UNICHECK
by Turnitin

Ім'я користувача:
приховано налаштуваннями конфіденційності

ID перевірки:
1015642962

Дата перевірки:
19.06.2023 11:29:53 EEST

Тип перевірки:
Doc vs Library

Дата звіту:
19.06.2023 11:31:25 EEST

ID користувача:
100011372

Назва документа: **ОК-42 Кустов Ілля**

Кількість сторінок: **44** Кількість слів: **8049** Кількість символів: **63422** Розмір файлу: **1.81 MB** ID файлу: **1015289156**

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

4.78%
Схожість

Найбільша схожість: **0.7%** з джерелом з Бібліотеки (ID файлу: **1013801157**)

Пошук збігів з Інтернетом не проводився

4.78% Джерела з Бібліотеки | **297** ..... Сторінка 46

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | **1**

Підозріле форматування | **7 сторінок**

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

CPU	Central Processing Unit, центральний процесор
GPU	Graphics Processing Unit, графічний процесор
ПЛІС	Програмована логічна інтегральна схема
SIMD	Single Instruction Multiple Data
ALU	Arithmetic Logic Unit, арифметично-логічний блок
CUDA	Compute Unified Device Architecture
Host	хост, центральний процесор комп’ютера
Device	пристрій, графічний процесор
Kernel	ядро, функція графічного процесора
Grid	сітка, масив блоків
Block	блок, масив потоків
Warp	варп, угруповання потоків по 32 штуки усередині блоку
Thread	потік (нитка), найменша незалежна послідовність запрограмованих інструкцій

РЕФЕРАТ

Пояснювальна записка дипломного проєкту: 84 сторінки, 14 рисунків, 3 таблиці, 17 посилань, 1 додаток.

Об’єкт проєктування – програма швидкого обчислення оберненої матриці з використанням графічного процесора в середовищі CUDA.

Мета виконання дипломного проєкту полягає в дослідженні особливостей програмування з використанням графічних прискорювачів та в порівнянні швидкодії обчислень для центрального та графічного процесорів.

В дипломному проєкті було розроблено програму швидкого обчислення оберненої матриці з використанням графічного процесора.

ГРАФІЧНИЙ ПРОЦЕСОР, ГРАФІЧНИЙ ПРИСКОРЮВАЧ, ВІДЕОКАРТА, ПРОГРАМУВАННЯ, C++, CUDA.

ЗМІСТ

ВСТУП

В сучасних концепціях і підходах до розв’язання обчислювальних задач одну з найважливіших ролей відіграють паралельні обчислення. Паралельні обчислення дозволяють найбільш ефективно використовувати ресурси обчислювальної системи і підвищувати швидкість розв’язання задач. Такий підхід широко використовується при обробці великих об’ємів даних і для рішення комплексних задач, що потребують великих об’ємів ресурсів.

Серед різноманіття апаратних засобів сучасних обчислювальних систем важливими є пристрої для візуального подання інформації, зокрема графічні контролери. Ці пристрої забезпечують ефективну взаємодію з обчислювальною системою. Початково графічний контролер був розроблений як компонент комп'ютера, що відповідає за генерацію графічних елементів на дисплеї, працюючи незалежно від центрального процесора. За останні роки, у зв'язку зі збільшенням вимог до швидкості та якості візуалізації графічних зображень, графічні контролери отримали великий набір якісно нових функцій. Сприяючи збільшенню обчислювальних можливостей комп'ютера, з'явилися технології неграфічних обрчислень загального призначення на графічних процесорах, які дозволяють значно пришвидшити роботу з великими об’ємами даних. В наш час такі обчислення широко використовуються при проведенні фізичних симуляцій, під час аналізу та прогнозування фінансових моделей, при розробці штучного інтелекту, в медицині і в багатьох інших галузях.

Сучасний графічний процесор включає в себе значну кількість математичних виконавчих блоків, призначених для великої кількості обчислювально інтенсивних програм. Архітектура сучасних графічних прискорювачів є гнучкою, що разом з високорівневими мовами програмування та програмно-апаратними архітектурами надає широкі можливості та робить їх доступними. Компанія Nvidia, одна з перших, що вдосконалювала технології неграфічних обчислень на графічних контролерах, розробила платформу під назвою CUDA.

CUDA – це програмно-апаратна архітектура паралельних обчислень, яка дозволяє значно збільшити обчислювальну продуктивність завдяки використанню графічних процесорів. У порівнянні з попередніми моделями програмування, ця архітектура реалізована з урахуванням прямого доступу до апаратних можливостей графічних прискорювачів. CUDA надає середовище проєктування з власним компілятором та бібліотеками, що дозволяє програмістам і розробникам створювати програмне забезпечення для вирішення складних обчислювальних завдань за менший час завдяки багатоядерній обчислювальній потужності графічних процесорів. CUDA дозволяє включати в текст програм на мовах C та C++ виклики підпрограм, що виконуються на графічних процесорах Nvidia. Архітектура CUDA надає розробникам максимальний контроль над апаратними можливостями графічного контролера, дозволяючи організовувати доступ до набору інструкцій графічного прискорювача та керувати його пам'яттю.

В цьому дипломному проєкті було розглянуто особливості програмування з використанням графічних прискорювачів, а також особливості програмування в середовищі CUDA. В перших трьох розділах було висвітлено особливості архітектури графічних контролерів, розглянуто програмні засоби для роботи з пристроями CUDA, а також описано методи їх програмування. В процесі виконання дипломного проєкту, була створена програма, яка дозволяє порівняти швидкодію центрального та графічного процесорів при обчисленні прикладних математичних задач. Результати виконання програми дозволяють побачити фактичну ефективність, яку можна отримати використовуючи графічний контролер в якості обчислювального пристрою.

1 ПАРАДИГМА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

1.1 Основні поняття технології паралельних обчислень

Парадигма паралельних обчислень є підходом до розв'язання обчислювальних задач, який включає в себе методи і технології, що дозволяють виконувати обчислення паралельно з використанням декількох обчислювальних ресурсів. Паралельні обчислення виникли з необхідності розв'язувати задач, які потребують великої кількості обчислювальних ресурсів або мають значні обсяги даних. Цей підхід широко застосовується в різних областях, таких як обробка великих даних, наукові дослідження, комп'ютерне моделювання, штучний інтелект та інші.

Паралельні обчислення – одночасне використання кількох ресурсів обчислювальної системи для розв’язування задач. Основна ідея полягає в тому, щоб розбити задачу на менші підзадачі та виконувати їх одночасно для прискорення обчислювального процесу. Такий підхід є способом реалізації паралелізму в обчисленнях.

Паралелізм – це сукупність математичних, алгоритмічних, програмних і апаратних засобів, що забезпечують можливість паралельного виконання задачі. Обчислювальна задача має допускати розбиття на незалежні підзадачі, які можна виконувати одночасно. Паралельні обчислення вимагають врахування питань синхронізації, керування доступом до ресурсів, поділу даних та комунікації між обчислювальними пристроями. За допомогою використання кількох обчислювальних ресурсів, що працюють паралельно, можна досягти вирішення задачі за коротший проміжок часу, порівняно з використанням лише одного ресурсу.

Паралельні обчислення передбачають використання обчислювальних систем з кількома незалежними обчислювальними одиницями. Такі системи можуть включати мультипроцесорні пристрої та багатоядерні процесори. До таких пристроїв також можна віднести графічний контролер.

1.2 Особливості архітектури графічного контролера

Графічний контролер – це електронний пристрій, частина комп'ютера, призначена для генерації та обробки зображень з подальшим їхнім виведенням на екран периферійного пристрою. Сучасні графічні контролери містять вбудований графічний процесор, який здійснює додаткову обробку зображень, звільнюючи від цих задач центральний процесор.

Графічний процесор забезпечує набагато вищу пропускну здатність команд і більшу ширину смуги пам'яті, ніж центральний процесор, при аналогічній ціні та енергоспоживанні. Багато програм використовують ці вищі можливості, щоб працювати швидше на графічному процесорі, ніж на центральному процесорі. Інші обчислювальні пристрої, такі як ПЛІС, також дуже енергоефективні, але пропонують набагато меншу гнучкість програмування, ніж графічні процесори. Ця різниця в можливостях між GPU і CPU існує тому, що вони розроблені з різними цілями. У той час як CPU призначений для максимально швидкого виконання послідовності операцій, які називаються потоками, і може виконувати кілька десятків потоків паралельно, GPU призначений для паралельного виконання тисяч потоків (амортизуючи повільнішу однопотокову продуктивність для досягнення більшої пропускної здатності). Графічний процесор спеціалізується на високопаралельних обчисленнях і тому спроєктований таким чином, що більше транзакцій присвячено обробці даних, а не кешуванню даних та управлінню потоками.

Виділення більшої кількості транзисторів для обробки даних, наприклад, для обчислень з плаваючою комою, є вигідним для високопаралельних обчислень. Графічний процесор може приховати затримки доступу до пам'яті під час обчислень, замість того, щоб покладатися на великі кеші даних і складне управління потоками, щоб уникнути довгих затримок доступу до пам'яті, які є дорогими з точки зору транзисторів. Загалом, програма має поєднання паралельних і послідовних частин, тому системи розробляються з поєднанням графічних і центральних процесорів, щоб максимізувати загальну продуктивність.

Додатки з високим ступенем паралелізму можуть використовувати цю масивну паралельну природу GPU для досягнення вищої продуктивності, ніж на CPU.

Історія розвитку графічних прискорювачів налічує декілька десятиліть і починається з простих пристроїв, які використовувалися для виведення текстової інформації на екран монітора комп'ютера. З плином часу, разом зі збільшенням потужності комп'ютерів, з'являлася необхідність у покращенні можливостей графічних прискорювачів.

На початку 1990-х років з'явилася нова платформа для розвитку графічних контролерів - графічні операційні системи, зокрема Microsoft Windows. Це стало приводом для створення нового ринку графічних прискорювачів, які могли б підтримувати графічні операційні системи та полегшувати відображення зображень на екрані. Таким чином, на початку 1990-х років користувачі почали купувати для своїх персональних комп'ютерів 2D прискорювачі відображення. Ці прискорювачі мали апаратну підтримку операцій з растровими зображеннями, що полегшувало роботу з графікою та використання графічних операційних систем.

Приблизно в той самий час у світі професійних комп'ютерів компанія Silicon Graphics у 1980-х роках популяризувала використання тривимірної графіки на різних ринках, включаючи урядові та оборонні програми і науково-технічну візуалізацію, а також надавала інструменти для створення приголомшливих кінематографічних ефектів. У 1992 році Silicon Graphics відкрила програмний інтерфейс для свого обладнання, випустивши бібліотеку OpenGL. Silicon Graphics планувала використовувати OpenGL як стандартизований, незалежний від платформи метод для написання додатків 3D-графіки.

Випуск NVIDIA GeForce 256 ще більше розширив можливості споживчого графічного обладнання. Вона містила 32-64 МБ пам'яті і шину пам'яті 128-біт. Вперше обчислення трансформації та освітлення можна було виконувати безпосередньо на графічному процесорі, тим самим збільшуючи потенціал для створення ще більш цікавих візуально додатків. Оскільки трансформація і освітлення вже були невід'ємними частинами графічного конвеєра OpenGL,

GeForce 256 поклав початок природному розвитку, коли все більше і більше графічного конвеєра буде реалізовано безпосередньо на графічному процесорі.

З точки зору паралельних обчислень, випуск NVIDIA серії GeForce 3 у 2001 році є, мабуть, найважливішим проривом у технології графічних процесорів. GeForce 3 була оснащена графічним процесором NV20 з 57 млн. транзисторів. Вона мала 128 МБ пам'яті DDR, шину пам'яті 128-біт і 480 потокових процесорів. Серія GeForce 3 стала першим в обчислювальній індустрії чіпом, що реалізував новий на той час стандарт DirectX 8.0 від Microsoft. Цей стандарт вимагав, щоб сумісне обладнання містило як програмовані вершинні, так і програмовані піксельні етапи зафарбовування. Вперше розробники отримали певний контроль над тим, які саме обчислення будуть виконуватися на їхніх графічних процесорах.

Поява графічних процесорів з програмованими конвеєрами привернула увагу багатьох дослідників до можливості використання графічного обладнання для обчислень загального призначення. Однак на початку розвитку таких обчислень на GPU панував заплутаний загальний підхід, оскільки стандартні графічні API, такі як OpenGL і DirectX, ще не дозволяли прямо виконувати довільні обчислення на GPU поза межами графічного API. Це змушувало дослідників експериментувати з використанням графічного API для виконання обчислень загального призначення, забезпечуючи зовнішній вигляд їхніх задач у вигляді традиційного рендерингу на GPU.

По суті, графічний процесор обманом змушували виконувати завдання, що не стосуються рендерингу, змушуючи ці завдання виглядати так, ніби вони є стандартним рендерингом. Цей підхід був дуже винахідливим, але й дуже заплутаним.

Незважаючи на те, що цей підхід давав хороші результати, його модель програмування мала обмеження, які ускладнювали процес програмування та робили його недоступним для більш широкого кола розробників. Існували жорсткі обмеження на ресурси, оскільки програми могли отримувати вхідні дані лише від декількох вхідних кольорів і кількох текстурних одиниць. Крім того, були серйозні обмеження на те, де і як програміст міг записувати результати в пам'ять, тому

алгоритми, що вимагають можливості запису в довільні місця в пам'яті (розкид), не могли працювати на GPU. Більше того, було майже неможливо передбачити, як ваш конкретний графічний процесор буде працювати з даними з плаваючою комою, якщо він взагалі підтримує роботу з даними з плаваючою комою, тому більшість наукових обчислень не могли використовувати графічний процесор. Не було достатньо ефективного методу для налагодження будь-якого коду, що виконувався на GPU, що значно ускладнювало проєктування будь-яких

До того ж, кожен, хто хотів використовувати GPU для виконання обчислень загального призначення, повинен був вивчити OpenGL або DirectX, оскільки вони залишалися єдиними засобами, за допомогою яких можна було взаємодіяти з графічним процесором. Це означало не лише зберігання даних у графічних текстурах і виконання обчислень шляхом виклику функцій OpenGL або DirectX, але й написання самих обчислень на спеціальних графічних мовах програмування, відомих як шейдерні мови.

Усе це сильно обмежувало використання графічних прискорювачів у неграфічних обчисленнях. У зв'язку з цим, виробники графічних прискорювачів розуміли необхідність створення зручних рішень для використання графічних процесорів у неграфічних задачах. Одним з таких рішень стала платформа CUDA.

1.3 Особливості архітектури CUDA

CUDA - це платформа паралельних обчислень загального призначення та модель програмування, яка використовує механізм паралельних обчислень у графічних процесорах NVIDIA для вирішення багатьох складних обчислювальних задач більш ефективним способом, ніж на CPU.

У листопаді 2006 року NVIDIA представила перший в індустрії графічний процесор з підтримкою DirectX 10, GeForce 8800 GTX, яка зображена на рисунку 1.1.

Рисунок 1.1 Графічний прискорювач Nvidia GeForce 8800 GTX

GeForce 8800 GTX також став першим графічним процесором, побудованим на архітектурі NVIDIA CUDA. Архітектура NVIDIA CUDA представляла собою сукупність нових компонентів, які були розроблені для зняття обмежень, що заважали попереднім графічним процесорам виконувати обчислення загального призначення. Таким чином, NVIDIA CUDA дозволяла графічним процесорам виконувати складні обчислювальні завдання, які раніше могли бути виконані тільки на центральних процесорах (CPU). Починаючи з GeForce 8800 GTX, NVIDIA розпочала створення нового ринку для графічних процесорів, які могли бути використані для розв'язання складних обчислювальних задач загального призначення.

На відміну від попередніх поколінь, які розділяли обчислювальні ресурси на вершинні та піксельні шейдери, архітектура CUDA включає уніфікований конвеєр шейдерів, що дозволяє кожній арифметико-логічній одиниці (ALU) на чіпі керувати програмою, яка має намір виконувати обчислення загального призначення. Оскільки NVIDIA планувала використовувати нове сімейство графічних процесорів для обчислень загального призначення, ці ALU були створені відповідно до вимог IEEE для арифметики з плаваючою комою одинарної точності і розроблені для використання набору інструкцій, пристосованих для загальних обчислень, а не спеціально для графіки.

Крім того, виконавчим блокам на GPU було дозволено довільний доступ до пам'яті для читання та запису, а також доступ до програмно-керованого кешу, відомого як спільна пам'ять. Всі ці особливості архітектури CUDA були додані для того, щоб створити графічний процесор, який би не лише добре виконував традиційні графічні задачі, але й відмінно справлявся з обчисленнями.

Прагнення NVIDIA надати споживачам продукт як для обчислень, так і для графіки не могло зупинитися на виробництві апаратного забезпечення з CUDA.

Однак, проблемою залишалася архітектура. Незалежно від того, скільки функцій NVIDIA додала до своїх чіпів для полегшення обчислень, все одно не було способу отримати доступ до цих функцій без використання OpenGL або DirectX. Це не тільки вимагало від користувачів продовжувати маскувати свої обчислення під графічні проблеми, але й продовжувати писати свої обчислення на графічно-орієнтованій шейдерній мові, такій як GLSL від OpenGL або HLSL від Microsoft.

Щоб охопити максимально можливу кількість розробників, NVIDIA взяла промисловий стандарт C і додала відносно невелику кількість ключових слів, щоб використати деякі особливості архітектури CUDA. Через декілька місяців після виходу GeForce 8800 GTX компанія NVIDIA випустила компілятор для цієї мови, CUDA C. Таким чином, CUDA C стала першою мовою, спеціально розробленою компанією-виробником графічних процесорів для полегшення обчислень загального призначення на графічних процесорах.

Окрім створення мови для написання коду для GPU, NVIDIA також надає спеціалізований апаратний драйвер для використання величезної обчислювальної потужності архітектури CUDA. Користувачам більше не потрібно володіти знаннями графічних програмних інтерфейсів OpenGL або DirectX, а також не потрібно змушувати свою задачу виглядати як задачу комп'ютерної графіки

Зараз CUDA постачається з програмним середовищем, яке дозволяє розробникам використовувати C++ як мову програмування високого рівня. Також підтримуються інші мови, інтерфейси прикладного програмування або підходи на основі директив, такі як FORTRAN, DirectCompute, OpenACC.

1.4 Програмні аспекти використання графічного контролера

Щоб краще зрозуміти принципи роботи з графічним процесором необхідно розглянути його будову. Найкраще буде порівняти будову графічного процесора і центрального процесора.

Центральний процесор - створений, спрямованим на мінімізацію затримок в обчисленнях шляхом швидкого отримання результатів. Для цього він має значну кеш-пам'ять, що зменшує середню затримку даних, та лише кілька високопродуктивних арифметично-логічних блоків для швидкого обчислення результатів. Сучасні моделі CPU також широко використовують паралелізм на рівні інструкцій для попереднього обчислення часткових результатів з метою додаткового зменшення затримок.

Натомість, графічний процесор спрямований на підвищення пропускної здатності. Завдяки великій кількості паралельних процесорів, GPU не може надати кеш-пам'ять для кожного з них з розміром, схожим на розмір кеш пам’яті в CPU. Це зумовлює частіші звернення до повільніших типів пам'яті та затримки. Однак, якщо графічний процесор має більше потоків, ніж фізичних ядер, він може "перевантажуватися" потоками та приховувати затримки, швидко перемикаючи виконання між ними.

Загалом, потоки на графічному процесорі є менш вимогливими за потоки на центральному процесорі, що робить перемикання між ними більш ефективним. Навіть якщо затримки можуть бути більшими, можливість швидкого перемикання потоків та конвеєризації додаткових інструкцій забезпечує високу пропускну здатність GPU під час виконання завдання. Таким чином, ефективність від використання графічних процесорів для обробки може зрости, чим більше потоків використовується для даної обчислювальної задачі.

Графічні прискорювачі мають безліч обчислювальних ядер, зазвичай кілька тисяч, проте вони об'єднані в блоки та мають спільні елементи, включаючи регістри. Архітектура ядра GPU та логічних елементів значно простіша, ніж на CPU.

Важливо зазначити, що зазвичай графічний прискорювач та процесор не ділять пам'ять між собою, тому запис даних на графічний прискорювач та повернення результату є окремими операціями. На рисунку 1.2 показано приклад розподілу ресурсів чіпа для CPU та GPU.

Рисунок 1.2 Розподіл ресурсів для CPU та GPU

Оглядаючи обмеження та можливості роботи з GPU, можна зазначити наступне:

– Якщо ми виконуємо обчислення на GPU, не можна виділити лише одне ядро, буде виділений цілий блок ядер (32 для NVIDIA).

– Усі ядра виконують одні й ті самі інструкції, але з різними даними, такі обчислення називаються Single-Instruction-Multiple-Data або SIMD.

– Через відносно простий набір логічних блоків та спільних регістрів, GPU не дуже прихильний до гілок, а також до складної логіки в алгоритмах.

З іншого боку, робота з GPU відкриває наступні можливості:

– Ефективне прискорення SIMD-обчислень.

CUDA комбінує високорівневу функціональність бібліотек з високою продуктивністю низькорівневих інструкцій. Це дозволяє зручно створювати програми з її використанням. Крім того, CUDA постійно покращує підтримку стандарту C++, і поставляється з готовим набором бібліотек для різних сценаріїв

використання. CUDA також часто розкриває можливості GPU першою, навіть до того, як вони з'являються в інших API. Це робить CUDA універсальним підходом до програмування на GPU зручним для управління кодом між різними архітектурами.

Хост (host) представляє собою центральний процесор (CPU) комп'ютера, тоді як пристрій (device) - це графічний процесор (GPU), що знаходиться на відеокарті. Кожен з процесорів має власну окрему пам'ять, і фізично розділені пам'ять хоста і пам'ять пристрою. Хост контролює обчислювальний процес і відповідає за виклик функцій, які забезпечують обмін даними між різними видами пам'яті і запускають завдання на пристрої. Кожному такому завданню відповідає ядро (kernel), яке є функцією, розробленою за спеціальними правилами.

У CUDA є два типи функцій: функції ядра та функції пристрою. Функції ядра можуть бути викликані безпосередньо з хосту та мають конфігурацію запуску, яка включає вбудовані типи, структури та вказівники як параметри. Крім того, глобальний покажчик вказує на область пам'яті, доступ до якої мають функції. Функції ядра не можуть повертати значення, вони повинні бути типу void.

Функції пристрою, з іншого боку, можуть бути викликані лише з ядер або інших функцій пристрою. Вони не мають конфігурації запуску та параметрів з ядер або функцій пристрою. Замість цього, пристрій кваліфікує функції та вказує на область пам'яті, доступ до якої має функція.

Зазвичай у CUDA-програмах під час роботи з пристроєм містяться такі етапи:

– виділяється пам'ять під дані на пристрої;
– копіюються дані з пам'яті хоста в пам'ять пристрою;
– на пристрої виконуються ядра;
– результати копіюються з пам'яті пристрою в пам'ять хоста.

При виконанні обчислень на GPU одночасно запускається велика кількість паралельних процесів, так звані потоки. У моделі програмування CUDA потік - це найнижчий рівень абстракції для виконання обчислень або операцій з пам'яттю.

Усі потоки, запущені на виконання, об'єднуються в складну структуру, відому як сітка (grid). Сітка є масивом блоків (block), який може бути одновимірним, двовимірним або тривимірним. Кожен блок також є масивом потоків (thread), який може бути одновимірним, двовимірним або тривимірним. При цьому всі блоки, які утворюють сітку, мають однакову розмірність і розмір, який задається заздалегідь. Схематичне зображення ієрархії потоків міститься на рисунку 1.3.

Рисунок 1.3 Ієрархія потоків в CUDA

Кожен блок у сітці має свою адресу, що складається з одного або двох невід'ємних цілих чисел (індекс блоку в сітці). Аналогічно кожен потік всередині блоку також має також свою адресу - одне, два або три невід'ємних цілих числа, що задають індекс потоку всередині блоку. Оскільки одне й те саме ядро виконується одночасно дуже великою кількістю потоків, то для того, щоб ядро могло однозначно визначити номер потоку (а отже, і елемент даних, який потрібно обробляти), використовуються вбудовані змінні threadIdx (індекс поточного потоку в обчисленні на GPU, має тип uint3) і blockIdx (індекс поточного блоку в обчисленні на GPU, має тип uint3). Кожна з цих змінних є тривимірним цілочисельним вектором.

Також ядро може отримати розміри сітки та блоку через вбудовані змінні gridDim (розмірність сітки, має тип dim3) і blockDim (розмірність блоку, також має тип dim3). Подібний поділ усіх потоків є ще одним загальним прийомом використання CUDA: вихідне завдання розбивається на набір окремих підзадач, які розв'язуються незалежно одна від одної. Кожній такій підзадачі відповідає блок потоків.

При цьому кожне підзавдання спільно вирішується всіма потоками свого блоку. Варп (warp) - це набір з 32 потоків у блоці, організований таким чином, що всі потоки в варпі виконують одну і ту ж інструкцію. Об'єднання потоків у варпи відбувається окремо для кожного блоку; таким чином, усі потоки одного варпа завжди належать одному блоку. При цьому потоки можуть взаємодіяти між собою тільки в межах блоку. Потоки різних блоків взаємодіяти між собою не можуть.

Однак є способи непрямої взаємодії потоків у блоці. Вони можуть взаємодіяти один з одним через спільну (shared) пам'ять. Кожен блок має доступ до окремого обсягу швидкої діленої пам'яті, яку можуть спільно використовувати всі потоки блоку. З огляду на те, що потоки блоку можуть не фізично виконуватися паралельно (тобто, ми маємо справу з не чистою SIMD-архітектурою, де потоки прозоро управляються), механізм синхронізації потоків блоку необхідний для запобігання проблем з одночасним доступом до спільної пам'яті.

Потоки CUDA мають можливість доступу до різних областей пам'яті під час виконання обчислень. Кожен потік має свою власну приватну локальну пам'ять, блок потоків має спільну пам'ять, що доступна для всіх потоків блоку і існує протягом життєвого циклу блоку. Блоки потоків в кластері блоків потоків можуть здійснювати читання, записи та атомізацію в спільну пам'ять один одного. Усі потоки мають доступ до однієї глобальної пам'яті.

Крім того, є ще дві області пам'яті, які доступні для читання всіма потоками: константна та текстурна пам'ять. Глобальна, константна та текстурна пам'яті оптимізовані для різних використань пам'яті. Текстурна пам'ять також пропонує різні режими адресації та фільтрацію даних для деяких специфічних форматів

даних. Області глобальної, константної та текстурної пам'яті є постійними для всіх запусків ядра з тією ж самою програмою. На рисунку 1.4 зображено ієрархію пам’яті CUDA.

Рисунок 1.4 Ієрархія пам’яті CUDA

CUDA надає простий спосіб синхронізації, відомий як бар'єрна синхронізація. Для виконання цього використовується вбудована функція syncthreads(), яка блокує потоки блоку, які її викликають, доти, поки всі потоки блоку не увійдуть у цю функцію. Таким чином утворюється "бар'єри" всередині ядра, перед яким усі потоки блоку мають зупинитися, чекаючи завершення роботи інших потоків блоку.

Використання архітектури CUDA дозволяє відокремити фрагменти коду звичайної програми для виконання на графічному процесорі. CUDA дає можливість спростити написання програм для графічних прискорювачів і покращує їх продуктивність.

2 ВИБІР ЗАСОБІВ РОЗРОБЛЕННЯ

2.1 Середовище Visual Studio

Для написання програми використовується середовище розроблення Microsoft Visual Studio, що дозволяє виконувати весь цикл проєктування програмного забезпечення в одному місці. Це комплексне інтегроване середовище розроблення (IDE), що надає можливість писати, редагувати, налагоджувати та збирати код, а також розгортати програми. Visual Studio містить компілятори, інструменти для завершення коду, контроль коду, розширення та інші можливості, що покращують кожен етап процесу розроблення програмного забезпечення.

Окрім редагування та налагодження коду, Visual Studio надає підтримку різних мов програмування. Visual Studio інтегрує контроль версій для роботи з кодом у команді. Більше того, це середовище розроблення надає можливості для впорядкування та редагування вмісту, включаючи редактор коду з підказками IntelliSense та мітками, які показують дії, а також провідник рішень та подання класів. Крім того, Visual Studio дозволяє створювати додатки для будь-якої платформи та компілювати та збирати програми з використанням декількох опцій налаштування. Розробники можуть також налаштовувати власну конфігурацію збірки для своїх проєктів.

Важливим для цього дипломного проєкту є те, що Visual Studio підтримує мову програмування C++ та середовище розроблення CUDA. Це дозволяє зручніше організувати написання програми, а також легше відслідковувати і виправляти помилки.

В дипломному проєкті використовувалась Visual Studio Comunity 2022. Для того щоб встановити її і налаштувати необхідні компоненти, потрібно завантажити Visual Studio Installer на офіційному веб-сайті Microsoft. Visual Studio Installer дозволяє встановити необхідну версію Visual Studio, а також необхідні компоненти і модулі. Варто зауважити, що деякі модулі займають багато дискового простору.

Для роботи з C++ необхідно встановити модуль який називається «Desktop development with C++». Це пакет різних інструментів для формування, налагодження та компіляції комп'ютерних програм на C++. Його необхідно вибрати в списку компонентів під час встановлення Visual Studio, або ж його можна додати пізніше.

2.2 Мова програмування для виконання проєкту

Мова програмування C++ є однією з найпоширеніших мов у світі програмування. Створена у 1983 році, C++ поєднує в собі можливості мови C з об'єктно-орієнтованим підходом, що дозволяє програмістам розробляти ефективні та масштабовані програмні продукти.

Однією з головних переваг C++ є її можливість бути використаною на будь-якій операційній системі та в різних сферах програмування, включаючи системне програмування, розробку ігор, наукові дослідження та багато іншого. Мова C++ має високу швидкодію і низький рівень абстракції, що дозволяє точно контролювати процеси в програмі та ефективно використовувати ресурси. Також вона підтримує декілька парадигм програмування, включаючи об'єктно-орієнтовану, процедурну, функціональну і залежну від шаблонів, що надає розробникам гнучкість і можливість використовувати найкращі практики з різних областей програмування.

Використання сторонніх бібліотек є однією з ключових можливостей, яку надає мова C++, що дозволяє розширювати можливості програми та забезпечувати зручне взаємодію з іншими додатками.

Одним з яскравих прикладів використання сторонніх бібліотек є платформа CUDA, розроблена компанією Nvidia з підтримкою мови C++. Ця платформа дозволяє розробникам зручно об'єднувати обчислення на графічному та центральному процесорах в рамках однієї програми.

2.3 Середовище CUDA Toolkit

CUDA Toolkit від NVIDIA є надійним середовищем розроблення для створення високопродуктивних програм з використанням GPU-прискорення. За допомогою цього інструментарію можна розробляти, оптимізовувати та розгортати програми на вбудованих системах з GPU-прискоренням, настільних робочих станціях, корпоративних центрах обробки даних, хмарних платформах та суперкомп'ютерах. CUDA Toolkit включає бібліотеки з GPU-прискоренням, інструменти для налагодження та оптимізації, компілятор C/C++ та бібліотеку часу виконання для розгортання додатку.

Разом з CUDA Toolkit надаються декілька додаткових інструментів, наприклад Visual Profiler. Це крос-платформний інструмент профілювання продуктивності, який надає розробникам важливу інформацію для оптимізації CUDA C/C++ додатків. Він дозволяє швидко і зручно виявляти проблеми з продуктивністю за допомогою таблиць і графіків, а також автоматизовано аналізувати програми для виявлення вузьких місць і отримання пропозицій щодо оптимізації.

CUDA Toolkit відмінно інтегрується з Visual Studio, надаючи користувачам можливість створювати проєкти на C++ з використанням CUDA, компілювати код за допомогою компілятора CUDA, відлагоджувати проєкт за допомогою CUDA Debugger, а також використовувати додатковий функціонал прямо в середовищі редактора Visual Studio.

В дипломному проєкті використовувалась CUDA Toolkit версії 12.1. Щоб встановити її, необхідно завантажити і запустити інсталяційний файл з офіційного веб-сайту Nvidia. Під час встановлення можна обрати деякі додаткові модулі. Варто зауважити, що перед встановленням CUDA Toolkit, необхідно мати вже встановлені драйвера Nvidia для графічного прискорювача, а також вже встановлену Visual Studio. Тоді, CUDA Toolkit автоматично інтегрується у Visual Studio і одразу же після встановлення буде можливо створювати проєкти з використанням CUDA.

3 ПРОЄКТУВАННЯ ПРОГРАМИ

3.1 Алгоритм знаходження оберненої матриці

Для порівняння ефективності обчислень на графічному та на центральному процесорах, а також для відзначення особливостей програмування CUDA в цьому дипломному проєкті використовується програма знаходження оберненої матриці. Операції з матрицями чудово піддаються паралелізму, що дозволяє якнайкраще використати графічний процесор при обчисленнях.

Обернена матриця – це така матриця, результат множення якої на початкову дасть одиничну матрицю. Обернена матриця позначається як A^{-1}, а отже справедливим є таке рівняння: $A \times A^{-1} = E$, де E – одинична матриця. Обернена матриця існує лише до квадратної невиродженої матриці. Невиродженою називається матриця визначник, або детермінант, якої не дорівнює 0, $\det(A) \neq 0$.

Існує декілька точних методів знаходження оберненої матриці. До них відносяться: метод Гауса-Жордана, за допомогою матриці алгебраїчних доповнень, а також з використанням LU або LUP розкладу. В цьому дипломному проєкті було використано метод Гауса-Жордана, оскільки він є найпростішим і чудово піддається паралелізму.

Метод Гауса-Жордана - метод, що використовується для розв'язування квадратних систем лінійних алгебраїчних рівнянь, знаходження оберненої матриці, знаходження координат вектора в заданому базисі або відшукання рангу матриці.

Якщо справа до квадратної матриці дописати одиничну матрицю того ж порядку і за допомогою елементарних перетворень над рядками перетворити отриману матрицю так, щоб початкова матриця стала одиничною, то матриця, отримана за допомоги одиничної, буде оберненою матрицею до початкової.

Для прикладу, знайдемо обернену матрицю до даної $A = \begin{bmatrix} 4 & 3 \\ 6 & 2 \end{bmatrix}$.

Спочатку дописуємо справа одиничну матрицю:

$$\begin{bmatrix} 4 & 3 & ¿ & 1 & 0 \\ 6 & 2 & ¿ & 0 & 1 \end{bmatrix}$$

Тепер, необхідно за допомогою елементарних перетворень над рядками матриці перетворити ліву частину матриці в одиничну.

1-й рядок ділимо на 4:

$$\begin{bmatrix} 1 & 0{,}75 & ¿ & 0{,}25 & 0 \\ 6 & 2 & ¿ & 0 & 1 \end{bmatrix}$$

Від 2-ого рядка віднімаємо 1-ий рядок, помножений на 6:

$$\begin{bmatrix} 1 & 0{,}75 & ¿ & 0{,}25 & 0 \\ 0 & -2{,}5 & ¿ & -1{,}5 & 1 \end{bmatrix}$$

2-ий рядок ділимо на -2,5:

$$\begin{bmatrix} 1 & 0{,}75 & ¿ & 0{,}25 & 0 \\ 0 & 1 & ¿ & 0{,}6 & -0{,}4 \end{bmatrix}$$

І від 1-ого рядка віднімаємо 2-ий рядок, помножений на 0,75:

$$\begin{bmatrix} 1 & 0 & ¿ & -0{,}2 & 0{,}3 \\ 0 & 1 & ¿ & 0{,}6 & -0{,}4 \end{bmatrix}$$

Зліва ми отримали одиничну матрицю, а справа шукану обернену до матриці А. Отже, оберненою до матриці $A=\begin{bmatrix} 4 & 3 \\ 6 & 2 \end{bmatrix}$, є матриця $A^{-1}=\begin{bmatrix} -0{,}2 & 0{,}3 \\ 0{,}6 & -0{,}4 \end{bmatrix}$.

В загальному, алгоритм Гауса-Жордана можна поділити на два етапи: прямий та зворотній хід.

При прямому ході спочатку відбувається нормалізація рядка, щоб головний елемент став дорівнювати одиниці. Після чого цей рядок віднімається від решти рядків так, щоб всі елементи, окрім головного елемента, в цих рядках стали рівні нулю. Далі необхідно повторити ці кроки для наступного головного елемента. На рисунку 3.1 зображено блок-схему прямого ходу алгоритму Гауса-Жордана.

Рисунок 3.1 Блок-схема алгоритму прямого ходу Гауса-Жордана

Зворотній хід, виконується від останнього до першого рядка. В ньому виконується нормалізація останнього рядка до одиниці. Після цього останній

рядок віднімається від решти рядків так, щоб у всіх рядках, крім останнього, елемент під останнім головним елементом став рівним нулю. Далі необхідно повторити ці кроки для попереднього головного елемента. На рисунку 3.2 зображено блок-схему зворотного ходу алгоритму Гауса-Жордана.

Рисунок 3.2 Блок-схема алгоритму зворотного ходу Гауса-Жордана

Після закінчення зворотного ходу права частина розширеної матриці містить шукану обернену, тому залишається переписати її в окрему матрицю. На рисунку 3.3 зображено блок-схему алгоритму запису оберненої матриці з розширеної.

Рисунок 3.3 Блок-схема алгоритму запису оберненої матриці

При написанні програми, яка буде обчислюватиі обернену матрицю згідно з цим алгоритмом, доведеться використати додаткові матриці, а саме одиничну, розширену і тимчасову матрицю. В залежності від рівня оптимізації програми, деякі додаткові матриці можна не використовувати, але для наочності роботи алгоритму вони були додані в блок-схемах. На рисунку 3.4 зображено блок-схему алгоритму заповнення тимчасової, одиничної і розширеної матриць.

Рисунок 3.4 Блок-схема алгоритму заповнення тимчасових матриць

Отож, повний алгоритм знаходження оберненої матриці, складається з чотирьох кроків:

– Створення додаткових тимчасових матриць, а саме тимчасової одиничної та розширеної;

– Здійснюється прямий хід Гауса-Жордана;

– Здійснюється зворотний хід Гауса-Жордана;

– Здійснюється запис оберненої матриці з розширеної матриці.

На рисунку 3.5 зображено повний алгоритм знаходження оберненої матриці методом Гауса-Жордана.

Рисунок 3.5 Блок-схема повного алгоритму знаходження оберненої матриці

3.2 Розроблення програми для CPU

Для реалізації алгоритму Гауса-Жордана на мові програмування C++, в цьому дипломному проєкті були використані одновимірні масиви чисел з плаваючою комою, тобто одновимірні масиви типу float. Використання двовимірних масивів типу float або інших типів, наприклад vector, хоча і є зручнішим в процесі програмування, проте сильно програє в швидкості обчислення. До того ж, CUDA не підтримує сторонні типи даних. Тому, щоб забезпечити швидкодію і спростити реалізацію програми під GPU, використовувався саме одновимірний масив float.

Варто звернути увагу на особливості мови C++ при роботі з масивами. В C++ перед використанням масиву він обов'язково має бути ініційований, тобто обов'язково повинен бути виділений фіксований розмір масиву або виділено фіксований розмір ділянки пам'яті для елементів даного масиву. Розмір масиву не може змінюватися в процесі його використання. Також, C++ автоматично не очищає пам'ять від невикористовуваних змінних, а отже, після використання масиву необхідно очистити ділянку пам'яті, яку він використовував.

Приклад того, як створювати та ініціювати масив типу float:

```
float* array = new float[10]; //створюється масив розміром 10
```

В цьому прикладі символ зірочки «*» означає, що array є вказівником на масив float. Подібним чином можна легко задати необхідний розмір масиву. Для прикладу, створимо матрицю розміром 4 на 4:

```
int size = 4;
float* array = new float[size*size];
```

В цьому прикладі, для виділення пам'яті під матрицю 4 на 4, необхідно виділити пам'ять для одновимірного масиву розміром 16. Але замість того, щоб вручну прописувати розмір масиву, була використана додаткова змінна. Такий підхід дозволяє не помилитися і зекономити час при ініціації масиву.

Тепер розглянемо, як отримати доступ до елементів масиву. В C++, щоб отримати доступ до елементу масиву, достатньо написати ім'я масиву, а справа від нього вказати індекс елементу, до якого ми звертаємось.

Для того, щоб реалізувати звертання до елементу масиву, як до елементу матриці, необхідно вказати індекси рядка і стовпця елемента матриці до якого ми звертаємося. Для прикладу розглянемо матрицю 3 на 3. Для матриці 3 на 3, необхідно використати масив розміром 9 елементів, а отже індекс останнього елемента рівний 8. При цьому, якщо рахувати індекс рядка та стовпця останнього елемента то вони рівні 2 – 2.

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} = \begin{bmatrix} 0-0 & 0-1 & -20 \\ 1-0 & 1-1 & 1-2 \\ 2-0 & 2-1 & 2-2 \end{bmatrix}$$

Для того, щоб зручніше звертатися до необхідних елементів матриці, наприклад при обробці за допомогою циклів, потрібно використати наступну конструкцію:

```
array[i*size+j];
```

При такому зверненні i – це індекс рядка матриці, а j – це індекс стовпця. Такий підхід дуже зручно використовувати при переборі елементів через подвійний цикл. Для прикладу, наведімо фрагмент програми, що дозволяє вивести всі елементи матриці в консоль:

```
for(int i=0; i<size; i++){
for(int i=0; i<size; i++){
        cout << array[i*size+j] <<" "; //виводимо елемент масиву в консоль
}
cout << endl; //переходимо на наступний рядок консолі
}
```

В цьому прикладі використовується два цикли for, перший для перебору рядків матриці, другий для перебору стовпців. Завдяки такому підходу, можна максимально зручно та ефективно реалізовувати різноманітні операції з матрицями, а також заощаджувати ресурси комп'ютера.

Тепер розглянемо повну реалізацію основної програми пошуку оберненої матриці. Варто зауважити, що для загострення уваги на самому алгоритмі Гауса-Жордана, було пропущено вимірювання часу роботи програми. Також використовується спрощена версія алгоритму, яка краще піддається паралелізму і не використовує надлишкові тимчасові змінні, ніж та яка була подана в підрозділі 3.1. В якості початкової матриці буде використано масив float* copy – який є копією оригінальної матриці, а результат буде записаний в матрицю invertedmatrix.

Створимо тимчасову матриці:

```
float* temp = new float[size * size];
```

Тепер заповнимо тимчасову і обернену матрицю значеннями. В оберненій матриці елементи на головній діагоналі дорівнюють 1, а всі решта дорівнюють 0, а елементи тимчасової дорівнюють елементам початкової матриці:

```
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        temp[i * size + j] = copy[i * size + j];
        invertedmatrix[i * size + j] = 0;
    }
    invertedmatrix[i * size + i] = 1;
}
```

Тепер реалізуємо обчислення оберненої матриці за алгоритмом Гауса-Жордана:

```
for (int k = 0; k < size; k++) {
    // Нормалізуємо поточний рядок
    float pivot = temp[k * size + k];
    for (int i = 0; i < size; i++){
        temp[k * size + i] /= pivot;
        invertedmatrix[k * size + i] /= pivot;
    }
    // Віднімаємо поточний рядок від решти рядків
    for (int i = 0; i < size; i++) {
```

```
        if (i != k) {
            float factor = temp[i * size + k];
            for (int j = 0; j < size; j++) {
                temp[i * size + j] -= factor * temp[k * size + j];
                invertedmatrix[i * size + j] -= factor * invertedmatrix[k * size + j];}}
    }
  }
```

Після завершення обчислення звільняємо пам'ять від непотрібних змінних:

```
delete[] temp;
```

3.3 Розроблення програми для GPU

В підрозділі 1.3 вже було коротко розглянуто особливості архітектури CUDA, тому в цьому підрозділі увага буде більше привернута до написання коду. Варто звернути увагу на те, що CUDA дозволяє максимально спростити та автоматизувати цей процес, а отже програмісту залишається лише організувати свій код таким чином, щоб він як найефективніше піддавався паралелізму.

Щоб почати обчислення на GPU, необхідно визначити яка частина коду буде виконуватися на ньому. Для цього в CUDA існують спеціальні функції __global__ і __device__. Функція __global__ може бути викликана з хоста, а функція __device__ може бути викликана лише з пристрою, тобто її виклик потрібно робити всередині функції __global__. Синтаксис функції ядра виглядає так:

```
__global__ void kernel(){ //тіло функції }
```

При виклику функції ядра необхідно вказати розмір сітки та розмір блоків, що будуть використовуватися. Ці розміри можна задати звичайним числом, але в CUDA існує структура dim3, яка дозволяє задати розмір одразу трьох вимірів. Для роботи з матрицями буде достатньо двох вимірів. Отже, виклик функції ядра, та задання розмірів сітки і блоків буде виглядати так:

```
dim3 block_size(16, 16);
dim3 grid_size(32, 32);
```

kernel <<<grid_size, block_size >>> (); //виклик функції

Цей код здійснить виклик функції ядра, яка буде використовувати сітку розміром 32 на 32 блоки, кожен з яких розміром 16 на 16 потоків. Для того щоб ефективно розділити виконання програми на таку кількість потоків в CUDA існують наступні змінні:

– gridDim.x,y,z – кількість блоків у сітці у визначеному напрямку;
– blockDim.x,y,z – кількість потоків у блоці у визначеному напрямку;
– blockIdx.x,y,z – індекс поточного блоку в сітці у визначеному напрямку;
– threadIdx.x,y,z – індекс поточного потоку в блоці у визначеному напрямку.

Використовуючи ці змінні, можна визначити за які обчислення буде відповідати кожен конкретний потік. Для прикладу, якщо визначено розмір сітки 32 на 32 та розмір блоку 16 на 16, це означає, що в загальному є 512 на 512 потоків. Тоді, використавши конструкцію int x = blockDim.x * blockIdx.x + threadIdx.x; можна отримати почерговий індекс кожного з 512 в напрямку x. Для прикладу, наведімо фрагмент програми, що дозволяє додати 1 до всіх елементів матриці розміром size на size:

```
__global__ void kernel(float* matrix, int size) {
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    if (x < size && y < size)
    {
        matrix[x*size+y] += 1;
    }
}
```

В цьому прикладі напрямок x, використовується як індекс рядка, а напрямок y, як напрямок стовпця. Щоб випадково не звернутися до неіснуючих елементів матриці, використовується перевірка if (x < size && y < size).

Якщо необхідно, щоб усі потоки всередині блоку почали виконання певної частини коду одночасно, потрібно використати функцію __syncthreads().

Для того, щоб працювати з пам'яттю пристрою в CUDA є функції cudaMalloc(), cudaMemset() і cudaMemcpy(). Наведемо приклад використання цих функцій:

```
float * d_matrix; //вказівник на матрицю для GPU
//виділимо пам'ять пристрою для матриці розміром size на size
cudaMalloc((void**)&d_matrix, size * size * sizeof(float));
//заповнимо матрицю нулями
cudaMemset(d_matrix, 0, size * (2 * size) * sizeof(float));
//скопіюємо матрицю з пам'яті хоста в пам'ять пристрою, і назад
cudaMemcpy(d_matrix, matrix, size * size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(matrix, d_matrix, size * size * sizeof(float), cudaMemcpyDeviceToHost);
```

Варто наголосити, що при програмуванні функції ядра, код повинен бути найбільш простим, варто уникати багатьох умов, та циклів, а також використовувати більш прості операції над змінними.

Тепер розглянемо повну реалізацію основної програми знаходження оберненої матриці, для CUDA. Варто зауважити, що для загострення уваги на самому алгоритмі Гауса-Жордана, було пропущено вимірювання часу роботи програми. В якості початкової матриці буде використано масив float* copy – який є копією оригінальної матриці, а результат буде записаний в invertedmatrix.

Встановлюємо розмір блоків, та сітки, які буде використовувати CUDA:

```
dim3 grid_size(1024, 1024);
dim3 block_size(32, 32);
```

Створюємо матриці, які будуть використовуватися лише на пристрої і виділяємо під них пам'ять:

```
float* d_inverted, * d_matrix;
//виділяємо пам'ять
```

cudaMalloc((void**)&d_inverted, size * size * sizeof(float));

cudaMalloc((void**)&d_matrix, size * size * sizeof(float));

Записуємо дані з масиву, що зберігається в пам'яті хоста, в пам'ять пристрою:

cudaMemcpy(d_matrix, copy, size * size * sizeof(float), cudaMemcpyHostToDevice);

Заповнюємо обернену матрицю нулями:

cudaMemset(d_inverted, 0, size * size * sizeof(float));

Викликаємо функцію ядра, яка заповнить головну діагональ оберненої матриці одиницями:

identity_kernel << <grid_size.x, block_size.x >> > (d_inverted, size);

Викликаємо функцію ядра, яка знайде обернену матрицю методом Гауса-Жордана:

```
for (int k = 0; k < size; k++) {
    inverse_kernel << <grid_size.x, block_size.x >> > (d_matrix, d_inverted, size, k);
}
```

Записуємо інвертовану матрицю, що зберігається в пам'яті пристрою, в пам'ять хоста:

cudaMemcpy(invertedmatrix, d_inverted, size * size * sizeof(float), cudaMemcpyDeviceToHost);

Після завершення обчислення звільняємо пам'ять пристрою від непотрібних змінних:

```
cudaFree(d_inverted);
cudaFree(d_matrix);
```

Тепер детальніше розглянемо самі функції ядер. Так само, як в програмі для CPU, заповнимо обернену матрицю:

```
__global__ void identity_kernel(float* inverse, int size) {
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    // Заповнюємо головну діагональ одиницями
    if (x < size) {
```

```
        inverse[x * size + x] = 1; }}
```

Тепер реалізуємо пошук оберненої матриці алгоритмом Гауса-Жордана:

```
__global__ void inverse_kernel(float* matrix, float* inverse, int size, int k) {
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    // Нормалізуємо поточний рядок
    float pivot = matrix[k * size + k];
    if (x < size) {
        matrix[k * size + x] /= pivot;
        inverse[k * size + x] /= pivot;
    }
    __syncthreads();
    // Віднімаємо поточний рядок від решти рядків
    if (x<size && x != k) {
        float factor = matrix[x * size + k];
        for (int j = 0; j < size; j++) {
            matrix[x * size + j] -= factor * matrix[k * size + j];
            inverse[x * size + j] -= factor * inverse[k * size + j];
        }}
    __syncthreads();
}
```

3.4 Розроблення додаткових підпрограм

Варто пам’ятати, що обернена матриця існує лише до квадратної невиродженої матриці. Отже, перед тим як здійснювати обчислення оберненої матриці, необхідно бути впевненим, що дана матриця невироджена. Для цього достатньо визначити чи детермінант матриці рівний 0, якщо так – матриця вироджена. Розглянемо код програми, яка здійснює перевірку на невиродженість:

```
bool CheckSingular(float* matrix, int size)
    {
    float* mat = new float[size * size]; //створюємо тимчасову матрицю
```

```
Copy1DFloat(mat, matrix, size);
float det = 1.0f; //змінна детермінант
//перевірка на невиродженість
for (int k = 0; k < size; k++) {
   float pivot = mat[k * size + k];
   //якщо матриця невироджена
   if (pivot == 0) {
      delete[] mat; //видаляємо тимчасову матрицю
      return true; //повертаємо значення true – тобто матриця є виродженою
   }
   det *= pivot;
   // елімінація Гауса
   for (int j = k + 1; j < size; j++) {
      float factor = mat[j * size + k] / pivot;
      for (int i = k + 1; i < size; i++) {
         mat[j * size + i] -= factor * mat[k * size + i];
      }
   }
}
delete[] mat; // видаляємо тимчасову матрицю
return false; // повертаємо значення false – тобто матриця є невиродженою
}
```

Для того, щоб порівняти ефективність роботи виконання програми на CPU і GPU, необхідно заміряти час виконання кожної з програм. Це можна зробити декількома способами, проте в C++ є вбудована бібліотека <chrono>, яка надає необхідні для цього функції. Щоб її підключити необхідно написати наступний код:

```
#include <chrono>
using namespace std::chrono; //це дозволить кожен раз не звертатися до класу chrono
```

Для того, щоб відміряти час виконання певного фрагменту коду, достатньо зберегти системний час перед та після його виконання, а різниця цих двох значень і буде часом виконання даного фрагменту. Код для виконання цієї задачі виглядає наступним чином:

```
// зберігаємо початковий час
auto start = high_resolution_clock::now();
// фрагмент коду, час виконання якого ми замірюємо
…
// зберігаємо кінцевий час
auto stop = high_resolution_clock::now();
// віднімаємо від кінцевого початковий час в мікросекундах
auto duration = duration_cast<microseconds>(stop - start);
//зберігаємо отримане значення в змінну цілого числа типу long int
long int time = duration.count();
```

Таким чином, ми отримуємо змінну time, в якій збережено час виконання вказаного фрагменту коду.

Використовуючи вищеописані алгоритми, а також можливості консольного вводу C++, можна створити програму зі зручним текстовим інтерфейсом. На рисунку 3.6 зображено зовнішній вигляд інтерфейсу готової програми.

Рисунок 3.6 Зовнішній вигляд текстового інтерфейсу програми

3.5 Результати виконання програми

Для того щоб побачити справжню ефективність роботи графічного процесора необхідно провести швидкісні порівняльні тести з великою вибіркою.

Для проведення таких тестів було написано додаткові фрагменти коду, які дозволяють створити випадкові невироджені матриці в заданому проміжку розмірів, знайти їх обернену на CPU і GPU, а також зберегти час обчислення обох пристроїв до текстового файлу. Отримані дані можна легко проаналізувати.

Було проведено швидкісний тест для вибірки матриць розмірами від 4*4 до 1024*1024 з кроком 1. Під час тесту швидкість роботи CPU постійно падає, а швидкість роботи GPU тримається на сталому рівні, змінюючись в рамках похибки і завантаженості GPU іншими задачами. Максимальний час обчислення на CPU складає 6 958 060 мікросекунд (тобто 6,9 секунд) для матриці 1024*1024, а час обчислення на GPU для цієї матриці складає 1 854 мікросекунд, що означає приріст швидкості в 3585,89 разів. В середньому, приріст швидкості складає 1029,27 разів. Графіки до цього тесту зображено на рисунку 3.7.

Рисунок 3.7 Графік порівняння швидкості CPU і GPU для розмірів від 4*4 до 1024*1024

За отриманими результатами, GPU стабільно отримує перевагу в швидкості починаючи з розміру матриці 43*43, коли швидкість обчислення CPU починає стрімко падати. Це можна побачити на рисунку 3.8, де зображено графік для меншого діапазону.

Рисунок 3.8 Графік порівняння швидкості CPU і GPU для розмірів від 4*4 до 128*128

Також було проведено тест для вибірки матриць розмірами від 64*64 до 1024*1024 з кроком 64, аби закріпити результати минулого тесту. Максимальний час обчислення на CPU складає 6 792 226 мікросекунд (тобто 6,7 секунди) для матриці 1024*1024, а час обчислення на GPU для цієї матриці складає 2 128 мікросекунд, що означає приріст швидкості в 3191,84 разів. Графік до цього тесту зображено на рисунку 3.9.

Рисунок 3.9 Графік порівняння швидкості CPU і GPU для розмірів від 64*64 до 1024*1024 з кроком 64

Згідно з результатами попередніх тестів, стало очевидно, що чим більше даних, тим більш ефективним є використання GPU. Тому було проведено тест для вибірки матриць розмірами від 250*250 до 5250*5250 з кроком 5250. Нажаль, після матриці розміром 4500*4500 результати обчислень перестали співпадати

(алгоритм не достатньо оптимізований для графічного процесора), а отже подальші результати не мають змісту. Якщо ж не брати до уваги подальші дані, то максимальний час обчислення на CPU складає 526 254 666 мікросекунд (тобто 8,7 хвилин) для матриці 4500*4500, а час обчислення на GPU для цієї матриці складає 8 292 мікросекунд, що означає приріст швидкості в 63 465,35 разів. Графік до цього тесту зображено на рисунку 3.10.

Рисунок 3.10 Графік порівняння швидкості CPU і GPU для розмірів від 250*250 до 5250*5250 з кроком 250

ВИСНОВКИ

У дипломному проєкті розглянуто особливості паралельних обчислень з використанням графічних контролерів.

Проведено дослідження особливостей використання графічних процесорів для виконання прикладних обчислень.

Розглянуто використання середовища CUDA для створення та відлагодження програм мовою C++ для апаратних засобів графічних контролерів.

Для тестових порівняльних експериментів з використанням центрального та графічного процесорів розроблено програму обчислення оберненої матриці методом Гауса-Жордана.

Виконано порівняльні тести для вибірки квадратних матриць різних розмірів з порівнянням швидкості знаходження оберненої матриці за допомогою центрального та графічного процесорів. Тести проведено для вибірок розмірів матриць: від 4 на 4 до 1024 на 1024 з кроком в 64 розміри, і від 64 на 64 до 1024 на

1024 з кроком в 64 розміри, а також від 250 на 250 до 5250 на 5250 з кроком в 250 розмірів.

Проведені дослідження показали, що для матриць з невеликими розмірами, до 40 на 40, перевагу у швидкості обчислень має центральний процесор. Для більших за розмірами матриць виграш у швидкодії отримує графічний процесор.

В результаті тестових дослідів виявилося, що ефективність паралельних обчислень зростає зі збільшенням кількості даних для обробки. Найбільше порівняльне значення швидкодії обчислень центральним і графічним процесорами, отримана в тестах, складає 6346535% на користь графічного процесора.

У розділі "Техніко-економічне обґрунтування" розглянуто витрати на розробку та впровадження дипломного проєкту.

У розділі "Охорона праці та безпека життєдіяльності" розглянуто основні положення про електробезпеку, пожежну безпеку, подано заходи і вимоги безпеки та ергономіки при роботі за комп'ютером.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Антонюк В. А. Програмирование на видеокартах (GPGPU) : навч. посіб. Москва : Фіз. ф-т МГУ ім. М.В. Ломонос., 2015. 48 с.

2. Історія розвитку відеокарт. Інтернет-магазин F.ua. URL: https://blog.f.ua/ua/articles/istoriya-razvitiya-videokart.html (дата звернення: 17.05.2023).

3. Метод Гауса. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Метод_Гауса (дата звернення: 17.05.2023).

4. Метод Гаусса - Жордана. Википедия - свободная энциклопедия. URL: https://ru.wikipedia.org/wiki/Метод_Гаусса_-_Жордана (дата звернення: 17.05.2023).

5. Невироджена матриця. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Невироджена_матриця (дата звернення: 17.05.2023).

6. Обернена матриця. Вивчення математики онлайн. URL: https://ua.onlinemschool.com/math/library/matrix/inverse/ (дата звернення: 17.05.2023).

7. Обернена матриця. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Обернена_матриця (дата звернення: 17.05.2023).

8. Определитель. Википедия – свободная энциклопедия. URL: https://ru.wikipedia.org/wiki/Определитель (дата звернення: 17.05.2023).

9. Технология Програмирования CUDA : навч. посіб. / Д. Н. Тумаков та ін. Казань : Казанс. держ. ун-т, 2017. 112 с.

10. Тишков Д. Вычисления на GPU – зачем, когда и как. Плюс немного тестов. Хабр. URL: https://habr.com/ru/companies/dbtc/articles/498374/ (дата звернення: 17.05.2023).

11. Обратная матрица. Википедия – свободная энциклопедия. URL: https://ru.wikipedia.org/wiki/Обратная_матрица (дата звернення: 17.05.2023).

12. CUDA and Applications to Task-based Programming : навч. посіб. / B. Kerbl та ін. CUDA Tutorial, 2022. 77 с. URL: https://cuda-tutorial.github.io.

13. CUDA C++ Programming Guide : документація. Nvidia, 2023. 496 с.

14. CUDA C++ Programming Guide. NVIDIA Documentation Hub - NVIDIA Docs. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (дата звернення: 17.05.2023).

15. CUDA - Memory Hierarchy. The Beard Sage. URL: http://thebeardsage.com/cuda-memory-hierarchy/ (дата звернення: 17.05.2023).

16. CUDA Toolkit Documentation 12.1. NVIDIA Documentation Hub - NVIDIA Docs. URL: https://docs.nvidia.com/cuda/ (дата звернення: 17.05.2023).

17. What is Visual Studio?. Microsoft Learn. URL: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022 (дата звернення: 17.05.2023).

Схожість

Джерела з Бібліотеки 297

1	Студентська робота	ID файлу: 1013801157	Навчальний заклад: National University Ostroh Academy	46 Джерело	0.7%
2	Студентська робота	ID файлу: 1014867678	Навчальний заклад: National Aviation University	4 Джерело	0.53%
3	Студентська робота	ID файлу: 1004077827	Навчальний заклад: Donetsk National Technical Universit	2 Джерело	0.51%
4	Студентська робота	ID файлу: 1003867921	Навчальний заклад: National Technical University of Ukra	2 Джерело	0.48%
5	Студентська робота	ID файлу: 1029890	Навчальний заклад: Lviv Polytechnic National University		0.43%
6	Студентська робота	ID файлу: 1011524563	Навчальний заклад: National Technical University of Ukra	3 Джерело	0.39%
7	Студентська робота	ID файлу: 1014362249	Навчальний заклад: National Technical University of Ukra	2 Джерело	0.37%
8	Студентська робота	ID файлу: 1000792122	Навчальний заклад: National Technical University of Ukraine "Kyi...		0.27%
9	Студентська робота	ID файлу: 1015102599	Навчальний заклад: Taras Shevchenko National University of Kyiv		0.26%
10	Студентська робота	ID файлу: 1012989973	Навчальний заклад: Vasyl Stus Donetsk National Univers	27 Джерело	0.25%
11	Студентська робота	ID файлу: 1014506306	Навчальний заклад: Lviv Polytechnic National University	78 Джерело	0.24%
12	Студентська робота	ID файлу: 1013820504	Навчальний заклад: National University Ostroh Academy	10 Джерело	0.24%
13	Студентська робота	ID файлу: 1005414699	Навчальний заклад: Ternopil Volodymyr Hnatiuk Nationa	8 Джерело	0.22%
14	Студентська робота	ID файлу: 1000329846	Навчальний заклад: Lviv Polytechnic National University		0.21%
15	Студентська робота	ID файлу: 46403	Навчальний заклад: Lviv Polytechnic National University	2 Джерело	0.2%
16	Студентська робота	ID файлу: 1005715215	Навчальний заклад: Zaporizhzhya National University		0.17%
17	Студентська робота	ID файлу: 1011459306	Навчальний заклад: National Technical University of Ukraine "Kyi...		0.17%
18	Студентська робота	ID файлу: 1003818832	Навчальний заклад: Lviv Polytechnic National University		0.17%
19	Студентська робота	ID файлу: 1015183619	Навчальний заклад: Lutsk National Technical University		0.16%
20	Студентська робота	ID файлу: 1000762850	Навчальний заклад: National Technical University of Ukr	16 Джерело	0.14%

21	Студентська робота	ID файлу: 1005941383	Навчальний заклад: Lviv Polytechnic National University	2 Джерело	0.14%
22	Студентська робота	ID файлу: 1011568615	Навчальний заклад: Vasyl Stus Donetsk National University		0.14%
23	Студентська робота	ID файлу: 1015233302	Навчальний заклад: National Technical University of Ukraine “Kyi...		0.12%
24	Студентська робота	ID файлу: 1000040826	Навчальний заклад: National University of Life and Environmenta...		0.12%
25	Студентська робота	ID файлу: 1011471585	Навчальний заклад: National Technical University of Ukr	34 Джерело	0.12%
26	Студентська робота	ID файлу: 8319731	Навчальний заклад: Donetsk National Technical University		0.12%
27	Студентська робота	ID файлу: 1005761940	Навчальний заклад: National Technical University of Ukr	13 Джерело	0.11%
28	Студентська робота	ID файлу: 1004190091	Навчальний заклад: Uzhhorod National University	32 Джерело	0.11%
29	Студентська робота	ID файлу: 1009542325	Навчальний заклад: National Technical University of Ukraine “Kyi...		0.11%
30	Студентська робота	ID файлу: 1011458098	Навчальний заклад: National Technical University of Ukraine “Kyi...		0.11%
31	Студентська робота	ID файлу: 8467723	Навчальний заклад: Lviv Polytechnic National University		0.1%
32	Студентська робота	ID файлу: 1008273211	Навчальний заклад: National Technical University of Ukraine “Kyi...		0.1%