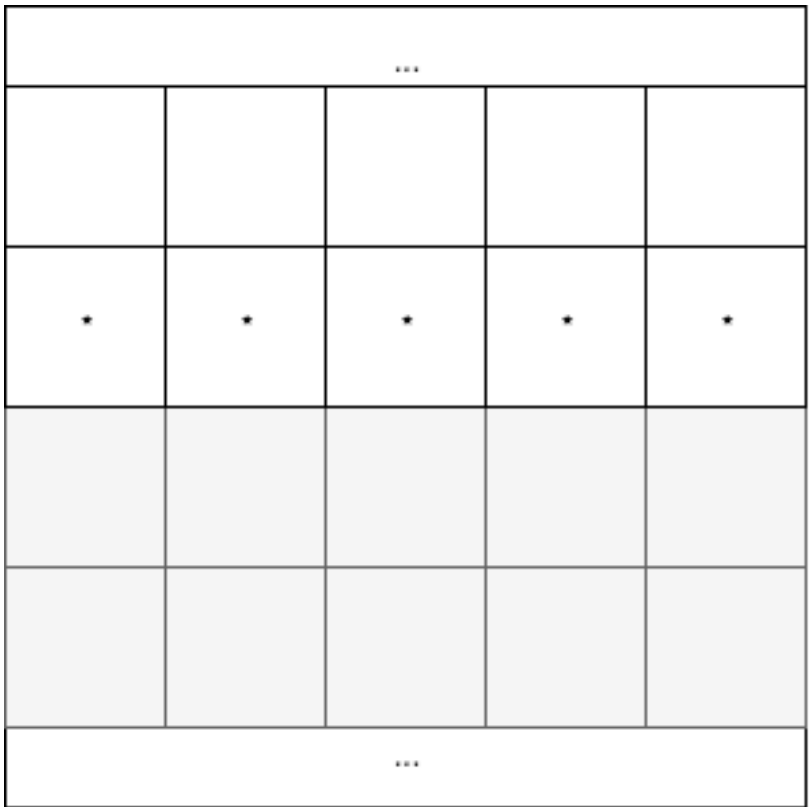

Рис. 4.2.2. Вузли матриці для поділення навпіл

На рис. 4.2.2 можна побачити як нижня частина виокремлена і для повних розрахунків потребує додаткового слою з сусіднього блоку. Таким чином весь блок необхідних даних для однієї картки будет не N * N, а N * ((N / 2) + 1). Індексація майже не змінюється, бо вона буде мати той самий характер як і в однокартковій архітектурі.

Взявши діагональний варіант розбиття елементів внутрішнього блоку буде (N + 1) * N / 2. Та в додачу до цього в нас з’являється діагональні елементи які теж потрібно обрахувати. Таким чином отримуємо 3 блоки даних (N + 1) * N / 2, (N + 1) * N / 2, N + 2 * (N - 1). Задача синхронізації в такому випадку буде ще важчою бо з кожного блоку потрібно в кожен інший скопіювати елементи.

У такому випадку в нас є 3 блоки які ми можемо обраховувати паралельно, а після кожної операції синхронізувати всі граничні точки розрізів (блоки з зірочками у центрах). Враховуючи архітектуру з двома картками ми будемо навантажувати одну з них більше а також потрібно дуже обережно синхронізувати блоки.

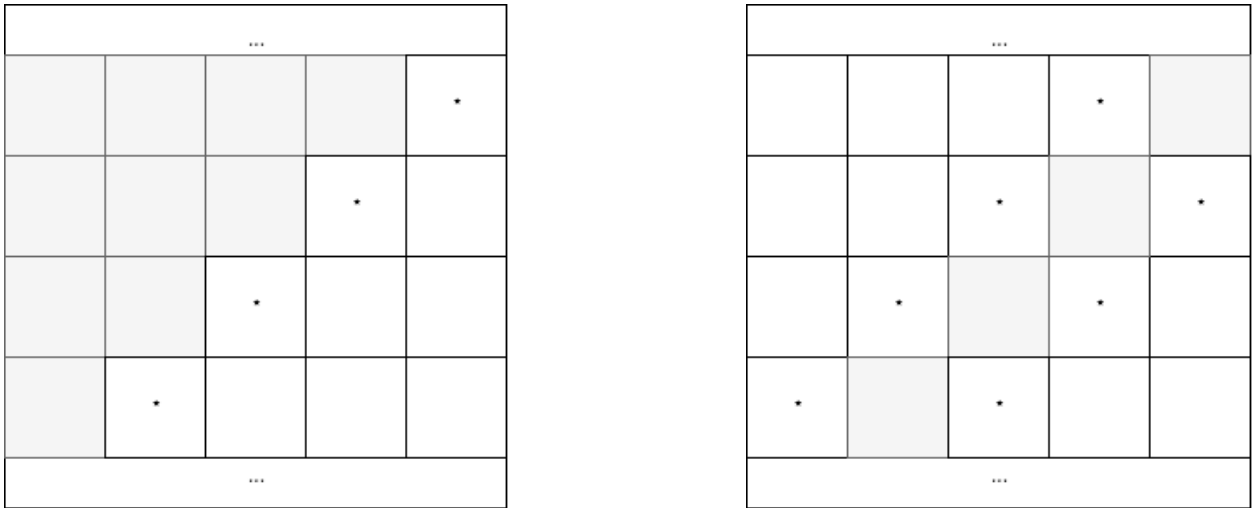

Рис. 4.2.3. Вузли матриці для діагонального трьохблочного поділення

Розглянемо інший діагональний варіант з побудовою лише 2 блоків. Нехай ми включимо діагональні елементи у кожен з трикутних блоків і будемо обчислювати діагональні елементи лише на одній картці (не дублюючи операції) а потім копіювати отримані значення.

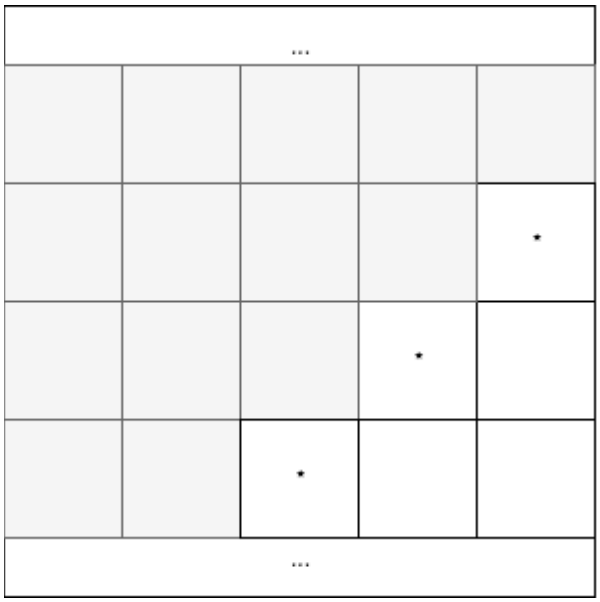

Рис. 4.2.4. Вузли матриці для діагонального двоблочного поділення

Таким чином отримуємо 2 блоки (N + 1) * N / 2 + (N - 1). Порівнючи з трьохблочним діагональним розбиттям ми зберігли N комірок та 1 операцію синхронізації. Порівнюючи з розбиттям навпіл ми зберігли N / 2 комірок пам’ті що виявляється не таким суттєвим покращенням. Залишилось відпо вісти на питання як зберігати такі блоки у пам’яті картки. Якщо для однокарткової архітектури ми зберігали матрицю розрівнюючи по рядках кожні N елементів, то у трьокутному блоці ми можемо розрівняти також по рядкам, але кожен наступний ряд буде займати на 1 комірку менше.

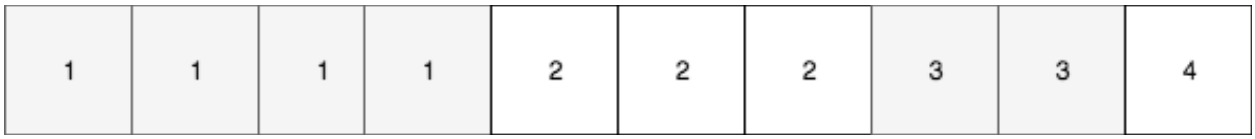

Рис. 4.2.5. Перетворення діагонального блоку до одновимірного масиву

Тож розбивши даним описаним вище чином ми маємо на кожній картці лише необхідні данні і збережемо пам’ять. Такий підхід буде достатньо легко масштабуватись з ростом кількості GPU - кожен блок можно бити навпіл і додавати синхронізації з сусідними блоками у випадку поділення навпіл. Для трикутного блоку при масштабуванні буде все більше турбувати проблема індексації. Тож з цього можна зробити висновок, що для подальшої роботи краще взяти метод поділення навпіл.

Синхронізація - велике питання для подібної задачі. Для горизонтального розбиття після кожного ітеративного кроку нам потрібно синхронізувати граничні слої по розрізу. У архітектурі з 1 GPU весь комплекс робот з пам’яттю ми виконували засобами CPU директив на початку та по закінченні роботи основного модуля. Але тепер нам потрібно робити це після кожної ітерації і використання CPU для цього не підходить, бо копіювання з пам’яті картки до оперативної пам’ті не надто швидке у загальному випадку. Окрім того після копіювання потрібно виокремити потрібні рядки з матриць та повернути їх на сусідні картки.

Для такого роду задач не погано було б напряму скопіювати дані з картки на картку, і CUDA з версії compute 2.0 це дозволяє, для цього потрібно

включити доступ копіювання між картками та скопіювати весь блок даних або лише потрібну частку.

Внутрішня робота поділеного блоку ідентична до роботи цілого блоку архітектури 1 CPU + 1 GPU, тобто ми маємо ті самі розподіли на блоки сіток і ниток як і в попередній задачі.

Розпишемо отриманий метод поділення навпіл:

1. визначити змінні для зберігання масивів на 2 картках. Оптимальні параметри, поточні обчислення внутрішнього та зовнішнього блоку лише тих частин за які дана картка відповідальна
2. записати в визначені змінні початкові значення та необхідні параметри, розрівнюючи матриці у одновимірні масиви
3. почати цикл ітерацій
    1. запустити на кожній картці ядро шаблону та ядро переносу даних
    2. скопіювати попарно з кожної на кожну картку отримані дані на перетинах зрізу
    3. повторити цикл ітерацій до закінчення

4.3 Розбиття даних для архітектури 2-х GPU методу верхньої релаксації

Розглядаючи алгоритм розподілу даних між картками для методу верхньої релаксації одразу стає зрозуміло, що застосувати алгоритм від методу Річардсона буд не можливо за рахунок властивості метода використовувати вже обраховані значення точок на поточному кроці.

Розглянемо можливе поділення даних використовуючи схему дії для архітектури 1 CPU + 1 GPU. В нас є 2 етапи роботи обчислень:

1. обчислення перших кроків починаючи з одного кута доки усі внутрішні точки не будуть обраховані хоча б 1 раз. Для цього потрібно 2*(N-2)-1 операцій які проходять по кожній з діагоналей матриці наступним чином:

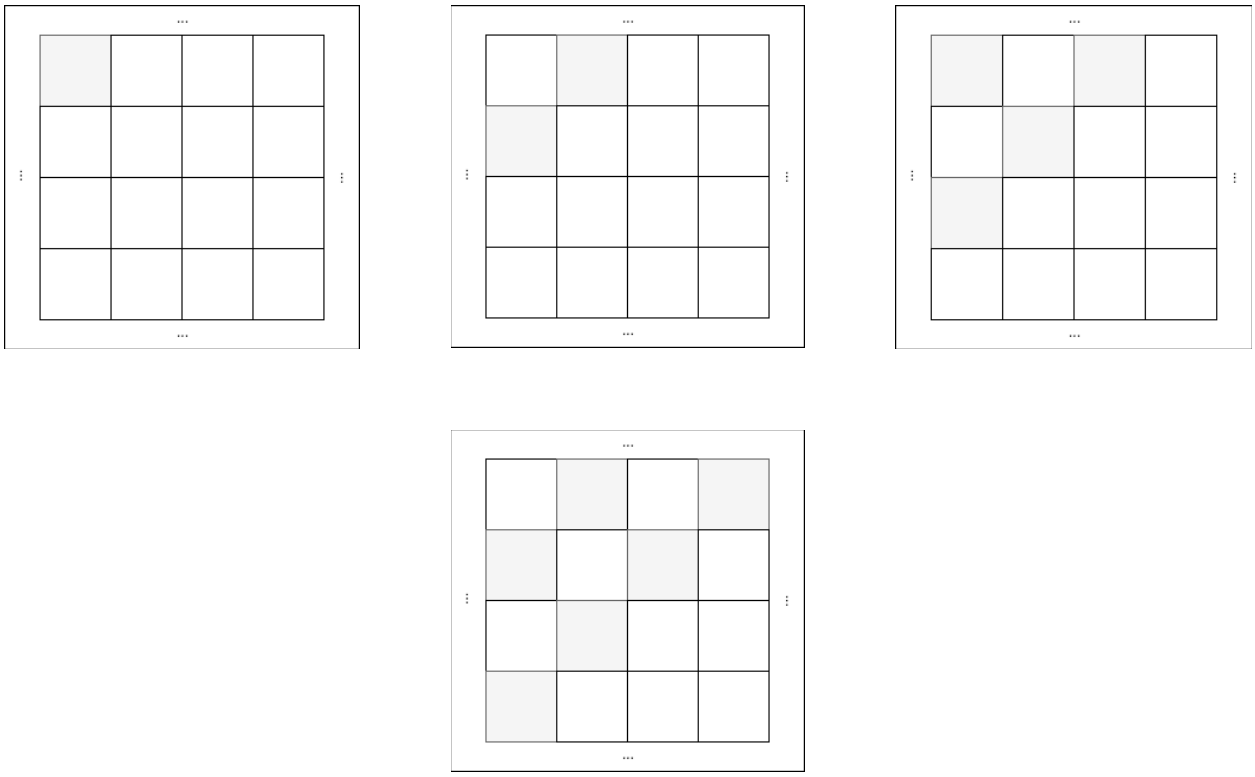

Рис. 4.3.1. - 4 кроки першого етапу ітерації

2. обчислення наступних кроків вже можна розпаралелити таким чином що за одну ітерацію циклу ми проводимо обчислення чорних вузлів а потім червоних.

Більшість операцій звичайно припадає на другий крок методу, тож почнемо розглядати поділенням для нього припускаючи, що попередню обробку для нас проведено на одній з карток чи навіть на самому CPU.

Першим з варіантів є діагональне розбиття запропоноване для методу Річардсона цієї ж архітектури. Таким чином ми за 1 крок ітерації методу будемо проводити 4 виклика ядра на картці - червоне для першої картки, червоне для другої картки, чорне для першої картки, чорне для другої картки. Зважаючи на те, що виклики ядер для блоків поділених на 4 будуть вимагати збільшенної розмірності задачі для отримання суттєвих прискорень.

Розбиття навпіл буде мати таку саму проблему 4 блоків, а також підвищує складність індексації на картці. Розглянемо інші варіанти розбиття.

Нехай ми будемо мати на кожній картці весь блок даних та кожна з карток буде відповідати за чорний чи червоний крок ітерації. Таким чином ми

ділимо кількість операцій між картками вдвічи. Такий підхід гарний у обчисленні бо добре інтегрується з вже існуючою схемою. Ми будемо максимально використовувати особливості архітектури з 2 GPU а також не витрачати додаткові операції для підтримання структури блоків.

Розглядаючи червоно-чорний спосіб поділення даних останнім питанням є спосіб синхронізувати обчислені точки між картками. Нагадаємо, що в архітектурі 1 CPU + 1 GPU нам після кожної ітерації потрібно отримати дані на хост для обчислення максимальної похибки в кожній з точок тож тут ми можемо скористуватись логічним копіюванням на хості. А саме після кожної ітерації мі копіюємо 2*(N-2)*(N-2) з карток на хост, синхронно злити червоні вузли з результату червоної картки та чорні вузли з чорної картки. Після цього ми відправляємо результат на обидві картки.

Розпишемо отриманий метод червоно-чорного поділення:

1. визначити змінні для зберігання масивів на 2 картках. Оптимальні параметри, поточні обчислення внутрішнього та зовнішнього блоку пересилаючи усі дані на обидві картки. Визначити відповідність ядер до червоних і чорних ітерацій
2. записати в визначені змінні початкові значення та необхідні параметри, розрівнюючи матриці у одновимірні масиви
3. провести первинне обчислювання для перших 2*(N-2)-1 ітерацій
4. почати цикл ітерацій
    1. запустити на відповідних картках червоне та чорне ядра
    2. скопіювати отримані ітерації на хост для обчислення критерію виходу з циклу а також для синхронізації результатів обчислень
    3. повторити цикл ітерацій

4.4 Використання архітектури 2 CPU + 2 GPU

Наступним кроком масштабування задачі є збільшення кількості опрацьовуючих ядер центрального процесора разом з графічними картками. Зважаючи на те що CPU лише робить підготовчі послідовні роботи та опрацьовує результати важко сказати як отримати від цього прискорення

обчислень, особливо якщо порівнювати лише чистий час виконання математичних обчислень.

Розглянемо таку архітектуру з наступної точки зору - головний цикл обчислень запустить 2 потока, які будуть опрацьовуватись різними ядрами процесору. Таким чином ми маємо 2 зв'язки потік + GPU. Додавши синхронізацію потоків наприкінці ітерації будемо певні що жоден потік не розсинхронує наші обчислення.

Збільшення кількості ядер з центрального процесору не надасть суттєвого прискорення, бо всі математичні обчислення виконуються на графічній картці, а частина задач CPU полягає лише у послідовному запуску ядер та контроль за синхронізацією після кожної ітерації. Прискорення отримане в результаті одночасного запуску ядер завдяки 2-м потокам буде мінімальне враховуючи тактову частоту центрального процесора.

4.5 Вплив збільшення розмірності задачі

Розглянемо випадок з 3-х розмірною сіткою та як зміниться алгоритм відносно неї. Окрім явного оновлення різницевої схеми першою задачею є спосіб зберігання даних на хості та пристрої. Якщо на хості ми можемо визначити n - вимірний вектор, матрицю, куб тощо, то для пристрою нам в будь якому разі потрібно переводити всю систему до 1-вимірного масиву. Тож індексація стане сложніша, синхронізація та обробка даних зміниться так само відповідно до нових індексів.

За всіма іншими ознаками ми будемо мати еквівалентну до попередньої задачі.

4.6 Ефективність алгоритмів з багатьма GPU

Кількість арифметичних операцій, що виконує один GPU на і-му кроці алгоритму, оцінюється величиною:

$$O_p \cong \frac{1}{p}(M(N-2)^2 t_G + 2pNt_B)$$

покажемо це:

На *i*-му кроці алгоритмів на GPU виконується обчислення за різницевою схемою, тож ми маємо завжди сталу кілкість арифметичних операцій для

оглянутих методів. А кількість комірок для опрацювання маємо у внутрішньому блоці матриці. Після опрацювання нам потрібно переслати граничні елементи по розрізу.

Нехай гібридний комп'ютер реалізує архітектуру гіперкуб. Тоді для гібридного алгоритму мають місце наступні оцінки прискорення:

$$S_p = \frac{T_1}{T_p} = \frac{6t_c + \frac{(N-2)^2}{n_0} Mt_G}{6t_c + \frac{(N-2)^2}{n_0 p} Mt_G + 2pt_B log_2 p}$$

Для цього спочатку оцінемо величину T_1:

$$T_1 = q(6t_C + \frac{(N-2)^2}{n_0} Mt_G + t_1)$$

Це значення виходить з того скільки конкретних арифметичних операцій потрібно для обрахування наступної ітерації. Оскільки для обмінів між обчислювальними пристроями використовується алгоритм «дерева», то загальний час мультирозсилки одного блоку можна оцінити величиною $t_B log_2 p$. Таким чином маємо таку оцінку величини T_p:

$$T_p \cong q(6t_c + \frac{(N-2)^2}{n_0 p} Mt_G + 2pt_B log_2 p + t_1)$$

Оскільки на практиці час обміну блоком між CPU та GPU набагато менший, чим час потрібний на обчислення, а також обмін може виконуватися на фоні обчислень, то величиною t_1 можна знехтувати. Виходячи з отриманих формул отримуємо описану вище оцінку прискорення.

4.7 Програмна реалізація та чисельні експерименти

Для реалізації алгоритмів для подібної архітектури, якщо порівнювати з варіантом описаним у частині 3, з'являється розподіл даних та міжкарткове пересилання даних.

Розрахунки проводились на вузлі кластеру Inparcom-G, які мають наступні характеристики:

- Процесори: 2 Xeon 5606 (4 ядра з частотою 2.13 ГГц);
- Графічні прискорювачі: 2 Tesla M2090 (6 Гб пам'яті);

- Об’єм оперативної пам’яті: 24 Гб;
- Комунікаційне середовище: InfiniBand 40 Гбіт/с (з підтримкою GPUDirect), Gigabit Ethernet.

Також на вузлах встановлена бібліотека MKL 10.2.6 та CUDA 5.0.

Для модельної задачі:

$$\Delta u = 2\sin(y) - x^2\sin(y)$$

$$u = x^2\sin(y) + 1$$

В таблицях 1. - 3. Показано часи виконання методів на відповідних архітектурах, при кроках розбиття, що рівні 10, 20, 40, 160. У таблицях рядок «Загальний час» - це час виконання всієї програми. Рядок «Головний цикл» - час виконання безпосередньо самого методу. Всі часи приводяться в секундах. Похибка встановлена на рівні $1e^{-5}$. Для отримання більш достовірних результатів експерименти проводяться декілька разів і обчислюється усередковане значення часу.

	Метод Річардсона		Метод верхньої релаксації	
	1 GPU	2 GPU	1 GPU	2 GPU
Головний цикл	0,0049	0.0058	0,0386	0.097
Загальний цикл	2,82	3.27	10,2	13,78

Табл. 4.7.1.Часові характеристики роботи методів для розбиття з кроком 10

	Метод Річардсона		Метод верхньої релаксації	
	1 GPU	2 GPU	1 GPU	2 GPU
Головний цикл	0.00702	0.00694	0,13	0.18285
Загальний цикл	2.93	3.36	33,6342	35,78

Табл. 4.7.2.Часові характеристики роботи методів з кроком 20

	Метод Річардсона		Метод верхньої релаксації	
	1 GPU	2 GPU	1 GPU	2 GPU
Головний цикл	0.0102	0.00879	0.763	0.6372
Загальний цикл	3.17	3.44	60.2	63.23

Табл. 4.7.3.Часові характеристики роботи методів з кроком 40

	Метод Річардсона		Метод верхньої релаксації	
	1 GPU	2 GPU	1 GPU	2 GPU
Головний цикл	0.0405	0.0323	1.43	0.837
Загальний цикл	120	120	108.9	105.3

Табл. 4.7.4.Часові характеристики роботи методів з кроком 160

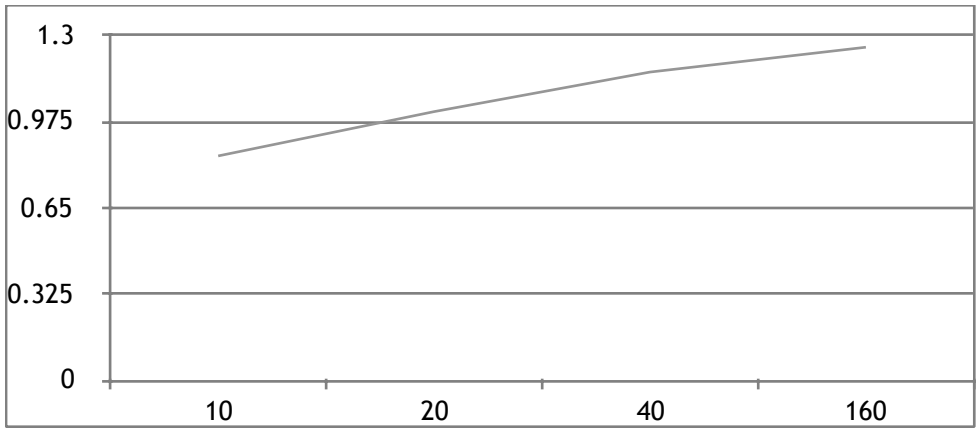

Рис. 4.7.5. Прискорення методу Річардсона в залежності від кроків

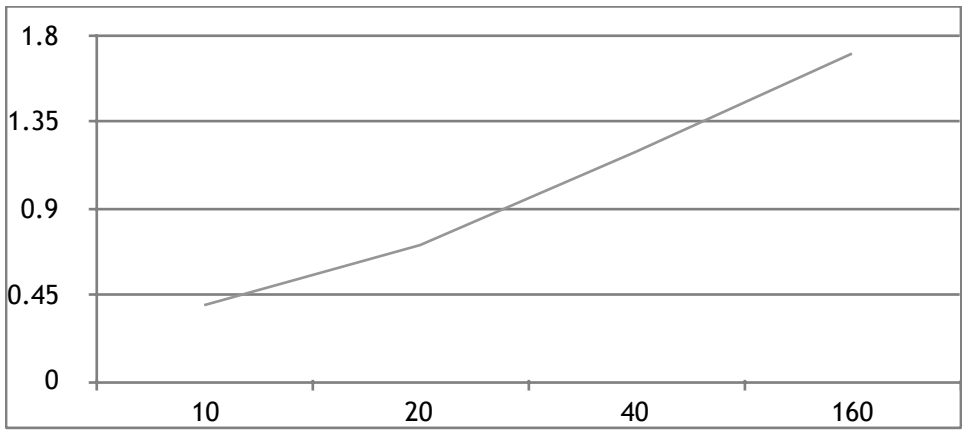

Рис. 4.7.6. Прискорення методу верхньої релаксації в залежності від кроків

Загальний цикл займає суттєво більше часу через обчислення всіх додаткових параметрів для розв’язання системи. А саме найбільшу частину часу займає обчислення власних чисел. Починаючи з бібліотеки cuda 8.0 для

розробників доступні методи знаходження власних чисел для щільних та розряджених матриць на графічних прискорювачах. Нажаль кластер Inparcom-G на даний момент має максимальну версію встановленої бібліотеки cuda 7.5 а тому розв’язання цієї підзадачі залишається на головному процесорі за допомогою бібліотеки ALGLIB.

Результати експерементального дослідження показують, що отриманий алгоритм дійсно покращує швидкість обчислень завдяки використання декількох графічних прискорювачів.

Висновки

В даній кваліфікаційній роботі отримані наступні результати:

Досліджено сучасні архітектури комп'ютерних систем, програмні інтерфейси і технології для розпаралелювання програм на комп'ютерах гібридної архітектури.

Розроблено та експериментально досліджено гібридні алгоритми ітераційних методів Річардсона та верхньої релаксації розв'язування СЛАР з використанням оптимального набору параметрів для архітектур 1 CPU + 1 GPU, 1 CPU + 2 GPU.

Створено відповідне програмне забезпечення, що використовує бібліотеки ALGLIB, MKL і технології CUDA. На мові CUDA запрограмовано власні обчислювальні ядра, що використовуються при обчисленнях і дають суттєве скорочення часу виконання одної ітерації алгоритму на графічних процесорах.

Отримані в даній роботі результати показують, що використання графічних прискорювачів для розв'язання СЛАР може бути ефективним засобом прискорення обчислень, а збільшення кількості GPU для розв'язання таких задач покращує ефективність для великих розмірностей.

Список використаної літератури

1. Технологии параллельного программирования [Електронний ресурс] // Лаборатория Параллельных Информационных Технологий, НИВЦ МГУ. – 2009. – Режим доступу :http://parallel.ru/

2. Химич А.Н., Молчанов И.Н., Мова В.И. и др. «Численное программное обеспечение MIMD-компьютера Инпарком» - Киев: Наукова думка, - 2007. - 222с.

3. Технологічна платформапрограми «Університетський кластер». – Режим доступу :http://unihub.ru/

4. Дістанційна платформа для національного відкритого університету– Режим доступу : http://intuit.ru/

5. CUDA C Programming Guide Version 4.2. — Santa Clara: Nvidia, 2012. — 173 p.

6. Самарский А.А., Николаев Е.С. Методы решения сеточных уравнений. – М.: Наука. 1978. – 592 с.

7. Jason Sanders, Edward Kandrot An Introduction to General Purpose GPU Programming – Addison Wesley.

8. Tristan Perryman – Runtime compilation with NVIDIA CUDA as a Programming tool – Imperial College London Department of Computing

9. Химич А.Н., Чистякова Т.В., Баранов А.Ю. Автоматический адаптивный решатель СЛАУ для гибридных систем – Міжнародна конференція "Високопродуктивні обчислення" HPC-UA’2011 (Україна, м. Київ, 12-14 жовтня 2011 року)

10. Химич А.Н. Параллельные алгоритмы решения задач вычислительной математики / Химич А.Н., Молчанов И.Н., Попов А.В. и др. – Киев: Наукова думка, - 2008. – 248 с.

11. Медведев А.В., Свешников В.М., Турчановский И.Ю - Распараллеливание решения сеточных уравнений на квазиструктурированных сетках с использованием графических ускорителей

12. Матвеева Н.О. – Решение эллиптического дифференциального уравнения в частных производных на графическом процессоре в технологии CUDA

13. Сидорук В.А. Гібридні трикутні ітераційні алгоритми на основі одно вузлової архітектури // Збірник матеріалів міжнародної наукової координаційної наради «Інформаційні проблеми комп'ютерних систем, юриспруденції, енергетики, економіки, моделювання та управління» (ICSM-2014). – Тернопіль, 2014

Додаток А. Керуюче ядро методу Річардсона 1 CPU + 1GPU

```
__global__ void myshab(double *temp, int n, double *all) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = index + N + 1 + 2 * (int) (index / (N - 2));
  if (index < n) {
    temp[index] = -4 * all[lindex] + all[lindex - N] + all[lindex + N] + all[lindex - 1]
+ all[lindex + 1];
  }
}

// B, Shablon, Tau, firstAppr, iteration number
__global__ void mykernel(double *a, double *b, double *c, double *d, int n, int i,
double *all) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = index + N + 1 + 2 * (int) (index / (N - 2));
  if (index < n) {
    d[index] = (-a[index] + b[index]) * c[i] + d[index];
    all[lindex] = d[index];
  }
}

  for (int i = 1; i < maxIter + 1; ++i) {
    myshab <<<N - 2, N - 2>>>(d_b, N * N - 4 * N + 4, d_g);
    mykernel <<<N - 2, N - 2>>>(d_a, d_b, d_c, d_d, N * N - 4 * N + 4, i, d_g);
  }
```

Додаток Б. Керуюче ядро методу верхньої релаксації 1 CPU + 1 GPU

```
__global__ void mykernel(double *rightSide, double wOpt, double *fa, double
*diff, int n, double *all, int i, int j,
 int litN) {
 int index = threadIdx.x;
 int row = (i - j + 1) / 2;
 int index1 = row * litN + (((j - i) == 1)? j : litN) - 1 + index * (litN - 1);
 int lindex = index1 + N + 1 + 2 * (int) (index1 / (N - 2));
 if (index1 < n) {
  fa[index1] = (-rightSide[index1] + all[lindex - N] + all[lindex + N] + all[lindex -
1] + all[lindex + 1] -
    4 * (1 - 1. / wOpt) * all[lindex]) * wOpt / 4.;
  diff[index1] = fa[index1] - all[lindex];
  all[lindex] = fa[index1];
 }
}
__global__ void my_red_black_kernel(double *rightSide, double wOpt, double
*fa, double *diff, int n, double *all,
 int litN, int first) {
 int index = threadIdx.x + blockIdx.x * blockDim.x;
 int lindex = index + N + 1 + 2 * (int) (index / (N - 2));
 int row = (int)(index / litN);
 int str = index % litN;
 if (index < n && ((row + str) % 2 == first)) {
  fa[index] = (-rightSide[index] + all[lindex - N] + all[lindex + N] + all[lindex - 1]
+ all[lindex + 1] -
    4 * (1 - 1. / wOpt) * all[lindex]) * wOpt / 4.;
  diff[index] = fa[index] - all[lindex];
  all[lindex] = fa[index];
 }
}

 do {
  my_red_black_kernel<<<N - 2, N - 2>>>(d_rs, wOpt, d_fa, d_diff, N * N - 4 *
N + 4, d_all, n, 1);
  my_red_black_kernel<<<N - 2, N - 2>>>(d_rs, wOpt, d_fa, d_diff, N * N - 4 *
N + 4, d_all, n, 0);
  cudaMemcpy(diff, d_diff, size * (N * N - 4 * N + 4), cudaMemcpyDeviceToHost);
```

```
  maxDiff = findMaxInVector(diff, N * N - 4 * N + 4);
  ++k;
} while (maxDiff > eps);
```

Додаток В. Керуюче ядро методу Річардсона 1 CPU + 2 GPU

```
__global__ void myshab(double *temp, int n_row, int n_col, int plus, double *all)
{
  int col = threadIdx.x + blockIdx.x * blockDim.x;
  int row = threadIdx.y + blockIdx.y * blockDim.y;
  int index = row * n_col + col;
  if (index >= n_row * n_col) return;
  int lindex = index + N + 1 + 2 * (int) (index / (N - 2)) + (N * plus);
  temp[index] = -4 * all[lindex] + all[lindex - N] + all[lindex + N] + all[lindex - 1]
+ all[lindex + 1];
}

// B, Shablon, Tau, firstAppr, iteration number
__global__ void mykernel(double *a, double *b, double *c, double *d, int n_row,
int n_col, int plus, int i,
  double *all
) {
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int index = row * n_col + col;
    if (index >= n_row * n_col) return;
    int lindex = index + N + 1 + 2 * (int) (index / (N - 2)) + (N * plus);
    d[index] = (-a[index] + b[index]) * c[i] + d[index];
    all[lindex] = d[index];
}

 for (int j = 1; j < maxIter + 1; ++j) {
    for (size_t i = 0; i < GPU; i++) {
      cudaSetDevice(cudas[i]);
      int plus = i * ((int)(N / 2) - 1);
      myshab <<<numBlocks, threadsPerBlock>>>(d_b[i], n_row, n_col, plus,
d_g[i]);
      mykernel <<<numBlocks, threadsPerBlock>>>(d_a[i], d_b[i], d_c[i], d_d[i],
n_row, n_col, plus, j, d_g[i]);
    }
    for (size_t i = 0; i < GPU; i++) {
      int index = N * ((int)(N / 2) + (i == 0 ? -1 : 0));
      // copy all to other.
```

```
    cudaMemcpy(
      &d_g[i == 0 ? 1 : 0][index],
      &d_g[i][index],
      size * N,
      cudaMemcpyDefault);
  }
}
```

Додаток Г. Керуюче ядро методу верхньої релаксації 1 CPU + 2 GPU

```
__global__ void mykernel(double *rightSide, double wOpt, double *fa, double
*diff, int n, double *all, int i, int j,
 int litN) {
 int col = threadIdx.x + blockIdx.x * blockDim.x;
 int row = (i - j + 1) / 2;
 int index = row * n_col + col;
 int index1 = row * litN + (((j - i) == 1)? j : litN) - 1 + index * (litN - 1);
 int lindex = index1 + N + 1 + 2 * (int) (index1 / (N - 2));
 if (index1 < n) {
  fa[index1] = (-rightSide[index1] + all[lindex - N] + all[lindex + N] + all[lindex -
1] + all[lindex + 1] -
    4 * (1 - 1. / wOpt) * all[lindex]) * wOpt / 4.;
  diff[index1] = fa[index1] - all[lindex];
  all[lindex] = fa[index1];
 }
}
__global__ void my_red_black_kernel(double *rightSide, double wOpt, double
*fa, double *diff, int n, double *all,
 int litN, int first) {
 int index = threadIdx.x + blockIdx.x * blockDim.x;
 int lindex = index + N + 1 + 2 * (int) (index / (N - 2));
 int row = (int)(index / litN);
 int str = index % litN;
 if (index < n && ((row + str) % 2 == first)) {
  fa[index] = (-rightSide[index] + all[lindex - N] + all[lindex + N] + all[lindex - 1]
+ all[lindex + 1] -
    4 * (1 - 1. / wOpt) * all[lindex]) * wOpt / 4.;
  diff[index] = fa[index] - all[lindex];
  all[lindex] = fa[index];
 }
}

 do {
  for (size_t i = GPU - 1; i >=0; i++) {
   cudaSetDevice(cudas[i]);
   my_red_black_kernel<<<N - 2, N - 2>>>(d_rs, wOpt, d_fa, d_diff, N * N - 4
* N + 4, d_all, n, cudas[i]);
```

```
    }
  for (size_t i = 0; i < GPU; i++) {
      int index = N * ((int)(N / 2) + (i == 0 ? -1 : 0));
      // copy all to other.
      cudaMemcpy(
        &d_fa[i == 0 ? 1 : 0],
        &d_fa[i],
        size * (N * N - 4 * N + 4),
        cudaMemcpyDefault);
      cudaMemcpy(
        &d_all[i == 0 ? 1 : 0],
        &d_all[i],
        size * (N * N),
        cudaMemcpyDefault);
      cudaMemcpy(
        &d_diff[i == 0 ? 1 : 0],
        &d_diff[i],
        size * (N * N - 4 * N + 4),
        cudaMemcpyDefault);
    }

   cudaMemcpy(diff, d_diff, size * (N * N - 4 * N + 4), cudaMemcpyDeviceToHost);
    maxDiff = findMaxInVector(diff, N * N - 4 * N + 4);
    ++k;
  } while (maxDiff > eps);
```