Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____*Комп'ютерних наук*_____
(повна назва)
Кафедра _____*Системотехніки*_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

_____другий (магістерський)_____
(освітньо-кваліфікаційний рівень)

_____*Дослідження методів пошуку оптимальних шляхів*_____
_____*на графах з використанням GPU*_____
(тема роботи)

Виконав: студент VI курсу, групи *СПРм-19-2*
_____Каяфюк Р. В._____
(прізвище, ініціали)
Спеціальність __122– Комп'ютерні науки___
_____
(шифр і назва напряму, спеціальності)
Тип програми____освітньо-наукова_____
(освітньо-професійна або освітньо-наукова)
Освітня програма __системне проектування_
(повна назва освітньої програми)
Керівник _____доц. каф. ст Панкратов О.В.__
(прізвище, ініціали)

Допускається до захисту
Зав. кафедри СТ _____ ___*проф. Гребеннік І.В.*___
(підпис) (прізвище, ініціали)

2021 р

Харківський національний університет радіоелектроніки

Факультет_____*Комп'ютерних наук*_____

Кафедра_____*Системотехніки*_____

Рівень вищої освіти_____другий (магістерський)_____

Спеціальність_____*122 – Копьютерні Науки*_____
(код і повна назва)

Тип програми _____освітньо-наукова_____
(освітньо-професійна або освітньо-наукова)

Освітня програма_____Системне проектування_____
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«____»_____ 20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Студентові *Каяфюку Родіону Віталійовичу*

(прізвище, ім'я, по батькові)

1. Тема роботи *Дослідження методів пошуку оптимальних шляхів на графах з використанням графічного процесора*

затверджена наказом по університету від "29" *березня* **2021р. №** *389СТ*

2. Термін подання студентом роботи до екзамініціонної комісії _____

3. Вихідні дані до роботи *Дослідити можливості застосування алгоритмів пошуку мінімального шляху на графі використовуючи апаратні можливості графічних процесорів. Перелік використовуваних програмних засобів: ОС Microsoft Windows 10, Інтегроване середовище розробки Microsoft Visual Studio 2017, MySQL Workbench 8.0, Jupyter Notebook, Nvidia CUDA Compiler, Nvidia Toolkit 10.0, пакет Anaconda 3. Технічне забезпечення: IBM-сумісний ПК з МП Pentium Gold 4415U та GPU Nvidia GeForce 940 MX.*

4. Перелік питань, що потрібно опрацювати в роботі

4.1 Вступ. 4.2 Аналіз проблемної області. 4.3 Вибір алгоритмів, технології, архітектури та програмних засобів. 4.4 Розробка тестуючої системи 4.5 Підготовка до роботи з GPGPU CUDA. 4.6 Розробка та оптимізація реалізації мурашиного алгоритму з використанням CUDA. 4.7 Представлення результатів роботи з алгоритмом Мурашиної Колонії 4.8 Розробка та оптимізація реалізації Хвильового алгоритму з використанням CUDA. 4.9 Представлення

результатів роботи з Хвильовим алгоритмом 4.10 Розробка та оптимізація реалізації алгоритму Беллмана-Форда з використанням CUDA. 4.11 Представлення результатів роботи з алгоритмом Беллмана-Форда 4.12 Представлення загальних результатів дослідження

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, плакатів)

5.1 Задача пошуку мінімального шляху. 5.2 General Purpose GPU. 5.3 Математична постановка задачі. Методи вирішення. 5.4 Алгоритм Мурашиної Колонії. 5.5 Хвильовий алгоритм 5.6 Алгоритм Беллмана-Форда 5.8 GPGPU CUDA. 5.9 Порівняння швидкодії GPU та CPU. 5.10 Структурна схема паралельних реалізацій алгоритму Мурашиної Колонії. 5.11 Основні віхи розробки мурашиного алгоритму на GPU 5.12 Процес розробки Хвильового алгоритму 5.13 Процес розробки алгоритму Беллмана-Форда 5.14 Застосовані техніки 5.15 Ітогова ефективність

КАЛЕНДАРНИЙ ПЛАН

По р. №	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	*Отримання завдання кваліфікаційної роботи*	*29.03.2021 – 29.03.2021*	
2.	*Аналіз завдання, літератури та аналогів з теми кваліфікаційної роботи*	*29.03.2021 -04.04.2021*	
3.	*Вибір алгоритму, технологій та програмних засобів для реалізації завдання кваліфікаційної роботи*	*04.04.2021 – 14.04.2021*	
4.	*Проектування тестуючої системи*	*14.04.2021 – 17.04.2021*	
5.	*Розробка тестуючої системи*	*17.04.2021 – 18.04.2021*	
6.	*Розробка та оптимізація імплементацій задачі*	*18.04.2021 – 30.04.2021*	
7.	*Аналіз результатів*	*30.04.2021 – 02.05.2021*	
8.	*Оформлення пояснювальної записки та програмної документації*	*02.05.2021 – 07.05.2021*	
9.	*Оформлення графічної частини та презентаційних матеріалів комп'ютерного захисту*	*07.05.2021 – 10.05.2021*	
10.	*Представлення на рецензування*	*10.05.2021*	
11.	*Представлення кваліфікаційної роботи в ДЕК*	*18.05.2021*	

Студент_____Каяфюк Р.В.

(підпис)

Керівник роботи _____доц. каф. ст Панкратов О.В,

(підпис)

РЕФЕРАТ

Кваліфікаційна робота: 99 стор., 19 рис., 2 додатки, 26 джерел. Графічний матеріал атестаційної роботи містить 15 плакатів.

Мета роботи – підвищення ефективності вирішення задачі пошуку мінімального шляху на графі.

Об’єкт дослідження – можливості оптимізації алгоритмів що вирішують задачу пошуку мінімального шляху на графі шляхом застосування до них ідей масивного паралелізму що забезпечуються використанням графічних процесорів.

Результати роботи – варіант алгоритму мурашиної колонії пошуку мінімального шляху з використанням ідей масивного паралелізму та його імплементація на GPU.

Область застосування – логістичні системи, картографія, наукомісткі розрахунки, аналіз даних, пошук оптимальної послідовності дій для недетермінованої машини.

CUDA, GPGPU, ГРАФІЧНИЙ ПРОЦЕСОР, ЗАДАЧА ПОШУКУ МІНІМАЛЬНОГО ШЛЯХУ НА ГРАФІ, ОПТИМІЗАЦІЯ МУРАШИНОЇ КОЛОНІЇ, МАСИВНИЙ ПАРАЛЕЛІЗМ, ФОРД-БЕЛЛМАН, ХВИЛЬОВИЙ АЛГОРИТМ

ABSTRACT

Qualification work: 99 p., 19 pic., 26 source, 2 applications. Graphic material attestation work contains 15 posters.

The purpose of the work - increasing the efficiency of the shortest path problem implementations.

The object of the research - possibilities to optimize shortest path problem algorithms using the ideas of massive parallelism that provided by GPU.

The results of the work - shortest path problem solving based on ant colony optimization using the ideas of massive parallelism and its implementation on GPU.

Scope - logistics systems, cartography, high-tech calculations, data science, searching the optimal sequence of actions for a non-deterministic machine.

CUDA, GPGPU, GPU, SHORTEST PATH PROBLEM, ANT COLONY OPTIMIZATION, SIMD, BELLMAN-FORD, ALGORITHM LI

ЗМІСТ

ВСТУП ……7
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ……8
1.1 Аналіз стану проблеми пошуку найкоротшого шляху на графі ……8
1.2 Аналіз архітектур для роботи з GPU ……12
2 ВИБІР АЛГОРИТМУ, ТЕХНОЛОГІЇ, АРХІТЕКТУРИ ТА ПРОГРАМНИХ ЗАСОБІВ ……16
2.1 Вибір алгоритму та GPGPU……16
2.2 Постановка задачі……16
2.3 Вибір програмних засобів ……17
3 РОЗРОБКА ТЕСТУВАЛЬНОЇ СИСТЕМИ ……19
3.1 Проектування тестуючої системи ……19
3.2 Реалізація тестуючої системи ……23
4 ПІДГОТОВКА ДО РОБОТИ З GPGPU CUDA ……26
4.1 Внутрішня модель CUDA ……26
4.2 Робота з пам’яттю у CUDA……27
4.3 Складові CUDA-додатку та основні принципи його реалізації ……29
4.4 Характеристика використовуємого GPU ……32
5 РОЗРОБКА ТА ОПТИМІЗАЦІЯ РЕАЛІЗАЦІЇ МУРАШИНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ CUDA ……34
5.1 Загальна характеристика процесу розробки та оптимізації……34
5.2 Глобальна структура реалізації оптимізаційного алгоритму Мурашиної Колонії на GPU ……35
5.3 Версія 0 ……37
5.4 Версії 1-2 ……39
5.5 Версія 3 ……40
5.6 Версії 4-6 ……44
5.7 Версія 7 ……45
5.8 Версія 8 ……49
6 ПРЕДСТАВЛЕННЯ РЕЗУЛЬТАТІВ РОБОТИ З АЛГОРИТМОМ МУРАШИНОЇ КОЛОНІЇ ……...50
6.1 Ітоговий вигляд алгоритму ……50
6.2 Характеристика алгоритму ……54
7 РОЗРОБКА ТА ОПТИМІЗАЦІЯ РЕАЛІЗАЦІЇ ХВИЛЬОВОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ CUDA ……57
7.1 Глобальна структура реалізації Хвильового алгоритму на GPU ……57
7.2 Початкова ініціалізація ……58
7.3 Оптимізація розповсюдження хвилі ……...59
7.4 Оптимізація відновлення шляху ……60
8 ПРЕДСТАВЛЕННЯ РЕЗУЛЬТАТІВ РОБОТИ ХВИЛЬОВИМ АЛГОРИТМОМ ……....62
8.1 Ітоговий вигляд алгоритму ……62

8.2 Характеристика алгоритму ……63
9 РОЗРОБКА ТА ОПТИМІЗАЦІЯ АЛГОРИТМУ БЕЛЛМАНА-ФОРДА ВИКОРИСТАННЯМ CUDA ……64
9.1 Глобальна структура реалізації Хвильового алгоритму на GPU ……64
9.2 Підготовка масиву ……65
9.3 Покращення значень ……...66
10 ПРЕДСТАВЛЕННЯ РЕЗУЛЬТАТІВ РОБОТИ АЛГОРИТМОМ БЕЛЛМАНА-ФОРДА ……....67
10.1 Ітоговий вигляд алгоритму ……..67
10.2 Характеристика алгоритму ……..…68
11 ПРЕДСТАВЛЕННЯ ЗАГАЛЬНИХ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ ……..…69
11.1 Порівняння алгоритмів ……..69
11.2 Типові прийоми використані для оптимізації розрахунків ……...70
ВИСНОВОК ……72
ПЕРЕЛІК ПОСИЛАНЬ ……73
ДОДАТОК А (код використаний у проекті) ……75
ДОДАТОК Б (графічні матеріали) ……84

ВСТУП

Для великої кількості практичних задач давно були розроблені оптимальні послідовні алгоритми (коли задача фактично не може бути вирішена за менше число операцій). Та поява багатопоточних систем (спочатку - на рівні ідеї, далі - на рівні фактичної реалізації) поставила цю оптимальність під питання, оскільки дала змогу реалізовувати алгоритми з більшим числом загальних операцій, але, при цьому, меншим діаметром, і, таким чином кращою ефективністю та асимптотичною складністю. Але проблемою залишалось те, що фактично більшість багатоядерних процесорів - складалися лише з декількох ядер, і реалізація дійсно масивної багатопоточної системи потребувала величезних ресурсів і при цьому - було все ж сильно обмежена по максимальній кількості потоків що ми можемо достатньо ефективно використовувати паралельно.

Новим же кроком а напрямку оптимізації вирішень типових задач стала можливість використовувати графічні процесори - що саме заточені під масивні паралельні розрахунки та навіть у бюджетному варіанті вміють працювати з тисячами паралельних потоків - для неграфічних розрахунків (GPGPU). Це дало можливість актуалізувати ще одну групу алгоритмів котрі мали ще менший діаметр, але занадто велике число необхідних паралельно працюючих потоків для ефективної реалізації з використанням CPU.

Однією із задач рішення котрої так і не отримало ефективної оптимізації є пошук найкоротшого шляху на графі, а задача ця є типовою і активно використовується у різноманітних сферах. Тому нашою задачею стане дослідження того як ми можемо оптимізувати рішення цієї задачі використовуючи можливості GPU.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз стану проблеми пошуку найкоротшого шляху на графі

Задача пошуку найкоротшого шляху є однією з основних задач теорії графів, її суть - пошук послідовності ребер що з`єднують між собою дві вершини та мають мінімальну з можливих суму ваг. Важливість проблеми обґрунтовується її широким практичним застосуванням - картографія (пошук оптимального шляху між точками на карті, найкоротший шлях у мережі доріг), радіоелектроніка (трасування печатних плат та з'єднувальних провідників на поверхні мікросхем), комп'ютерні мережі (задачі маршрутизації на різних рівнях), Data Science (аналіз взаємозв'язків при представленні даних у вигляді графу) та інші.

Є декілька модифікацій постановки задачі - це пошук найкоротшого шляху безпосередньо між двома вершинами, пошук найкоротшого шляху з певної вершини або до певної вершини відносно всіх інших вершин, а також - пошук найкоротших шляхів між усіма парами вершин на графі. Також для певних задач - та у певних алгоритмах ми дізнаємося лише відстань, при цьому не маючи змоги відновити сам шлях.

У нашому випадку ми розглядаємо саме задачу пошуку шляху між двома вершинами та з відновленням шляху, вирішення саме її є актуальною проблемою на даний час, оскільки проблемою усіх алгоритмів пошуку найкоротшого шляху - є велика кількість "зайвої" інформації що стосується інших найкоротших шляхів у графі котру ми знаходимо в процесі пошуку найкоротшого шляху між двома вершинами та що ми надалі не використовуємо. У випадку коли ми розглядаємо задачу пошуку найкоротших шляхів між усіма вершинами - ми такої зайвої інформації майже не отримуємо, для найкоротших шляхів зв'язаних з однією вершиною - отримуємо, але не критично багато, у випадку ж пошуку найкоротшого шляху між двома вершинами - більшість отриманої інформації зрештою нам не знадобиться, а отже у нас є значно більше простору для можливих оптимізацій.

Актуальною є проблема вирішення задачі пошуку найкоротшого шляху саме з використанням GPU через те, що використання багатопоточності GPU дозволяє нам отримувати вирішення типових задач з кращою асимптотичною складністю і таким чином працювати з великими об'ємами даних використовуючи значно менші потужності та часові затрати. В той же час вирішення задачі пошуку мінімального

шляху у багатопоточному форматі не є очевидним і, принаймні у відкритих джерелах, ефективні його реалізації відсутні. Тому розглядання можливостей для застосування цього потужного інструменту до вирішення задачі пошуку найкоротшого шляху є важливим та актуальним.

Перейдемо безпосередньо до аналізу наявних алгоритмічних рішень, розглянемо історію їх розробки, сильні та слабкі сторони, асимптотичну та абсолютну складність їх роботи, придатність до створення паралельних багатопоточних реалізацій (на котрих, власне, і ґрунтується можливість використання у розрахунках GPU). Далі будемо використовувати при оцінці наступні скорочення:

- V - vertices - кількість вершин у графі;
- E - edges - кількість ребер у графі.

Алгоритм Дейкстри, розроблений у 1959 році нідерландським вченим Едсгером Дейкстрою. Знаходить усі найкоротші шляхи що виходять з однієї вершини. Як і переважна більшість алгоритмів пошуку найкоротшого шляху - засновується на пошуку в ширину. Також його можна охарактеризувати як використання динамічного програмування з інтелектуальним повним перебором. Кожна вершина зберігає інформацію про найкоротший шлях до стартової. Ми, обходячи в ширину граф, розглядаємо усіх сусідів, і, якщо використання ребра, що розглядається, є оптимальнішим за використання шляху з котрого було отримане попереднє значення для даної вершини - ми його оновлюємо, у випадку якщо усі сусіди розглянуті - ми виключаємо вершину з переліку вершин що необхідно розглянути. Асимптотично працює це, при наївній реалізації за O(V^2 + E), при використанні складних структур даних для зберігання - за O(VlogV + ElogV) та за O(VlogV + E) але з великими прихованими константами, тобто у найгіршому випадку, коли ми працюємо з повнозв`язним графом, при будь-якій реалізації, це буде не швидше за O(V^2). Точний алгоритм. Також не очевидно як його реалізовувати багатопоточно, оскільки у нас використовується динаміка та будуть конфлікти по даним.

Алгоритм Беллмана - Форда розроблений 1969 року незалежно один від одного Річардом Беллманом та Лестером Фордом. Знаходить найкоротші шляхи, що виходять з однієї вершини. Знову використовується динаміка. Ми зберігаємо інформацію про найкоротший шлях до вершини і що складається з не більше за j ребер. У динаміці ми ітеративно знаходимо усі шляхи з одного ребра, далі, додаючи

усі сусідні ребра - двох, і так V-1 раз, оскільки найкоротший шлях не може складатися з більш ніж V вершин (оскільки в кожну вершину ми заходимо максимум один раз). Перевагою даного алгоритму є те, що він вміє працювати з ребрами вагою менше нуля. Асимптотично працює за O(V*E), тобто, в гіршому випадку - O(V^3), що дуже повільно, але на розріжених та невеликих графах дає гарні результати за рахунок невеликих прихованих коефіцієнтів. Паралельна реалізація не є очевидною, знов через необхідність “оновлювати” однакові позиції у пам’яті у різних потоках, що викличе або велике уповільнення (при синхронізації доступу до пам’яті з використанням семафорів) або невизначену поведінку. Точний алгоритм.

Алгоритм A* (A-стар) був розроблений 1968 року Пітером Хартом, Нільсом Нільсоном та Бертрамом Рафаелем. По суті є розширенням алгоритму Дейкстри, що був описаний вище, за рахунок використання евристичної функції при виборі наступної вершини до розглядання. Є алгоритмом пошуку по першому кращому співпадінню. Знаходить найкоротший шлях між двома вершинами. Ми підтримуємо список вершин, що вже були розглянуті, а також впорядковану чергу вершин до розгляду, до черги додаємо усі сусідні вершини, вони зберігаються в відсортованому вигляді по значенню суми шляху пройденого до даної вершини та евристичної оцінки відстані до вершини, до котрої ми будуємо найкоротший шлях. У найгіршому випадку - асимптотично працює як і алгоритм Дейкстри, за O(E), але на випадковому графі - працює помітно швидше, але при цьому - потребує помітно більше місця для зберігання даних. Також є точним алгоритмом, котрий завжди знаходить рішення. У багатопоточному режимі можна паралельно попередньо розрахувати евристичні значення (що, правда, не дає великого приросту), якщо говорити стосовно основного етапу алгоритму - ситуація аналогічна до наведених вище.

Хвильовий алгоритм (Алгоритм Лі) був запропонований у 1961 році у якості формалізації алгоритмів трасування печатних плат. Перше що слід відмітити - алгоритм працює лише з планарними графами, однак для більшості практичних задач це не є важливим, і область задач, що він покриває є достатньо великою (картографія, радіоелектроніка залишаються у повному обсязі). Знаходить найкоротший шлях між двома вершинами. Також є модифікацією пошуку в ширину, у якій ми працюємо з одиничними ребрами (до такого вигляду можна перетворити довільний граф), ми спочатку позначаємо всі вершини як прохідні/непрохідні,

позначаємо стартову та фінішну вершини. Далі роздаємо значення “1” усім сусіднім вершинам до стартової, далі усім сусіднім вершинам до вершин зі значенням один - роздаємо значення “2”, і так далі поширюємо хвилю по нашому графу, на певному кроці досягаючи фінішу. Цей алгоритм активно використовується в радіоелектроніці та у побудуванні оптимальних шляхів у комп’ютерних іграх (покрокові стратегії, шахо-подібні ігри, та ж змійка). Асимптотична складність буде ідентична до складності звичайного пошуку в ширину - O(E), але ми будемо мати кращі результати на довільних графах. Якщо казати про багатопоточну реалізацію - ми можемо собі спробувати це дозволити за рахунок того, що хвиля розходиться у усіх напрямах, але конфлікти щодо запису даних без використання семафорів вирішити все-таки складно, є варіант з використанням декількох потоків що оброблюють одночасно вершини, що точно не можуть мати спільних сусідів (ми можемо це собі дозволити у зв’язку з методом додання вершин у чергу розгляду що використовується у даному алгоритмі, що заснований на їх фізичному місці знаходження). Працювати це буде все також за O(E), але з дуже приємним прихованим коефіцієнтом. Проблематикою залишається неможливість застосування на довільному графі та необхідність трансформування графа до одиничного.

Оптимізаційний алгоритм мурашиної колонії, був запропонований 1992 року доктором наук Марко Доріго і засновується на принципах пошуку шляху від мурашника до джерела їжі у природі. Є чимало модифікацій алгоритму заснованих на однаковій ідеї, ми будемо розглядати версію що знаходить найкоротший шлях між двома вершинами (саме такою була перша запропонована реалізація). Загальна ідея наступна: ми маємо певну кількість мурах, котрі спочатку розміщуються у стартовій вершині, для кожного ребра ми зберігаємо інформацію про кількість феромона на ребрі та його вагу, мурахи будують шляхи керуючись імовірністним вибором ребер у шляху відносно їх довжини (мінімізуємо) та відносно кількості на ребрі феромону (максимізуємо). Після того як кожна мураха завершила побудову шляху (зайшла у фінішну вершину або у глухий кут) ми зрізаємо значення феромонів на усіх ребрах на певний фіксований відсоток (випаровування феромону), а далі оновлюється уздовж побудованих шляхів (кожне ребро через котре проходить шлях, отримує приріст феромону зворотно пропорційний до довжини шляху) і, таким чином, чим більше шляхів пройшло через ребро та чим вони були коротші - тим більше приросту феромону це ребро отримає. Таким чином ми ітеративно зведемо рішення до одного шляху, котрий жадібно відновлюється.

Асимптотичну оцінку дати доволі складно, оскільки вона сильно залежить від подробиць окремої реалізації. При однопоточній реалізації, ми маємо n мурах, обираємо наступний крок у шляху за O(VlogV) з маленьким прихованим коефіцієнтом (оскільки нам потрібно розрахувати значення переходів по кожному ребру (V) та вибрати одне з них (logV)), шлях складається з максимум V виборів наступних кроків, отже фаза будування шляхів працює за O(n*V^2logV) з маленькими прихованими коефіцієнтами. Оновлюємо ми за O(E) - випаровування феромону - та за O(n*V) - оновлення значень феромону вздовж побудованих шляхів, отримуємо O(E + n*V). Загальна складність виходить O(n*V*V*logV + E + n*V), або, якщо брати по найвищій степені полінома, O(n*V^2*logV). До того ж алгоритм є приблизним та не гарантує 100% точності знаходження найкоротшого шляху. Але є одна важлива відмінність від розглянутих вище - він дуже гарно розпаралелюється на багатопоточну реалізацію, для того щоб виявити можливу для досягнення асимптотичну складність, уявімо що ми працюмо з нескінченною множиною паралельних потоків: кожна мураха працює окремо і незалежно один від одної (забираємо з асимптотики n), далі, на кожному кроці вибору ребра - ми можемо організувати вибір за logV, з використанням паралельної префікс-суми та паралельного пошуку, забираємо з асимптотики V, на другому етапі - виконуємо випаровування за O(1) та оновлення вздовж шляхів за O(V). Таким чином отримуємо O(V*logV + V) у гіршому випадку (навіть на повнозв’язних графах), що вище наведених вище алгоритмів.

1.2 Аналіз архітектур для роботи з GPU

Графічні процесори у сучасних відеокартах використовуються у якості пришвидшувача трьох вимірної графіки, їх архітектура побудована таким чином щоб максимально ефективно оброблювати та відображати комп’ютерну графіку, за рахунок спеціальної конвеєрної архітектури, що для цих задач підходить краще ніж архітектура типового центрального процесора. Якщо центральний процесор містить у кращому випадку декілька ядер та глобально орієнтований на послідовні розрахунки, то GPU, навіть на ідейному плані, з самого початку будувалася як багатоядерна структура для масивних паралельних розрахунків. Утім GPU має і пов’язаний з цим ряд послаблень/спеціальних умов котрі треба мати на увазі: менша кількість команд що може виконувати процесор, а також потоки не є цілком

незалежними, тобто вони мають одночасно виконувати однорідні операції - тому сила і ефективність GPU проявляється саме у задачах що здатні розбиватися на велику кількість ідентичних за функціональним змістом потоків.

Використання GPU для роботи з не графічними об’єктами називається GPGPU. Вперше подібна система була створена у 1978 році, початок активного ж розвитку цього напрямку припав на кінець дев’яностих - початок двохтисячних років, коли на графічних процесорах Nvidia були вирішені такі задачі як розрахунок діаграм Вороного, мультиплікація матриць та розрахунок часткових диференціальних рівнянь.

Є два основних виробники графічних процесорів - Nvidia та ATI Technologies - вони очевидно мають власні архітектури, це є CUDA для процесорів Nvidia та ATI Stream Technology для процесорів ATI Technologies. Програмна реалізація для CUDA виконується через CUDA SDK, для ATI Stream Technologies - у якості програмного інтерфейсу використовується OpenCL - фраємворк що стандартизується консорціумом Khronos Group, що включає такі компанії як AMD, Apple, ARM, Intel, Nvidia, Sony Computer Entertainment. Далі розглянемо особливості та переваги тих чи інших архітектур, очевидно що будемо розглядати CUDA у контексті CUDA SDK, а ATI Stream Technolgy у контексті OpenCL. Окрім OpenCL є ще один крос-апаратний інтерфейс реалізований Microsoft, що називається DirectCompute.

CUDA SDK - працює лише з графічними процесорами Nvidia, дає змогу розробнику на свій розсуд організовувати доступ до графічного або тензорного пришвидчувача, а також керувати його пам’яттю. Працює зі спрощеними діалектами C, C++ та Fortran. Представлена 2007 року. До складу програмної архітектури входить власний компілятор nvcc. У цій архітектурі використовується модель пам’яті grid, кластерне моделювання потоків та SIMD інструкції (одиничні інструкції виконувані на багатьох даних). Активно використовується у наукових розрахунках, у таких областях як астрофізика, обчислювальна біологія та хімія, моделювання динаміки рідин, електромагнітні взаємодії, комп’ютерна томографія, сейсмічний аналіз та багато іншого. Порівняно з іншими GPGPU виділяються наступні переваги:

- інтерфейс програмування заснований на стандартній мові C з певними обмеженнями, що помітно спрощує та пришвидшує процес набуття навичок роботи з CUDA;

- наявна розподілена пам’ять (shared memory) дозволяє створювати кеш з більш широкою пропускною полосою;

- більш ефективні транзакції між пам’яттю центрального процесора та відео-пам’яттю;

- наявна повна підтримка цілочисельних та побітових операцій на апаратному рівні;

- підтримка компіляції GPU коду засобами відкритого LLVM (універсальна система аналізу, трансформації та оптимізації програм, що реалізує віртуальну машину);

- наявність великої кількості бібліотек;

- наявність великої кількості навчальних статей що спрощують початок роботи з CUDA.

OpenCL - фреймворк для створення багатопоточних програм з використанням як GPU так і CPU, є реалізацією техніки GPGPU, є відкритим стандартом. До ATI Stream був доданий ще з першої версії (2008 рік), у CUDA підтримка з’явилася з 2010 року, але не пряма, а лише у вигляді драйверу що використовує функціонал CUDA SDK, що робить його використання на CUDA за замовчуванням повільнішим за CUDA SDK. Мова OpenCL є по факту модифікацією C++ з певними обмеженнями.

Якщо порівнювати безпосередньо CUDA з OpenCL - можна сказати наступним чином, CUDA має більшу кількість бібліотек та засобів, більше документації та підтримки від розробників, та ширше ком’юніті. У той же час OpenCL є більш універсальним, дозволяє створювати крос-платформні рішення, дозволяє працювати з відеокартами AMD Technologies, котрі, у середній ціновій категорії працють швидше у якості GPGPU, за рахунок того що Nvidia блокують частину функціоналу, для просування карток нового покоління, має більш ефективні механізми синхронізації, OpenCL є глобально складнішим для роботи за технологію CUDA.

Також окремо можна виділити DirectCompute - інтерфейс програмування гетерогенних додатків націлених на виконання неграфічних задач загального користування, що є складовою DirectX - графічного API що розроблюється Microsoft і здатен працювати на IBM-сумісних машинах під операційними системами сімейства Windows. Взаємодію з DirectCopmpute підтримують ключові розробники графічних процесорів - Nvidia та AMD. Працює він у обох випадках “поверх” CUDA

та AMD Stream відповідно, та до того ж є менш гнучким за OpenCL. Активно застосовується цей інтерфейс у задачах пов’язаних с фізичними (ігровими) двигунами.

2 ВИБІР АЛГОРИТМУ, ТЕХНОЛОГІЇ, АРХІТЕКТУРИ ТА ПРОГРАМНИХ ЗАСОБІВ

2.1 Вибір алгоритмів та архітектури GPGPU

Оскільки GPGPU передбачає багатопоточну реалізацію з однорідними операціями, ключовим критерієм є якість паралелізації алгоритму та можливий спідап, що отримується за рахунок використання GPU. Серед розглянутих алгоритмів, більшість заснована на пошуку в ширину, що створює проблеми з паралельною реалізацією, оскільки нам або необхідно додавати окрему евристику, за якою ми будемо ділити вершини на групи, та розглядати у кожному потоці лише ці вершини, або використовувати м'ютекси на всіх операціях запису, що сильно сповільнює роботу алгоритму. Лише один з наведених алгоритмів не відноситься до сімейства заснованих на пошуку в ширину - це мурашиний алгоритм, котрий гарно паралелиться і, до того ж, отримує при цьому значний асимптотичний приріст, тому його реалізація буде найбільш потенційно ефективною та має більше простору для оптимізацій. Також Мурашиний Алгоритм дуже прямо асоційований з принципами GPGPU, оскільки ми виконуємо ідентичні операції для великого масиву мурах. Виходячи з цього, оптимальним до реалізації буде Мурашиний Алгоритм.

Окрім цього також ідеї для можливого паралелізму є у Хвильовому алгоритмі, оскільки для нього на кожному кроці відбувається незалежне оновлення у різних клітинках.

Також давайте оберемо оберемо один із інтуїтивно погано здатних до паралелізму алгоритмів – Алгоритм Беллмана-Форда, щоб на його прикладі побачити що ми можемо зробити для вирішення задачі перенесення на GPGPU класичних BFS-like алгоритмів.

При виборі архітектури GPGPU ми в першу чергу відштовхуємося від наявної у нас для експериментів відеокарти, а нашому випадку це Nvidia GeForce 940MX, що підтримує як OpenCL, так і CUDA 5го покоління (Maxwell). Наша робота є у більшій степені дослідницькою, і тому ключовою для неї є не гнучкість, а ефективність, а отже правильним буде вибрати використання CUDA SDK, що дає нам кращу швидкодію для карток Nvidia ніж OpenCL, котрий працює поверх CUDA. DirectCompute ми не будемо використовувати з тих же міркувань - він сповільнить

глобальну швидкодію через необхідність витрачання зайвого часу на взаємодію з CUDA.

2.2 Постановка задачі

Ітогова задача зводиться до дослідження можливостей застосування ідей масивного паралелізму до трьох різнопланових алгоритмів пошуку мінімального шляху – алгоритму Мурашиної колонії, Хвильового алгоритму, та алгоритму Беллмана-Форда. Необхідно створити алгоритмічні моделі імплементації цих алгоритмів на GPU, знайти можливі техніки оптимізації та максимально їх оптимізувати, виявивши при цьому типові прийому що можуть використовуватися при перенесення алгоритмів на графах з CPU на графічні процесори.

2.3 Вибір програмних засобів

У архітектурі CUDA ми працюємо з C-подібною мовою, є офіційний інструментарій для роботи з кодом що називається Nvidia Nsight, котрий нам дозволяє виконувати збірку, налагодження, профілювання і трасування гетерогенних додатків написаних з використанням усіх найбільш розповсюджених програмних архітектур, у тому числі CUDA C/C++. Інтегрується він до основних серед розробки, таких як Visual Studio та Eclipse. У нашому випадку будемо користуватися Visual Studio 2017, котра дозволяє працювати з CUDA витрачаючи менше часу на низько-рівневе налаштування а також кращий інструментарій для дебагу.

Ураховуючи те, що у ході роботи ми намагаємося максимізувати ефективність реалізації алгоритму, а також дослідити який фактичний приріст дають ті чи інші оптимізації, нам необхідно мати інструментарій для зберігання, аналізу та відображення результатів експериментів. А отже нам потрібна база даних до котрої ми будемо зберігати результати усіх експериментів та інтерфейс для роботи з нею. Використовувати ми будемо SQL базу даних, з СУБД MySQL та візуальним засобом проектування MySQL Workbench 8.0.

Для налаштування коректного функціонування CUDA у системі та використанні CUDA у свої гетерогенних додатках нам знадобиться CUDA Toolkit,

його інструментарій включає в себе прискорені GPU бібліотеки, засоби налагодження та оптимізації, компілятор CUDA C / C ++ і бібліотеку для розгортання програми.

Нам знадобиться інструментарій для генерації тестових даних, оскільки достатньо якісних вибірок даних знайдено не було, у якості такого пропонується використовувати C++ та Visual Studio.

Для відображення результатів проведення експериментів - пропонується використовувати дистрибутив Anaconda з мовою Python у зв’язці з командною оболонкою Jupyter Notebook, котра дозволяє у зручному форматі відображати у рамках одного документа різноманітні графіки, формули, коментарі та фрагменти коду. Для безпосередньо відображення графіків пропонується використання matplotlib.

3 РОЗРОБКА ТЕСТУЮЧОЇ СИСТЕМИ

3.1 Проектування тестуючої системи

Оскільки в процесі виконання дипломної роботи ми намагаємося отримати максимально ефективну реалізацію алгоритму, нам потрібно мати можливість швидко і якісно оцінювати точність і швидкодію алгоритму, а так само для представлення результатів - ми повинні мати інструмент для доступу і роботи з результатами оціночних експериментів. Виходячи з цих міркувань було вирішено реалізувати тестуючу систему.

В першу чергу - нам необхідні дані на яких ми можемо провести тестування, а саме - графи і набори параметрів. Нас в першу чергу цікавить робота з великими графами, котрі, при цьому, мають велику щільність (оскільки саме з щільними графами виникають проблеми у основних алгоритмів пошуку найкоротшого шляху, а алгоритм Мурашиної Колонії на GPU від кількості ребер залежить у меншій степені, оскільки масивний паралелізм що реалізується в ньому має працювати з графами будь якої щільності однаково). Оскільки нам доведеться генерувати великі масиви даних - хочеться робити це максимально швидко, а отже будемо використовувати найшвидший високорівневий інструмент програмування - мову c++. Нам необхідно буде згенерувати графи у форматі матриць суміжності, оскільки алгоритм у процесі виконання працює з аналогічними матрицями для представлення інших даних, а параметри мають включати вершини між котрими ми будемо шукати шлях та кількість ітерацій алгоритму що ми планує виконати - для того щоб при тестуванні оцінити залежність точності що ми отримуємо від кількості ітерацій, та у подальшому, відштовхуючись від цього виробити об’єктивний механізм вибору кількості ітерацій.

Потім нам необхідно спроектувати базу даних, що буде використовуватися для зберігання інформації про проведені експерименти, використовувані тестові дані та різні версії реалізацій, на котрих виконувалося тестування. База даних необхідна для того, що б мати можливість в автоматичному режимі зберігати інформацію про виконані запуски а потім отримувати до неї доступ (як для надання підсумкових результатів, так і, наприклад, отримання «помилкових» тестів для визначення слабких місць або помилок у реалізації алгоритму та подальшого їх виправлення). Перейдемо безпосередньо до проектування. Користуватися будемо

інструменталом MySQL Workbench, і через нього будемо взаємодіяти з базою даних у подальшому.

Нам знадобляться таблиці для представлення чотирьох основних сутностей:

- реалізація;
- експеримент;
- граф;
- набір параметрів.

Для реалізації нам потрібно зберігати інформацію про її фізичне положення у даних проекту, про число окремих мурах, що приймає участі у будуванні шляхів та коефіцієнти що застосовуються на різних етапах алгоритму, наприклад коефіцієнт випаровування.

Для графу ми зберігаємо інформацію про фізичне положення, кількість вершин у графі та кількість ребер у графі для можливості оцінки залежності швидкодії алгоритму від цих параметрів.

Для параметрів ми зберігаємо фізичне розміщення та безпосередньо параметри - початкова вершина, фінальна вершина, а також попереднє число ітерацій.

Експеримент - ключова сутність, для котрої ми зберігаємо результати експерименту, а саме:

- точність (відношення кількості запусків що знайшли мінімальний шлях, до загального числа запусків);
- абсолютна точність (була додана пізніше, починаючи з 7 реалізації, сутність даної метрики - середнє відхилення знайденого шляху від мінімального шляху, тобто для попередньої метрики для нас ситуації коли ми знайшли шлях на одиницю довший за мінімальний та у рази довший за нього є ідентичними, а для об’єктивної оцінки роботи алгоритму та пошуку слабких місць їх слід розглядати та оцінювати окремо);
- число запусків алгоритму;
- середній час роботи;
- бінарна інформація про те, чи завершив алгоритм коректно свою роботу (виставляється як один у випадку якщо ми зловили певну помилку всередині кернелів CUDA);

- інформацію про те, на якому графі/наборі параметрів/реалізації виконувався цей експеримент (зовнішні ключі, котрі стають частиною первинного складеного ключа).

На малюнку 3.1.1 зображена логічна схема спроектованої БД.

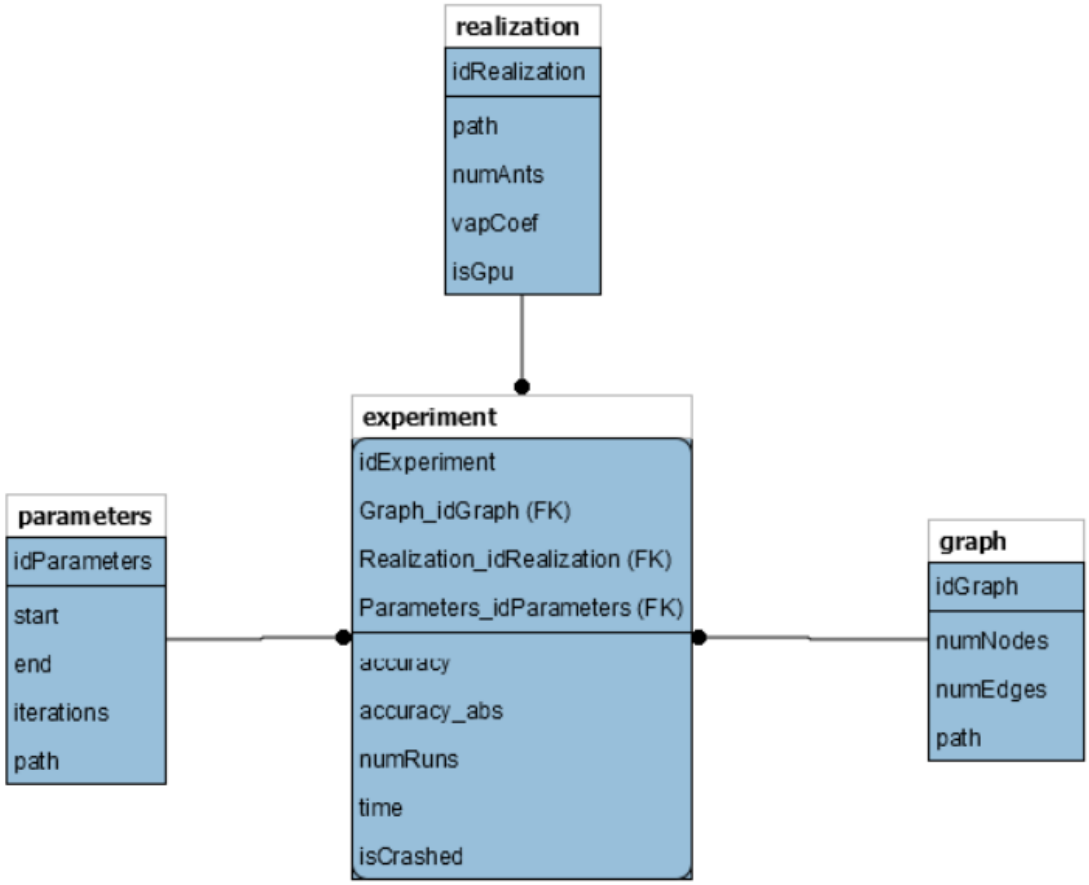

Рис. 3.1.1 - Логічна схема спроектованої БД

Фізично база буде реалізовуватися на MySQL, отже використаємо для проектування фізичної моделі EER діаграму з MySQL Workbench, що дозволяє напряму згенерувати та підняти базу даних з наявної діаграми. Встановимо типи даних, флаги (not null, autoincrement, unique та інші), зовнішні та первинні ключі. Зв’язок буде ініціюючим, оскільки експеримент не може існувати без фіксованих графу/параметрів/реалізації, за рахунок цього додаємо зовнішні ключі таблиці experiment до її складеного первинного ключа. Але зовнішні ключі не можуть повністю представити окремий експеримент, оскільки ми маємо змогу виконувати декілька експериментів з однаковими даними, а отже частиною складеного первинного ключа залишиться і id експерименту (інакше ми могли б виключити

його, і залишити у складі первинного ключа для таблиці experiment лише її зовнішні ключі).

На рисунку 3.1.2 зображена фізична схема БД.

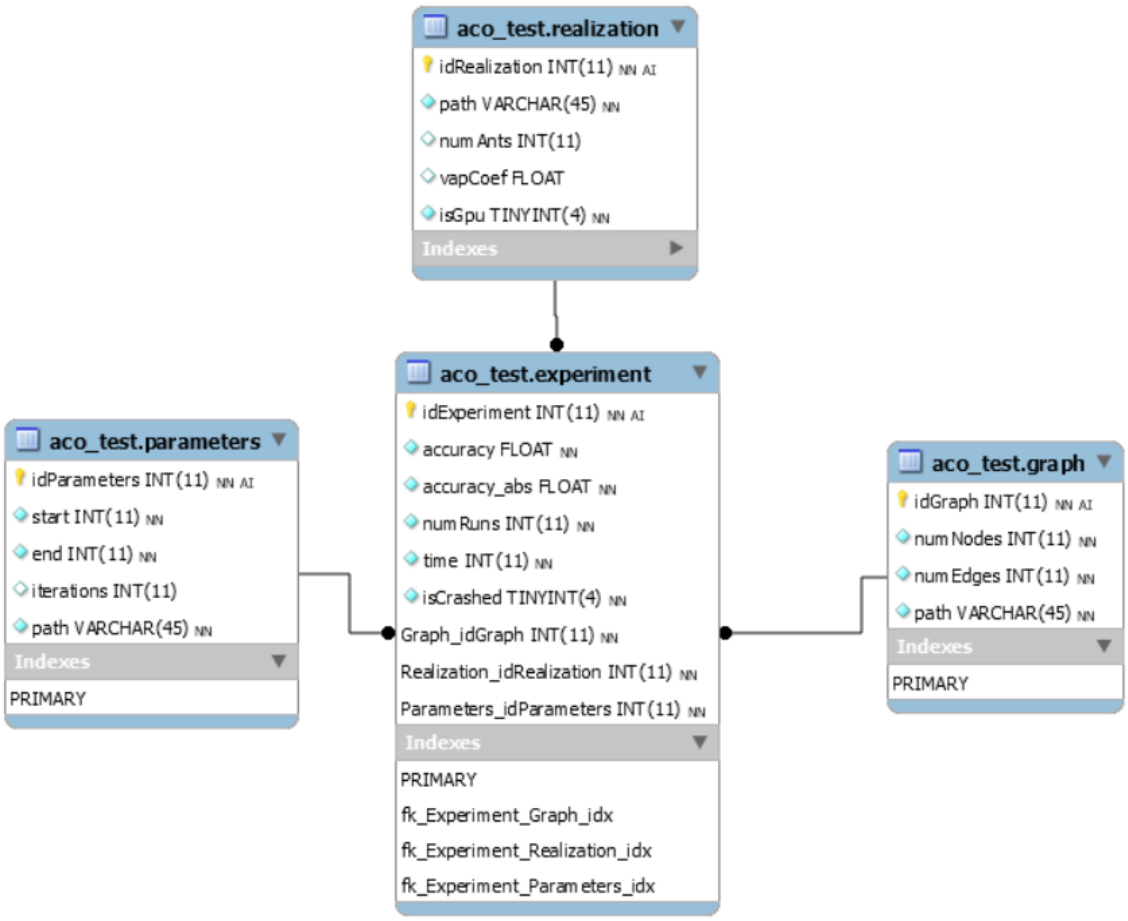

Рис. 3.1.2 - Фізична схема БД

Нам також знадобиться забезпечити взаємодію між базою даних і нашими реалізаціями. Для цього нам необхідно буде створити збережену процедуру, яка виконує додавання інформації про експеримент: в разі відсутності графа / реалізації / параметрів в бд раніше - буде створювати нові об’єкти відповідного типу і з відповідними даними у БД, після чого буде додавати новий запис до таблиці експериментів з відповідними результатами та id щойно доданих об’єктів, інакше - буде аналогічно додавати інформацію про проведений експеримент, але при цьому буде використовувати id знайдених графа / реалізації / параметрів доданих у попередніх експериментах.

Також нам потрібно буде створити обгортку для наших реалізацій, за допомогою якої будемо додавати дані до бази даних напряму під час виконання алгоритму, за рахунок звернення до SQL серверу та виклику збереженої процедури з відповідними параметрами. Для цього ми будемо користуватися бібліотекою mysql.h, що розповсюджується разом з пакетом MySQL та реалізує функції для взаємодії з базою даних.

Нам залишилося забезпечити можливість запуску тестування у автоматичному режимі (оскільки у нас є велика кількість тестових даних, і запускати все вручну не є можливим). Для цього пропонується реалізувати BAT-скрипт, що буде у циклі запускати обрану реалізацію з обгорткою та усіма можливими зв’язками графів і параметрів.

3.2 Реалізація тестуючої системи

Для генерації були написані дві консольні програми на C++, в першій ми йшли за кількістю вершин що дорівнює степеням двійки і для кожного такого значення генерували по 16 графів з вартостями переходів згенерованих по модулю 2 ^ 1 - 2 ^ 16, при цьому половина ребер випадково встановлювалась нулем, використовуючи функцію rand() та остаток від ділення на 2. Код створеного додатку є у додатку А(1). У другій же аналогічним чином ми генеруємо вершини між якими ми шукаємо шлях, і генеруємо кількість ітерацій які ми виконаємо (очевидно, що перевіряти точність після кожної, прагнучи до якогось епсілону ми не можемо, тому, поки не буде визначено механізм визначення числа ітерацій - будемо керуватися згенерованими, і, на їх фоні робити висновки як раз для пошуку даного механізму, а потім зможемо просто їх ігнорувати). Код створеного додатку є у додатку А(2). Окрему увагу слід виділити принципу найменування графів та параметрів, для найменування використовуються ступені двійки що відповідають кількості вершин та модуля по котрому були узяті ваги/параметри, що спрощує взаємодію з ними у автоматичному режимі. Також, окремі крайні випадки, були згенеровані вручну. На виході ми отримали 14 гігабайт тестових даних, на яких зможемо визначати коректність роботи нашого алгоритму і його ефективність, шукати «на крайні випадки», що могли бути втрачені при теоретичному опрацюванні.

Для реалізації БД ми використали функцію Forward Engineering до створеної раніше фізичної моделі. Отриманий генеруючий БД скрипт знаходиться у додатку А(3).

У згенерованій БД одразу була створена збережена процедура add_experiment, що приймає у якості параметрів номер стартової вершини, номер фінальної вершини, кількість ітерацій виконаних алгоритмом, фізичну адресу у котрій знаходиться файл з параметрами, кількість мурах задіяних у реалізації, коефіцієнт випаровування, булеву змінну що показує чи на GPU реалізований алгоритм, фізичну адресу де знаходиться реалізація, кількість вершин та ребер у графі, фізичну адресу за якою знаходиться матриця суміжності графу, точність отримана як результат експерименту, кількість запусків алгоритму за час котрих була визначена точність, середній час затрачений на роботу алгоритму, булева змінна що показує чи зловив алгоритм помилку та абсолютна точність. Використовуючи конструкцію IF-THEN-END IF ми шукаємо у базі даних граф/реалізацію/набір параметрів з такою ж фізичною адресою через SELECT COUNT FROM, і, якщо не знаходимо (тобто якщо COUNT = 0) - додаємо новий запис до відповідної таблиці використовуючи параметри передані до процедури. Потім виконуємо INSERT INTO до таблиці experiment, де у якості параметрів що стосуються зовнішніх ключів виконуємо підзапит SELECT шукаючи id об’єкту фізичне положення котрого співпадає з переданим у збережену процедуру. Код реалізованої процедури знаходиться у додатку А(4).

Перейдемо до обгортки. На початку, визначимо механізм передачі параметрів до реалізації - у якості параметрів будемо писати назви графів/набору параметрів, за замовчуванням визначені директорії в котрих вони зберігаються. Спочатку ініціюємо mysql з’єднання вказуючи порт на котрому активний sql-сервер, та параметри користувача для входу. У двох місцях (після завершення роботи алгоритму, та в функції що ловить CUDA помилки) створюємо sql-запити, що передаються у вигляді строки “CALL add_experiment(parameter1, parameter2 … parametern)” напряму до sql-серверу та виконуються там. Також, після кожної дії пов’язаної з взаємодією з MySQL ми додаємо хендлер що буде ловити помилки взаємодії з сервером у випадку їх наявності. Фрагменти обгортки знаходяться у додатку А(5).

Залишилося створити BAT-скрипт, для запуску окремої реалізації на тестування. Як ми згадували вище, в проекті - для кожного типу вхідних даних

(реалізації, графи, параметри) - окрема папка, тому взаємодіяти з ними досить легко. Використовувати ми будемо вкладені цикли, а наші графи та параметри заздалегідь були названі для зручного їх застосування, у форматі i_j. Цикли тут викликаються у форматі “for /L %%f in (1,1,16) do (“, де параметр /L ініціює появлення циклу, а %%f показує змінну вздовж котрої цикл виконується, усього чотири таких цикли (по два для графу та параметру), на нижньому рівні маємо безпосередньо виклик реалізації у форматі “start /MIN /wait ‘D:\source\Diplom\Realizations\’ 1.exe %%f_%%d %%i_%%j”, де параметр /MIN змушує відкривати кожний екземпляр у новому зверненому вікні, а параметр /wait змушую на кожному кроці циклу очікувати завершення виконання викликаного раніше вікна. Повний код BAT-скрипту знаходиться у додатку А(6).

4 ПІДГОТОВКА ДО РОБОТИ З GPGPU CUDA

У попередніх розділах був наведений поверхневий аналіз CUDA у порівнянні з іншими GPGPU системами, але для того щоб здійснити ефективну імплементацію алгоритму з використанням CUDA нам необхідно розуміти як саме вона працює на низькому рівні абстракції, який саме інструментал може запропонувати, принципи котрих необхідно дотримуватися при роботі з CUDA та з чого має складатися типовий CUDA-додаток, а також маємо налаштувати саму систему під взаємодію з CUDA.

4.1 Внутрішня модель CUDA

Для виконання розрахунків на CUDA використовуються так звані ядра (kernels), котрі за сутністю схожі на функції у звичайних с-подібних мовах. У цих ядрах ми реалізуємо код, що буде викликаний у усіх потоках що входять до сітки, для котрої буде викликане це ядро, а також визначаємо параметри що будуть передані до цього ядра з пам’ять “загальної видимості” (глобальної, константної та інших, детальніше про це у розділі 4.2).

Що з себе представляє сітка для виклику ядра. Це масив блоків, що може бути двох-вимірним, або, у версіях починаючи з 2.0 – трьох-вимірним. Максимальне число блоків у кожному вимірі - також обмежене та залежить від покоління та конкретної версії до котрої відноситься і апарат на котрому виконуються розрахунки. Кожен блок, у свою чергу є тривимірним і складається з потоків (threads). Особливістю саме блоків є те, що ми можемо синхронізувати потоки у межах одного блоку, а також маємо спільну швидку кеш пам’ять до котрої мають доступ лише потоки одного блоку (розподілена пам’ять). Кількість цієї пам’ять та можливий розмір кожного виміру блока також однозначно визначаються специфікацією.

На рисунку 4.1.1 відображена розрахункова модель GPGPU CUDA (kernel/greed/block/thread).

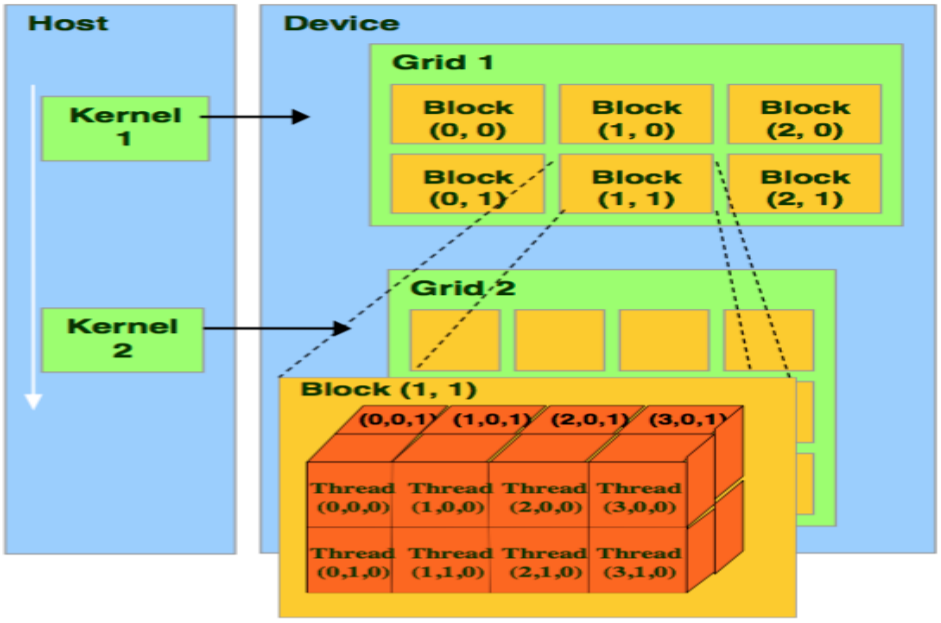

Рис. 4.1.1 - Розрахункова модель GPGPU CUDA

Така структура зумовлена саме необхідністю графічного процесору вирішувати графічні задачі, такі як виконувати обробку та відображення багатовимірної графіки, розрахунок текстур. Така структура дозволяє нам ділити графічні об’єкти на фрагменти та обробляти їх окремо у масивному паралельному режимі.

Далі, кожен потік, не виконується окремо - вони діляться на так звані варпи, що для всіх поколінь складають 32 потоки. Одна операція у межах варпу виконана на 32 потоках - є фактично однією операцією виконаною на cuda-ядрі (близько до принципу векторних процесорів), це є найменшою структурною одиницею з котрою ми працюємо, а саме тому реалізації на CUDA слід організовувати таким чином, щоб варпи були максимально повними, оскільки одна операція у варпі виконується з такою ж швидкістю як і 32. Звичайним паттерном для вирішення задач на CUDA є пряме виділення варпів (коли ми створюємо окреме ядро, котре працює безпосередньо з 32 потоками, та зводимо основне ядро до структури що викликає ці ядра та об’єднує результати їх роботи, що, зазвичай, дає чималий приріст швидкодії).

4.2 Робота з пам’яттю у CUDA

Коректна робота з пам’яттю для GPGPU систем - і CUDA не виняток - є надзвичайно важливою, оскільки ми працюємо з масивними паралельними розрахунками на системі котра призначена для швидкого виконання операцій саме у такій програмній структурі, а якщо при цьому кожна операція потребуватиме звернення до повільних типів пам’яті - отримана швидкодія буде дуже низької і саме робота з пам’яттю вважається найслабкішою та найбільш вразливою складовою програм CUDA.

У CUDA розділяють декілька типів пам’яті, що розрізнюються по швидкості доступу, області видимості, доступному об’ємі, особливостях стосовно адресації та основних операцій:

- Реєстрова пам'ять (register) є найшвидшою з усіх видів. Визначити кількість регістрів доступних GPU можна за допомогою функції cudaGetDeviceProperties. Розрахувати кількість регістрів, доступних одній нитці GPU, так само не складає труднощів, для цього необхідно розділити загальне число регістрів на додаток кількості ниток в блоці і кількості блоків в гріді. Всі регістри GPU 32 розрядні. У CUDA немає явних способів використання реєстрової пам'яті, всю роботу по розміщенню даних в регістрах бере на себе компілятор;

- Локальна пам'ять (local memory) може бути використана компілятором при великій кількості локальних змінних в будь-якої функції. За швидкісними характеристиками локальна пам'ять значно повільніше, ніж реєстрова. У документації від nVidia рекомендується використовувати локальну пам'ять тільки в найнеобхідніших випадках. Явних засобів, що дозволяють блокувати використання локальної пам'яті, не передбачено, тому при падінні продуктивності варто ретельно проаналізувати код і виключити зайві локальні змінні;

- Глобальна пам'ять (global memory) - найповільніший тип пам'яті, з доступних GPU. Глобальні змінні можна виділити за допомогою специфікатора __global__, а так само динамічно, за допомогою функцій з сімейства cudMallocXXX. Глобальна пам'ять в основному служить для зберігання великих обсягів даних, що надійшли на device з host'a, дане переміщення здійснюється з використанням функцій cudaMemcpyXXX. В алгоритмах, що вимагають високої продуктивності, кількість операцій з глобальною пам'яттю необхідно звести до мінімуму;

- Розподілена пам'ять (shared memory) відноситься до швидкого типу пам'яті. Пам'ять, що розділяється рекомендується використовувати для мінімізації звернення до глобальної пам'яті, а так само для зберігання локальних змінних функцій.

Адресація пам'яті, що між нитками потоку однакова в межах одного блоку, що може бути використано для обміну даними між потоками в межах одного блоку. Для розміщення даних в пам'яті, що використовується специфікатор __shared__, основну частину операцій пропонується виконувати саме взаємодіючи з пам’яттю даного типу;

- Константна пам'ять (constant memory) є досить швидкою з доступних на GPU. Відмінною особливістю константної пам'яті є можливість запису даних з хоста, але при цьому в межах GPU можливо лише читання з цієї пам'яті, що й обумовлює її назву. Для розміщення даних в константної пам'яті передбачений специфікатор __constant__. Якщо необхідно використовувати масив в константної пам'яті, то його розмір необхідно вказати заздалегідь, так як динамічне виділення на відміну від глобальної пам'яті в константної не підтримується. Для запису з хоста в константну пам'ять використовується функція cudaMemcpyToSymbol, і для копіювання з device'а на хост cudaMemcpyFromSymbol, як видно цей підхід дещо відрізняється від підходу при роботі з глобальною пам'яттю;

- Текстурна пам'ять (texture memory), як і випливає з назви, призначена головним чином для роботи з текстурами. Текстурна пам'ять має специфічні особливості в адресації, читанні і запису даних. Вона доступна для читання всіма мульти-процесорами. Вибірка даних здійснюється за допомогою текстурних блоків відео-чипа, тому надаються можливості лінійної інтерполяції даних без додаткових витрат. Кешується по 8 кілобайт на кожен мульти-процесор. Повільна, як глобальна.

4.3 Складові CUDA-додатку та основні принципи його реалізації

Щоб почати виконання розрахунків на GPU - у першу чергу необхідно обрати девайс на котрому розрахунки і будуть виконуватися, за допомогою функції cudaSetDevice() що є складовою API CUDA для взаємодії з хостом. Далі усі дані що знадобляться нам для розрахунків треба перенести з хосту на девайс, для цього створюються вказівники потрібного нам типу, традиційно у назві використовується літера d (device). Створені вказівники ми ініціалізуємо використовуючи функцію cudaMalloc(), таким чином виділяючи їм пам’ять на девайсі, а потім копіюємо туди дані з хосту використовуючи cudaMemcpy(). Ці дані знаходяться у глобальній пам’яті графічного процесора. Номінально ми можем отримати доступ з хосту до

вказівників на GPU, оскільки зберігаються вони як звичайні C++ вказівники, але адекватної інформації звідти не отримаємо.

Для запуску розрахунків потрібно створити ядро, про котре йшла річ у розділі 4.1. Для цього використовується структура формату __global__ void CoreName(parameters), у якості параметрів встановлюємо типи даних відповідно до даних що були передані на девайс від хосту і будуть задіяні при розрахунках у даному ядрі.

Усередині ядра для визначення того де саме ми знаходимося у нашій сітці користуємося запитами формату threadIdx.x, blockIdx.x - ми можемо визначити координати потоку/блоку по кожному з вимірів та загальний розмір кожного наявного виміру сітки/блоку. Зазвичай ключові значення фіксуються одразу і зберігаються.

Також ми маємо виділити масив для роботи з швидкою розподіленою пам’яттю, використовуючи ідентифікатори extern __shared__. Максимальна кількість розподіленої пам’яті на блок/ на мульти-процесор є дуже обмеженою та визначається специфікацією, у разі перебільшення ми отримаємо помилку. Кількість же розподіленої пам’яті що виділяється на кожен блок ядра, що запускається, визначається при його виклику (але її слід вказувати слідкуючи за специфікаційними обмеженнями).

Глобальна стратегія побудування ефективно працюючого ядра - є перенесення максимуму операцій на взаємодію з розподіленою пам’яттю. Таким чином, основний паттерн - це перенесення всіх даних у розподілену пам’ять на початку виконання ядра, потім виконання на ній усіх розрахунків, і наприкінці - перенесення даних назад до глобальної пам’яті.

Ми не можемо отримувати значення з ядра напряму для дебагу, для цього використовується наступний трюк: ми створюємо додатковий глобальний масив потрібного розміру у пам’яті GPU, передаємо його до ядра, і, на потрібному нам моменті (наприклад - на кожному кроці алгоритму) записуємо туди всі дані котрі хочемо перевірити. Після завершення роботи - просто копіюємо їх до хосту і працюємо як зі звичайним масивом.

Синхронізація, якщо не брати до уваги графічні процесори останнього покоління, доступна лише у межах одного блоку, для її виконання викликається функція __syncthreads(). Це накладає обмеження на реалізацію, оскільки ми не можем напряму виконувати у блоці підзадачі для інших блоків. Блоки мають бути

незалежними, і, для, наприклад, реалізації групових операцій, ми повинні виконати повністю фрагмент розрахунків у блоці (наприклад знайти префікс-суму для кожного блоку), винести знайдені дані до глобальної пам’яті і потім вже з ними виконувати аналогічні операції.

Кожне ядро також може виконуватися паралельно з іншими, тому, якщо ми хочемо щоб вони виконувалися послідовно - нам необхідно вручну їх синхронізувати використовуючи функцію cudaDeviceSynchronize().

Також слід відмітити механізм реалізації отримання випадкових чисел у середині кернелу CUDA, оскільки це також не є тривіальним. Ми не можемо ефективно генерувати окреме випадкове число у реальному часі без попередньої підготовки. Для цього нам знадобиться проініціалізувати curandState з використанням для цього окремого ядра, де ми для кожного потоку що має генерувати певну послідовність випадкових чисел (у нашому випадку - на кожному кроці побудови шляху), ми використовуємо функцію curand_init(). Якщо помилитися при ініціалізації - наприклад неправильно визначивши кількість резидентних структур (потоків/блоків) що потребують генерації послідовностей випадкових чисел - ми отримаємо повтори однакових значень з певним інтервалом, наявність котрих доволі складно виявити (бо інтервал може бути високим), і, ми будемо бачити що алгоритм працює некоректно, але будемо шукати помилки у його реалізації, а не у генерації випадкових чисел. Аналогічно ситуація сталася і у процесі розробки.

Для отримання ж самого випадкового числа усередині кернелу - ми, під час запуску кернелу, передаємо туди, у якості параметру, наявний ініціалізований curandState, та при необхідності викликаємо усередині функцію curand_uniform(), де, у якості параметра передаємо елемент масиву curandState. що відповідає за генерацію випадкових чисел для структурної одиниці у котрій ми зараз знаходимось. Код кернелу що ініціалізує curandState представлений у додатку А(8).

Інтерфейс для розробки у CUDA не має вбудованого обробника помилок і у випадку помилки усередині ядра CUDA - ми не дізнаємося якого типу була помилка та під час виконання котрого з ядер вона трапилася. Для того щоб ці помилки ловити, реалізуємо функцію-обробник. Код функції-обробника є у додатку А(7). Далі нам треба навчитися застосовувати її до CUDA-команд що викликаються, для цього задефайнимо #define checkCudaErrors(val) check((val), #val, __FILE__,

__LINE__), та будемо обгортати у checkCudaErrors кожну функцію API CUDA що ми викликаємо у програмі.

Після завершення розрахунків, нам потрібно перенести їх результати з девайсу назад на хост. Для цього ми використовуємо ту ж функцію cudaMemcpy(), але у якості останнього параметру встановлюємо cudaMemcpyDeviceToHost що показує напрям копіювання.

Наприкінці роботи з GPU нам необхідно вручну звільнити виділену для розрахунків глобальну пам’ять, використавши до всіх створених вказівників функцію cudaFree().

4.4 Характеристика використовуємого GPU

Нам слід визначити характеристики, котрі визначає специфікація CUDA для використовуємої нами картки, для того, щоб при реалізації знаходитися у допустимих специфікацією рамках та, разом з тим, використовувати наявні ресурси оптимально. Тобто реалізація орієнтована під конкретний девайс буде працювати і на девайсах новішого покоління, але не буде для них оптимальною.

Ми працюємо з карткою Nvidia GeForce 940 MX, що відноситься до покоління Maxwell зі специфікацією версії 5.0. Відображати будемо не усі параметри, а лише ті, що впливають на особливості реалізації:

- Максимальна розмірність сітки: 2147483647 x 65535 x 65535
- Максимальне число потоків у блоці: 1024;
- Максимальне число незалежних блоків на мульти-процесор: 32;
- Максимальне число незалежних варпів на мульти-процесор: 64;
- Максимальне число незалежних потоків на мульти-процесор: 2048;
- Максимальне число розподіленої пам’яті на блок: 48 КБ;
- Розмір константної пам’яті: 512 КБ;
- Кількість мульти-процесорів: 3;
- Кількість ядер CUDA (шейдерних процесорів): 384;
- Максимальне число операцій на кернел: 512 * 10^9.

Під “незалежними” маються на увазі ті, що виконуються істинно паралельно. Одним з ключових висновків що ми отримуємо з даної характеристики є кількість істинно паралельних потоків, що можуть виконуватися на девайсі: 3 мульти-процесори * 64 варпи * 32 потоки = 6144 потоків. Для відеокарт навіть цього ж

покоління це є доволі невеликим числом, через малу кількість мульти-процесорів. Наприклад для відеокарти того-ж покоління Maxwell GTX 750 Ti ми маємо 10240 потоки, а для останнього покоління - наприклад TITAN RTX, що коштує близько 2.5 тис. доларів - ми маємо 72 мульти-процесори та близько 74 тисяч істинно паралельних потоків.

Псевдо-паралельно ж, для нашої графічної картки, кількість потоків обмежується пам’яттю що ми на них виділяємо, оскільки верхня границя розмірів сітки згідно специфікації (9.2230906e+18) фактично не є досягаємою.

5 РОЗРОБКА ТА ОПТИМІЗАЦІЯ РЕАЛІЗАЦІЇ МУРАШИНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ CUDA

5.1 Загальна характеристика процесу розробки та оптимізації

Першою поставленою ціллю - була безпосередня імплементація алгоритму у довільному вигляді, не звертаючи особливої уваги на швидкодію та отриману ефективність. Потрібно було отримати основу, відштовхуючись від котрої ми б мали змогу виконувати подальші оптимізації та генерувати ідеї для більш широкого розпаралелювання.

Наступні версії це або оптимізація слабких місць минулої реалізації або втілення нових ідей що стосуються безпосередньо алгоритму.

Розробка та проміжне тестування відбувалися у Visual Studio, а до остаточних реалізацій поточної версії додавалася обгортка, що була описана у розділі 3.2, та разом компілювалося з використанням компілятору NVCC через командну строку з використанням команди формату nvcc -Xcompiler "/MD" 1.cu -o 1, на виході ми отримуємо .exe файл.

Для отриманого .exe файлу ми запускаємо BAT-скрипт описаний у розділі 3.2, по завершенню його роботи - отримуємо записи у базі даних що стосуються реалізації що тестується. С цих даних ми вибираємо набори вхідних даних невеликого розміру для котрих ми отримуємо неправильне рішення виконуючи запит до бази даних формату select from where, повний код типового запиту представлений у додатку А(9). Ми беремо саме графи невеликого розміру, що б ми могли наглядно представити їх матриці суміжності, побудовані мурахами дороги, матрицю феромонів та інші дані що допоможуть проаналізувати поведінку реалізації на цих даних та знайти що викликало помилку.

Знайдені у процесі аналізу результатів окремої версії крайні випадки, проблеми, речі на котрі необхідно звертати увагу у подальших реалізаціях або при розгляданні результатів тестування подальших версій, знайдені причини помилок записувалися до спеціального файлу, ітогове наповнення цього файлу знаходиться у додатку Б(1).

Також підтримувалися ще декілька файлів, для спрощення роботи з проектом та аналізу виконаної діяльності. Це файли:

- patch.txt, зберігає інформацію про ключові зміни що були внесені для отримання наступних версій, текст у додатку Б(2);

- versions.txt, зберігає коротку характеристику кожної версії реалізації, показуючи на чому вона заснована, текст у додатку Б(3);

- ideas.txt, до цього файлу записувалися глобальні ідеї по зміні реалізації, котрі можуть значно вплинути на швидкодію або точність та потребують великих зусиль для їх імплементації, але котрі не входять то поточної версії, текст файлу у додатку Б(4).

При розробці нової версії, ми намагаємося полагодити коректність роботи на помилкових тестах згідно знайдених помилок, та/або намагаємося оптимізувати реалізацію глобально (наприклад, за рахунок покращення принципів роботи з пам’яттю), та/або намагаємося застосувати знайдені нові глобальні ідеї покращення алгоритму.

Далі перейдемо безпосередньо до розглядання розроблених версій, відмічаючи при цьому ключові їх особливості, отриману швидкодію та застосовані глобальні ідеї.

5.2 Глобальна структура реалізації оптимізаційного алгоритму Мурашиної Колонії на GPU

Щоб мати змогу орієнтуватися у наявних версіях та особливостях їх реалізації, опишемо загальну структуру алгоритму, що реалізовувався, з точки зору саме програмної архітектури CUDA.

Глобально у нас є дві послідовні частини алгоритму, котрі завжди мають знаходитися у окремих кернелах:

- Побудова шляхів;

- Оновлення значень феромонів.

Забігаючи наперед, другий етап у останніх реалізаціях був розбитий ще на окремі два кернели, що було зумовлено необхідністю зробити між блочну синхронізацію через використання у кожному блоці на другому етапі масиву значень отриманих у різних блоках на першому.

Етап побудови шляхів складається з циклу що йде по кожній мурасі до тих пір, доки ми не потрапимо у глухий кут, або не потрапимо до фінішної вершини шлях до котрої нам і необхідно знайти.

На кожній ітерації циклу ми можемо виділити по декілька виконуваних ключових під етапів:

- розрахунок коефіцієнтів вірогідності переходу по шляху для усіх сусідніх ребер графу;

- підготування масиву для вибору оптимального ребра (виконується з використанням inclusive_scan);

- вірогідністний вибір ребра для переходу відносно знайдених коефіцієнтів ефективності (виконується з використанням upper_bound);

- додання нової вершини до шляху.

Другий глобальний етап - оновлення значень феромонів також можна розділити на декілька під етапів:

- зниження значення феромонів у усіх вершинах на фіксований коефіцієнт випаровування;

- розрахунок довжин побудованих шляхів;

- оновлення значень ребер, що є складовими побудованих шляхів котрі дісталися фінальної вершини, на значення обернено пропорційне до довжини цього шляху.

На рисунку 5.2.1 відображена глобальна структурна схема алгоритму що була описана вище.

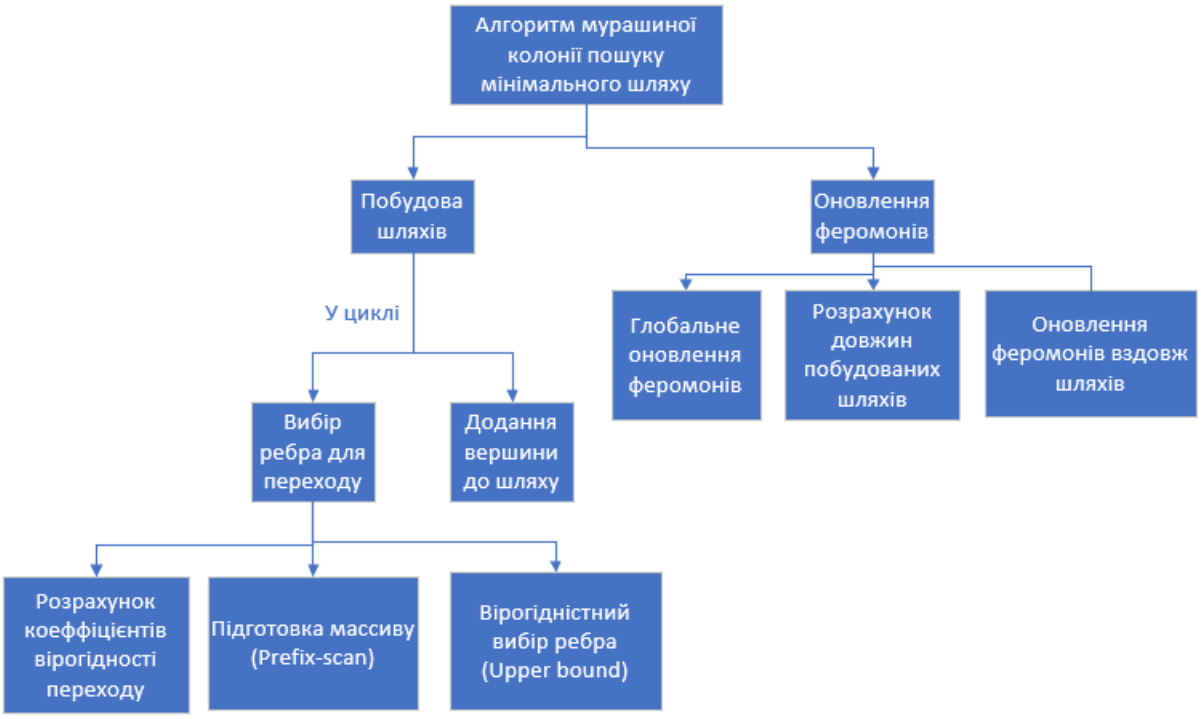

Рис. 5.2.1 - Структурна схема реалізацій алгоритму

Також, у кожній версії, окрім відмінностей у реалізації окремих етапів алгоритму, можуть відрізнятися кількість працюючих мурах, степенні коефіцієнти що характеризують залежність вибору шляху від довжини/кількості феромонів на ребрі, степенні коефіцієнти що визначають залежність додаваємої кількості феромонів на ребрах від довжини шляхів що через ці ребра проходять, способи визначення числа ітерацій алгоритму (повторень зв'язки пошук шляху + оновлення значень феромонів) та коефіцієнт випаровування.

5.3 Версія 0

Як згадувалося у розділі 5.1, першою ціллю була пряма імплементація алгоритму. Засновувалася вона на ідеї виділення кожній мурасі окремого потоку, оскільки процес побудови кожного шляху є незалежним, а працюють вони з загальними даними, та максимізації числа мурах для отримання максимальної точності (чим більше мурах - тим більше вірогідність що коректний шлях буде знайдений хоча б однією з них).

Таким чином, ми використовували кількість блоків по числу наявних CUDA-ядер (шейдерних процесорів), 384, а кожному блоку давали максимально можливе число потоків (1024), де кожен з потоків кожного блоку представляв собою незалежну мураху.

Охарактеризуємо особливості структури кернелу побудови шляхів. Кожна мураха має зберігати інформацію про те, котрі вершини були вже використані у побудові шляху, тому кожна мураха напочатку у циклі по кількості вершин записує до shared memory E булевих значень (одиниць), а потім відмічає стартову вершину шляху як ту що вже була використана (нуль). Далі починається цикл побудови шляху. Розрахунок коефіцієнтів вірогідності переходу виконується у циклі по E, самі коефіцієнти зберігаються до розподіленої пам'яті, а розраховуються як додаток булевого значення що показує чи використана була вершина, на кількість феромону та ділення на довжину ребра до вершини, попередньо перевіряється чи не нульове це ребро.

Підготовка масиву (префікс-скан) виконується з використанням thrust::inclusive_scan(), thrust - бібліотека, що э аналогом STL для алгоритмів на GPU і дозволяє використовувати їх з великим рівнем абстракції (навіть не працюючи

безпосередньо з CUDA, можна виконувати розрахунки з його використанням просто викликаючи функції з бібліотеки.

Виконується цей префікс-скан за O(log(n)), але, при цьому, використовує не наявні потоки, а викликає нове ядро з новими потоками, що, як пізніше виявилося, доволі сильно сповільнює роботу алгоритму, оскільки така кількість ядер/потоків не може одночасно розраховуватися для використовуємого нами графічного процесора.

Далі, про те, що стосується вірогідністного вибору ребра. Для цього ми генеруємо випадкове число від 0 до 1, механізм генерації був описаний у розділі 4.3. Множимо згенероване число на префікс-суму, і отримуємо випадкове число від 0 до максимуму префікс суми. Тепер ми маємо знайти перше число серед префікс-суми, що більше або дорівнює згенерованому, для цього використовуємо функцію thrust::upper_bound, що виконується за O(1), але, як і у попередньому випадку, вимагає генерації нового ядра та нових потоків, що додає до O(1) великий прихований коефіцієнт.

Що стосується ядра оновлення феромонів, сітку ми використовуємо аналогічної розмірності. Глобальне оновлення феромонів (випаровування) відбувається у кожному потоці що менший за кількість вершин - у циклі проходяться усі ребра що виходять з даної вершини, тобто виконується за O(V). Далі розділена пам’ять ініціалізується нулями і у циклі кожна мураха прораховує довжину побудованого шляху за O(V). Оновлення значень феромонів вздовж побудованих шляхів виконується аналогічно, кожна мураха додає до кожного ребра на побудованому шляху число зворотно пропорційне довжині цього шляху (кількість фермонів за замовчуванням, число котрим вони ініціалізуються, ділена на довжину шляху).

Оцінимо асимптотичну складність що ми отримуємо при такій реалізації. При побудові шляхів ми ініціалізуємо за O(V), на кожному кроці циклу розрахунок коефіцієнтів виконується за O(V), префікс-сума за O(logV), вибір ребра для переходу за O(1), всього у нас максимум E ітерацій циклу, таким чином для етапу побудови шляхів отримуємо O(V + V*(V + logV + 1), за найвищою степенню - O(V^2). При оновленні феромонів, випаровування відбувається за O(V), розрахунок довжин за O(V) і оновлення ребер вздовж шляхів O(V), таким чином загальна складність зводиться до O(V). Повна асимтотична складність - по старшій степені - залишається O(V^2).

Така асимптотика не краща за асимптотичну складність що пропонується звичайними лінійними алгоритмами, а до того ж ми маємо великі приховані коефіцієнти, зумовленість котрих була описана вище у цьому ж розділі. До того ж були виявлені ще деякі проблеми. Кожна мураха потребує у розподіленої пам’яті принаймні V+V*4 місця (ми зберігаємо V булевих змінних що відображають відвіданість вершин а також V float змінних для виконання операцій з коефіцієнтами ефективності переходу), кожен блок може виділити максимум 48КБ і таким чином для, наприклад, графу на 10^3 вершин ми можемо виділити лише по 9 мурах на блок, що дуже не ефективно, так як у нас простоюють “зайві” потоки, а у той же час для задіяних потоків послідовно виконуються операції що могли б бути розпаралелені, і що в принципі знищує ідею “кожній мурасі - по потоку” на великих графах. Також була виявлена проблема пов’язана з конфліктами доступу до пам’яті на етапі оновлення феромонів вздовж шляхів, так як при такому варіанті реалізації можлива ситуація коли ми одночасно оновлюємо одне і те ж ребро у глобальній пам’яті у двох різних потоках, що провокує невизначену поведінку.

5.4 Версії 1-2

Зміни у 1 та 2 версіях мінорні, тому об’єднані до одного розділу. У 1 версії була змінена політика взаємодії з бібліотекою thrust, замість використання розпаралелювання на GPU, ми виконуємо алгоритми thrust лінійно у тому ж потоці у котрому будується шлях, таким чином ми погіршуємо асимптотичну складність префікс-суми та upper bound, але сильно покращуємо приховані коефіцієнти. Фактичний приріст від цього - у 3-4 рази. Також у 1 версії ми оптимізували роботу в кернелах, за рахунок попереднього запису до швидкої пам’яті використовуємих id блоків, потоків та інше, а також попереднього розрахунку активно використовуємих виразів (наприклад, ми часто використовували значення blockDim.x * E, для орієнтування у матрицях, тепер це записано у одну змінну та при необхідності - викликається напряму. Це дало приріст у 1/8 разів.

Версія 2 була здебільшого тестовою, в ній ми спеціально сповільнили роботу реалізації за рахунок заміни розподіленої пам’яті на глобальну у всіх кернелах. Це було зроблено для того що б розглянути залежності точності результату від кількості мурах, налаштувати степенні коефіцієнти та перевірити працездатність реалізації алгоритму на великих даних (на котрих ми запустити реалізацію при

попередній моделі роботи з пам’яттю не могли через недостачу розподіленої пам’яті на блок).

5.5 Версія 3

У 3 версії була реалізована одна із глобальних ідей, що з’явилися у якості альтернатив для вирішення проблем виникших у версії 0. Як було вказано у розділі 5.3, при спробі максимізації числа мурах шляхом виділення кожній окремого потоку, ми потрапляємо до парадоксальної ситуації, коли, через недостачу швидкої пам’яті у нас простоюють зайві потоки, у той час коли самі операції усередені використовуємих потоків виконуються лінійно, замість того щоб бути розпаралеленими. Таким чином ми використовуємо не усі ресурси і при цьому маємо слабку асимптотику.

Ідея що була реалізована у версії 3 складається у тому, що кожна мураха представляється не потоком, а блоком потоків, котрі реалізують колективні операції усередині неї, за рахунок чого ми знижуємо кількість мурах, але максимально щільно використовуємо доступні ресурси і через це отримуємо кращу асимптотику і приховані коефіцієнти.

У ядрі побудови шляхів, кожен потік ініціалізує одну булеву змінну, і виконує це за O(1), цикл по елементам шляху залишається, але змінюється наповнення кожної його ітерації. На етапі визначення вірогідністних коефіцієнтів кожен потік блоку працює з ребром до вершини що відповідає його номеру у блоці, таким чином ця стадія також виконується за O(1).

Процес розрахунку префікс-суми сильно змінився, оскільки наявні бібліотеки не вміють розпаралелювати рішення на конкретні виділені потоки і нам довелося створювати його вручну. Використаємо наївний алгоритм що був розроблений Хіллісом і Стілі у 1986 році [9], у котрому виконується цикл по log2(n) елементів, і на кожній i-тій ітерації кожен k-тий елемент сумується до елемента що знаходиться на k+2^i позиції у масиві i, таким чином працює за O(log2(n)). Схема роботи алгоритму зображена на рисунку 5.5.1.

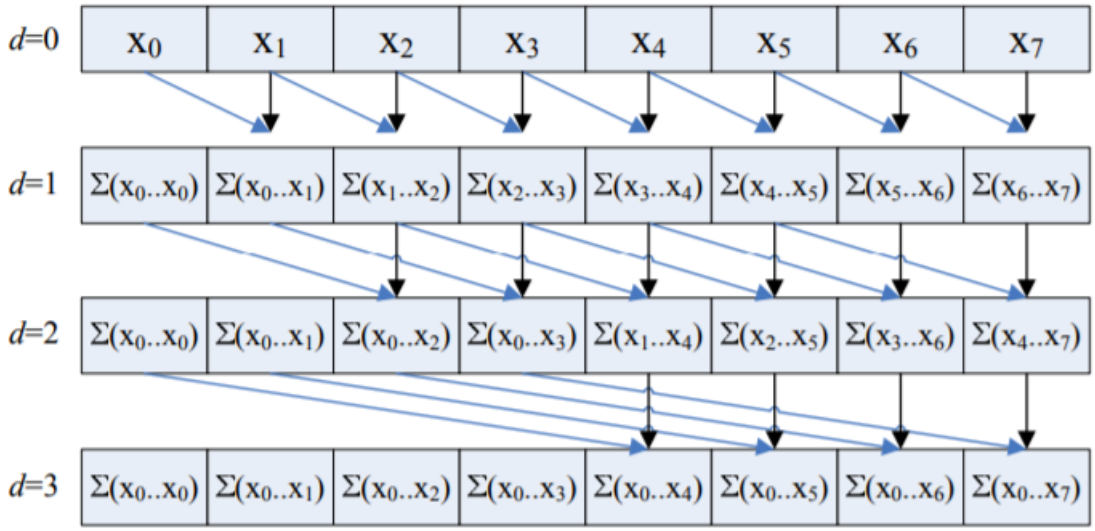

Рис. 5.5.1 - Алгоритм Хілліса-Стілі

Для імплементації цього алгоритму ми створюємо цикл що йде до кількості вершин графу по степеням двійки, в ідеальному варіанті, для максимальної швидкодії, ми маємо використовувати два масиви записуючи послідовно результати то в один то в інший, але у нас обмежена розподілена пам’ять і ми будемо працювати одним, для цього ітерація розбивається на два кроки, на першому кожен потік запом’ятовує вершину значення котрої має бути доданим до нього, потім виконується синхронізація усіх потоків між собою, щоб можна було бути певним що кожен потік зафіксував число що має додати до наявного значення до того як це значення буде змінено. На другому кроці - відбуваються призначення збережених значень до вершин відповідних кожному потоку, таким чином реалізація працює за O(log2(n)), що відповідає асимптотиці запланованого алгоритму.

Також ми змінили механізм рандомізації, так як кожен потік має отримати ідентичне число, ми додали до розподіленої пам’яті ще один слот під float, у нульовому потоці ми отримуємо випадкове число, зберігаємо його до розподіленої пам’яті і далі з усіх інших потоків взаємодіємо безпосередньо з ним.

Upper bound також не має готових реалізацій розпаралелених на фіксованих потоках, тому ми його також реалізували вручну. Масивно-паралельна реалізація upper bound є тривіальною, кожен потік розглядає вершину що відповідає його положенню у блоці та сусідню вершину, той що зустріне “перепад” - тобто ситуацію коли елемент зліва менший за число позицію котрого ми шукаємо, а елемент справа - більший або дорівнює йому, окремо розглядаються ситуації з крайніми потоками. Очевидно що у монотонно зростаючому масиві котрий ми отримуємо після

виконання префікс-скану можлива лише одна позиція що відповідає умовам, а отже ми її знайдемо і не отримаємо конфлікту по даним. Працює така реалізація за O(1).

Розглянемо зміни ядра що займається оновленням феромонів. Для випаровування кожен потік оновлює значення у ребрах що дорівнюють його глобальному номеру у сітці по модулю від загальної кількості потоків (кількість блоків * кількість потоків у блоці), при виділення достатнього числа потоків - працювати це буде за O(1). Змінена була ініціалізація, тепер кожен блок зберігає інформацію про вагу ребер що входять до шляху котрий відповідає положенню блоку у сітці, і кожен потік блоку ініціалізує одне значення що відповідає переходу з вершини з положенням відповідним положенню потоку у блоці. Виконується це за O(1).

Для розрахунку довжини шляху ми використовуємо паралельну реалізацію суми, у котрій ми в циклі по степеням двійки за наївною структурою бінарного дерева за логарифм від числа вершин у шляху знаходимо суму відстаней на цьому шляху.

Етап оновлення значень фермонів вздовж шляхів довелося винести у окреме ядро через те, що нам необхідно мати інформацію про довжину усіх шляхів для того щоб ефективно реалізувати оновлення даних. Проблема у тому, що ми не можемо йти вздовж усіх шляхів паралельно та оновлювати значення, оскільки у двох різних шляхах у один час може опинитися одне й те саме ребро і тоді ми отримаємо невизначену поведінку програми. Для того щоб це виправити була використана ідея що була представлена у рамках вирішення задачі комівояжера на GPU у 2012 році [10]. Вона складалася у тому, що ми представляємо шляхи не у вигляді послідовності вершин, а у векторі де на і-тій позиції знаходиться вершина до котрої ми переходимо з і-тої, і ми, таким чином, беручи один стовбець у матриці доріг, можемо бути впевнені що він не буде конфліктувати з іншими і можемо обчислювати їх паралельно. В рамках же ядра, ми плануємо при розгляданні і-того стовбця завантажити до розподіленої пам’яті і-ту строку з матриці феромонів та сам стовбець з матриці доріг, оновити вся значення в циклі та вивантажити результат назад до глобальної пам’яті.

На рисунку 5.5.2 зображена схема пропонуємого до реалізації алгоритму оновлення значень феромонів вздовж доріг.

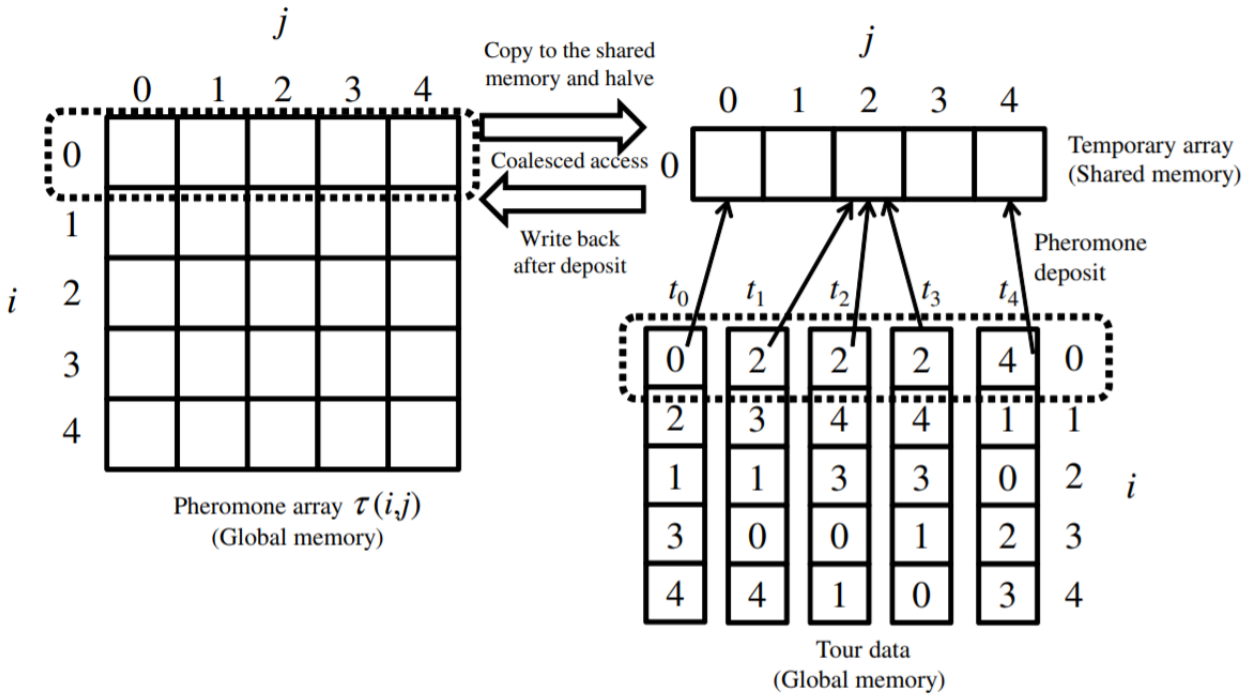

Рис. 5.5.2 - Схема алгоритму оновлення феромонів вздовж доріг

Для імплементації цієї ідеї, окрім створення нового ядра, ми маємо змінити і ядро пошуку шляхів, що б знайдені шляхи записувалися у потрібному нам вигляді, і першу частину ядра оновлення феромонів, оскільки тепер у нас інший механізм визначення завершення шляху. Тепер ми на етапі ініціалізації у другому ядрі (коли переносимо інформацію про довжину ребер через котрі проходять шляхи) спочатку помічаємо ребро що виходить з фінального елементу -inf, а якщо у процесі ініціалізації ми з певної вершини потрапимо до нього (максимум це може бути справедливим для однієї вершини), то встановимо значення в нуль. Таким чином у ситуації коли шлях не потрапив у фінальну вершину - він у нас буде від’ємного розміру і не буде розглядатись.

Для передачі інформації про довжини шляхів між ядрами був створений новий глобальний масив dsums що зберігає інформацію про знайдені довжини шляхів, наприкінці ядра глобального оновлення феромонів та розрахунку довжин шляхів ми записуємо отриманні значення до глобальної пам’яті, а у новому ядрі при ініціалізації завантажуємо їх до розподіленої пам’яті. В новому ядрі ми тепер виконуємо оновлення вздовж шляхів у циклі по кількості мурах, додаючи числа напряму до глобальної пам’яті, тобто, асимптотично це O(1), оскільки не залежить від розміру вхідних даних.

Оцінимо ітогову отриману асимптотику, фаза пошуку шляхів відбувається за O(1 + V*(1 +log2(V) + 1), по найвищому степені - O(V*log2(V)), та, окрім того, можемо помітити що знизилася, у порівнянні з попередніми реалізаціями, не лише найвища степінь, а й інші складові, тобто фактичне прискорення стало більшим за V/log2(V).

Фаза оновлення шляхів відбувається за O(1 + 1 + log2(V) + 1), за найбільшою степенню - O(log2(V). Фінальна асимптотична складність для цієї реалізації становить O(V*log2(V)), що вже краще за наявні лінійні аналоги.

Також, змінилася кількість пам’яті що задіяна блоками, нам тепер необхідно E*sizeof(bool) + E*sizeof(float) + 2 для блоків задіяних у побудові шляху, та E*sizeof(float) для блоків задіяних при оновленні феромонів, за рахунок чого ми маємо змогу проводити тестування на більших графах.

Загалом, отримана для цієї реалізації асимптотична складність - вже є достатньою і подальшим завданням стала оптимізація прихованих коефіцієнтів та степенних коефіцієнтів що застосовуються на різних етапах алгоритму, для підвищення точності.

5.6 Реалізації 4-6

Зміни у 4-6 версіях мінорні, тому їх було вирішено об’єднати до одного розділу. У четвертій версії ми займалися конкретно підбором коефіцієнтів, проблема котра була виявлена на неправильних тестах - це ситуації коли ми маємо настільки великий процентний розрив між довжинами альтернативних ребер, що до більшого з них мурахи просто не потраплять. Щоб виправити це, ми, знизили вагу вірогідністного розриву, за рахунок автомиточного додавання до довжини ребра константного значення при розрахунку коефіцієнту вірогідності переходу. Також ми максимізували вагу довжини отриманого шляху при визначенні числа феромонів зо будуть додані до ребра, таким чином, що якщо хоч одна мураха знайте коректний шлях - то приріст буде достатнім щоб переважити всі альтернативи. Сумісно ці зміни сильно підвищили точність на невеликих графах, але знизили для ситуацій коли ми не можемо знайти коректний шлях на перших ітераціях, таким чином середня точність майже не змінилась.

Версія 5 - це намагання змусити один блок працювати з більшим числом вершин. У попередніх версіях, на один блок могло приходитися до 2^10 вершин

(2^20 ребер), але у нас в блоках достатньо вільної розподіленої пам’яті для роботи з 2^11-2^12. Для реалізації цієї ідеї ми усі розрахунки де потік додає відповідне своїй позиції значення, на цикли що виконують аналогічні операції для усіх значень з позицією що по модулю від кількості потоків у блоці дорівнює позиції цього потоку. Але ми отримали занадто сильне сповільнення, через необхідність багатократного використання порівняльного модулю у всіх потоках, і, дивлячись на результати попереднього тестування, не побачили сенсу у відправці на повне тестування.

У версії 6 ми розробили механізм визначення кількості ітерацій що слід здійснити алгоритму. Нагадаємо, що раніше кількість ітерацій подавалася на вхід і була згенерована попередньо. Дивлячись на результати проведених експериментів на інших версіях, ми визначили що оптимальне рішення ми отримуємо з більшою вірогідністю тоді, коли у процесі виконання значення феромонів наближаються до одиниці, але не стають меншими за неї. Таким чином ми вивели формулу (5.6.1) що визначає за скільки кроків кількість феромонів стане менше одиниці за рахунок лише випаровування.

$$n = \log(1/def)/\log(vap_coef) \tag{5.6.1}$$

Де n-кількість ітерацій, def - кількість фермону за замовчуванням, vap_coef - коефіцієнт випаровування;

Використання такого механізму вибору кількості ітерацій помітно повисило точність реалізації.

5.7 Версія 7

У версії 7 ми займалися оптимізацією прихованих коефіцієнтів, за рахунок використання більш ефективних алгоритмів у кернелах, а також максимального перенесення даних з котрими ми у кернелах взаємодіємо багатократно до розподіленої пам’яті.

У кернелі побудови шляхів ми замінили алгоритм пошуку префікс-суми. Раніше ми використовували наївний алгоритм Хілліса-Стілі, котрий не є оптимальним. Для цієї версії ми реалізували алгоритм, що у науковій літературі називається Work-Efficient, це є ексклюзивним алгоритмом (алгоритмом Блеллоха

[11]), де ми отримуємо префікс суму для усіх елементів крім останнього, тому його довелося виправити. Асимптотично він працює так як і алгоритм Хілліса-Стілі, але в процесі ми виконуємо значно менше операцій, що провокує зниження прихованих коефіцієнтів. Основою алгоритму є використання розповсюдженого для паралельних обчислень паттерну: збалансованих дерев. Ідея полягає в тому, щоб побудувати збалансоване бінарне дерево на вхідних даних і пройти по ньому до кореня і назад щоб розрахувати префікс сумму. Бінарне дерево з n листями має log n рівнів, а кожен рівень d∈ [0, logn) має 2^d вузли. Якщо ми виконуємо одне додавання на вузол, то виконуватимемо O(n) додавань за один обхід дерева. У алгоритмі ж Хілліса-Стілі ми виконуємо O(nlogn) додавань. Алгоритм складається з двох фаз: проход вверх до кореня, та зворотній проход. Слід усвідомлювати що ми не створюємо структури “бінарне дерево”, це лише визначає механізм роботи потоків на кожному кроці обходу.

На першій фазі алгоритму ми проходимо циклом до log2(E/2), на кожному кроці кожен потік додає між собою дві вершини, з кроком між собою в 2^i, де i - номер ітерації циклу. У результаті роботи першої фази алгоритму ми отримуємо часткові суми. На рисунку 5.7.1 зображена схема першої фази Work-Efficient алгоритму.

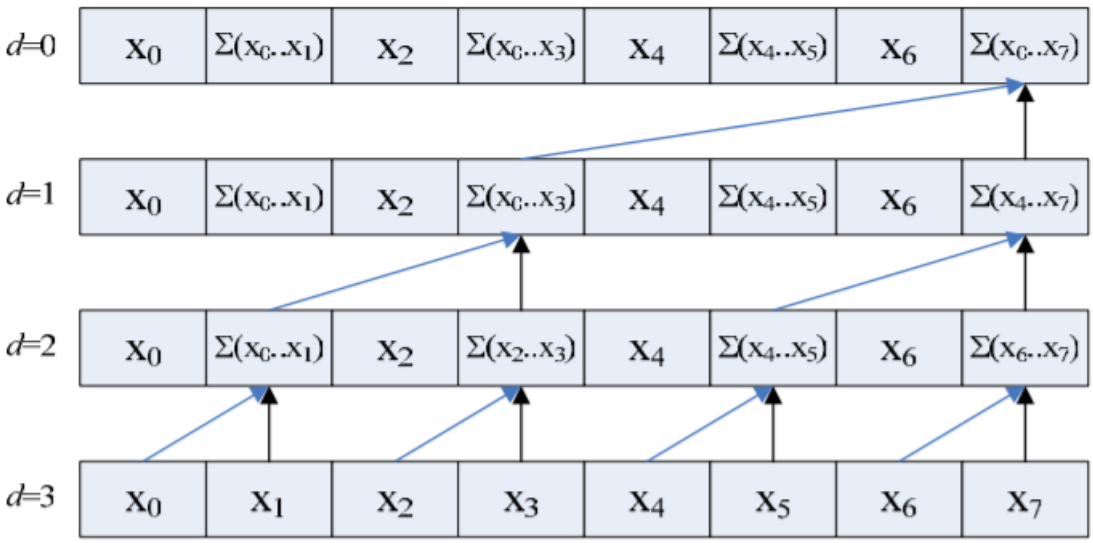

Рис. 5.7.1. Перша фаза Work-Efficient алгоритму

Псевдокод першої фази Work-Efficient алгоритму побудови префікс-суми Блеллоха:

```
for d := 0 to log2n - 1 do
     for k from 0 to n – 1 by 2d + 1 in parallel do
          x[k + 2d + 1 - 1] := x[k + 2d - 1] + x [k + 2d + 1 - 1]
```

У фазі розгортки ми обходимо дерево з кореня, використовуючи часткові суми для побудови префікс-суми на масиві, використовуючи часткові суми, обчислені на попередній фазі. Зауважимо, що оскільки це ексклюзивне сканування (тобто загальна сума не включена в результати), між фазами ми маємо ініціалізувати останній елемент масиву нулем. Цей нуль поширюється назад до головної частини масиву під час фази розгортки. На протязі циклу виконуються дії аналогічні першій фазі, але в зворотному порядку (зверху вниз), та 1 додаткова операція присвоєння для кожному кроці.

На рисунку 5.7.2 зображена схема другої фази Work-Efficient алгоритму побудови префікс-суми Блеллоха.

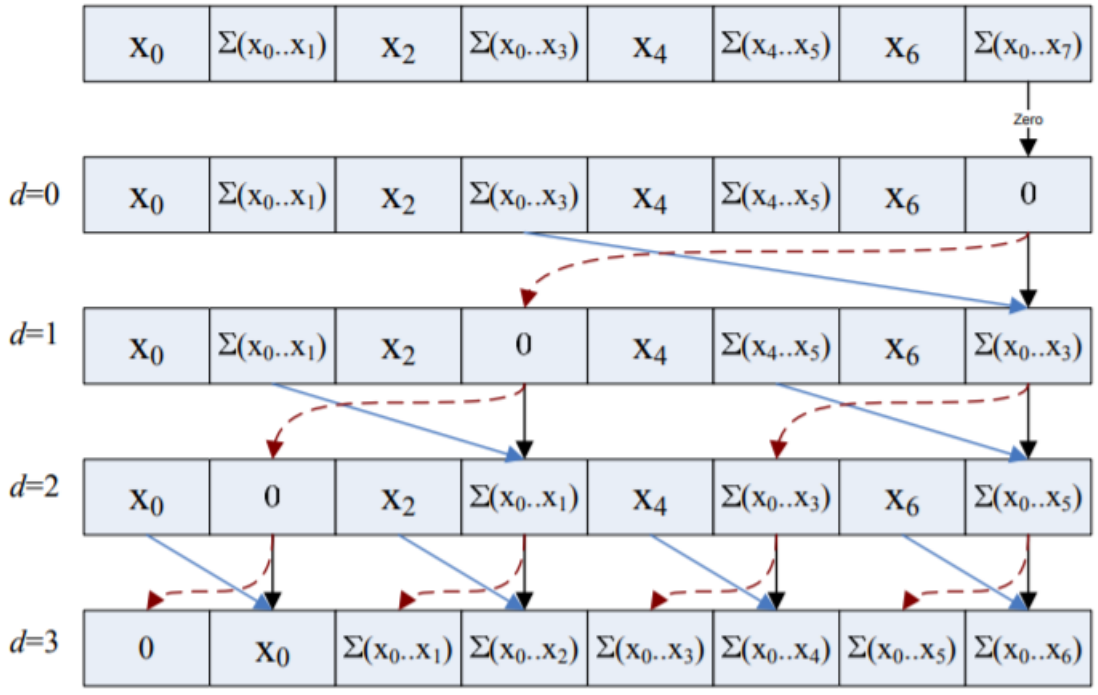

Рис. 5.7.2 - Схема фази розгортки

Псевдокод фази розгортки Work-Efficient алгоритму побудови префікс-суми Блеллоха:

```
x[n - 1] := 0
for d := log2n down to 0 do
```

```
for k from 0 to n – 1 by 2d + 1 in parallel do
     t := x[k + 2d - 1]
     x[k + 2d - 1] := x [k + 2d + 1 - 1]
     x[k + 2d + 1 - 1] := t + x [k + 2d + 1 - 1]
```

Також ми зайнялися оптимізацією найбільш вразливого місця CUDA - роботи з пам’яттю - намагаючись перенести усі данні до котрих ми звертаємося більше одного разу до розподіленої пам’яті. У ядрі пошуку шляхів зробити оптимальніше іже не було можливості, оскільки ми максимально по здійснюємо максимум E взаємодій з матрицею феромонів та матрицею суміжності, а що б покрити усі можливі випадки - нам доведеться занести до розподіленої пам’яті E^2 елементів, що не вигідно. До матриці доріг - ми записуємо E значень, не виконуючи на них жодних розрахунків, що також не можна зробити оптимальніше.

У ядрі глобального зрізання феромонів та розрахунку довжин шляхів ми також по одному разу зверталися до елементів матриці доріг та матриці суміжності і зробити це оптимальніше не можемо навіть теоретично.

У ядрі оновлення феромонів вздовж шляхів ми з масивом знайдених довжин працювали лише однією операцією, що є оптимальним, а от у циклі що відповідає безпосередньо за оновлення феромонів - ми багатократно додаємо значення напряму до матриці феромонів у глобальній пам’яті, що є повільним і може бути оптимізованим. Оскільки кожен блок відповідає за переходи з однієї вершини, то нам достатньо завантажити до розподіленої пам`яті E ребер з глобальної пам’яті, далі оновлювати їх значення в циклі, та завантажувати назад. Також ми багатократно звертаємося до матриці шляхів, для того що б визначити по котрому ребру відбувався переход з кожної вершини. Оскільки кількість мурах у нас фіксована і завідомо менша за розмірність блоку - ми можемо собі дозволити завантажити стовбець цієї матриці до розподіленої пам’яті. Більш того, якщо ми завантажимо лише один стовбець - то частина потоків блоку буде простоювати на стадії ініціалізації, і у нас залишаються потоки на останньому циклі, оскільки він виконуються лінійно по кількості мурах в одному потоці. Це дозволяє нам завантажити не одну строку з матриці феромонів та один стовпець з матриці шляхів, а по декілька (оптимальним числом було 3), таким чином задіявши потоки що раніше простоювали (ураховуючи механізм варпів - це є важливим).

5.8 Версія 8

У 8 версії ми звернули увагу на механізм відновлення шляху. Раніше ми виконували його на CPU обираючи максимальне значення феромонів серед ребер кожної вершини починаючи зі стартової, і переходячи до іншої. Це працювало у гіршому випадку за O(E), що нас не влаштовує. Щоб пришвидшити, ми додали ще одне ядро до GPU, у котрому ми попередньо розраховуємо максимуми серед ребер з кожної вершини, виконується це за O(V). І уже з попередньо розрахованими максимумами ми на CPU будуємо шлях за O(V).

6 ПРЕДСТАВЛЕННЯ РЕЗУЛЬТАТІВ РОБОТИ З АЛГОРИТМОМ МУРАШИНОЇ КОЛОНІЇ

6.1 Ітоговий вигляд алгоритму

Використовувані об’єкти:

- Adj - матриця суміжності;
- Pher - матриця феромонів;
- Roads - матриця доріг;
- start - початкова вершина;
- end - фінальна вершина;
- V - кількість вершин графу;
- road - вектор оптимальних ребер для переходу з вершини;
- def - кількість феромонів за замовчуванням;
- ants - кількість мурах;
- block_size - максимальна кількість потоків на блок згідно GPU;
- vap_coef - коефіцієнт випаровування;
- pow_pher - степенний коефіцієнт важливості феромонів;
- pow_dist - степенний коефіцієнт важливості довжини ребер;
- pow_road - степенний коефіцієнт важливості довжини шляху.

Вхідні дані:

Граф у вигляді матриці суміжності Adj, кількість вершин V, стартова та фінішна вершини, start та end відповідно.

Хід алгоритму:

Матриця феромонів Pher розмірності (V, V), заповнюється значеннями за замовчуванням def. Оптимальний def - максимальне значення що ми плануємо зустріти в графі, згідно предметної області, або згідно попередньої оцінки. Матриця доріг Roads розмірністю (ants, V), заповнюється значеннями start.

У глобальну пам’ять GPU переносимо Adj, Pher, Roads. Далі виконуємо ітерації алгоритму у кількості n = log(1.0 / def) / log(vap_coef). На кожній ітерації виконуємо три ядра.

Ядро пошуку шляхів. Для оптимальної швидкодії, одній марусі знадобиться V потоків для обробки графу з V елементами, тому розмірність сітки визначаються як добуток кількості мурах на відношення кількості вершин до розміру блоку, ceil(ants

* V/block_size). Виділяємо (threadnum + 3) * sizeof(float) + (threadnum * sizeof(bool) простору у розподіленій пам’яті, масив tmp на threadnum+1 float для зберігання даних розрахунків (+1 через те що у нас використовується exclusive scan), масив used на threadnum bool для інформації про використованість вершин, int та float змінні для зберігання інформації про поточну вершину prev та згенероване випадкове число myrand. Ініціалізуємо used нулями паралельно у всіх потоках, у 0 потоці визначаємо start вершину used. В циклі по V з додатковою умовою на used[end] == 0 виконуємо побудову шляху. На кожній ітерації, перебираємо усі наявні ребра з поточної вершини, для кожної паралельно розраховуємо значення коеффіцієнту вірогідності переходу й записуємо до tmp: tmp[thid] = pow(Pher[edge], pow_pher)/pow(Adj[edge]). Використовуючи алгоритм Work Efficient Prefix Scan будуємо паралельну префікс суму, але на початку reverse step ми не лише обертаємо останній елемент (що зберігає інформацію про inclusive scan) у нуль, а попередньо зберігаємо його у tmp[threadnum]. Генеруємо випадкове число від 0 до tmp[threadnum] і зберігаємо його до myrand. Виконуємо паралельний upper_bound для згенерованого числа відносно розрахованої префікс-суми і додаємо знайдену позицію до матриці доріг у вигляді переходу з поточної вершини, оновлюємо значення prev.

Ядро глобального випаровування феромонів та розрахунку довжин шляхів. Розмірність сітки по кількості мурах. Число потоків - по threadnum. Пам’яті виділяємо threadnum*sizeof(float). Також створюємо глобальний масив sums, для зберігання довжин шляхів. У ядрі виділяємо розподілений масив tmp на threadnum float елементів. Розподіляємо всі ребра між усіма потоками та виконуємо випаровування (Pher[thid]*=vap_coef), tmp[end] встановлюємо як -inf, Записуємо до tmp всі довжини ребер що складають шлях що розглядається у поточному блоці. Для цього кожен поток розглядає ребро що виходить з вершини що відповідає його порядковому номеру (tmp[thid] = Adj[road[block_id, thid]]), і, якщо потрапляє до фінальної вершини, встановлює tmp[end] у нуль. Таким чином, якщо ми не потрапили у шляху до фінальної вершини - він буде менше нуля. Далі реалізуємо параллельну сумму, а отримане значення, згідно обраного степенного коефіцієнта, - записуємо у відповідну ячейку масиву sums: sums[block_id] = 1/pow(tmp[0], pow_road). Ми одразу обчислюємо значення, оскільки тут вони виконуються один раз, а у наступному ядрі - викликаються багатократно.

Ядро оновлення феромонів вздовж побудованих шляхів. Нам необхідна кількість блоків ceil(V/threadnum). Shared memory - ants*sizeof(float). До розподіленої пам’яті завантажуємо значення масиву sums. Виконуємо цикл по кількості мурах, кожен потік оновлює значення ребер що виходять з вершини що відповідає його загальному порядковому номеру, додаючи значення з shared memory, якщо вони більші за нуль.

Після завершення циклу алгоритму займаємося відновленням шляху. Перша фаза виконується на GPU в ядрі вибору оптимальних ребер. Сітка розмірністю ceil(V/threadnum). Виділяємо 2*V*sizeof(float) розподіленої пам’яті. В глобальній пам’яті створюємо масив на V float елементів road, для збереження оптимальних ребер з кожної вершини. Кожен потік представляє собою строку матриці феромонів. Ми проходимо у циклі по цій строці, обираючи максимальний елемент, та записуємо до масиву road. Друга фаза виконується на CPU, ми починаючи з start елементу переходимо по оптимальним ребрам із road розраховуючи суму довжин та записуючі самі ребра до масиву.

Псевдокод алгоритму:

```
set(ants,def,vap_coef,pow_pher,pow_dist,pow_road)//default values
read(Adj, start, end)
V:= Adj.size
Pher := [def for [1,V*V]]
Roads := [start for [1,ants*V]]
Road := [0 for [1:V]]
MemCpyToDevice(Adj as dAdj, Pher as dPher, Roads as dRoads, Road as
dRoad)
for i in [1, log(1.0 / def) / log(vap_coef)]do
     //inside first kernel, ant - id of current ant
     __shared__ used[], x[], rand, cur
     for id in [0,V - 1] in parallel do
          used[id] := false
     used[start] := true
     for turn in [1,V] do
          for id in[0,V-1] in parallel do
               x[id]:=  used[id]  *  dPher[cur,  id]^pow_pher/dAdj[cur,
          id]^pow_dist
          //prefix scan
          for d in [0,log2V - 1] do
```

```
                for k from 0 to V − 1 by 2d + 1 in parallel do
                x[k + 2d + 1 - 1] := x[k + 2d - 1] + x [k + 2d + 1 - 1]
          x[n] := x[n - 1]
          x[n - 1] := 0
          for d := log2V down to 0 do
                for k from 0 to V − 1 by 2d + 1 in parallel do
                     t := x[k + 2d - 1]
                     x[k + 2d - 1] := x [k + 2d + 1 - 1]
                     x[k + 2d + 1 - 1] := t + x [k + 2d + 1 - 1]
          //prefix scan ends
          rand := random(0, x[n])
          //upper bound
          for d in [1, V] in parallel do
                if x[d - 1] < rand and x[d] >= rand do
                     used[d] := True
                     dRoads[ant, cur] := d
                     cur := d
     // inside second kernel
     __shared__ sizes[]
     for id in [0, V*V-1] in parallel do
          dPher[id] := dPher[id] * vap_coef
     sizes[end] := -inf
     for id in [0, V-1] in parallel do:
          sizes[id] := dAdj[id, dRoad[ant, id]]
          if dRoad[ant, id] = end do
                sizes[end] := 0
     //sum
     for s:= 0 to log2V-1 do
          for id in [0,V - 1] in
                if id % 2^(s + 1) = 0 do
                     sizes[id] := sizes[id] + sizes[id + 2^s];
     dSum[ant] := 1/(sizes[0]^pow_road)
     // inside third kernel
     __shared__ sizes[]
     for i in [0, ants - 1] in parallel do
          sizes[i] := dSum[i]
     for i in [0, V - 1] in parallel do
          for j in [0, ants - 1] dp
                dPher[i,dRoad[j,i]] := dPher[i,dRoad[j,i]] + sizes[j]
//inside 4th kernel
```

```
__shared__ pos[], val[]
for i in [0,V-1]  in parallel do
     for j in [0, V-1] do
          if dPher[i,j] > val[i] do
               val[i] := dPher[i, j]
               pos[i] := j
for i in [0,V-1] in parallel do
     dRoad[i] := pos[i]
MemCpyToHost(dRoad as Road)
// on host
Path := []
range := 0
pos := start
for v in [0,V] do
     range += Adj[pos, road[pos]]
     Path.push(road[pos]]
```

6.2 Характеристика алгоритму

Отримана асимптотична складність: O(V*logV)

Для відображення коректних графіків швидкодії довелося згенерувати додаткові графи, оскільки графи для тестування були згенеровані по степеням двійки і не давали чіткого розуміння про характер. Код генерації додаткових графів у додатку А(10). Самі графіки відображались використовуючи бібліотеки pandas та matplotlib. Приклад коду для відображення графіку відображений у додатку А(11).

На рисунку 6.2.1 відображений графік швидкодії на останній версії.

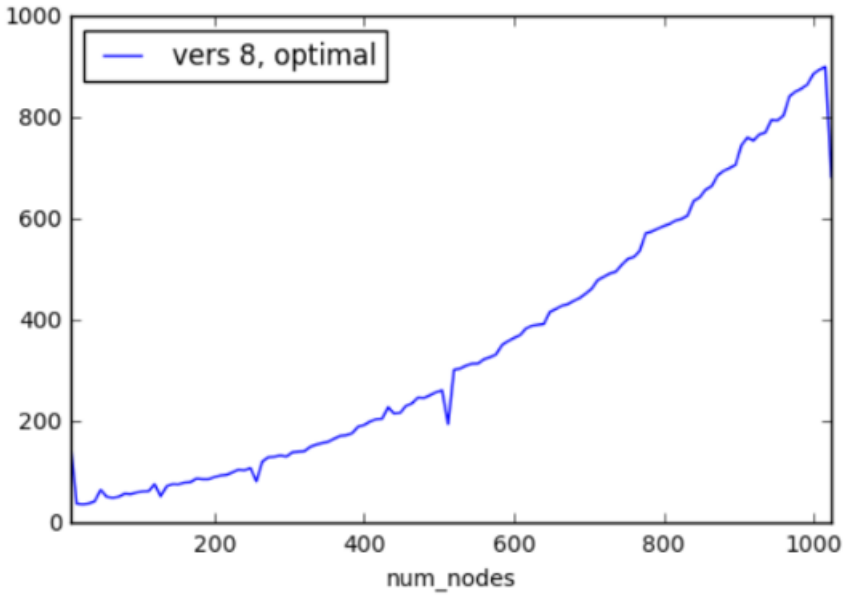

Рис. 6.2.1 - Швидкодія на останній версії реалізації

На рисунку 6.2.2 зображено графік порівняння фактичної швидкодії першої та останньої реалізації.

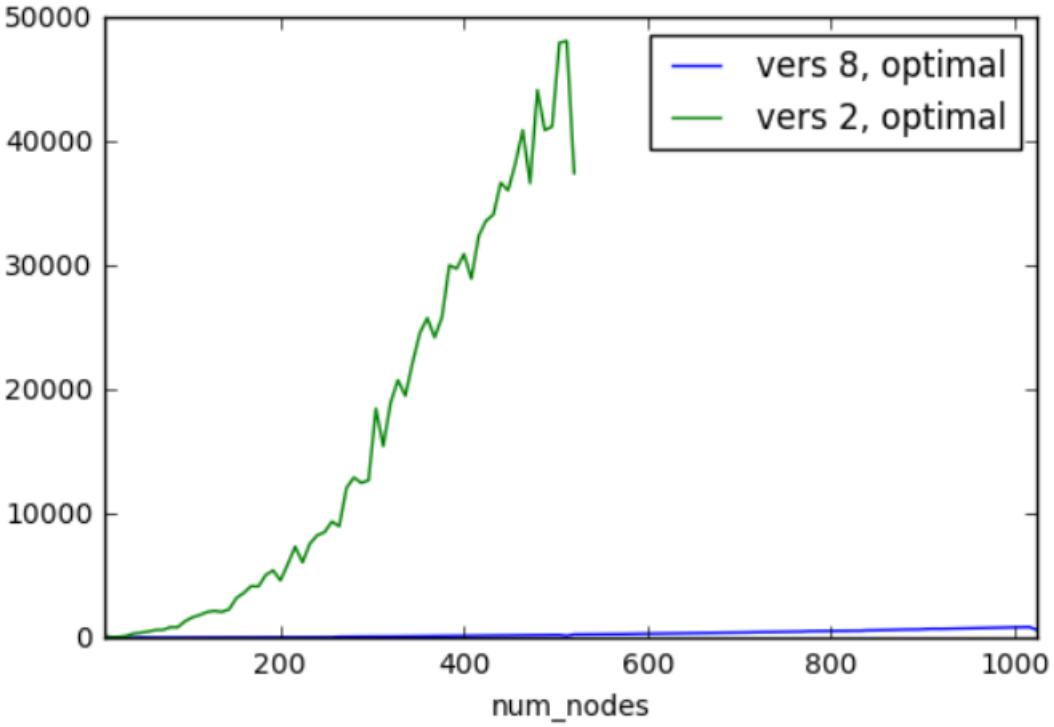

Рис. 6.2.2 – Порівняння швидкодії першої та останньої реалізації

Отримана складність по пам’яті: O(V^2)

Кількість резидентних потоків у GPU для оптимальної швидкодії: V*ants

Точність: при максимізації числа мурах, на версіях 1-3, досягала 98.4%, коли ми знизили кількість мурах для отримання оптимальної швидкодії у рамках наявного графічного процесора - отримали близько 85%. На рисунку 6.2.3 зображені результати запитів щодо точності реалізацій.

```
df[df.realization.isin([3,3])].mean()
```

id	27456.500000
accuracy	0.915141
numRuns	10.000000
time	3914.328345
iscrushed	0.000000
graph	71.877641
realization	3.000000
params	80.754401
acc_abs	0.984139

```
df[df.realization.isin([5,5])].mean()
```

id	23207.500000
accuracy	0.604755
numRuns	10.000000
time	347.627589
iscrushed	0.000000
graph	96.685734
realization	5.000000
params	96.565678
acc_abs	0.849558

Рис. 6.2.3 – Точність алгоритму на 2 та 4 версіях

Це свідчить про те, що при наявності достатнього технічного ресурсу, ми отримуємо високу точність. Також слід пам’ятати, що налаштування коефіцієнтів це окрема задача, котра залежить, в тому числі, від предметної області. Також оптимальні коефіцієнти можна підібрати з використанням нейронних мереж, таким чином ще повисивши точність (оскільки поточні версії коефіцієнтів підбиралися методом проб та помилок).

7 РОЗРОБКА ТА ОПТИМІЗАЦІЯ РЕАЛІЗАЦІЇ ХВИЛЬОВОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ CUDA

7.1 Глобальна структура реалізації Хвильового алгоритму на GPU

Хвильовий метод відноситься до класу методів «динамічного програмування». Алгоритм складається з двох етапів: «поширення хвилі» і «відновлення шляху». Для здійснення першого етапу вводиться матриця, в якій елементи зіставляються деяким цілим числах для кожної клітини просторової сітки, накладеної на розглянуту прямокутну область. У клітку призначення записується число «0». Решта осередків позначаються, що не пройдені хвилею. Осередок «0» випускає хвилю першого покоління і записує значення «1» в сусідні доступні осередки, які ще не були відвідані. Кожне нове покоління хвилі: позначає осередки як пройдені; породжує хвилю наступного покоління, додаючи в осередку нової хвилі своє значення, збільшене на «одиницю». Процес зупиняється при виконанні одного з наступних умов:

а) коли поточне положення (початок шляху) було позначено як пройдене хвилею;

б) коли всі доступні осередки були відвідані. Якщо в процесі роботи алгоритму початкова точка все-таки була досягнута хвилею, то в масив зберігається шлях від поточного положення робота до клітини призначення.

Відновлення шляху - це другий етап роботи хвильового алгоритму. Для його виконання потрібно помістити в шляховий список початкову точку: поточний стан. Далі зробити кілька ітерацій: вибрати комірку, сусідню з останньою в списку, тобто таку, яка була б позначена числом на одиницю менше, ніж остання в шляховому списку. Продовжувати процес до тих пір, поки в шляховому списку не виявиться осередок з позначкою «0», тобто осередок призначення.

Також нам необхідно мати певні початкові значення, ще до того як ми почнемо масивне розповсюдження хвилі. Їх встановленням будемо займатися до початку безпосереднього виконання алгоритму, на так званій фазі «початкової ініціалізації».

На рисунку 7.1.1 відображена глобальна структурна схема алгоритму що була описана вище.

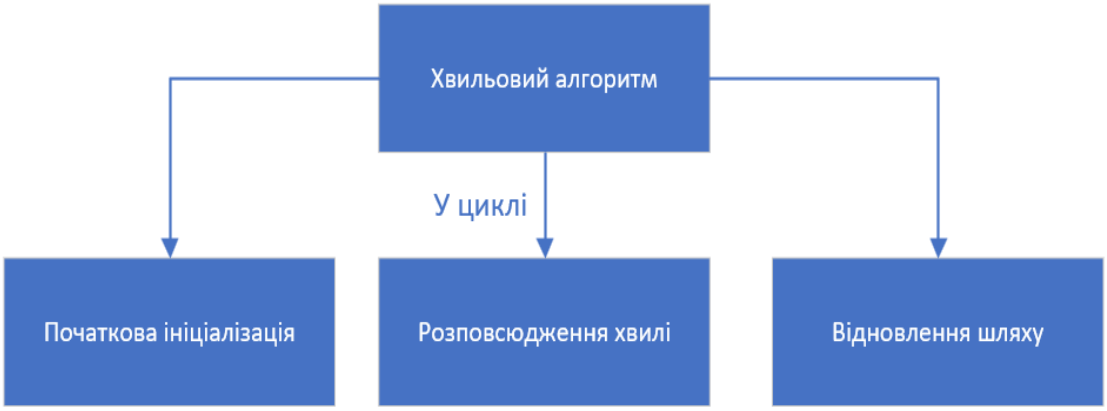

Рис. 7.1.1 - Структурна схема реалізацій алгоритму

7.2 Початкова ініціалізація

На етапі початкової ініціалізації ми повинні якось відмітити початкову, та результуючу вершини, та додати певні початкові значення для всіх інших полів, оскільки ми будемо працювати з кожним поле однаково. Початкова вершина має бути 0, щоб далі будувати розповсюдження хвилі, стосовно фінальної вершини – у нас є декілька варіантів, або зробити для неї певне унікальне значення і намагатися від нього зрозуміти коли вона буде досягнута, або проініціалізувати її так як і всі інші, і робити перевірку за межами блоку. Почнемо з другого варіанту. Всі інші вершини необхідно проініціалізувати значеннями, котрі точно більші за найбільше можливе значення шляху. Нехай це буде значення відповідне до кількості вершин на графі. Оскільки кількість вершин відповідає найбільшому можливому шляху. Ми можемо виконати це лінійно, тоді всі значення будуть встановлені за O(V). Але також ми можемо виконати це з використанням gpgpu, за O(1), але зі значно більшими коефіцієнтами. Котрий підхід є об’єктивно вигіднішим можна виявити лише емпіричним методом, та, це залежить від конкретних розмірностей графу та потужностей CPU/GPU. Оскільки ми займаємося більш дослідничими справами, давайте будемо вважати, що наша GPGPU достатньо потужна щоб використовувати

перший підхід. Наприклад, для графу на 25 вершин таблиця після початкової ініціалізації буде виглядати наступним чином (рисунок 7.2.1):

25	25	25	25	25
25	0	25	25	25
25	25	25	25	25
25	25	25	25	25
25	25	25	25	25

Рис. 7.2.1 – Приклад значень таблиці після початкової ініціалізації

7.3 Оптимізація розповсюдження хвилі

Одна з ключових ідей на котрих може базуватись ефективна версія хвильового алгоритму для GPGPU – це паралелізм операцій на етапі розповсюдження хвилі, оскільки при лінійній роботі – нам необхідно виконувати O(n) на кожному кроці (хоча і з меншим за одиницю коефіцієнтом). У контексті масивного паралелізму кожен thread має виконувати ідентичні операції. У такому випадку, спробуємо виділити кожному потоку конкретне місце у масиві за оновлення котрого він буде відповідати. А такому випадку на кожному кроці нам необхідно буде виконати O(1) операцій (фактично 5, чотири порівняння та 1 оновлення значення). Слід відмітити що при такому підході загальна кількість операцій буде значно більша за лінійну реалізацію, але вони будуть безумовними і зможуть виконуватися істинно паралельно.

Наступне питання котре слід визначити – якою буде умова завершення роботи? Інтуїтивно хочеться перевіряти чи проініціалізована фінальна вершина, але це зламає нам усю ефективність. Пропонується прийняти наступну умову: коректний шлях, у гіршому випадку, знаходиться за O(path_lenth). При цьому максимальна можлива довжина шляху – відповідає кількості вершин. Тож якщо ми виконаємо V повторень – то можемо бути певними що шлях буде знайдено. Але в одночас з цим, нам не хочеться виконувати забагато зайвих ітерацій. Це гарна модель для гіршого випадку, але у середньому значенні порівняння може бути навіть більш вигідним. Проблема ж порівнянь в свою чергу, що нам необхідно буде виходити за межі кернелів.

Виявляється що ми маємо певні проблеми з пам’яттю, коли намагаємося оновлювати усі поля одночасно: ми можемо одночасно оновляти і зчитувати одне й те саме значення. Але давайте зобразимо як у нас насправді відбувається процес розповсюдження, звернемо увагу на рисунок 7.3.1, де зображений приклад розповсюдження хвилі.

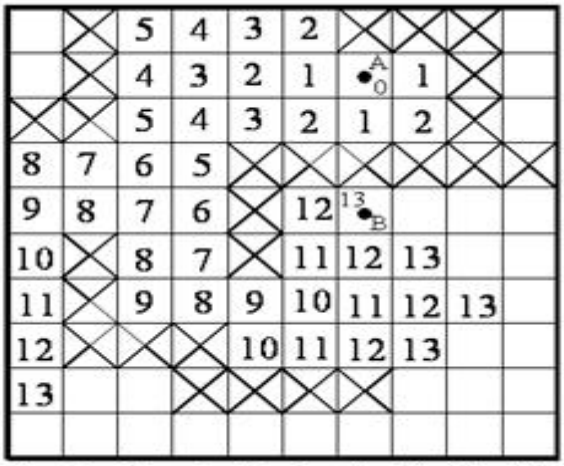

Рис. 7.3.1 – Приклад розповсюдження хвилі

Числа у клітинках відповідають кроку розповсюдження на котрому була відкрита певна вершина. Якщо придивитися – ми можемо помітити, що усі сусідні вертикально чи горизонтально клітинки містять різні числа, тобто були оновлені у різні епохи. Це наводить на думку про те як ми можемо ефективно вирішити проблему з пам’яттю: ми можемо «давати» на оновлення не усі клітинки одразу, а лише парні, або не парні з них. І при цьому ми вивільнимо половину потоків, тобто зможемо зменшити їхню кількість вдвічі, і до того ж не втратимо у швидкодії. У подальшому можна подумати над тим, як ще виключити зі списку на виконання певну частину завідомо непотрібних операцій (котрих у нас, насправді, доволі багато).

7.4 Оптимізація відновлення шляху

Наївна реалізація відновлення шляху працює за O(V), з коефіцієнтом 4. Ми тут просто перебираємо усі сусідні вершини і переходимо у найменшу, починаючи з останньої. Оскільки кожен наступний елемент знаходиться послідовно, використати

тут ефективно технології GPGPU не виглядає можливим. Тож, тут використаємо CPU.

8 ПРЕДСТАВЛЕННЯ РЕЗУЛЬТАТІВ РОБОТИ З ХВИЛЬОВИМ АЛГОРИТМОМ

8.1 Ітоговий вигляд алгоритму

Використовувані об’єкти:

- Adj – матриця вершин;
- len – ширина матриці вершин;
- start - початкова вершина;
- end - фінальна вершина;
- V - кількість вершин графу;
- block_size - максимальна кількість потоків на блок згідно GPU;

Вхідні дані:

Граф у вигляді матриці вершин Adj, кількість вершин V, стартова та фінішна вершини, start та end відповідно.

Хід алгоритму:

Матриця вершин Adj, заповнюється значеннями за замовчуванням, V. Adj[start] заповнюється нулем.

У глобальну пам’ять GPU переносимо Adj . Далі виконуємо ітерації алгоритму у кількості V. На кожній ітерації виконуємо одне з двох ядер: ядро парного або непарного розповсюдження хвилі.

Ядро розповсюдження хвилі – кожен поток отримує (не) парну вершину ‘a’ і виставляє у ній значення як min(Adj(a-1), Adj(a+1), Adj(a – len), Adj(a+ len)).

Після завершення циклу алгоритму займаємося відновленням шляху. Повертаємо дані із глобальної пам’яті GPU до CPU та рекурсивно перебираємо цсі сусідні вершини, вибираючи найменшу, починаючи з Adj(end).

Псевдокод алгоритму:

```
read(Adj, V, start, end)
len := Adj[_].len()
MemCpyToDevice(Adj as dAdj)
for i in [0, V] do
     if i.is_even()do
          //inside even kernel, a – current cell
          for a in [0,V/2] in parallel do
```

```
                a := a * 2
                dAdj[a]: = min(dAdj[a-1], dAdj[a+1], dAdj[a - len],
          dAdj[a + len]
     else do
          //inside odd kernel, a - current cell
          for a in [0,V/2] in parallel do
                a := a * 2 - 1
                dAdj[a]: = min(dAdj[a-1], dAdj[a+1], dAdj[a - len],
          dAdj[a + len]

MemCpyToHost(dAdj as Adj)
// on host
Path := []
range := Adj[end]
pos := end
while pos != start do
     Path.push(pos)
     Pos := x where x in [pos-1, pos+1, pos-len, pos+len] and Adj[x]
is minimal
Path.push(start)
Path.reverse()
```

8.2 Характеристика алгоритму

Отримана асимптотична складність: O(V)

Отримана складність по пам’яті: O(V)

Кількість резидентних потоків у GPU для оптимальної швидкодії: V

Точність: це є точний алгоритм

9 РОЗРОБКА ТА ОПТИМІЗАЦІЯ РЕАЛІЗАЦІЇ АЛГОРИТМУ БЕЛЛМАНА-ФОРДА З ВИКОРИСТАННЯМ CUDA

9.1 Глобальна структура реалізації оптимізаційного алгоритму Беллмана-Форда на GPU

По-перше, для достатньо вигідної роботи з графом, він повинен бути представлений у доволі специфічній формі. Розглянемо структуру зберігання зваженого графа, так як в подальшому вона буде згадуватися і перетворюватися. Граф задається в стислому CSR форматі. Даний формат широко поширений для зберігання розріджених матриць і графів. Для графа з N вершинами і M ребрами необхідно три масиву: xadj, adjncy, weights. Масив xadj розміру N + 1, інші два - 2 * M, так як в неорієнтованому графі для будь-якої пари вершин необхідно зберігати пряму і зворотну дуги.

Принцип зберігання графа наступний. Весь список сусідів вершини I знаходиться в масиві adjncy з індексу xadj [I] до xadj [I + 1], не включаючи його. За аналогічним індексам зберігаються ваги кожного ребра з вершини I. Для ілюстрації на рисунку 9.1.1 зліва показаний граф з 5 вершин, записаний за допомогою матриці суміжності, а праворуч - в CSR форматі.

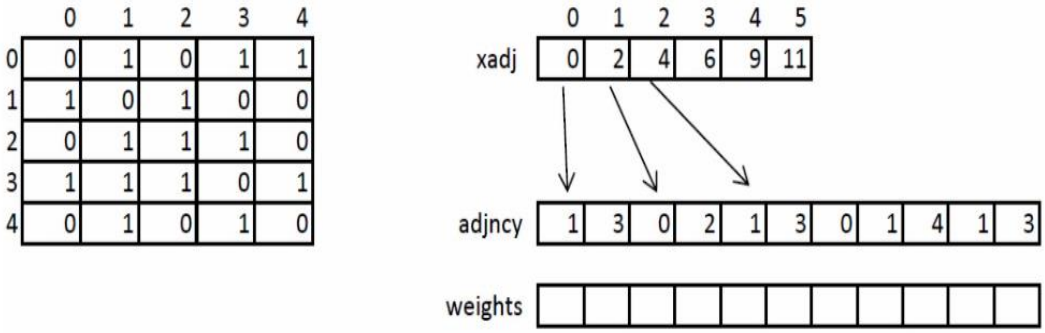

Рис. 9.1.1 – Приклад CSR формату

Окрім цього нам необхідно виконати ще певне розширення цієї структури, а саме – привести до координатного вигляду.

Далі у циклі ми будемо намагатися покращувати значення для кожної пари ребер.

На рисунку 9.1.2 відображена глобальна структурна схема алгоритму що була описана вище.

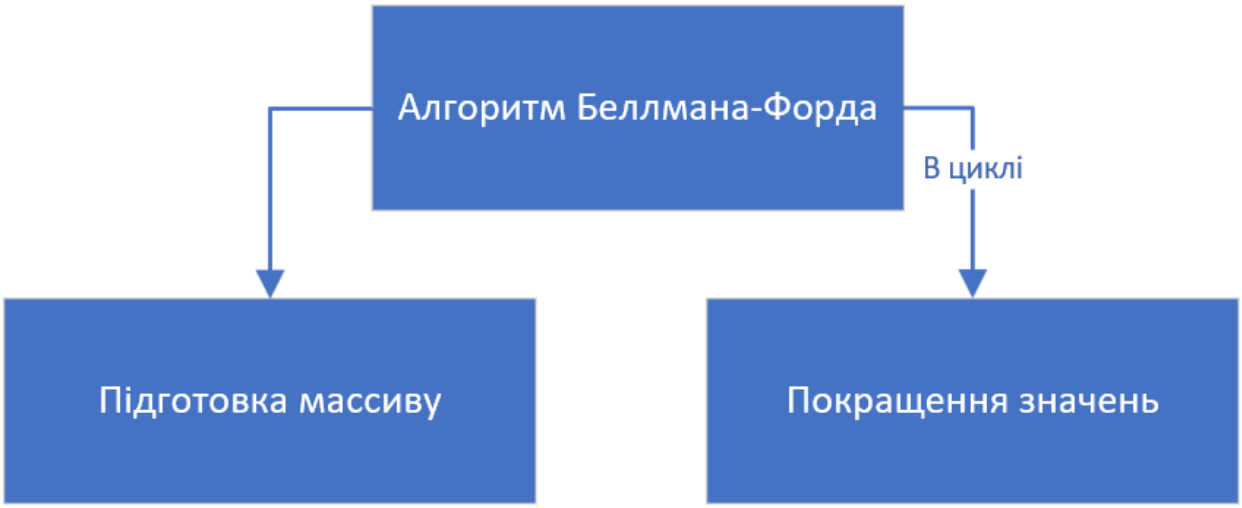

Рис. 9.1.2 - Структурна схема реалізацій алгоритму

9.2 Підготовка масиву

Структуру що була наведена у попередньому розділі треба ще трохи розширити. Ми замінюємо xadj на новий масив, котрий буде у собі зберігати початкові вершини дуг. Тепер для цієї пари массивів нам необхідно забрати повтори. Це все працює за O(E). І вже безпосередньо до розрахунків граф буде мати наступний вигляд (рисунок 9.2.1).

startV	0	0	1	1	2	2	3	3	3	4	4
endV(adjncy)	1	3	0	2	1	3	0	1	4	1	3
weights											

Рис. 9.2.1 – Граф в COO форматі

9.3 Покращення значень

Паралелізм тут заснований на тому, що кожне ребро може оновлюватися окремо від інших, а також на тому, що ми гарантуємо, за рахунок виключення на попередньому кроці повторів, що з кожним ребром одночасно буде працювати лише один поток.

Тож приготуємо ядро, де кожна нитка обробляє два ребра (пряме і зворотне), намагаючись поліпшити відстань по одному з них. І саме на цьому етапі у нас виникає гарантія, що вони обидва модифіковані не будуть. На відміну від алгоритму Форда-Беллмана, де кожне ребро проглядається послідовно, в реалізованому на GPU алгоритмі може виникнути ситуація «гонки» потоків - коли два або більше потоку заходять оновити одну і ту ж комірку dist [I]. Покажемо, що в даному випадку алгоритм залишиться коректним.

Припустимо, що є два потоки K1 і K2, які хочуть оновити осередок dist [I]. Можливі два випадки. Перший - одна з двох ниток записала найменше значення. Тоді на наступній ітерації для цих двох ниток умова не виконається, а значення в осередку dist [I] виявиться мінімальним. Другий - одна з двох ниток записати не мінімальне значення. Тоді на наступній ітерації для одного з потоків умова виконається, а для іншого - ні. Тим самим, результат буде однаковий в обох випадках, але досягнутий за різну кількість ітерацій.

Але при такому підході у нас виникає проблема з конфліктами запису/читання з пам’яті. Конфлікти виникають у ситуації коли два різних потоки незалежно вирішують оновити одне й те ж значення одночасно, або звернутися до значення що оновлюється. Отже, щоб вирішити проблему конфліктів, нам необхідно відмовитись від незалежного опрацювання кожного ребра, а працювати кожним потоком з однією конкретною вершиною. Це нас насправді дуже сповільнює на нерозріджених графах.

У нас також є можливість перевіряти чи не завершився пошук після кожного кроку, але це буде потребувати додаткового O(logn) на кожній ітерації. Це має сенс використовувати на щільних графах.

10 ПРЕДСТАВЛЕННЯ РЕЗУЛЬТАТІВ РОБОТИ З АЛГОРИТМОМ БЕЛЛМАНА-ФОРДА

10.1 Ітоговий вигляд алгоритму

Використовувані об’єкти:

- madj – матриця вагів;
- xAdj – мап ребер;
- Adjincy – вектор ребер;
- start - початкова вершина;
- weights – вектор ваг;
- V - кількість вершин графу;

Вхідні дані:

Граф у вигляді CSR : xadj, adj, кількість вершин V, стартова вершина start.

Хід алгоритму:

Перетворюємо граф у формат COO: створюємо вектор startV, де для кожного елемента adj додаємо відповідну згідно xadj стартову вершину. Видаляємо все повтори.

У глобальну пам’ять GPU завантажуємо madj, xadj, adj. Далі виконуємо ітерації алгоритму у кількості V. На кожній ітерації виконуємо ядро, у котрому кожен потік – це окрема вершина, і він у циклі «релаксує» всі вихідні ребра.

Після цього повертаємо до CPU пам’яті масив weights.

Псевдокод алгоритму:

```
read(madj, xadj, adj, start)
weights := []
MemCpyToDevice(madj as dmadj, xadj as dxadj, adj as dadj, weights as
dweights)
for i in [0, V] do
     //inside main kernel, j
     for j in [xadj[i], xadj[i + 1]]:
          if dweights[dadj[j]] > dweights[i] + dmadj[i,dadj[j]]:
               dweights[dadj[j]] = dweights[i] + dmadj[i,dadj[j]]

MemCpyToHost(dweights as weights)
// on host
```

10.2 Характеристика алгоритму

Отримана асимптотична складність: O(V^2)

Отримана складність по пам’яті: O(E)
Кількість резидентних потоків у GPU для оптимальної швидкодії: V
Точність: це є точний алгоритм

11 ПРЕДСТАВЛЕННЯ ЗАГАЛЬНИХ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

11.1 Порівняння алгоритмів

Було досліджено три алгоритми, алгоритм Мурашиної Колонії, Хвильовий алгоритм та алгоритм Беллмана-Форда. Порівняємо отриману ефективність (рисунок 11.1.1):

	Ефективність оригінального CPU алгоритму	Ефективність розробленого GPU алгоритму
Алгоритм Мурашиної Колонії	O(V^2*logV)	O(VlogV)
Хвильовий Алгоритм	O(V^2)	O(V)
Алгоритм Беллмана-Форда	O(V*E)	O(V^2)

Рис. 11.1.1 – Порівняння отриманої ефективності

Для алгоритму мурашиної колонії та для хвильового алгоритму ми отримали спідап на O(V), оскільки вони ідейно добре підходять для паралелізму, для Беллмана-Форда ж ми отримали O(E/V), ща на повних графах еквівалентне O(V), а на розріджених – O(1). Для алгоритмів же з BFS сімейства – результати можливого спідапу будуть не кращі за Беллмана-Форда.

11.2 Типові прийоми використані для оптимізації розрахунків

У процесі дослідження ми використовували різні прийоми для перенесення алгоритмів на GPU. Визначимо їх:

11.2.1 Представлення графу специфічною структурою

При роботі з графом на GPU, коли ми хочемо щоб кожен поток мав ідентичні завдання, нам не завжди підходить стандартне представлення графу у вигляді матриці суміжності, або списку ребер. Корисним може виявитися переведення графу у вигляд іншої структури, що, для конкретно взятого алгоритму, дозволить повноцінно розпаралелити завдання, наприклад COO або CSR.

11.2.2 Розбиття even/odd

Деякі підзадачі можуть потребувати взаємодії з сусідніми полями у пам’яті. Для того, щоб у такому випадку позбутися можливих конфліктів доступу, ми можем розбити матрицю на дві групи кліток, наприклад по принципу even/odd, та обробляти їх по черзі. Це дасть гарантію на відсутність колізій, і при цьому помітного сповільнення не буде.

11.2.3 Використання CPU

Як не дивно, але важливим при роботі з GPU є вміння оцінити наскільки для конкретної підзадачі більш або менш ефективним є використання CPU і у випадку коли CPU є більш ефективним – слід віддавати частину розрахунків туди, не зважаючи на те що основна направленість алгоритму – GPU.

11.2.4 Декомпозиція операцій

При роботі з алгоритмами, часто окремі масивні операції (знайти максимум, знайти суму і так далі) визначаються окремими мінімальними об’єктами декомпозиції. При роботі ж з GPU треба завжди намагатися декомпонувати їх до ще

нижчого рівня, знаходити найменші одиниці операцій. Таким чином, наприклад, ми можемо виконати паралельний предрозрахунок максимумів на рядках, і потім за значно меншу асимптотику знайти загальний максимум. Або можемо розбити операції перфікс-суми на декілька етапів, кожен з котрих може бути ефективно розпаралелений.

11.2.5 Мануальний контроль пам’яті

GPU має доволі своєрідну модель пам’яті, котра залишає багато місця як для оптимізацій, так і для деоптимізацій. При роботі з GPU важливо визначати як саме ми будемо взаємодіяти з тим чи іншим об’єктом, і вручну додавати до кешу ті об’єкти, з котрими відбувається багатократна взаємодія, і навпаки, дані до котрих буде лише одне звернення – у розподіленій пам’яті будуть лише займати місце, і мати навіть меншу ефективність.

11.2.6 Заміна крокових алгоритмів на паралельно ефективні

Як правило, алгоритми пошуку найкоротшого шляху мають певний перелік стандартних алгоритмів що використовуються на певних кроках основного алгоритму. Слід пам’ятати, що нам, при виконанні паралельної реалізації необхідно не переробити кожен крок на GPU, а знайти найбільш ефективний варіант цього кроку. Наприклад, стандартний Upper Bound працює за log(V), у той же час існує ефективна реалізація котра знаходить потрібне значення за O(1). Ми можемо написати масивний UpperBound, де операції будуть виконуватися одразу декількома потоками, але працювати все ще за O(logV), або знайти досить тривіальний масивний алгоритм за O(1).

ВИСНОВОК

У ході виконання атестаційної роботи були розроблені GPU імплементації трьох різних алгоритмів пошуку мінімального шляху на графі, а саме – алгоритмів Мурашиної Колонії, Беллмана-Форда, Хвильового. Для всіх них вдалося побудувати алгоритмічні моделі що дають виграш у ефективності у порівнянні зі стандартними реалізаціями. Отримана ефективність є кращою до O(V). Найбільш вдалим є імпелементація хвильового алгоритму, що зміг досягти ефективності у O(V), при тому, що стандартні точні алгоритми пошуку мінімального шляху, працюють не швидше за O(E).

Були виявлені та запропоновані способи, прийоми для переносу CPU алгоритмів на графах на GPU.

Виявлені ідеї та алгоритмічні моделі стали основою для доповіді на 25-му Міжнародному молодіжному форумі «РАДІОЕЛЕКТРОНІКА ТА МОЛОДЬ В XXI ст.», а також використані у статті [26].

У подальшому найбільш перспективними для дослідження є Хвильовий алгоритм та алгоритм Мурашиної колонії, котрі отримали значне пришвидшення. Для алгоритму Мурашиної колонії достатньо отримати більш місткі GPGPU, а для Хвильового алгоритму вирішити проблему швидкого перетворення довільного планарного графу до одиничного вигляду, що, наразі, робить сферу його застосування не такою широкою. Алгоритми ж засновані на BFS – для масивного паралелізму підходять слабко, та значного приросту ефективності не отримують.

ПЕРЕЛІК ПОСИЛАНЬ

1. Dorigo,M.,et al., "Ant Colony Optimization ", IEEE COMP. INTELLI. MAGAZINE, PP. 28-39, NOVEMBER 2006.

2. A. V. Aho, J. E. Hopcroft, J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley Longman, Boston, MA, USA, 1974.

3. R. E. Tarjan, A unified approach to path problems, Journal of the ACM 28 (3) (1981) 577–593

4. Shortest path problem [Електронний ресурс]. - Режим доступу: URL: https://en.wikipedia.org/wiki/Shortest_path_problem

5. General-purpose computing on graphics processing units [Електронний ресурс]. - Режим доступу: URL: https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units

6. Знайомство з технологією CUDA [Електронний ресурс]. - Режим доступу: URL: https://www.overclockers.ua/software/nvidia-cuda/

7. Ant colony optimization [Електронний ресурс]. - Режим доступу: URL: http://www.scholarpedia.org/article/Ant_colony_optimization

8. CUDA:Робота з пам’яттю (частина 1) [Електронний ресурс]. - Режим доступу: URL: https://habr.com/ru/post/55461/

9. W. Daniel Hillis, Guy L. Steele. Data parallel algorithms. Magazine Communications of the ACM - Special issue on parallelism, 1986, Том 29 Випуск 12. - с. 1170-1183

10. Akihiro Uchida, Yasuaki Ito, Koji Nakano. An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem. Third International Conference on Networking and Computing, 2012.

11. Guy E. Blelloch. “Prefix Sums and Their Applications”. Synthesis of Parallel Algorithms, Morgan Kaufmann,1990.

12. Cecilia,J.M., et al., "Enhancing Data Parallelism for Ant Colony Optimization on GPUs ". Journal of Parallel and Distrib. Computing, 2012.

13. Patterson,D., et al., "Programming Massively Parallel Processors ". A Hands-on Approach, Morgan Kaufmann Publishers is an imprint of Elsevier, USA, 2012.

14. M. S. Hung, and J. J. Dovoky, A computational study of efficient shortest path algorithms, Computers. Ops. Res. 15, 567-576 (1988).

15. E.W. Dijkstra, A note of two problems in connection with graphs. Num. Math. 1, 269-271 (1959)

16. Klimov M.A.., Vorotnikov S.A. Vybornov N.A. Method of calibration of local navigation system of mobile robots. Caspian journal: control and high technonogy. -2017 .- № 1 (37) .- С. 106 – 115]

17. Kozadayev A.S., Dubovitskiy Ye.V. Realization of wave algorithm for determination of shortest route on platitude at modeling of tracks with obstacles

18. Motorin D. E., Popov S. G. Multi-Criteria Path Planning Algorithm for a Robot on a Multilayer Map. Informatsionnoupravliaiushchie sistemy [Information and Control Systems], 2018, no. 3, pp. 45–53 (In Russian). doi:10.15217/issn1684-8853.2018.3.45] 7. Dmitrii Motorin, Serge Popov, Vladimir Muliukha, “Collision-Free Path Planning Algorithm for Group of Robots in Spatio-Situational Uncertainty”, PROCEEDING OF THE 21ST CONFERENCE OF FRUCT ASSOCIATION, Helsinki, Finland, pp. 456-461, ISSN 2305-7254

19. A. V. Goldberg, H. Kaplan and R. F. Werneck, "Reach for A*: Efficient point-to-point shortest path algorithms", Proceedings of the Eighth Workshop on Algorithm Engineeringand Experiments (ALENEX), pp. 129-143, Jan. 2006.

20. F. Baghayeri, H. Deldari and D. Bahrepour, "Divide and conquer skeleton on GPU", 2014 International Congress on Technology Communication and Knowledge (ICTCK) Mashhad, pp. 1-6, 2014.

21. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, The MIT Press, Sep. 2009.

22. E. W. Dijkstra, "A Note On Two Problems In Connexion With Graphs", NUMERISCHE MATHEMATIK, vol. 1, no. 1, pp. 269-271, 1959.

23. R. Bellman, "On A Routing Problem", Quarterly Of Applied Mathematics, vol. 16, no. 1, pp. 87-90, 1958.

24. H. Ortega-Arranz, Y. Torres, D. Llanos and A. Gonzalez-Escribano, A New Gpu-Based Approach To The Shortest Path Problem, pp. 505-511, 2013.

25. U. Meyer and P. Sanders, "A-Stepping: A Parallelizable Shortest Path Algorithm", Journal Of Algorithms, vol. 49, no. 1, pp. 114-152, 2003.

26. Дубинський, В.М, Лисенко, Б.С., Каяфюк, Р.В., Панкратов, О.В., Романова, Т.Є. Опукла багатокутна оболонка для пари складених об'єктів. Збірник "Кібернетика та комп'ютерні технології" № 2 за 2021 рік.