СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної випускної роботи

освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)

спеціалізація „Інженерія програмного забезпечення”
(назва спеціалізації)

на тему „Реалізація нейронних мереж на графічних процесорах за допомогою API Vulkan”

Виконав: студент групи ІПЗ-20д _____ С. А. Чушин
(підпис) (ініціали і прізвище)

Керівник _____ В. О. Лифар
(підпис) (ініціали і прізвище)

Завідувач кафедри _____ О.І. Захожай
(підпис) (ініціали і прізвище)

Рецензент___Ратов Д.В._____

Київ – 2024

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра інформаційних технологій та програмування
Освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)
спеціалізація „Інженерія програмного забезпечення”
(назва спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Захожай О.І.
“___” _____ 2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ ВИПУСКНУ РОБОТУ СТУДЕНТУ

Чушин Сергій Андрійович
(прізвище, ім’я, по батькові)

1. Тема роботи: Реалізація нейронних мереж на графічних процесорах за допомогою API Vulkan

керівник роботи Доц., д.т.н. Лифар В.О.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджений наказом університету від “_06_” _травня_2024 року №171/15.15-С

2. Строк подання студентом роботи 08.06.2024р.
3. Вихідні дані до роботи: Об’єктом даної роботи є розробка CNN з прискоренням на GPU та Vulkan API

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Вступ.
аналіз предметної області; вибір програмних засобів для розробки продукту; розробка продукту; висновки. Перелік використаних джерел.
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслеників)
6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 30.03.2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання кваліфікаційної випускної роботи	Строк виконання етапів	Примітка
1	Одержання завдання на виконання роботи	30.03.24	виконано
2	Укладання і погодження з керівником плану і етапів виконання роботи	06.04.24	виконано
3	Узагальнення даних літературних джерел, укладання розділу «Аналіз предметної галузі»	13.04.24	виконано
4	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху	20.04.24	виконано
5	Укладання та тестування програмного продукту	27.04.24	виконано
6	Укладання, оформлення та погодження пояснювальної записки з керівником	04.05.24	виконано
7	Здача готової пояснювальної записки на кафедру	08.06.24	виконано
8	Укладання доповіді і презентації	10.06.24	виконано

Студент _____ С. А. Чушин
підпис (ініціали і прізвище)

Керівник роботи _____ В. О. Лифар
підпис (ініціали і прізвище)

ЛИСТ ПОГОДЖЕННЯ І ОЦІНЮВАННЯ
дипломної роботи студента гр. ІПЗ-20д Чушин С. А.

Науковий керівник:

Професор, д.т.н. _____ Лифар В.О.

Оцінка наукового керівника: _____

Рецензент _____
ПІБ, місто роботи, посада

Оцінка рецензента: _____

Кінцева оцінка за результатами захисту:

Голова ЕК
Професор кафедри ІТП
д.т.н. _____ Меняйленко О.С.
підпис

РЕФЕРАТ

Робота "Прискорення згорткових нейронних мереж на графічних процесорах за допомогою Vulkan API" представляє собою важливе дослідження в області штучного інтелекту та обчислювальної техніки. Метою цієї роботи є вивчення можливостей прискорення обчислень згорткових нейронних мереж (CNN) на графічних процесорах (GPU) з використанням Vulkan API. В контексті цього дослідження були використані сучасні підходи та методи програмування для оптимізації алгоритмів CNN і їх виконання на GPU.

Дослідження почалося з аналізу особливостей роботи згорткових нейронних мереж та їх значення в сучасних додатках, що використовують комп'ютерний зір. Було проведено огляд існуючих технологій прискорення CNN, зокрема у контексті використання Vulkan API, яке надає високий рівень контролю над обчисленнями на GPU.

У роботі було детально проаналізовано та порівняно різні підходи до прискорення CNN на GPU, враховуючи ефективність, швидкість та оптимальність виконання алгоритмів. Особлива увага приділялася реалізації алгоритмів з використанням Vulkan API з метою досягнення максимальної продуктивності та швидкодії виконання.

У результаті проведених досліджень та розробок було створено прототип системи прискорення CNN на GPU з використанням Vulkan API. Цей прототип дозволяє ефективно виконувати згорткові операції нейронних мереж і має потенціал для подальшої оптимізації та розвитку.

Зміст

ВСТУП

Значна кількість сучасних додатків у сфері комп'ютерного зору, штучного інтелекту та обробки зображень вимагають ефективного обчислення згорткових нейронних мереж (CNN). Згорткові нейронні мережі, завдяки своїм універсальним здібностям у вирішенні задач класифікації, детекції та сегментації зображень, стали необхідним інструментом у багатьох областях, від медицини до автомобільної промисловості.

За останні кілька років, з огляду на зростання обсягу даних та складності моделей, виникає потреба у використанні потужних обчислювальних ресурсів для навчання та виконання CNN. Графічні процесори (GPU) завдяки своїм паралельним обчислювальним можливостям стали важливим інструментом у цьому контексті.

Одним із ефективних інтерфейсів для використання обчислювальних можливостей GPU є Vulkan API. Vulkan забезпечує високий рівень контролю над обчисленнями на GPU та дозволяє оптимізувати алгоритми для максимальної продуктивності.

У цій роботі досліджується прискорення виконання CNN на GPU з використанням Vulkan API. Основна мета полягає у розробці та оптимізації алгоритмів згорткових операцій для максимального використання потенціалу графічних процесорів у виконанні нейронних мереж.

Такий вступ надає читачеві загальне уявлення про те, що дослідження спрямоване на використання GPU та Vulkan API для прискорення обчислень у згорткових нейронних мережах, а також вказує на важливість і актуальність даної теми.

РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД

У сучасному світі штучний інтелект та глибоке навчання (Deep Learning) набули значного поширення у різних галузях, від розпізнавання зображень та обробки природної мови до автоматизації бізнес-процесів та розробки автономних систем. Зростання обсягів даних та складності моделей вимагає ефективних методів обробки та обчислень, що спонукає дослідників та інженерів шукати способи прискорення глибокого навчання. Одним з найбільш перспективних напрямів у цій сфері є використання графічних процесорів.

Графічні процесори, спочатку розроблені для обробки зображень та рендерингу відео, завдяки своїй архітектурі виявилися надзвичайно ефективними для виконання паралельних обчислень. Це робить їх ідеальним інструментом для виконання завдань глибокого навчання, які часто потребують обробки великих обсягів даних і виконання мільйонів операцій одночасно. У порівнянні з центральними процесорами, GPU можуть забезпечити значно вищу продуктивність та швидкість обчислень.

1.1 Програмні платформи та інструменти для обчислень на GPU

Для використання потужностей GPU у загальноцільових обчисленнях розроблено кілька програмних платформ та інструментів:

Рис. 1.0 CUDA

CUDA (Compute Unified Device Architecture): Платформа та набір інструментів від NVIDIA, що дозволяють розробникам писати програми

для виконання на GPU. CUDA підтримує мови програмування, такі як C, C++ та Python, що полегшує інтеграцію у існуючі програмні проекти.

Рис. 1.1 OpenCl

OpenCL (Open Computing Language): Відкрита стандартна платформа, яка підтримується різними виробниками апаратного забезпечення, включаючи AMD, Intel та NVIDIA. OpenCL забезпечує універсальність та портативність, дозволяючи писати код, що може виконуватися на різних типах пристроїв, включаючи GPU, CPU та інші акселератори.

Рис. 1.2 Vulkan

Vulkan API: Новий стандарт для графічних та обчислювальних обчислень, який забезпечує низькорівневий доступ до апаратного забезпечення. Vulkan API, розроблений консорціумом Khronos Group, дозволяє ефективно використовувати потужності сучасних GPU для виконання обчислень загального призначення.

Microsoft DirectCompute – це API (Application Programming Interface), що є частиною DirectX, призначеного для виконання обчислень загального призначення на графічних процесорах (GPU). Запущений компанією Microsoft, DirectCompute дозволяє розробникам використовувати потужність сучасних графічних карт для виконання

високопродуктивних обчислень, що виходять за рамки традиційного рендерингу графіки.

1.2 Концепти програмування GPGPU

1. Паралельні обчислення

Графічні процесори складаються з численних ядер, що дозволяє виконувати тисячі операцій одночасно. Паралельні обчислення включають розподіл великої задачі на менші частини, які можуть виконуватися незалежно одна від одної.

Data Parallelism (Паралелізм даних): Це основний тип паралелізму, використовуваний у GPGPU, де одна й та сама операція виконується над різними частинами даних. Наприклад, додавання двох векторів елемент за елементом.

Task Parallelism (Паралелізм завдань): Виконання різних завдань одночасно на різних ядрах GPU. Це менш поширено для GPU, оскільки вони більше оптимізовані для паралелізму даних.

2. Блоки і потоки

В архітектурі CUDA від NVIDIA, наприклад, поняття блоків і потоків є фундаментальними.

Threads (Потоки): Окремі обчислювальні одиниці, що виконують завдання. Потоки організовуються у блоки.

Blocks (Блоки): Групи потоків. Кожен блок виконується на одному мультипроцесорі (SM – Streaming Multiprocessor).

Grids (Сітки): Сукупність блоків, що запускаються одночасно на GPU.

3. Ядро (kernel)

Ядро – це функція, що виконується на GPU. Вона визначає, які обчислення повинні бути виконані кожним потоком. Ядра запускаються з боку хост-програми (зазвичай працюючої на CPU) з вказівкою конфігурації сітки потоків, яка визначає кількість блоків і потоків у кожному блоці.

4. Пам'ять

GPU має кілька типів пам'яті, кожен з яких має свої особливості і призначення.

Global Memory (Глобальна пам'ять): Найбільш ємна і повільна пам'ять, доступна всім потокам.

Shared Memory (Спільна пам'ять): Швидша пам'ять, що розділяється між потоками одного блоку. Використовується для обміну даними між потоками і для оптимізації доступу до глобальної пам'яті.

Registers (Реєстри): Найшвидша пам'ять, доступна окремим потокам. Обмежена кількість реєстрів може впливати на продуктивність.

Constant Memory (Константна пам'ять): Пам'ять для зберігання даних, які не змінюються під час виконання ядра.

Texture Memory (Текстурна пам'ять): Використовується для оптимізованого доступу до даних, які часто читаються.

5. Обробка даних та синхронізація

Thread Synchronization (Синхронізація потоків): Координація роботи потоків для забезпечення правильного порядку виконання операцій, наприклад, через бар'єри синхронізації.

Memory Coalescing (Об'єднання доступу до пам'яті): Оптимізація доступу до глобальної пам'яті, коли сусідні потоки доступаються до послідовних адрес пам'яті.

6. Обробка помилок та відлагодження

Програмування для GPU може бути складним через паралельну природу обчислень, тому відлагодження та обробка помилок є важливими аспектами.

Відлагодження: Використання інструментів, таких як NVIDIA Nsight або AMD CodeXL, для відстеження і діагностики помилок у коді для GPU.

Обробка помилок: Виявлення та обробка помилок, що виникають під час виконання обчислень на GPU, наприклад, через виклики спеціальних функцій обробки помилок.

7. Оптимізація продуктивності

Профілювання: Використання інструментів для профілювання (наприклад, NVIDIA Visual Profiler) для аналізу продуктивності і виявлення вузьких місць у коді.

Латентність і пропускна здатність: Оптимізація використання ресурсів GPU для зменшення латентності та максимізації пропускної здатності обчислень і пам'яті.

1.3 Vulkan Compute

Vulkan було розроблено з обов'язковою підтримкою обчислень: якщо пристрій може запускати Vulkan, він може запускати обчислювальні шейдери(compute shaders). Обчислювальні шейдери у Vulkan мають першокласну підтримку в API і можуть використовуватися для чисто обчислювальних навантажень без будь-якого графічного виводу, так званий "headless compute".

1.3.1 Основні концепти Vulkan Compute

Командні буфери (Command Buffers) є основними одиницями, що містять записи команд, які можуть бути виконані GPU. Вони дозволяють

групувати та організовувати обчислювальні завдання. Команди записуються в командний буфер і потім виконуються на GPU.

Черги (Queues) є каналами, через які командні буфери подаються на виконання. Vulkan підтримує різні типи черг для графічних, обчислювальних і трансферних операцій. Сімейства черг (Queue Families) є групами черг, що підтримують певний набір операцій.

Шейдери (Shaders) включають обчислювальні шейдери (Compute Shaders), які є програмами, що виконуються на GPU для виконання обчислювальних задач. Вони пишуться на мові GLSL або SPIR-V. SPIR-V є міжпроміжним репрезентативним форматом, який використовується для компіляції шейдерів.

Дескриптори та буфери (Descriptors and Buffers) включають дескриптори, які є інтерфейсами для зв'язку шейдерів з пам'яттю, та буфери, що є контейнерами для даних, що використовуються обчислювальними шейдерами.

1.3.2 Переваги Vulkan Compute

Основна перевага vulkan над CUDA, OpenCl, це кросплатформеність. Vulkan підтримується майже на всіх популярних платформах, та на gpu різних виробників. Це дозволяє виконувати складні обчислення, такі як виконання нейронних мереж, у тому числі на мобільних телефонах.

Дослідження показує потенційне використання машинного навчання в інтерактивних і високочастотних додатках:

- Анімація персонажів (фазова функція нейронних мереж і т.д.)
- Обробка зображень на весь екран (антиаліасинг, збільшення роздільної здатності, відновлення зображень, DLSS та інше)
- Боти неперсонажів (AlphaStar, OpenAI Five та інше)
- Генерація зображень (GAN, вогонь, дим та хмари та інше)

Поточні рішення машинного навчання мають відносно великі накладні витрати на взаємодію(interop) між різними програмними рішеннями:

- Взаємодія зі сторонніми фреймворками (Python TensorFlow, PyTorch, OpenCL та інше) призводить до виникнення вузких місць по пам’яті(bottlenecks), коли CPU/GPU не виконують корисну роботу.
- Обмін даними з зовнішніми API може бути складним через різницю в моделях пам'яті і може потребувати додаткових копіювань.

РОЗДІЛ 2. МОДЕЛЬ ПРОЕКТУ ТА ПОСТАНОВКА ЗАДАЧІ

Ми реалізуємо нейронну мережу CNN (Згорточна нейронна мережа). Навчання та виконання якої буде прискорено за допомогою GPU та Vulkan API.

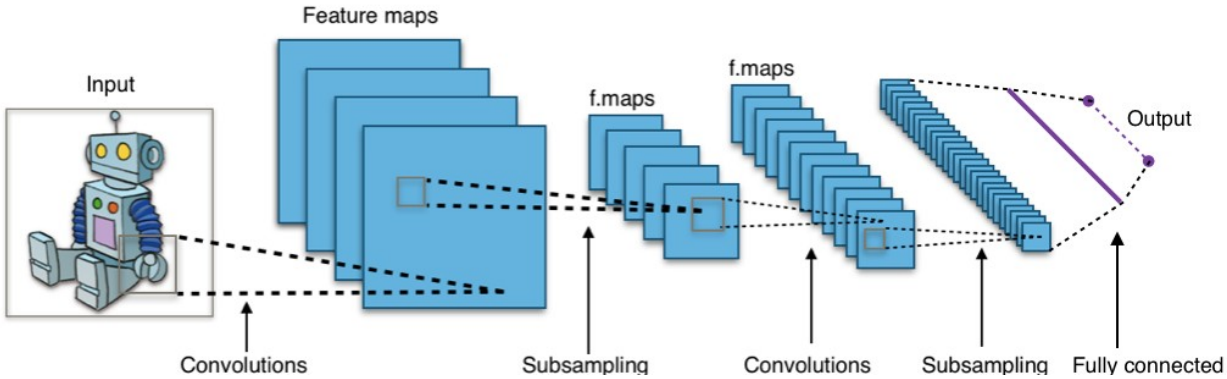

Рис 2.1. Архітектура згорточної нейронної мережи

Згорточна нейронна мережа (Convolutional Neural Network, CNN) — це тип глибокої нейронної мережі, який зазвичай використовується для аналізу візуальних даних. CNN успішно застосовуються в різних завданнях комп'ютерного зору, таких як класифікація зображень, виявлення об'єктів, сегментація зображень, розпізнавання обличчя та інші.

CNN обробляють вхідні дані через послідовність шарів, кожен з яких виконує конкретну функцію. Спочатку зображення проходить через кілька згорточних і шарів підвибірки, які витягують важливі особливості та зменшують розмірність даних. Після цього дані проходять через кілька повнозв'язних шарів, які виконують класифікацію або інші задачі.

Переваги CNN

- Автоматичне виділення ознак: CNN автоматично виявляють важливі ознаки на зображеннях без потреби ручного визначення.
- Перевірена ефективність: CNN показали високу точність у різних завданнях комп'ютерного зору.

• Масштабованість: CNN можуть ефективно працювати з великими наборами даних та складними зображеннями.

Застосування CNN

• Класифікація зображень: Розпізнавання об'єктів на зображеннях та віднесення їх до певних категорій.

• Виявлення об'єктів: Локалізація та ідентифікація об'єктів на зображеннях.

• Сегментація зображень: Розділення зображень на сегменти для виділення конкретних об'єктів або областей.

• Розпізнавання облич: Ідентифікація та верифікація осіб на фотографіях або відео.

• Аналіз медичних зображень: Виявлення патологій на медичних зображеннях, таких як рентгенограми або МРТ.

2.1 Основні компоненти CNN

Вхідний шар (Input Layer) конволюційної нейронної мережі (CNN) є першим шаром, який отримує та обробляє вхідні дані, зазвичай у вигляді зображень. Основна функція цього шару полягає в тому, щоб підготувати вхідні дані для подальшої обробки в наступних шарах мережі. Вхідний шар отримує дані з зовнішнього джерела. У випадку зображень ці дані представлені у вигляді багатовимірних масивів (тензорів), де розміри масиву відповідають висоті, ширині та кількості каналів зображення (наприклад, RGB-канали).

Вхідний шар підготовлює дані для подальшої обробки в наступних шарах. Це включає нормалізацію даних, тобто приведення значень пікселів до діапазону [0, 1] або [-1, 1], що покращує стабільність і швидкість навчання моделі.

Вхідний шар зберігає просторову структуру даних, що дозволяє наступним шарам використовувати цю інформацію для виявлення патернів, текстур і інших особливостей зображення.

Формат вхідних даних

Зазвичай вхідні дані представляються у вигляді тривимірного тензора з розмірами H×W×C, де:

- H — висота зображення (кількість рядків пікселів).
- W — ширина зображення (кількість стовпців пікселів).
- C — кількість каналів (наприклад, 3 для кольорових зображень RGB або 1 для чорно-білих зображень).

2. Конволюційний шар застосовує операцію згортки (convolution) до вхідних даних за допомогою набору фільтрів (ядер), які сканують вхідні дані для виявлення локальних патернів. Кожен фільтр відповідає за виявлення конкретного типу особливостей, таких як вертикальні або горизонтальні краї.

Конволюційний шар застосовує операцію згортки (convolution) до вхідних даних за допомогою набору фільтрів (ядер), які сканують вхідні дані для виявлення локальних патернів. Кожен фільтр відповідає за виявлення конкретного типу особливостей, таких як вертикальні або горизонтальні краї.

Фільтри(Ядра) — це невеликі матриці, які сканують вхідні дані. Розмір фільтра зазвичай менший, ніж розмір вхідного зображення (наприклад, 3x3, 5x5). Кожен фільтр має свої власні ваги, які налаштовуються під час навчання.

Згортка — це операція, при якій фільтр переміщується по всьому зображенню, обчислюючи скалярний добуток між значеннями пікселів вхідного зображення і вагами фільтра. Результати згортки формують вихідну карту ознак (feature map).

Підрізання визначає, наскільки далеко фільтр пересувається кожного разу. Наприклад, stride=1 означає, що фільтр переміщується на один піксель за раз, а stride=2 означає, що фільтр пересувається на два пікселі за раз.

Заповнення додає рамку навколо вхідного зображення, щоб контролювати розмір вихідної карти ознак. Zero padding (заповнення нулями) часто використовується для збереження розмірів вхідного зображення після згортки.

Результат операції згортки для пікселя (i, j) вихідної карти ознак можна виразити як:

$$Output(i,j) = \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} Input(i+m, j+n) \cdot Filter(m,n) + Bias$$

де:

- Input(i+m, j+n) — значення пікселя вхідного зображення.
- Filter(m,n) — значення фільтра.
- Bias — зміщення (bias).

Вихідна карта ознак — це результуючий масив, який зберігає результати операції згортки. Кожен елемент вихідної карти ознак відповідає за присутність певної ознаки у відповідному регіоні вхідного зображення.

Переваги конволюційного шару

1. Локальні зв'язки: Конволюційний шар використовує локальні зв'язки, що дозволяє виявляти локальні патерни у вхідних даних. Це особливо корисно для аналізу зображень, де важливі локальні особливості, такі як краї та текстури.

2. Спільні ваги: Фільтри в конволюційних шарах використовують спільні ваги, що зменшує кількість параметрів і знижує ризик перенавчання. Це також дозволяє виявляти ті самі особливості в різних частинах зображення.

3. Просторове зменшення: Конволюційні шари часто поєднуються з шарами підвибірки (pooling layers), які зменшують розмірність вихідних даних, зберігаючи важливу інформацію і зменшуючи обчислювальні витрати.

3. Шар активації (Activation Layer) є важливим компонентом нейронних мереж, зокрема конволюційних нейронних мереж (CNN). Основна мета цього шару полягає в додаванні нелінійності до моделі, що дозволяє нейронній мережі навчатися складним патернам і функціям. Без нелінійних функцій активації нейронні мережі не могли б ефективно моделювати складні дані.

2.1.1 Основні функції активації

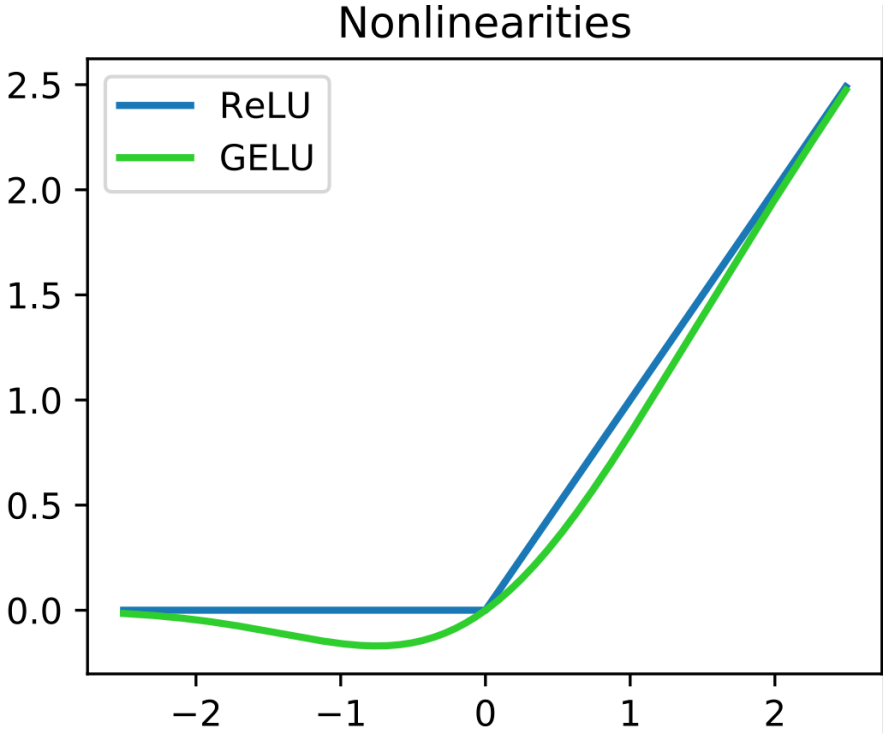

Рис. 2.2 ReLU

1. ReLU (Rectified Linear Unit) є найбільш поширеною функцією активації в сучасних нейронних мережах. Вона визначається як: $ReLU(x)=max(0,x)$ ReLU обнуляє всі негативні значення і пропускає позитивні без змін. Це сприяє швидкому і ефективному тренуванню моделі.

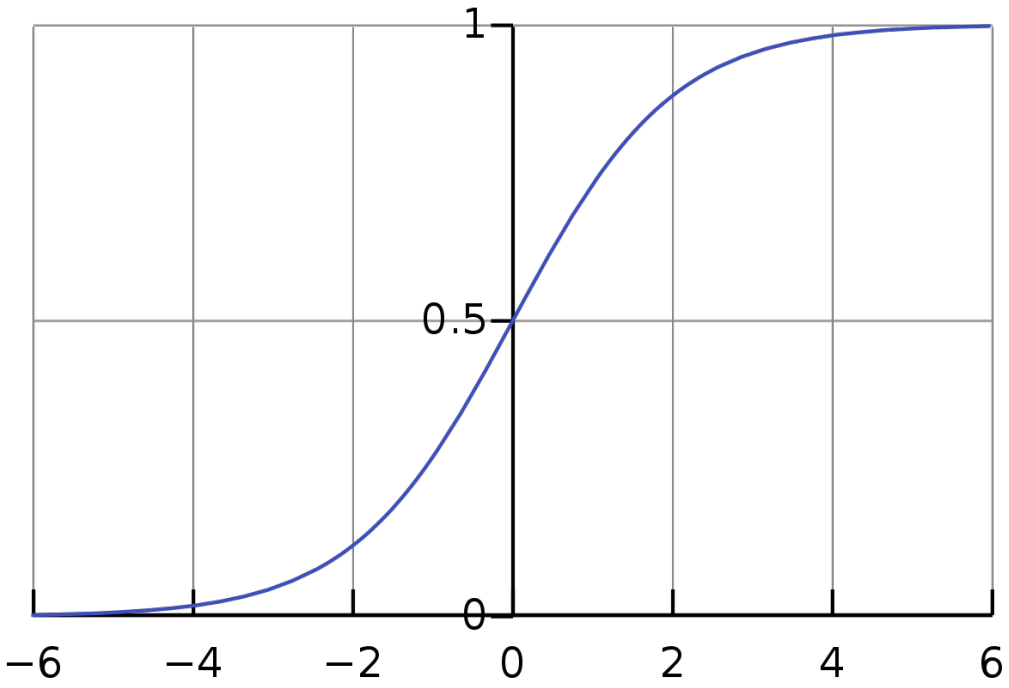

Рис. 2.3 Sigmoid

2. Сигмоїдна функція стискає вхідні значення до діапазону [0, 1]. Вона визначається як: $Sigmoid(x)=\frac{1}{1+e^{(-x)}}$ Сигмоїдна функція часто використовується в вихідних шарах для задач бінарної класифікації.

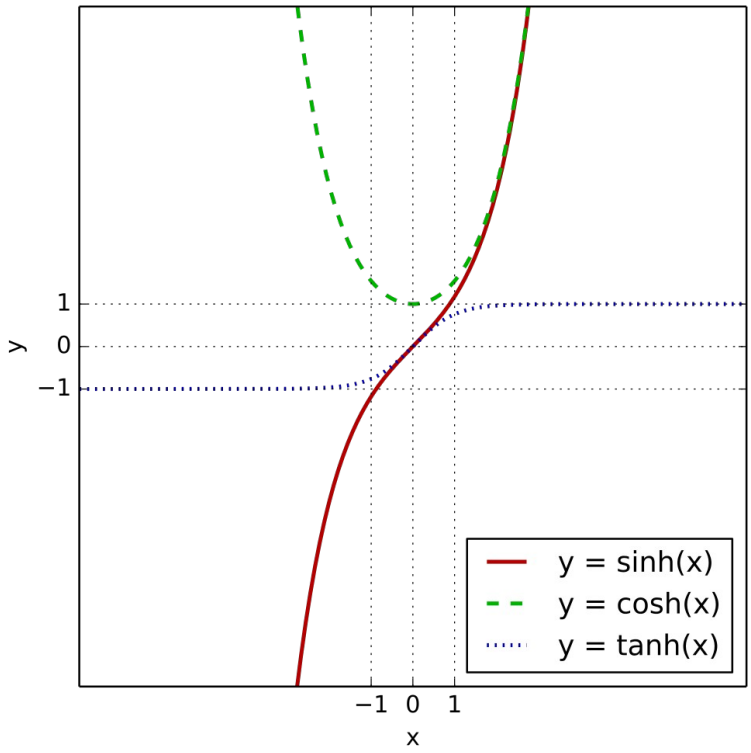

Рис. 2.4. Tanh

3. Функція тангенса гіперболічного стискає значення до діапазону [-1, 1]. Вона визначається як: $\tanh(x)=\frac{e^{x}-e^{(-x)}}{e^{x}+e^{(-x)}}$ Функція Tanh є зсунутою версією сигмоїдної функції, яка має середнє значення 0, що може поліпшити швидкість тренування.

4. Leaky ReLU — це модифікація ReLU, яка дозволяє невелику пропускну здатність для негативних значень: ReLULeaky $ReLU(x)=max(\alpha x, x)$ де α — мале значення, наприклад 0.01.

Основна роль шару активації це

Нелінійність: Функції активації вводять нелінійність у модель, що дозволяє нейронній мережі навчатися складним патернам. Без цієї нелінійності нейронні мережі були б еквівалентними до одного лінійного перетворення.

Розпізнавання складних патернів: Завдяки нелінійності, мережа може розпізнавати складні патерни і взаємозв'язки у вхідних даних, що є

важливим для багатьох завдань, таких як розпізнавання зображень, обробка мови тощо.

Нормалізація: Деякі функції активації, такі як Sigmoid і Tanh, нормалізують вихідні значення, що може бути корисним для стабільності тренування.

4. Шар підвибірки (Pooling Layer) є важливим компонентом конволюційних нейронних мереж. Його основна мета полягає в зменшенні просторових розмірів вхідних даних, що допомагає знизити обчислювальні витрати та кількість параметрів у моделі, а також робить моделі менш чутливими до зміщень і деформацій у вхідних даних та допомагає зменшити ризик перенавчання моделі, оскільки зменшується кількість параметрів і обчислювальних операцій.

Шар підвибірки зменшує розмір вхідного тензора за допомогою операції підвибірки із заданим розміром вікна (kernel size) і кроком (stride). Наприклад, для Max Pooling з вікном розміром 2x2 і кроком 2, шар вибирає максимальне значення з кожного підрегіону розміром 2x2 і пересувається на 2 пікселі по горизонталі та вертикалі.

Шари підвибірки бувают різних типів. Ось основні з них.

Max Pooling обирає максимальне значення з кожного підрегіону вхідного зображення. Це найпоширеніший тип підвибірки, який ефективно витягує найбільш виразні ознаки.

Average Pooling обчислює середнє значення для кожного підрегіону вхідного зображення. Цей тип підвибірки використовується рідше, але може бути корисним у деяких задачах.

Global Pooling обчислює максимальне або середнє значення для всього вхідного зображення, що дозволяє значно зменшити розмірність вихідного тензора. Використовується перед повнозв'язними шарами в кінці моделі.

5. Полносвязний шар (Fully Connected Layer), також відомий як шар щільного з'єднання (Dense Layer), є ключовим компонентом нейронних мереж, зокрема в останніх шарах конволюційних нейронних мереж (CNN). Його основна мета полягає у виконанні класифікації або регресії шляхом об'єднання ознак, витягнутих попередніми шарами, та прогнозування кінцевих вихідних значень.

Принцип роботи полносвязного шару

Полносвязний шар складається з нейронів, кожен з яких пов'язаний з усіма нейронами попереднього шару. Це означає, що кожен нейрон у шарі отримує всі вихідні значення з попереднього шару як вхідні дані.

Основні компоненти полносвязного шару

1. Вхідні дані (Input): Вхідні дані до полносвязного шару зазвичай представлені у вигляді одномірного вектора, що є результатом згорткових і підвибіркових шарів.

2. Ваги (Weights): Кожен зв'язок між нейронами має відповідну вагу, яка множиться на значення вхідного нейрона. Ваги ініціалізуються випадковими значеннями і налаштовуються під час навчання для мінімізації функції втрат.

3. Зміщення (Bias): До кожного нейрона додається зміщення, яке дозволяє моделі краще підлаштовуватися до даних.

4. Функція активації (Activation Function): Після обчислення зваженого сумарного значення кожен нейрон застосовує нелінійну функцію активації, таку як ReLU, Sigmoid або Softmax, щоб отримати кінцеве значення нейрона.

Формула обчислення вихідного значення

Вихідне значення для нейрона j у полносвязному шарі можна виразити як:

$$y_j = \varphi\left(\sum_{i=1}^{n} w_{ij} x_i + b_j\right)$$

де:

- x_i — значення вхідного нейрона i,
- w_{ij} — вага, що зв'язує вхідний нейрон i з вихідним нейроном j,
- b_j — зміщення для нейрона j,
- ϕ — функція активації.

Використання в моделі

Полносвязний шар зазвичай використовується на завершальних етапах нейронних мереж для об'єднання витягнутих ознак і виконання кінцевої класифікації або регресії.

6. Вихідний шар (Output Layer) є заключним шаром у нейронних мережах, зокрема конволюційних нейронних мережах (CNN). Основна функція вихідного шару полягає у формуванні остаточних прогнозів або класифікацій на основі оброблених даних, що пройшли через попередні шари.

Вихідний шар приймає вхідні дані від попереднього шару (зазвичай полносвязного шару) і застосовує до них відповідну функцію активації, щоб отримати остаточні прогнози. Кількість нейронів у вихідному шарі відповідає кількості класів у задачі класифікації або кількості вихідних параметрів у задачі регресії.

Основні типи функцій активації для вихідного шару

1. Функція Softmax використовується в задачах багатокласової класифікації. Вона перетворює вихідні значення нейронів у ймовірності, що їх сума дорівнює 1. Формула $Softmax(z_i)=\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$ де z_i — вихідне значення нейрона, K — кількість класів.

2. Функція Sigmoid використовується для бінарної класифікації, повертаючи ймовірність належності до певного класу. Формула $Sigmoid(z)=\frac{1}{1+e^{(-z)}}$

3. Лінійна функція активації використовується в задачах регресії, де вихідне значення є неперервним. У цьому випадку вихідні значення не перетворюються, тобто: $Linear(z)=z$

2.2 Датасет

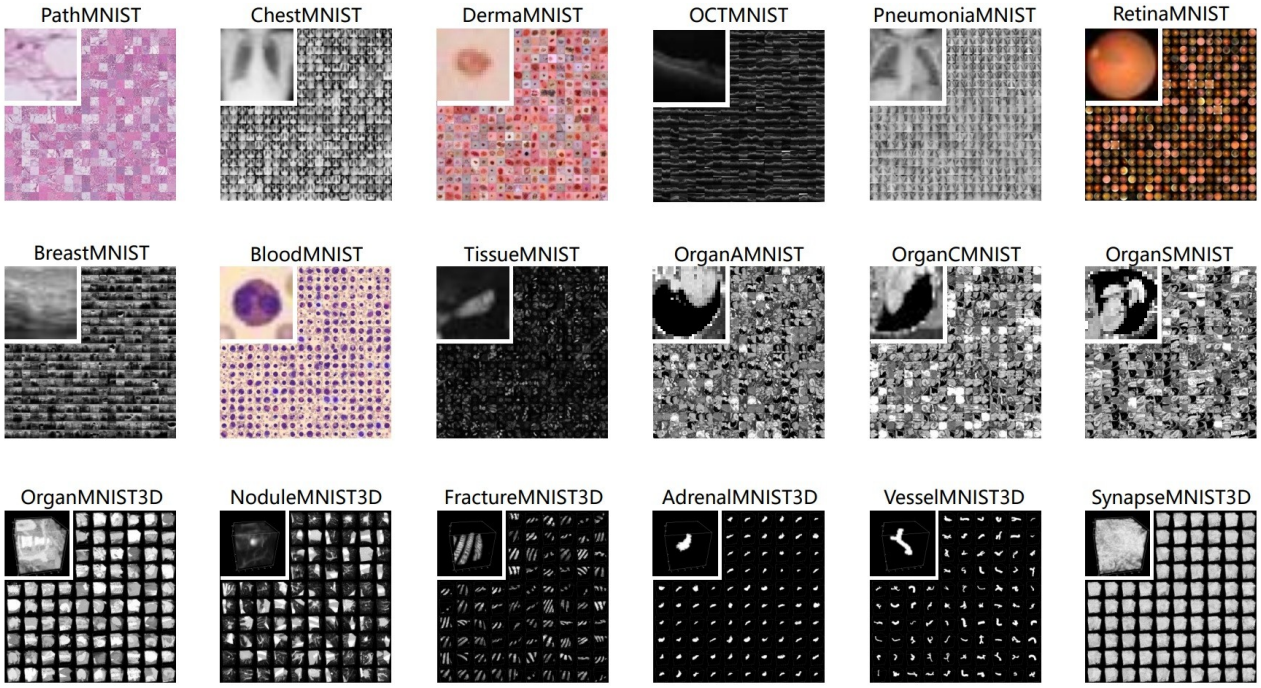

Рис. 2.5. MedMNIST Датасет

Для тренування CNN ми будем використовувати DermaMNIST - це велика колекція дерматоскопічних зображень поширених пігментних уражень шкіри з різних джерел. Набір даних складається з 10015 дерматоскопічних зображень, віднесених до 7 різних захворювань, що сформульовано як завдання багатокласової класифікації. Вихідні зображення розміром 3 × 600 × 450 змінено на 3 × 224 × 224.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ЗГОРТОЧНОЇ НЕЙРОННОЇ МЕРЕЖІ ТА ЇЇ ПРИСКОРЕННЯ НА GPU

3.1 Прискорення CNN на Vulkan API

В першу чергу виконнаня на GPU нам потрібно для пришвидшення навчання. Центральний процесор знаходиться в центрі і керує роботою різного обладнання в комп'ютерній системі. Графічні процесори, як для настільних/серверних систем, де ми зазвичай бачимо дискретні, так і для мобільних/вбудованих систем, де ми зазвичай бачимо інтегровані, є лише додатковими спеціалізованими обчислювальними пристроями в цій архітектурі. Графічні процесори мають власні процесори для виконання команд та ієрархію пам'яті для зберігання даних. Ці характеристики означають, що графічні процесори можуть виконувати програми асинхронно з центральним процесором. Але він не може робити це самостійно; він повинен отримувати робоче навантаження від центрального процесора.

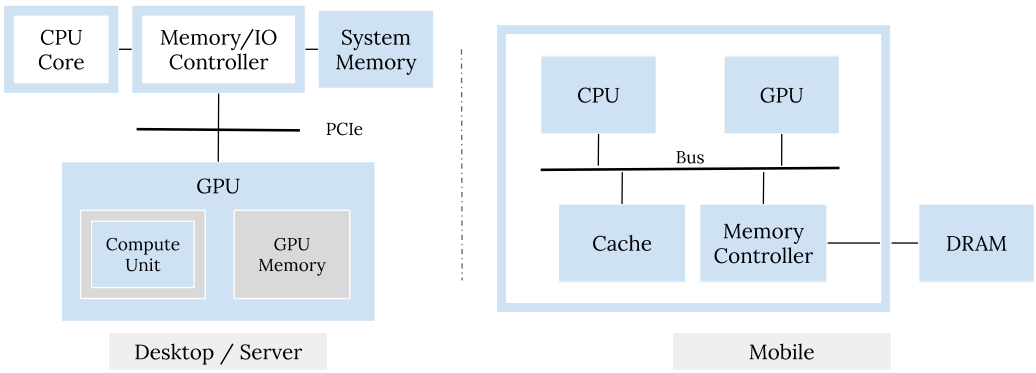

Рис. 3.1. Модель пам’яті

Тому, щоб використовувати GPU, CPU повинен 1) виявити його, 2) підготувати ресурси і програму для нього, 3) скласти і відправити робоче навантаження на нього для обробки, і 4) синхронізуватися з ним, щоб знати про хід роботи. Це в основному ті функціональні категорії, які можна було б очікувати від GPU API, який в основному виконується на

CPU і призначений для абстрагування від вищезазначеного. Крім того, оскільки GPU має високопаралельну структуру з багатьма обчислювальними блоками, він також потребує 4) внутрішньої синхронізації. Ви можете досить легко знайти виклики вищезгаданих категорій у різних API GPU, як для графіки, так і для обчислень; Vulkan тут не є винятком.

Однак, через свою явну природу, площа поверхні API Vulkan, що не дивно, більша, ніж у інших GPU API. Тому ви знайдете більше викликів API для вищезгаданих категорій у Vulkan. Крім того, для API епохи CUDA/OpenCL драйвер GPU внутрішньо керує глобальним контекстом програми і відстежує час життя різних об'єктів, що відповідають ресурсам GPU. Але для Vulkan, в обмін на низькі накладні витрати і явний контроль, відповідальність лежить на плечах розробників. Тому нам також доведеться вручну 5) керувати часом життя об'єктів на GPU.

3.2 Графічний та обчислювальний Vulkan API

Сьогодні, хоча ми можемо рендерити казкові сцени в AAA-іграх, ми все ще перебуваємо на стадії, коли використовуються короткі шляхи та наближення (наприклад, растеризація), тому що генерація геометричних примітивів та освітлення у найбільш природній спосіб (чиста трасування променів) все ще вимагає величезного обсягу обчислень, що перевищує можливості сучасних графічних процесорів. Це створює навантаження на графічний конвеєр та API. Наприклад, нам потрібно мати етапи з фіксованими функціями (збірка вхідних даних, растеризація і т.д.), торгуючи гнучкістю заради продуктивності; нам потрібно підтримувати різні «трюки» (наприклад, MSAA) безпосередньо в API, що обтяжує API.

Для виявлення GPU графіка додатково повинна враховувати безліч різних форматів зображень і обмежень на представлення.

Програма - це те місце, де графіка і обчислення відрізняються найбільше. Як було сказано вище, графіка використовує графічний конвеєр, який є сумішшю стадій з фіксованими функціями і програмованих стадій. Це призводить до величезних ускладнень, оскільки нам потрібно як конфігурувати ці стадії з фіксованими функціями, так і надавати шейдери для різних програмованих стадій. І вони повинні відповідати інтерфейсу. Обчислювальний конвеєр набагато простіший: він містить лише один програмований шейдер (або ядро у термінології CUDA/OpenCL).

Взаємодія між програмою та системою

Додаток Vulkan взаємодіє з системою Vulkan через екземпляр Vulkan, який містить специфічні для додатка стани. Додаток запитує доступні графічні процесори в системі і отримує список фізичних пристроїв. Потім програма продовжує запитувати доступні функції цих фізичних пристроїв і вибирає один або декілька з них для створення логічних пристроїв. Логічний пристрій - це uber-об'єкт, до якого програма звертається для виділення ресурсів, компіляції та виконання програм.

Пристрій Vulkan розкриває одну або більше черг, які є абстракцією обчислювальних блоків на графічному процесорі. Робочі навантаження (зв'язування ресурсів і програми та виконання програми) записуються в буфери команд і передаються в черги. Різні черги можуть обробляти роботу асинхронно, тому необхідна синхронізація, щоб гарантувати коректність залежностей між чергами; це семафори. Навіть всередині однієї черги команди можуть виконуватися позачергово, тому тут також потрібна синхронізація. (Це одне з місць, де Vulkan суттєво відрізняється від CUDA/OpenCL, де за замовчуванням усі команди виконуються і

завершуються у порядку надходження). Це в основному бар'єри конвеєра. Програма може знати про хід роботи на GPU від CPU через огорожі.

Виявлення GPU

Vulkan відкритий і масштабований. Ми можемо мати кілька реалізацій Vulkan в одній системі. Наприклад, зараз у ноутбуках часто можна побачити інтегрований графічний процесор та дискретний графічний процесор; вони обидва можуть підтримувати Vulkan. Різні реалізації Vulkan можуть мати різні архітектури від різних виробників обладнання (AMD, ARM, NVIDIA, Qualcomm тощо), орієнтовані на мобільні/вбудовані та настільні/серверні сценарії.

Для такого широкого охоплення Vulkan має багатий набір функціональних API-викликів для перерахування пристроїв і виявлення їхніх меж (залежних від реалізації мінімумів/максимумів певних властивостей), можливостей (необов'язкових функцій на рівні специфікації ядра, які можуть підтримуватися не всіма реалізаціями) і розширень (набір функцій, підтримуваних одним або декількома виробниками).

Ці виклики API зазвичай мають префікс vkEnumerate/vkGet. Важливі з них включають:

vkEnumeratePhysicalDevices: перераховує всі реалізації Vulkan у системі.

vkGetPhysicalDeviceProperties22: повертає величезну структуру, що містить різні властивості та обмеження пристроїв. Ми можемо об'єднати структури для Vulkan 1.0, 1.1 і 1.2 за допомогою pNext, щоб запитувати такі властивості, як ідентифікатор постачальника (з 1.0), назва пристрою (з 1.0), розмір підгрупи (з 1.1), підтримувані операції підгрупи (з 1.1), режим округлення з плаваючою комою (з 1.2) і важливі обмеження на обчислення, такі як максимальний розмір спільної пам'яті (з 1.0), максимальний розмір робочих груп (з 1.0).

vkGetPhysicalDeviceFeatures2: подібно до vkGetPhysicalDeviceProperties2, але для запиту додаткових можливостей рівня специфікації ядра. Ми також можемо з'єднати структури для Vulkan 1.0, 1.1 і 1.2 за допомогою pNext. Серед цікавих можливостей - підтримка обчислень у форматі float64/int16/int64 (починаючи з 1.0), підтримка вказівників на змінні (починаючи з 1.1), підтримка обчислень у форматі float16/int8 (починаючи з 1.2), підтримка моделі пам'яті Vulkan (починаючи з 1.2), підтримка адреси буферного пристрою (починаючи з 1.2).

vkEnumerateInstanceExtensionProperties/vkEnumerateDeviceExtensionProperties: запитує підтримку розширень Vulkan на рівні екземпляра/пристрою.

vkGetPhysicalDeviceQueueFamilyProperties2: запитує характеристики сімейства черг, наприклад, чи має воно можливості передачі/обчислення.

vkGetPhysicalDeviceMemoryProperties2: запитує характеристики пам'яті, такі як доступні купи пам'яті (для різних банків фізичної пам'яті) та типи (для різних сценаріїв використання, кешовані/некешовані/тощо, поверх певної купи).

Ці виклики API зазвичай викликаються на самому початку роботи програми, коли ми намагаємося виявити доступні реалізації Vulkan та їхні характеристики, вибрати одну з них для створення екземпляра/пристрою, а потім отримати з неї відповідну чергу:

vk{Create|Destroy}Instance: створює/знищує екземпляри Vulkan. При створенні нам потрібно надати інформацію про програму та запитати, які розширення Vulkan екземплярів увімкнути.

vk{Create|Destroy}Device: створює/знищує логічні пристрої Vulkan. Тут ми вказуємо, які розширення пристроїв увімкнути і скільки черг створити, а також їхні відносні пріоритети.

vkGetPhysicalDeviceMemoryProperties2 є чимось особливим, оскільки вона взаємодіє з розподілом буферів, що пояснюється у наступному розділі.

Підготовка ресурсів та програми

У Vulkan є два типи ресурсів: буфери (для неструктурованих даних у вигляді мішків слів) і зображення (для структурованих даних у певному форматі). Але я не буду торкатися зображень у цьому блозі. Вони є основним джерелом ускладнень. Вони можуть використовуватися в моделях машинного зору, але все ще не так часто зустрічаються в обчисленнях.

Для буферів у Vulkan відокремлено пам'ять та об'єкт хендла для наочності та зрозумілої моделі витрат, оскільки виділення пам'яті коштує дорого, і різні додатки можуть підходити до цього дуже по-різному. Тому краще уникати їхнього об'єднання, щоб не обмежувати вибір.

Виділення пам'яті

vk{Allocate|Free}Memory: створює/звільняє пам'ять. Для кращої продуктивності, зазвичай ми хочемо виконати велике виділення, а потім підвиділення.

vk{Map|UnMap}Memory: відображає виділену пам'ять, видиму хосту, щоб отримати вказівник на стороні процесора, щоб ми могли читати або записувати / розмотувати вказівник на стороні процесора після читання або запису.

vk{Flush|Invalidate}MappedMemoryRanges: очищає / анулює діапазон відображеної пам'яті.

vk{Map|UnMap}Memoryivk{Flush|Invalidate}MappedMemoryRanges також є прикладами явності Vulkan. Вони відображають характеристики архітектури графічного процесора. GPU може мати власну виділену пам'ять. Отримання вказівника з боку процесора на таку пам'ять означатиме надання можливості ядру виконувати відображення сторінок

та розподіл між адресними просторами програми та драйверів ядра. Крім того, через різні рівні кешу, записи на CPU або GPU можуть не бути одразу доступними або видимими для інших. Тому нам потрібне очищення (для CPU → GPU) або анулювання (для GPU → CPU). Я, звичайно, спрощую, оскільки за цим стоїть багато деталей, які чудово пояснюються в цій статті в блозі.

Буферний об'єкт та зв'язування

vk{Create|Destroy}Buffer: створює/знищує об'єкт буфера. При створенні буферного об'єкта, окрім його розміру, нам також потрібно вказати його використання, щоб драйвери могли оптимізувати роботу відповідно.

vkGetBufferMemoryRequirements2: запитує вимоги до пам'яті (наприклад, тип пам'яті, вирівнювання) для конкретного буфера.

vkBindBufferMemory2: прив'язує об'єкт буфера до конкретної пам'яті.

Вище наведено виклики API для керування буферними ресурсами. Далі розглянемо програмну частину, яка складається з обчислювального конвеєра.

Конвеєр дескрипторів та обчислень

Обчислювальний конвеєр є дуже простим і чистим у порівнянні з графічним конвеєром. Він містить лише один програмований етап обчислення шейдерів. Vulkan шейдери повинні бути виражені у двійковому форматі SPIR-V.

Обчислювальний шейдер посилається на буферні ресурси, які повинні бути надані середовищем виконання Vulkan. Ці буфери знаходяться у прив'язаних «слотах»; кожен слот має прив'язаний номер, а слоти організовано у різні набори. Обчислювальний конвеєр (що містить лише один обчислювальний шейдер) описує свої потреби у використанні буферів у вигляді схеми конвеєра, яка містить декілька наборів

дескрипторів. Кожен дескриптор є дескриптором деякого буферного ресурсу з точки зору шейдера.

Дескриптори забезпечують ще один рівень опосередкування, який дозволяє відокремити обчислювальний конвеєр від конкретних буферних ресурсів. Таким чином, ми можемо використовувати той самий обчислювальний конвеєр з різними буферами. Це допомагає амортизувати вартість обчислювальних конвеєрів, оскільки їх створення означає компіляцію SPIR-V у драйвері для GPU ISA. Це може бути дорого.

Перед виконанням Vulkan має зв'язати конкретні набори дескрипторів, які відповідають схемі обчислювального конвеєра. Дескриптори є конкретними об'єктами у графічному процесорі (оскільки вони мають бути виділені всередині драйвера графічного процесора); тому ми використовуємо об'єкти пулу для амортизації витрат на їх виділення.

vk{Create|Destroy}DescriptorSetLayout: створює/знищує макет для набору дескрипторів. При створенні нам потрібно вказати тип дескриптора для кожного взятого номера зв'язування. Для буферів це зазвичай UNIFORM_BUFFER або STORAGE_BUFFER або їхні _DYNAMIC варіанти.

vk{Create|Destroy}PipelineLayout: створює/знищує компонування для всього обчислювального конвеєра. В основному це вказує на декілька заданих компонувань.

vk{Create|Destroy}DescriptorPool: створює/знищує пули дескрипторів. Потрібно вказати максимальну кількість наборів, які він може підтримувати, і максимальну кількість дескрипторів для кожного типу.

vk{Allocate|Free}DescriptorSets: виділяє/звільняє дескриптори з/до пулу.

vkUpdateDescriptorSets: дійсно пов'язує набори дескрипторів з конкретними буферними ресурсами. Пам'ятайте, що дескриптори - це лише об'єкти дескрипторів. Вони повинні бути підкріплені конкретними буферами, перш ніж їх буде використано у виконанні.

Маючи набір дескрипторів, ми можемо створити конвеєр обчислень:

vkCreateShaderModule: створює модуль шейдерів з блобу SPIR-V.

vkCreateComputePipelines: створює конвеєр обчислень. Нам потрібно вказати модуль шейдеру SPIR-V для обчислення та схему конвеєра.

vkDestroyPipeline: знищує конвеєр обчислень.

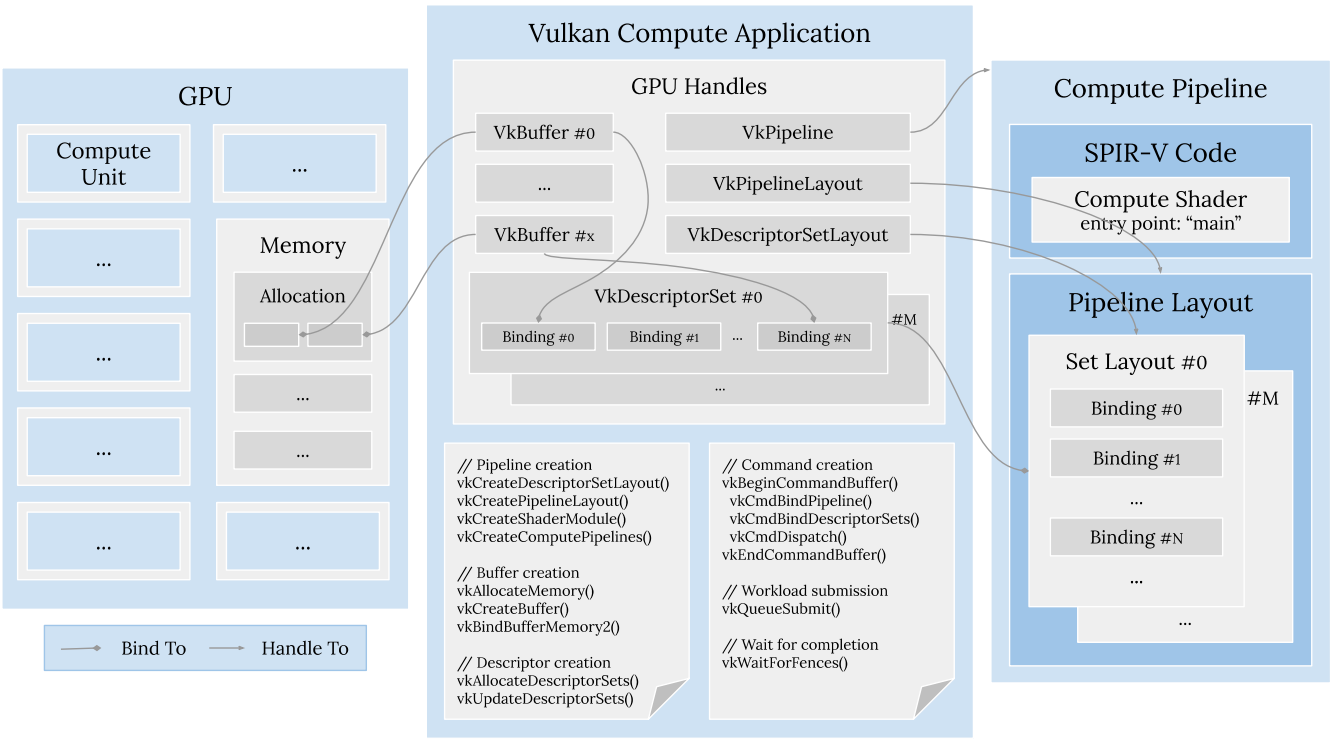

Рис. 3.2. Комунікація робочого навантаження

Як і центральний процесор, процесор всередині GPU також споживає потік команд. Ці команди готуються центральним процесором і записуються в буфер команд у Vulkan. Насправді у Vulkan буфер команд -

це лише абстракція для CPU, щоб обробляти робоче навантаження, можливо, у багатопотоковому режимі. На GPU немає концепції буфера команд (якщо не брати до уваги реальні потреби в буферному сховищі для цих команд), оскільки GPU просто бачить потік команд.

Буфери команд вимагають конкретних ресурсів від GPU, тому вони мають виділені об'єкти пулу, які допомагають амортизувати витрати на розподіл. Команди можна записувати у буфер команд після vkBeginCommandBuffer. Типовою послідовністю є прив'язка конвеєра обчислень (для запуску програми), прив'язка наборів дескрипторів (для посилання на ресурси буфера) та відправка (для запуску виконання). Потім vkEndCommandBuffer припиняє запис і переводить буфер команд у стан очікування, готовий до передачі у чергу за допомогою vkQueueSubmit.

vk{Create|Reset|Destroy}CommandPool: створює/скидає/знищує пул команд. Пул команд пов'язано з певною чергою команд, і виділені з нього буфери команд можуть бути передані лише до цієї черги.

vk{Allocate|Free}CommandBuffers: виділяє/звільняє командні буфери з/до пулу.

vk{Begin|End}CommandBuffer: починає/зупиняє запис буфера команд.

vkCmdBindPipeline: прив'язує обчислювальний конвеєр до буфера команд.

vkCmdBindDescriptorSets: прив'язує набори дескрипторів до командного буфера.

vkCmdDispatch{|Base|Indirect}: відправляє робоче навантаження.

vkQueueSubmit: додає буфери команд до черги для запуску асинхронного виконання на графічному процесорі.

Зауважте, що vkQueueSubmit може приймати декілька командних буферів разом, оскільки передача до черги є дороговартісною дією, тому її варто амортизувати. vkQueueSubmit також є місцем, де ми можемо приєднати примітиви синхронізації, про що буде сказано у наступному розділі.

Синхронізація

Синхронізація, мабуть, одна з найбільш непрозорих тем у Vulkan. Я просто зроблю вступ і не буду вдаватися в подробиці. (Щоб добре пояснити їх, може знадобитися багато постів у блозі! Ви вже можете знайти фантастичні вже існуючи, такі як цей, цей і цей).

Важливо пам'ятати, що за замовчуванням у Vulkan команди можуть виконуватися в неправильному порядку, тому для коректної роботи нам знадобиться явна синхронізація. Також важливо пам'ятати, що за замовчуванням Vulkan не гарантує когерентності пам'яті (як і базове апаратне забезпечення!), тому нам потрібно мати явні бар'єри пам'яті.

Загалом, існує декілька примітивів синхронізації для задоволення різних потреб:

vk{Create|Destroy}Fence: створює/знищує огорожі, які використовуються для очікування GPU на CPU.

vk{Create|Destroy}Semaphore: створює/знищує семафори. Тут варто зазначити одну річ: існує різниця між двійковими семафорами і семафорами часової шкали. Перші використовуються для внутрішньої синхронізації між чергами графічного процесора, але мають багато обмежень. Другі є потужнішим нещодавнім доповненням до Vulkan, яке знімає багато обмежень і дещо уніфікує примітиви синхронізації, покриваючи потреби синхронізації як на CPU, так і на GPU.

vkCmdPipelineBarrier: вставляє у буфер команд бар'єр конвеєра, який використовується для синхронізації команд в одній черзі.

vk{Create|Destroy}Event: створює/знищує події. Події також називають «розділеними бар'єрами». Вони мають окремі стадії сигналізації та очікування. І сигналізувати/очікувати може як CPU, так і GPU. Отже, у нас тут є кілька викликів API:

vkCmd{Set|Reset}Event: сигналізує/скидає на GPU; vkCmdWaitEvents: чекає на GPU.

vk{Set|Reset}Event: сигналізує/скидає на CPU; vkGetEventStatus: чекає на CPU.

За допомогою vkQueueSubmit ми можемо вказати список семафорів для очікування, список семафорів і один паркан для сигналізації.

І є великі молотки, які просто зливають всю чергу або пристрій: vkQueueWaitIdle і vkDeviceWaitIdle. Їх не варто використовувати, окрім випадків, коли ми виходимо з програми.

3.3 Прискорення виконання та навчання за допомогою Vulkan

Для того, щоб вирішити, яку частину мережі слід прискорити потрібно визначити найбільш затратну в обчислювальному плані частину мережі. Існує загальний консенсус, що шар згортки (convolution layer) є найбільш вимогливим.

Ми прискоримо шари субдискретизації/об'єднання, навіть якщо хоча там виконується лише 0.8% обчислень. Причиною цього є те, що піддискретизація/об'єднання може бути виконана практично паралельно зі згорткою паралельно зі згорткою, з мінімальними витратами апаратних ресурсів.

3.4 Архітектура програмного забезпечення

На рисунку показано спрощену версію архітектури чистого програмного забезпечення реалізації нашої мережі. Кожен шар містить набір попередньо навчених ваг які завантажуються перед тим, як мережа

починає обробляти зображення. Коли зображення надходить на перший шар, він виконує обчислення і передає результат на наступний шар.

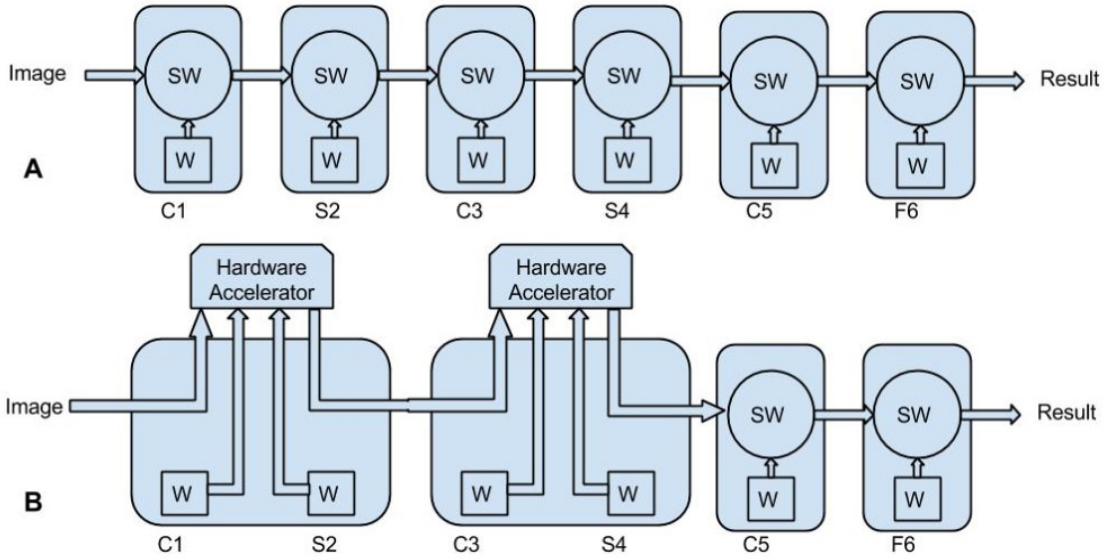

Рис. 3.3. Модель пам’яті

На рисунку показано, як було змінено оригінальне програмне забезпечення для використання прискорювача. Ми вирішили прискорити шар згортки і шар підвибірки/об'єднання середніх, тому ми написали новий програмний модуль який би обробляв обидві операції. Але замість того, щоб обчислювати операції у програмному забезпеченні, новий модуль передає вхідні дані та вагу на апаратний апаратному прискорювачу і витягує результат з обчислень. Хоча наша архітектура підтримує прискорення C5, ми утрималися від використання його в поточному вигляді. Причина в тому, що наразі прискорювач здатен обчислювати лише здатний обчислювати лише одну карту ознак за раз. Кожне обчислення супроводжується певну кількість накладних витрат, тобто передача даних на/з прискорювача та його конфігурування. Таким чином, для C5, який приймає 120 матриць 5 × 5, вхідні дані були настільки численними і малими, що це призвело б до надто великих накладних витрат, щоб для того, щоб бути ефективними.

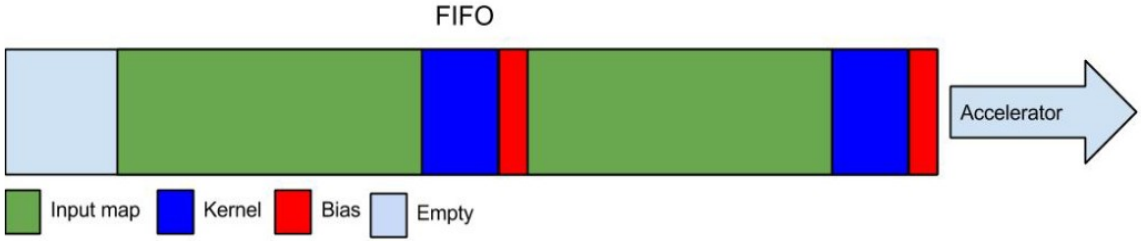

Рис. 3.2. Робота прискорювача за FIFO

Прискорювач був написаний з метою створення простого і зручного інтерфейсу для інтерфейсу з апаратним забезпеченням. Як вже було сказано, у поточному вигляді прискорювач здатен обчислювати карту функцій за один раз, тому вхідними даними для драйвера є всі дані, необхідні для обчислення згаданої карти особливостей. Тобто, набір зображень, їх відповідні ядра та зсув, середня константа об'єднання та її відповідний зсув. Потім драйвер передає ці дані прискорювачу і повертає обчислену карту особливостей. Через архітектуру прискорювача вхідні дані мають бути передаватися у певному порядку. Спочатку зміщення та середня константа об'єднання, вагові коефіцієнти - другими, а зображення - останніми. На рисунку показано, в якому порядку дані повинні знаходитись у FIFO-буфері прискорювача.

3.5 WGSL мова програмування для написання шейдерів

WGSL (WebGPU Shading Language) — це мова програмування для написання шейдерів, яка використовується з API WebGPU. Вона розроблена для забезпечення безпечного, продуктивного та передбачуваного коду для графічних та обчислювальних шейдерів. WGSL є частиною стандарту WebGPU, який є сучасним графічним API, спрямованим на заміну WebGL.

WGSL розроблений з акцентом на безпеку, що запобігає помилкам під час виконання. Це досягається через строгий синтаксис і типізацію. WGSL забезпечує ефективне використання ресурсів GPU, що дозволяє

досягти високої продуктивності. WGSL спеціально створений для використання з API WebGPU, що забезпечує оптимальну інтеграцію з сучасними веб-технологіями. Код, написаний на WGSL, може бути виконаний на різних платформах і пристроях, що підтримують WebGPU.

WGSL підтримує основні типи даних, такі як скалярні типи (int, float), вектори (vec2, vec3, vec4), матриці (mat2x2, mat3x3, mat4x4), а також користувацькі структури. Змінні можуть бути визначені як в межах функцій, так і глобально. Вони можуть бути різних типів, включаючи текстури, буфери та уніформи.

WGSL підтримує визначення користувацьких функцій для організації та повторного використання коду.

Підтримуються конструкції для управління потоком виконання, такі як умови (if, else), цикли (for, while) та інші.

WGSL використовується для написання шейдерів, які можна використовувати з API Vulkan через компіляцію у формат SPIR-V. Шейдери можуть бути створені для різних типів обчислень, таких як вершинні шейдери, фрагментні шейдери, а також обчислювальні шейдери, які ми і будемо використовувати.

Оскільки Vulkan використовує SPIR-V як проміжне представлення шейдерів, WGSL-шейдери потрібно компілювати у SPIR-V. Для цього можна використовувати інструменти, такі як Naga, яка підтримує компіляцію WGSL в SPIR-V.

Після компіляції WGSL-шейдера у SPIR-V, ми зможемо використовувати їх для нашої нейронної мережі.

3.6 Написання шейдерів для слоїв нейронної мережі

Розглянемо основні структури та кроки, використані в наведеному коді для створення комп'ютерного пайплайна в Vulkan.

Початку створюємо Shader Module з скомпільованого SPIR-V коду:

```
use ash::vk;

pub fn create_shader_module(device: &ash::Device, code: &[u8]) -> Result<vk::ShaderModule, vk::Result> {
    let create_info = vk::ShaderModuleCreateInfo::builder()
        .s_type(vk::StructureType::SHADER_MODULE_CREATE_INFO)
        .code_size(code.len() as _)
        .p_code(code.as_ptr() as *const u32)
        .build();

    let shader_module = unsafe {
        device.create_shader_module(&create_info, None)
    }?;

    Ok(shader_module)
}
```

VkShaderModuleCreateInfo: Ця структура використовується для опису параметрів створення shader module. Вона містить тип структури, розмір коду, та вказівник на код.

device.create_shader_module: Функція, яка створює shader module на основі наданої інформації. Якщо створення не вдалося, Повертається помилка.

```
use std::fs::File;
use std::io::{Error, Read};
use std::path::Path;
```

```
pub fn load_shader_code(file_path: &Path) -> Result<Vec<u8>, Error>
{
    let mut file = File::open(file_path)?;
    let mut code = Vec::new();
    file.read_to_end(&mut code)?;
    Ok(code)
}

fn main() -> Result<(), vk::Result> {
    // ... (Vulkan initialization, device creation, etc.)

    // Load shader code from file
    let shader_file_path = Path::new("path/to/shader.spv"); // Replace with
actual path
    let shader_code = load_shader_code(shader_file_path)?;

    // Create shader module
    let compute_shader_module = create_shader_module(&device,
&shader_code)?;

    // ... (Use the created shader module in the Vulkan pipeline)

    // Destroy shader module when no longer needed
    unsafe { device.destroy_shader_module(compute_shader_module,
None) }

    Ok(())
}
```

shaderCode: Це вектор, який містить завантажений SPIR-V код шейдера.

createShaderModule: Викликається функція, яка створює shader module з наданого коду.

```
use ash::vk;

let shader_stage_info = vk::PipelineShaderStageCreateInfo::builder()
    .s_type(vk::StructureType::PIPELINE_SHADER_STAGE_CREATE_
INFO)
    .stage(vk::ShaderStageFlagBits::COMPUTE)
    .module(computeShaderModule)
    .name("main")
    .build();
```

VkPipelineShaderStageCreateInfo: Ця структура описує стадію шейдера в пайплайні. Вона містить тип структури, стадію шейдера , модуль шейдера, та назву входної точки.

```
use ash::vk;

let pipeline_create_info = vk::ComputePipelineCreateInfo::builder()
    .s_type(vk::StructureType::COMPUTE_PIPELINE_CREATE_INFO)
    .stage(shaderStageInfo)
    .layout(pipelineLayout)
    .build();

let compute_pipelines = unsafe {
    device.create_compute_pipelines(vk::PipelineCache::null(), 1,
&[pipeline_create_info], None)
}?;
```

```
let compute_pipeline = compute_pipelines[0];
```

VkComputePipelineCreateInfo: Ця структура містить всю інформацію, необхідну для створення compute pipeline. Вона включає тип структури, стадію шейдера, та макет пайплайна.

vkCreateComputePipelines: Функція, яка створює compute pipeline на основі наданої інформації. Якщо створення не вдалося, викликається виключення.

```
use ash::vk;

let alloc_info = vk::CommandBufferAllocateInfo::builder()
    .s_type(vk::StructureType::COMMAND_BUFFER_ALLOCATE_IN
FO)
    .level(vk::CommandBufferLevel::PRIMARY)
    .command_pool(command_pool)
    .command_buffer_count(1)
    .build();

let command_buffers = unsafe
{ device.allocate_command_buffers(&alloc_info)? };
let command_buffer = command_buffers[0];

let begin_info = vk::CommandBufferBeginInfo::builder()
    .s_type(vk::StructureType::COMMAND_BUFFER_BEGIN_INFO)
    .flags(vk::CommandBufferUsageFlags::ONE_TIME_SUBMIT)
    .build();

if unsafe { vkBeginCommandBuffer(command_buffer, &begin_info) !=
VK_SUCCESS } {
```

```
        panic!("failed to begin recording command buffer!");
    }

    // ... (Record your command buffer commands here)

    unsafe { vkEndCommandBuffer(command_buffer) };
    Створення і початок командного буфера
    use ash::vk;

    // ... (Assuming you have the commandBuffer, computePipeline,
pipelineLayout, descriptorSet, imgWidth, and imgHeight available)

    // Bind compute pipeline
    unsafe {
        vkCmdBindPipeline(commandBuffer,
vk::PipelineBindPoint::COMPUTE, computePipeline);
    }

    // Bind descriptor sets
    unsafe {
        vkCmdBindDescriptorSets(
            commandBuffer,
            vk::PipelineBindPoint::COMPUTE,
            pipelineLayout,
            0,
            1,
            &descriptorSet,
            0,
            std::ptr::null(),
```

```
        );
    }

    // Calculate workgroup sizes
    let workgroup_x = (imgWidth as f32 / 16.0).ceil() as u32;
    let workgroup_y = (imgHeight as f32 / 16.0).ceil() as u32;

    // Dispatch workgroups
    unsafe {
        vkCmdDispatch(commandBuffer, workgroup_x, workgroup_y, 1);
    }

    // End command buffer recording
    if unsafe { vkEndCommandBuffer(commandBuffer) !=
VK_SUCCESS } {
        panic!("Failed to record command buffer!");
    }
    Запис команд у командний буфер

    let submit_info = vk::SubmitInfo::builder()
        .s_type(vk::StructureType::SUBMIT_INFO)
        .command_buffer_count(1)
        .p_command_buffers(&commandBuffer)
        .build();

    let result = unsafe { vkQueueSubmit(computeQueue, 1, &submit_info,
vk::Fence::null()) };

    if result != vk::Result::SUCCESS {
```

```
    panic!("Failed to submit command buffer: {:?}", result);
}

unsafe { vkQueueWaitIdle(computeQueue) };
VkQueue computeQueue;
vkGetDeviceQueue(device, computeQueueFamilyIndex, 0,
&computeQueue);
```

VkSubmitInfo: Структура, яка описує інформацію для відправки командного буфера у чергу. Включає тип структури (sType), кількість командних буферів (commandBufferCount), та вказівник на командні буфери (pCommandBuffers).

vkQueueSubmit: Відправляє командний буфер у вказану чергу для виконання.

vkQueueWaitIdle: Очікує завершення виконання всіх команд у черзі.

3.7 Шейдерний код

```
[[group(0), binding(0)]] var<storage, read> bottom_blob:
array<vec4<f16>>;
[[group(0), binding(1)]] var<storage, write> top_blob:
array<vec4<f16>>;
[[group(0), binding(2)]] var<storage, read> weight_blob:
array<vec4<f16>>;
[[group(0), binding(3)]] var<storage, read> bias_blob:
array<vec4<f16>>;
```

bottom_blob: Вхідні дані для обчислень.

top_blob: Вихідні дані після обчислень.

weight_blob: Вагові коефіцієнти для обчислень.

bias_blob: Зсуви для обчислень.

```
struct Parameters {
    w: i32,
    h: i32,
    c: i32,
    cstep: i32,
    outw: i32,
    outh: i32,
    outc: i32,
    outcstep: i32,
    kernel_w: i32,
    kernel_h: i32,
    dilation_w: i32,
    dilation_h: i32,
    stride_w: i32,
    stride_h: i32,
    bias_term: i32,
    activation_type: i32,
    activation_param_0: f32,
    activation_param_1: f32,
};
```

w, h, c: Розміри вхідного тензора.

cstep: Крок по каналах.

outw, outh, outc: Розміри вихідного тензора.

outcstep: Крок по каналах вихідного тензора.

kernel_w, kernel_h: Розміри ядра згортки.

dilation_w, dilation_h: Коефіцієнти дилатації для ядра.

stride_w, stride_h: Кроки згортки.

bias_term: Наявність зсуву (1 - є, 0 - немає).

activation_type: Тип активації (наприклад, 1 для ReLU).
activation_param_0, activation_param_1: Додаткові параметри для функції активації.
[[group(0), binding(4)]] var<uniform> params: Parameters;
Ця прив'язка забезпечує доступ до параметрів з структури Parameters.

```
[[stage(compute), workgroup_size(4, 1, 1)]]
fn main([[builtin(global_invocation_id)]] global_id: vec3<u32>) {
    let gx = i32(global_id.x) * 4;
    let gy = i32(global_id.y);

    let outsize = params.outw * params.outh;

    if (gx >= outsize || gy >= params.outc) {
        return;
    }

    var sum0: vec4<f16> = vec4<f16>(0.0);
    var sum1: vec4<f16> = vec4<f16>(0.0);
    var sum2: vec4<f16> = vec4<f16>(0.0);
    var sum3: vec4<f16> = vec4<f16>(0.0);

    if (params.bias_term == 1) {
        sum0 = bias_blob[gy];
        sum1 = sum0;
        sum2 = sum0;
        sum3 = sum0;
    }
```

```
    let maxk = params.kernel_w * params.kernel_h;
    let N = params.c * maxk;
    let gx4 = vec4<i32>(gx, gx + 1, gx + 2, gx + 3);
    let sy4 = gx4 / params.outw;
    let sx4 = gx4 % params.outw;
    let sxs4 = sx4 * params.stride_w;
    let sys4 = sy4 * params.stride_h;

    for (var z = 0; z < N; z = z + 1) {
        let sz = z / maxk;
        let kk = z % maxk;
        let ky = kk / params.kernel_w;
        let kx = kk % params.kernel_w;

        let x4 = sxs4 + kx * params.dilation_w;
        let y4 = sys4 + ky * params.dilation_h;

        let v0 = bottom_blob[sz * params.cstep + (sys4.x + ky *
params.dilation_h) * params.w + sxs4.x + kx * params.dilation_w];
        let v1 = bottom_blob[sz * params.cstep + (sys4.y + ky *
params.dilation_h) * params.w + sxs4.y + kx * params.dilation_w];
        let v2 = bottom_blob[sz * params.cstep + (sys4.z + ky *
params.dilation_h) * params.w + sxs4.z + kx * params.dilation_w];
        let v3 = bottom_blob[sz * params.cstep + (sys4.w + ky *
params.dilation_h) * params.w + sxs4.w + kx * params.dilation_w];

        let k0 = weight_blob[z * 8 + 0];
        let k1 = weight_blob[z * 8 + 1];
```

```
        let k2 = weight_blob[z * 8 + 2];
        let k3 = weight_blob[z * 8 + 3];
        let k4 = weight_blob[z * 8 + 4];
        let k5 = weight_blob[z * 8 + 5];
        let k6 = weight_blob[z * 8 + 6];
        let k7 = weight_blob[z * 8 + 7];

        sum0 = sum0 + v0 * k0 + v0 * k1 + v0 * k2 + v0 * k3 + v0 * k4 + v0 *
k5 + v0 * k6 + v0 * k7;
        sum1 = sum1 + v1 * k0 + v1 * k1 + v1 * k2 + v1 * k3 + v1 * k4 + v1 *
k5 + v1 * k6 + v1 * k7;
        sum2 = sum2 + v2 * k0 + v2 * k1 + v2 * k2 + v2 * k3 + v2 * k4 + v2 *
k5 + v2 * k6 + v2 * k7;
        sum3 = sum3 + v3 * k0 + v3 * k1 + v3 * k2 + v3 * k3 + v3 * k4 + v3 *
k5 + v3 * k6 + v3 * k7;
    }

    // Activation function (if needed)
    if (params.activation_type == 1) {
        // Example activation (ReLU)
        sum0 = max(sum0, vec4<f16>(0.0));
        sum1 = max(sum1, vec4<f16>(0.0));
        sum2 = max(sum2, vec4<f16>(0.0));
        sum3 = max(sum3, vec4<f16>(0.0));
    }

    let gi = gy * params.outcstep + gx;
    top_blob[gi + 0] = sum0;
    if (gx + 1 < outsize) { top_blob[gi + 1] = sum1; }
```

```
    if (gx + 2 < outsize) { top_blob[gi + 2] = sum2; }
    if (gx + 3 < outsize) { top_blob[gi + 3] = sum3; }
}
```

Основні моменти

Параметри глобального виклику: [[builtin(global_invocation_id)]] global_id: vec3<u32>: Ідентифікатор глобального виклику обчислень, який визначає робочу групу.

Перевірка меж:

```
if (gx >= outsize || gy >= params.outc) {
    return;
}
```

Якщо координати виклику виходять за межі обчислень, то функція завершується.

Ініціалізація сум:

```
var sum0: vec4<f16> = vec4<f16>(0.0);
var sum1: vec4<f16> = vec4<f16>(0.0);
var sum2: vec4<f16> = vec4<f16>(0.0);
var sum3: vec4<f16> = vec4<f16>(0.0);

if (params.bias_term == 1) {
    sum0 = bias_blob[gy];
    sum1 = sum0;
    sum2 = sum0;
    sum3 = sum0;
}
```

Ініціалізуються змінні для збереження суми, а якщо зсув присутній, то додається до початкових значень.

Цикл обчислень:

```
for (var z = 0; z < N; z = z + 1) {
    ...
    sum0 = sum0 + v0 * k0 + v0 * k1 + v0 * k2 + v0 * k3 + v0 * k4 + v0 * k5
+ v0 * k6 + v0 * k7;
    sum1 = sum1 + v1 * k0 + v1 * k1 + v1 * k2 + v1 * k3 + v1 * k4 + v1 * k5
+ v1 * k6 + v1 * k7;
    sum2 = sum2 + v2 * k0 + v2 * k1 + v2 * k2 + v2 * k3 + v2 * k4 + v2 * k5
+ v2 * k6 + v2 * k7;
    sum3 = sum3 + v3 * k0 + v3 * k1 + v3 * k2 + v3 * k3 + v3 * k4 + v3 * k5
+ v3 * k6 + v3 * k7;
}
```

Виконується цикл по всім каналах і елементам ядра, обчислюючи суму для кожного вихідного елемента.

Активація:

```
if (params.activation_type == 1) {
    // Example activation (ReLU)
    sum0 = max(sum0, vec4<f16>(0.0));
    sum1 = max(sum1, vec4<f16>(0.0));
    sum2 = max(sum2, vec4<f16>(0.0));
    sum3 = max(sum3, vec4<f16>(0.0));
}
```

Якщо задано тип активації, то застосовується функція активації (в даному випадку ReLU).

Запис результатів:

```
        let gi = gy * params.outcstep + gx;
        top_blob[gi + 0] = sum0;
```

```
    if (gx + 1 < outsize) { top_blob[gi + 1] = sum1; }
    if (gx + 2 < outsize) { top_blob[gi + 2] = sum2; }
    if (gx + 3 < outsize) { top_blob[gi + 3] = sum3; }
```

Записуються результати обчислень у вихідний буфер.

Цей код представляє типову реалізацію обчислювального шейдера, що виконує згортку з певними параметрами, використовуючи обчислювальні можливості Vulkan.

Шейдер обчислює максимальне значення в кожному вікні пулінгу.
Він ітерує по вікну пулінгу, визначеному kernel_size і stride.
Зберігає знайдене максимальне значення у вихідне зображення.

```
@group(0) @binding(0) var<storage, read> input_image: texture_2d<f32>;
@group(0) @binding(1) var<storage, write> output_image:
texture_storage_2d<rgba8unorm, write>;

struct Parameters {
    kernel_size: i32,
    stride: i32,
    input_width: i32,
    input_height: i32,
    output_width: i32,
    output_height: i32,
};
@group(0) @binding(2) var<uniform> params: Parameters;

@compute @workgroup_size(8, 8)
fn main(@builtin(global_invocation_id) global_id: vec3<u32>) {
    let gx = i32(global_id.x);
    let gy = i32(global_id.y);
```

```
    if (gx >= params.output_width || gy >= params.output_height) {
        return;
    }

    var max_value: f32 = -f32.infinity;

    for (var ky = 0; ky < params.kernel_size; ky++) {
        for (var kx = 0; kx < params.kernel_size; kx++) {
            let in_x = gx * params.stride + kx;
            let in_y = gy * params.stride + ky;

            if (in_x < params.input_width && in_y < params.input_height) {
                let value = textureLoad(input_image, vec2<i32>(in_x, in_y), 0).r;
                if (value > max_value) {
                    max_value = value;
                }
            }
        }
    }

    textureStore(output_image, vec2<i32>(gx, gy), vec4<f32>(max_value,
max_value, max_value, 1.0));
}
```

Прив'язки і Уніформи:

input_image: Вхідне зображення (2D текстура).

output_image: Вихідне зображення, куди будуть збережені результати.

params: Блок уніформів, що містить параметри для операції пулінгу.

Параметри:

kernel_size: Розмір ядра пулінгу (наприклад, 2 для 2x2 пулінгу).

stride: Крок операції пулінгу.

input_width, input_height: Розміри вхідного зображення.

output_width, output_height: Розміри вихідного зображення.

ВИСНОВКИ

У даній роботі було досліджено та реалізовано прискорення згорткових нейронних мереж (CNN) на графічному процесорі (GPU) за допомогою API Vulkan. Основною метою було виявлення можливостей та переваг цього підходу порівняно з іншими технологіями, такими як CPU, CUDA та OpenCL, у контексті оптимізації та прискорення обчислень нейронних мереж.

Під час дослідження було розглянуто наступні аспекти:

Було реалізовано згорткові шари, шар пулінгу та інші основні компоненти CNN з використанням API Vulkan. Це дозволило ефективно використовувати потужності графічного процесора для обчислень.

Використання Vulkan дозволило оптимізувати обчислення на GPU, використовуючи паралельність та гнучкість шейдерів для виконання складних обчислень.

Було проведено порівняння з використанням CPU, CUDA та OpenCL. Виявлено, що використання Vulkan може забезпечити значне прискорення обчислень у порівнянні з CPU, особливо при обробці великих обсягів даних.

Реалізована система показала хорошу масштабованість і ефективність при роботі з різними розмірами даних та конфігураціями мереж.

Отже використання Vulkan для прискорення CNN на GPU є ефективним та перспективним кросплатформенним підходом.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. https://registry.khronos.org/vulkan/specs/1.3/html/index.html — vulkan specification.
2. https://www.lei.chat/posts/what-is-vulkan-compute/ - What is Vulkan Compute?
3. A. Krizhevsky, I. Sutskever, and G. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." Advances in Neural Information Processing Systems, 2012.
4. https://www.nature.com/articles/s41597-022-01721-8 — MedMNIST dataset.
5. https://d2l.ai/ - Dive into Deep Learning. Interactive deep learning book with code, math, and discussions.
6. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9044238/ - Effect of neural network structure in accelerating performance and accuracy of a convolutional neural network with GPU/TPU for image analytics.
7. https://www.khronos.org/assets/uploads/developers/presentations/Vulkan-02-ML-SIGGRAPH-Jul19.pdf - Vulkan ML.
8. https://www.khronos.org/assets/uploads/developers/presentations/Tencent-ncnn-Universal-Neural-Network-Inference-with-Vulkan_Apr21.pdf - ncnn - universal neural network inference with vulkan.
9. Practical Convolutional Neural Networks By Mohit Sewak , Md. Rezaul Karim , Pradeep Pujari.
10. https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2353511/13656_FULLTEXT.pdf - Hardware Acceleration of Convolutional Neural Networks.