Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи прискорення периферійних обчислень на основі FPGA

(тема)

Виконав:
студент II курсу, групи СПм-21–2
Ільяшов О.А.
(прізвище, ініціали)

Спеціальність
123 – Комп'ютерна інженерія
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо–професійна або освітньо–наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: проф. Коваленко А.А.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 – Комп'ютерна інженерія
(код і повна назва)

Тип програми освітньо-наукова
(освітньо–професійна або освітньо–наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(Підпис)
"____" _____ 2023 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Ільяшову Олександру Андрійовичу
(Прізвище, ім'я, по батькові)

1. Тема роботи Методи прискорення периферійних обчислень на основі FPGA.

затверджена наказом по університету від " 03 " квітня 2023 р. № 318 Ст

2. Термін подання студентом роботи 17 травня 2023 р.

3. Вхідні дані до роботи
Внутрішня архітектура типової ПЛІС (FPGA), мова паралельного програмування OPENCL, існуючі методи апаратного прискорення на основі FPGA, основи контурного прискорення в гетерогенних системах, парадигми периферійних обчислень, апаратне забезпечення FPGA прискорювачів, фрейморк UltraShare Express software.

4. Перелік питань, що потрібно опрацювати в роботі
Вступ.
Аналіз літератури та постановка мети й задач дослідження.
Дослідження архітектури та засобів для FPGA прискорень.
Аналіз функціонування багаточергового FPGA – прискорювача.
Апаратне прискорення на FPGA з використанням OPENCL.
Висновки. Додаток.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів)

Слайд-презентація - 16 слайдів.

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначку консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ п./ п.	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз літератури за темою роботи	03.04.23–10.04.23	
2	Постановка мети та задач дослідження	11.04.23–12.04.23	
3	Багаточерговий FPGA – прискорювач	13.04.23–20.04.23	
4	Апаратне прискорення на FPGA	21.04.23–30.04.23	
5	з використанням OPENCL		
6	Експериментальна частина	01.05.23–09.05.23	
7	Аналіз результатів	10.05.23–11.05.23	
8	Подання кваліфікаційної роботи керівникові та	12.05.23–14.05.23	
9	її попередній захист		
10	Подання кваліфікаційної роботи на	15.05.23–16.05.23	
11	рецензування		

Дата видачі завдання 03.04. 2023 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Коваленко А.А.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 106 с., 29 рис., 11 табл., 19 джерел.

АПАРАТНЕ ПРИСКОРЕННЯ, FPGA, ПЕРИФЕРІЙНІ ОБЧИСЛЕННЯ, EDGE COMPUTING, IOT, OPENCL, CPU, GPU, MULTI-FPGA ПРИСКОРЕННЯ, ULTRASHARE EXPRESS FRAMEWORK

Мета кваліфікаційної роботи вивчення сучасних методів та інструментів прискорення периферійних обчислень з використанням FPGA.

У ході виконання кваліфікаційної роботи були досліджені переваги та недоліки існуючих методів для прискорення периферійних обчислень з використанням FPGA, особливості їх застосування у різних сферах. Розроблено модифікований метод прискорення периферійних робочих навантажень з використанням FPGA. Також були створені обчислювально інтенсивні програми в OpenCL й відображені на гетерогенній SoC-FPGA. Запропонована структуру на основі однієї команди під назвою MQMAI, яка використовує архітектуру з кількома чергами в програмному стеку для мінімізації конфліктів між різними програмами для доступу до різних прискорювачів FPGA. Впроваджені додатки до платформ інтеграції прискорювачів, які дозволяють спільно використовувати кілька прискорювачів FPGA між різними програмами користувача щодо вимог у хмарах і центрах обробки даних, також сформульовано задачі для подальшого дослідження.

ABSTRACT

Master's thesis: 106 pages, 29 figures, 11 tables, 19 sources.

HARDWARE ACCELERATION, FPGA, PERIPHERAL COMPUTING, EDGE COMPUTING, IOT, OPENCL, CPU, GPU, MULTI-FPGA ACCELERATION, ULTRASHARE EXPRESS FRAMEWORK

The goal of qualifying work is to study modern methods and tools for accelerating peripheral computing using FPGA.

During the qualifying work the advantages and disadvantages of existing methods for speeding up peripheral calculations using FPGA, features of their application in various fields were investigated. A modified method of accelerating peripheral workloads using FPGA has been developed. Computationally intensive programs were also created in OpenCL and displayed on a heterogeneous SoC-FPGA. A framework based on a single command called MQMAI is proposed, which uses a multi-queue architecture in the software stack to minimize conflicts between different programs accessing different FPGA accelerators. Implemented applications to accelerator integration platforms that enable the sharing of multiple FPGA accelerators between different user applications for cloud and data center requirements, and issues for further research are formulated

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CPU - Central Processing Unit

FPGA - Field-Programmable Gate Array

IoT – Internet of Things

HDL - Hardware Description Language

RTL - Register-Transfer Level

MTSA - Multi-Thread Single-Accelerator

MTMA - Multi-Thread Multi-Accelerator

CLB - Configurable Logic Blocks

HLS - High level Synthesis

IC - Integrated Circuit

AI - Artificial Intelligence

FaaS - FPGA-as-a-Service

RIFFA - Reusable Integration Framework for FPGA Accelerators

MQMAI - Multi-Queue Multi-Accelerator Interface

GPU - Graphics Processing Unit

BLE - Basic Logic Element

RAM - Random-Access Memory

BRAM - Block RAM

SoC - System on Chip

FIFO - First-In First-Out

DSP - Digital Signal Processing

PE - Processing Element

MIMD - Multiple-Instruction Multiple-Data

IDC - International Data Corporation

PLA - Programmable Logic Array

ВСТУП

Згідно зі звітом міжнародної корпорації даних (International Data Corporation (IDC)), очікується, що кількість пристроїв Інтернету речей (IoT) перевищить 41,6 мільярда протягом наступних п’яти років, і вони генеруватимуть близько 79,4 зетабайт даних [2]. Завдяки нещодавньому прогресу в комунікаційних технологіях відбулося величезне покращення в передачі даних у всьому світі. З появою технологій високочастотної передачі даних, таких як 5G, оркестровка та координація кінцевих пристроїв, пристроїв обробки даних і систем зберігання в усьому світі стала життєздатною в IoT [6]. IoT надає інноваційні рішення для різних питань і проблем, поєднуючи розумні та інтелектуальні пристрої та датчики. Технологія IoT призвела до появи та зростання хмарних обчислень [5]. Хмарні обчислення відіграють роль співпраці в IoT і використовуються для зберігання даних IoT і обробки даних, коли це потрібно. Хмарні обчислення є поточною тенденцією в архітектурі додатків наступного покоління.

Високий попит на обробку даних, необхідна енергія та вартість призводять до значного зростання хмарних і периферійних обчислень, щоб перенести більшу частину обробки даних на спільні потужні ресурси [8]. Іншим запропонованим рішенням для вирішення дорогого процесу переміщення даних є техніка обробки в сховищі, яка переміщує обробку даних у центри обробки даних і пристрої зберігання замість переміщення даних на робочі столи користувачів для обробки [4]. В останні кілька років також було введено концепцію периферійних обчислень, щоб уникнути надсилання величезної кількості даних на віддалені сервери в хмарах. Граничні обчислення визначаються як розподілена обчислювальна топологія, у якій обробка інформації розташована поблизу того місця, де речі та люди виробляють або споживають цю інформацію. Однак усі ці рішення покладаються на потужні процесори, здатні обслуговувати різних користувачів. Незалежно від вибраного ме-

тоду, існує загальний попит на потужні процесори в хмарах, периферійних середовищах і центрах обробки даних, щоб відповідати вимогам продуктивності обробки даних.

На відміну від графічних процесорів (GPU), FPGA пропонують гетерогенне середовище для різних типів прискорювачів. Ця унікальна функція дозволяє ПЛІС одночасно обслуговувати різні додатки та робить їх придатними для задоволення вимог різноманітних додатків у хмарах, периферійних системах і центрах обробки даних. Однак через обмежену кількість ресурсів і трудомісткий процес реконструкції виникає важливість спільного використання прискорювачів FPGA між різними програмами. Поточні найсучасніші схеми управління прискорювачем цього не роблять але забезпечте спільне використання прискорювача між кількома програмами одночасно. Щоб досягти цієї мети, нам потрібна підтримка системного програмного забезпечення, а також апаратний контролер для забезпечення безперебійного спільного використання прискорювача.

Високий попит на забезпечення необхідної обчислювальної потужності сучасних додатків із великим об’ємом даних і інтенсивними обчисленнями, а також вартість потужних процесорів призвели до того, що кінцеві пристрої перенесли більшу частину обробки даних у хмари, межі та центри обробки даних. Високий попит на забезпечення необхідної обчислювальної потужності сучасних додатків із великим об’ємом даних і інтенсивними обчисленнями, а також вартість потужних процесорів призвели до того, що кінцеві пристрої перенесли більшу частину обробки даних у хмари, межі та центри обробки даних. У хмарах, на периферії та в центрах обробки даних апаратні прискорювачі використовуються для значного збільшення обчислювальної потужності багатоядерних процесорів.

Серед апаратних прискорювачів значну увагу привернули програмовані вентильні матриці (FPGA) завдяки своїй видатній продуктивності та енергоефективності. Є дві основні проблеми при використанні FPGA як апаратної платформи для прискорення: як спроектувати ефективні прискорювачі, щоб

отримати високу продуктивність, і як ефективно надати програмним програмам доступ до прискорювачів FPGA. У нашій роботі визначимо останню проблему для проведення досліджень.

1 АНАЛІЗ ЛІТЕРАТУРИ ТА ПОСТАНОВКА МЕТИ Й ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Основні поняття, прискорення на основі FPGA

На відміну від графічних процесорів (GPU), FPGA пропонують гетерогенне середовище для різних типів прискорювачів. Ця унікальна функція дозволяє ПЛІС одночасно обслуговувати різні додатки та робить їх придатними для задоволення вимог різноманітних додатків у хмарах, периферійних системах і центрах обробки даних. Однак через обмежену кількість ресурсів і трудомісткий процес реконструкції виникає важливість спільного використання прискорювачів FPGA між різними програмами. Поточні найсучасніші схеми управління прискорювачем цього не роблять але забезпечте спільне використання прискорювача між кількома програмами одночасно.

Завжди існував пошук продуктивного та енергоефективного обладнання для задоволення зростаючих потреб обробки. Це розвинуло тенденції в апаратних архітектурах відповідно до закону Мура [15] зі збільшенням числа транзисторів щільність і частота. Тим не менш, з порушенням принципу масштабування Деннарда в середині 2000-х років неможливо збільшити частоти через обмеження потужності [8]. Це призвело до зміни парадигми в комп’ютерній архітектурі, де розробляються такі альтернативи, як багатоядерні процесори, графічні процесори (GPU) і гетерогенні системи на процесорі з виділеними апаратними прискорювачами. Сучасні програми передбачають обчислення експоненціального обсягу даних за менший час, а також системи реального часу вимагають відповідей із надзвичайно низькою затримкою. Приклади включають штучний інтелект (AI), когнітивні обчислення та хмарні обчислення. Таким чином, необхідний пошук систем апаратного прискорення та енергоефективності.

На даний момент найбільш привабливими платформами для апаратно-

го прискорення в хмарних і периферійних обчисленнях є обробка в сховищах, спеціальні інтегральні схеми (ASIC), графічні процесори (GPU) і програмовані вентильні матриці (FPGA). Дослідники в наукових колах і промисловості націлилися на широкий спектр програм, щоб прискорити їх використання апаратні прискорювачі. Наприклад, Google і IBM досліджують власну програму для штучного інтелекту (AI) і глибокого навчання з дизайном ASIC [3]. Facebook використовує прискорювачі ASIC для висновків ШІ, навчання ШІ та перекодування відео NVIDIA представила величезний асортимент оптимізованих прискорювачів, включаючи науку про дані та аналітику, штучний інтелект, медіа та розваги, медичну візуалізацію, безпеку та багато іншого для графічних процесорів, які будуть використовуватися її користувачами. Дослідники з Університету Іллінойсу в Урбана-Шампейн працюють над прискоренням додатків молекулярного моделювання на GPU [9]. На ПЛІС алгоритми включають як навчання, так і фази висновку глибокого навчання [7], обробка зображення та алгоритми зору, Hadoop MapReduce потокові програми, аналітика даних, фінансова аналітика, і багато інших застосувань були досліджені.

З наближенням до кінця ери масштабування Деннарда та багатоядерних процесорів [3-5] дослідники шукали нові методи для вирішення цієї вимогливої обчислювальної потужності. Серед усіх рішень розгортання гетерогенних архітектур показало багатообіцяюче підвищення продуктивності [9-11]. Гетерогенна архітектура складається з багатоядерних процесорів, які супроводжуються апаратними прискорювачами для виконання інтенсивних обчислювальних ядер різних типів програм. Ці апаратні прискорювачі в основному є спеціально розробленими схемами для конкретних операцій, які забезпечують набагато швидший час виконання порівняно з багатоядерними процесорами [6-8]. На рисунку 1.1 зображено деякі пристрої IoT та їхні вимоги до серверів і центрів обробки даних у хмарі, щоб отримати вигоду від їхніх потужних обчислювальних можливостей, які розширюються завдяки розгортанню апаратних прискорювачів.

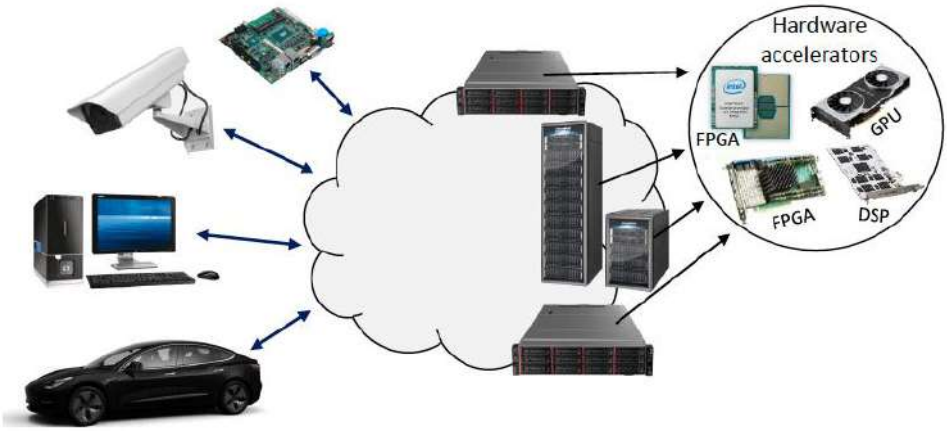


Рисунок 1.1 - FPGA, що використовуються для прискорення у хмарі

Рисунок 1.2 порівнює різні аспекти платформ апаратного прискорення. ASIC - це інтегральна мікросхема, призначена для спеціального застосування, а не для загального використання. Щоб розробити та виготовити дизайн ASIC, дизайнери часто використовують мову опису обладнання (HDL), таку як Verilog або VHDL, для опису функціональності ASIC. Конструкції ASIC дуже ефективні для прискорення конкретних програм. Конструкції ASIC дуже потужні та ефективні, однак вони повністю налаштовані під конкретні умови операцій і не може використовуватися для інших цілей і застосувань.

Графічний процесор — це процесор, спеціально розроблений для виконання інтенсивного рендерингу 3D-графіки. У той час як процесори складаються з кількох ядер (від чотирьох до восьми), графічні процесори складаються із сотень набагато менших ядер. Графічні процесори мають архітектуру Single Instruction Multiple Data (SIMD) і забезпечують масовий паралелізм для досягнення високої продуктивності обчислень [4]. Графічні процесори підходять для паралельних завдань з передачі даних, інакше вони повинні виконуватися послідовно і дуже неефективно.

	CPU	GPU	FPGA	ASIC
Overview	Sequential processor for general-purpose applications	Originally designed for graphics; now used in a wide range of computationally intensive applications	Flexible collection of logic elements and IP blocks that can be configured and changed in the field	Custom integrated circuit optimized for the end application
Processing	Single- and multi-core processor, plus specialized blocks: FPU, etc.	Thousands of identical processor cores	Configured for application	Application-specific: may include third-party IP cores
Programming	huge range of high-level languages (e.g., C, C++, Python, Java, Fortran); assembly language	OpenCL & Nvidia's CUDA API allow general-purpose programming (e.g., C, C++, Python, Java, Fortran)	Traditionally HDL (Verilog, VHDL); newer systems include C/C++ via OpenCL & SDAccel	---
Strengths	Versatility, multitasking, ease of programming	Massive processing power for target applications: video processing, image analysis, signal processing; Suitable for data parallel tasks	Configurable for specific application; configuration can be changed after installation; high performance per watt; accommodates massively parallel operation;	Custom-designed for application with optimum combination of performance and power consumption
Weaknesses	OS capability adds high overhead; optimized for sequential processing with limited parallelism	High power consumption, not suited to some algorithms; problems must be reformulated to take advantage of parallelism; Sequential execution	Difficult to program; second-longest development time; poor performance for sequential operations; not good for floating-point operations	Longest development time; high cost; cannot be changed without redesigning the silicon

Рисунок 1.2 - Порівняння різних аспектів апаратних прискорювачів

Графічні процесори дуже програмовані, однак дуже енергоємні. Через свою архітектуру SIMD графічні процесори не можуть підтримувати різні прискорювачі та бути спільними для кількох орендарів одночасно [13]. Однак, на відміну від ASIC, FPGA є гнучкими та реконфігурованими, і на відміну від графічних процесорів, вони дуже енергоефективні, тоді як за допомогою незернистого паралелізму вони можуть значно покращити продуктивність. ПЛІС відносно на порядок енергоефективніші та мають вищу продуктивність на ват порівняно зі звичайними процесорами, центральними процесорами та графічними процесорами. Враховуючи обмеження GPU, впродовж останніх двох десятиліть FPGA привернули величезну увагу в гетерогенних архітектурах, щоб прискорити інтенсивні обчислювальні програми [2]. У хмарах і центрах обробки даних поєднання різноманітності додатків і важливості енергоспоживання привернуло увагу до FPGA для розгортання для прискорення. Ця робота присвячена прискорювачам на основі FPGA, а також інтеграції та спільному використанню їх між програмами користувача.

Обробка в сховищі дозволяє обробляти дані в системах зберігання замість того, щоб передавати їх до механізмів обробки та потім обробляти. Обробка в сховищі дозволяє довести процес до парадигми даних до його кінцевих меж, використовуючи механізми обробки всередині блоків зберігання для обробки даних [9]. Ця техніка може значно покращити та прискорити час обробки різних додатків (особливо додатків з великими даними) шляхом усунення вартості передачі даних і затримки [3-4].

Незважаючи на переваги обробки в сховищі, для цього потрібен новий набір пристроїв зберігання, які підтримують обробку в сховищі, і його не можна додати до доступних наразі комп'ютерів і серверів у хмарах і центрах обробки даних як розширення.

1.1.1 Контурне прискорення в гетерогенних системах

Багато програм можуть отримати вигоду від обчислень на апаратних прискорювачах, починаючи від хмарних обчислень і закінчуючи великими даними та периферійними обчисленнями. Приклади таких застосувань включають: аналіз великої кількості даних на платформах великих даних, навчання та запуск моделей штучного інтелекту (AI) і машинного навчання в хмарі, обробка потоків запитів і даних від пристроїв IoT, а також моделювання та імітація поведінки наукових програм.

Різноманітні прискорювачі легко доступні для програм, щоб вибрати для своїх потреб обчислень у хмарі. Графічні процесори (GPU) є найбільш широко використовуваними, їх можна легко знайти в багатьох високопродуктивних і хмарних системах. Інші типи прискорювачів також стають дедалі доступнішими, наприклад, блоки обробки тензорів (TPU) у хмарі Google і програмовані вентильні матриці (FPGA) у хмарі Amazon (вузли F1). Ці прискорювачі мають різні можливості та обмеження. Наприклад, FPGA можна переналаштувати для запуску будь-яких програм, але вони можуть забезпечувати лише низьку тактову частоту; Графічні процесори можна програмува-

ти за допомогою мов високого рівня для прискорення високопаралельних програм; і TPU спеціально розроблені для глибокого навчання.

Щоб вирішити вищезазначені проблеми, ми вивчаємо, як прискорювачі з різною апаратною архітектурою можуть прискорювати різні типи циклів, які є основними будівельними блоками майже кожної програми, що потребує інтенсивних обчислень. Ці додатки зазвичай складаються з одного або багатьох вкладених і контрольованих циклів. Ці петлі можуть втілюють різні шаблони з точки зору типів і ступенів залежності та паралельності, і їх можна знайти в багатьох програмах.

Наприклад, алгоритми динамічного програмування складаються з одного або кількох вкладених циклів, де кожна ітерація залежить від іншої ітерації, яка вказує по діагоналі в просторі ітерацій. Таким чином, абстрагування загальних шаблонів циклів із додатків і розуміння того, як вони працюють на різних апаратних прискорювачах, є важливими кроками до оптимального використання прискорювачів для виконання різних додатків.

1.1.2 Multi-FPGA прискорення

В останні роки FPGA отримали величезну увагу у світі прискорення нейронних мереж. FPGA можуть надати унікальні переваги для прискорення згорткових нейронних мереж (CNN). По-перше, FPGA можуть гарантувати жорсткі межі затримки для вхідних запитів. Звичайні прискорювачі CNN, тобто графічні процесори, показали здатність прискорювати пакет запитів, використовуючи свою ферму процесорних ядер. На відміну від графічних процесорів, FPGA можуть використовувати свій глибокий конвеєр, що реконструюється, для обслуговування запитів у потоковому режимі та забезпечення передбачуваної низької затримки. По-друге, звичайні процесори зазвичай енергоємні, що ускладнює їх розгортання в середовищах з обмеженим споживанням енергії. По-різному, ПЛІС мають високу енергоефективність через низьку тактову частоту. Підсумовуючи, FPGA вважаються чудовою

платформою для прискорення розгортання CNN.

Постійно зростаюча складність нових CNN вимагає FPGA з більшою кількістю ресурсів, таких як пропускна здатність пам’яті та логічні блоки, щоб досягти низької затримки та високої пропускної здатності. Навіть високоякісні технології мікросхем FPGA можуть розмістити лише невелику частину цілої моделі CNN. Наприклад, Intel Stratix 10 FPGA може виконувати лише 5000 операцій множення накопичення (MAC) за тактовий цикл, що навіть менше, ніж загальна кількість операцій для одного рівня типового CNN, такого як VGG-16 або ResNet. У результаті їм не вистачає роботи з більш важкими CNN для наднизької затримки (менше десяти мілісекунд) і високої пропускної здатності (більше 60 зображень/кадрів на секунду). Така проблема є навіть більш важливою для прискорення більш інтенсивних обчислювальних операцій, наприклад, тривимірних (3D) згорток, які демонструють великий потенціал у програмах обробки відео. Цю проблему потенційно можна вирішити, використовуючи кластер FPGA, з’єднаних через інфраструктуру зв’язку з високою пропускною здатністю.

Досягти лінійного прискорення за допомогою рішення з кількома FPGA непросто. По-перше, нам потрібно мати ефективний дизайн на одній ПЛІС і досягти найсучаснішої продуктивності. Такі переваги продуктивності повинні бути відображені в прискоренні різних операцій CNN. По-друге, слід правильно керувати конвеєром із кількох FPGA, щоб гарантувати, що всі FPGA виконують корисну роботу для обробки вхідних запитів. По-третє, розбивка CNN, яка є процесом відображення різних частин моделі на різні FPGA, має бути виконана розумно, щоб переконатися, що робоче навантаження збалансовано між FPGA.

1.2 Апаратне прискорення на основі FPGA

У 1984 році Xilinx представила першу програмовану вентильну матри-

цю (FPGA) [1-2]. FPGA — це реконфігурована інтегральна схема, яка може виконувати одну або декілька логічних операцій. Ці операції можуть бути такими ж простими, як логічний вентиль (функція І чи АБО), або однією чи кількома складними функціями, такими ж комплексними, як багатоядерний процесор. ПЛІС є реконфігурованими, що означає, що користувачі можуть змінити їх функціональність у будь-який час, завантаживши їхні бінарні файли, що називаються бітовими файлами в контексті ПЛІС. Щоб запрограмувати FPGA, конструкцію потрібно закодувати на рівні регістру-передачі (RTL). RTL — це абстракція дизайну, яка моделює цифрову схему за допомогою потоку цифрових сигналів між регістрами та логічними операціями. Є три кроки від проектування RTL до створення бітового файлу, які виконуються інструментами проектування постачальників, такими як пакет проектування Xilinx Vivado. Ці три етапи – синтез, реалізація та запис біта конфігурації [5].

ПЛІС підтримують прискорення обчислень, оскільки вони можуть забезпечити продуктивність і енергоефективність. Крім того, FPGA можна інтегрувати з іншими системами, такими як процесори, графічні процесори. Нещодавні дослідження доводять, що платформи з комбінацією FPGA і ЦП досягають кращої продуктивності, ніж одні ЦП під час певних робочих навантажень. Дослідження показали, що, перевантажуючи ЦП з обчислювально інтенсивних частин на FPGA, можна досягти значного прискорення [9-10]. Цю ідею також підхопили та реалізували провідні виробники FPGA.

У 2015 році Intel придбала Altera для заохочення продуктів FPGA на ринках центрів обробки даних та Інтернету речей. У 2010 році корпорація Майкрософт запустила проект Catapult - технологію прискорення гіпермасштабування, яка інтегрує FPGA в центри обробки даних Microsoft для прискорення роботи в мережі, безпека, хмарні сервіси та штучний інтелект [11]. У проекті Amazon EC2 F1 використовуються настроюванні конструкції FPGA для прискорення обчислень у Amazon Web Services (AWS). У 2017 році Baidu розгорнула служби прискорення додатків Xilinx FPGA у своїй публічній хмарі [3].

ПЛІС привернули величезну увагу завдяки своїм унікальним характеристикам, оскільки їх досліджували в хмарних і периферійних обчисленнях і центрах обробки даних.

Реконструкція під час виконання дозволяє ПЛІС прискорювати різноманітні додатки з інтенсивними обчисленнями [5-9]. Незважаючи на привабливі вищезазначені особливості FPGA, деякі проблеми ускладнюють їх легку експлуатацію. Розробка ефективного прискорювача FPGA є одним із викликів розгортання FPGA заради прискорення. Ефективна конструкція прискорювачів FPGA вимагає глибоких знань проектування RTL за допомогою таких мов апаратного програмування, як VHDL і Verilog. Зазвичай розробники додатків — це програмісти, які зовсім не знають або мають дуже мало знань про програмування апаратного забезпечення. Таким чином, зазвичай досліджувати прискорювачі FPGA дуже складно, і для цього потрібна команда програмістів програмного та апаратного забезпечення. Однак розвиток інструментів високорівневого синтезу (HLS), таких як Xilinx SDx HLS [9] і LegUp, було полегшено розробку прискорювачів FPGA.

Покладаючись на інструменти HLS для автоматичного генерування проектів RTL на основі шаблонів і на основі шаблонів, необов’язково призведе до оптимального дизайну для підвищення продуктивності FPGA. Коди C/C++, як правило, є послідовними, тоді як підвищення продуктивності FPGA досягається за рахунок паралелізму. Таким чином, ідентифікація потенційних частин, які потрібно розпаралелювати, є дуже важливою, але займає багато часу.

Іншою важливою проблемою використання прискорювачів FPGA є інтеграція прискорювачів у користувацькі додатки, що працюють на ядрах центрального процесора. Щоб хост-додаток викликав прискорювач на FPGA, потрібна інтеграційна структура, яка, по суті, виконує три основні завдання:

1) встановлення зв’язку;

2) відображення запитів на відповідні прискорювачі;

3) обробку переміщень даних між хостами та FPGA.

Перша вимога потребує інфраструктури для передачі інформації та даних керування між хостами та FPGA для встановлення підключення. Друга вимога може бути такою ж простою, як наявність лише одного прискорювача на FPGA, прив’язаного до однієї програми на хості, і пересилання всіх запитів до цього єдиного прискорювача. У такому простому сценарії для виклику прискорювачів буде розроблено набір функцій блокування, щоб уникнути суперечок щодо доступу до прискорювача. У третій вимозі, обробці переміщень даних, головною метою є забезпечення достатньої пропускної здатності передачі даних, щоб прискорювачі могли подавати стільки, скільки вони можуть обробити дані.

Існують деякі структури та платформи, які були запропоновані для полегшення виклику прискорювачів FPGA з хост-комп’ютерів і серверів. Кілька таких фреймворків в основному розроблено для отримання повного високорівневого опису дизайну як вхідних даних і створення всього пакету проектування, включаючи програмну частину, відому як головна програма, прискорювачі FPGA та інтерфейсну частину для надання зв'язок між головною програмою та прискорювачами FPGA. Такі промислові платформи, як Xilinx SDAccel і Intel SDK, забезпечують таке середовище [4]. Завдяки автоматизації всього процесу проектування/впровадження цей підхід знімає тягар проектування апаратного забезпечення з дизайнерів; однак у нього є три головні недоліки:

1) вся FPGA призначена одній програмі для прискорення деяких її інтенсивних обчислювальних ядер;

2) прискорювачі на FPGA не можуть використовуватися іншими програмами хосту;

3) для кожного дизайну, повинні бути сплачені часові (продуктивні) витрати на впровадження дизайну та конфігурації FPGA.

У цьому підході виділення всіх ресурсів FPGA для однієї програми призводить до значного недовикористання, а враховуючи той факт, що FPGA є дорогими пристроями, це призводить до значної неефективності.

Рисунок 1.3 показує типову блок-схему хост-сервера в хмарі, краю або центрах обробки даних із багатоядерним процесором і FPGA, підключеними до хост-сервера через PCIe (Peripheral Component Interconnect Express).

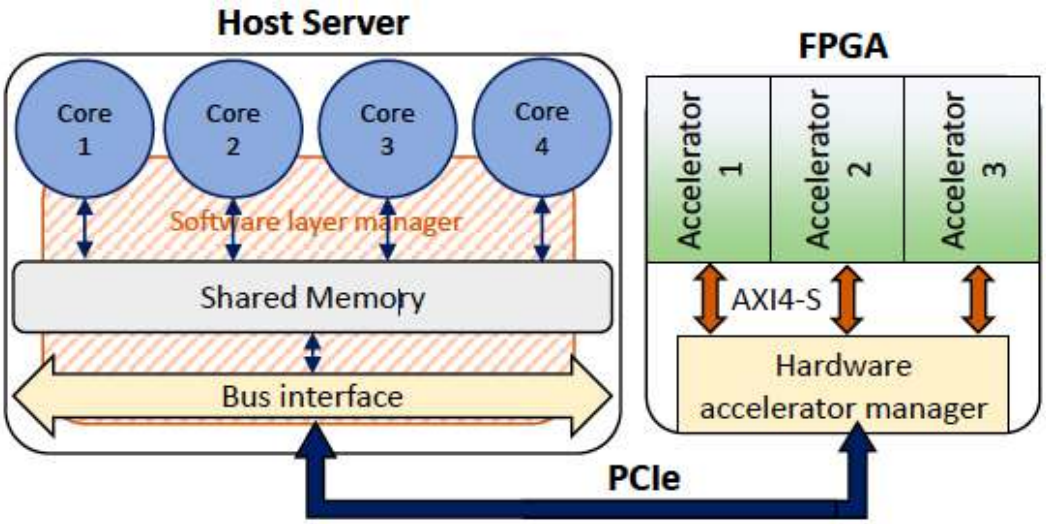


Рисунок 1.3 - Кілька прискорювачів FPGA використовуються для прискорення в хмарах і центрах обробки даних [2]

1.2.1 Спільне використання прискорювачів FPGA

На відміну від GPU та ASIC, FPGA можуть одночасно розміщувати різні прискорювачі. Це відкриває величезні можливості для покращення хмарних, периферійних і центрів обробки даних. У складних системах, таких як хмара, периферія та центри обробки даних, багато програм і потоків виконуються одночасно. Кожна програма має багато вимог і запитів щодо різних типів інтенсивних обчислювальних операцій, які можуть використовувати переваги прискорювачів FPGA. Наприклад, алгоритму згорткової нейронної мережі потрібно багато векторних операцій на кожному рівні, які зазвичай виконуються на різних ядрах обробки одночасно в конвеєрному механізмі.

Ці рівні можуть спільно використовувати ті самі прискорювачі на FPGA та надсилати їм запити. Ефективна система керування прискорювачем

повинна підтримувати кілька одночасних доступу до прискорювачів у середовищі без перевантажень. Однак у таких системах рисунок 1.3 показує типову блок-схему хост-сервера в хмарі, краю або центрах обробки даних із багатоядерним процесором і FPGA, підключеними до хост-сервера через PCIe (Peripheral Component Interconnect Express).

Комплексна структура, яка керує додатками для інтеграції прискорювача, складається з двох основних рівнів для обробки трьох вимог інтеграційної структури: управління програмним рівнем і менеджер прискорювача апаратного рівня. У таких структурах (показаних як менеджер апаратного прискорення та менеджер програмного прискорювача на рисунку 1.3) ці два рівні відповідають за керування запитами прискорювача на всьому шляху від додатків користувачів на хост-серверах до прискорювачів FPGA. Менеджер програмного рівня та контролер апаратного прискорювача отримують доступ до основної пам’яті хоста для обміну даними за допомогою програмування прямого доступу до пам’яті (DMA) і обмінюються даними (для обміну керуючими даними та інформацією) один з одним через регістри пристрою за допомогою запрограмованого вводу-виводу (PIO).) механізм.

1.3 Переваги апаратного прискорення на основі FPGA

Тут ми обговорюємо переваги FPGA для апаратного прискорення та ефективної обробки даних.

Можливість повторного налаштування. FPGA — це програмоване обладнання, яке можна швидко перепрограмувати протягом мілісекунд. Їх можна запрограмувати на будь-яку спеціальну апаратну функцію. Це дає їм гнучкість для постійного оновлення в польових умовах.

Закон Мура. Згідно з прогнозом Гордона Мура в 1965 році, кількість транзисторів на інтегрованому чіпі подвоюється приблизно кожні два роки [15]. Це також справедливо для FPGA, що дозволяє реалізувати на них повні апаратні системи з мінімальними ресурсами on-chip. Закон Мура дозволив ін-

тегрувати ПЛІС з іншими апаратними блоками, такими як центральні процесори, графічні процесори, DSP, ASIC і пам'ять на одному чіпі. Це підтримує розподіл робочого навантаження та паралельну обробку на FPGA.

Синтез високого рівня (HLS) для програмування FPGA. Методи програмування FPGA пройшли довгий шлях. Спочатку програмування FPGA було відведено лише розробникам обладнання, які володіли мовами низького рівня, такими як Verilog, VHDL, що також було трудомістким процесом. Зі збільшенням складності FPGA більшість постачальників FPGA тепер надають інструменти HLS для синтезу мов високого рівня (наприклад, C, C++, SystemC, OpenCL) до програмованої логіки. Це робить програмування FPGA легше та швидше. Крім того, HLS дозволяє досліджувати простір дизайну, оскільки дозволяє генерувати різні архітектури без необхідності змінювати опис поведінки.

Економічне та швидке створення прототипів. ПЛІС дають значні результати щодо скорочення часу виходу на ринок і витрат на розробку. FPGA вважаються економічно ефективною альтернативою ASIC, оскільки витрати на NRE є незначними.

Ефективна потужність. ПЛІС забезпечують меншу енергоспоживання, ніж процесори, завдяки спеціальним апаратним реалізаціям (рисунок 1.4). Дані можуть оброблятися паралельно з набагато меншою частотою швидше, ніж за допомогою ЦП.

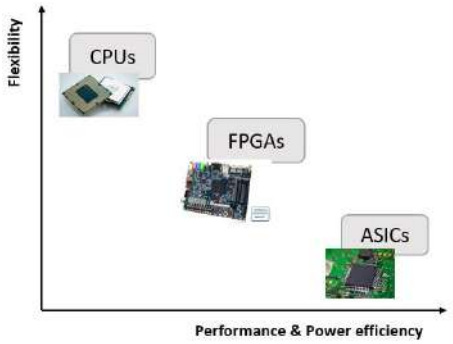


Рисунок 1.4 - Компроміс гнучкості та енергоефективності для CPU, FPGA та

ASIC

1.3.1 Проблеми FPGA для апаратного прискорення

Обговоримо існуючі обмеження FPGA.

Комунікаційні витрати. У гетерогенних системах, таких як SoC-FPGA, ефективність паралельної обробки обмежена через передачу даних на інтерфейсі. Тому нам потрібні архітектури для обробки накладних витрат на зв’язок, що є одним із основних факторів, що впливають на прискорення.

Програмування ПЛІС. Проте ми досягли точки, коли ми можемо синтезувати більшість популярних описів мов високого рівня на FPGA. є ще деякі обмеження. Програміст повинен бути достатньо обізнаним щодо апаратних мікроархітектур та інструментів компілятора. Компілятори мають складний процес компіляції з кількома етапами та циклами, які можуть зайняти кілька годин, щоб синтезувати єдиний дизайн.

1.4 Постановка мети та задач роботи

Швидке зростання додатків Інтернету речей (IoT) і штучного інтелекту викликало нову обчислювальну парадигму (edge computing). Програми периферійних обчислень, такі як відеоспостереження, автономне водіння та доповнена реальність, потребують високої обчислювальної інтенсивності та обробки в реальному часі.

Сучасні периферійні системи, як правило, базуються на стандартному апаратному забезпеченні загального призначення, такому як центральні процесори (CPU) і графічні процесори (GPU), які в основному призначені для великих, незалежних від часу завдань у хмарі та не відповідають потребам граничних навантажень. Крім того, ці системи зазвичай енергоємні і не підходять для крайових розгортань з обмеженими ресурсами. Така невідповідність програми та апаратного забезпечення викликає нову обчислювальну

магістраль для підтримки вимог до високої пропускної здатності, низької затримки та енергоефективності.

Основна проблема під час адаптації цих обчислювальних систем до різних областей полягає в тому, що це призводить до складності дизайну IC, яка ніколи раніше не була бачена, що робить їх дизайн надзвичайно складним.

Основною метою атестаційної роботи є вивчення та впровадження методів прискорення периферійних обчислень з використанням FPGA, які дозволяють спільно використовувати кілька прискорювачів FPGA між різними програмами користувача щодо вимог у хмарах і центрах обробки даних. Ми повинні дослідити доступні методи та платформи, та визнати їхні недоліки та запропонувати свій модифікований метод та фреймворк для спільного використання прискорювачів FPGA для досягнення максимальної продуктивності обчислень.

Головною метою є дослідження сучасного стану використання прискорювачів на основі FPGA та порівняння різних аспектів платформ апаратного прискорення. Для цього потрібно: зробити огляд сучасного стану FPGA, її архітектури, визначити існуючі проблеми FPGA для апаратного прискорення обчислень та виконати порівняльний аналіз існуючих технологій.

Мета кваліфікаційної роботи досягається послідовним вирішенням наступних задач, а саме треба:

- провести дослідження придатності FPGA для прискорення периферійних робочих навантажень;
- розробити модифікований метод прискорення периферійних робочих навантажень з використанням FPGA;
- розробити тестові сценарії на OpenCL для досягнення прискорення на архітектурі SoC-FPGA;
- запропонувати структуру, яка використовує архітектуру з кількома чергами в програмному стеку для мінімізації конфліктів між різними програмами для доступу до різних прискорювачів FPGA.

2 ДОСЛІДЖЕННЯ ЗАСОБІВ ДЛЯ FPGA - ПРИСКОРЕННЯ

2.1 Парадигми периферійних обчислень

Граничні обчислення - це нова парадигма, яка наближає обчислення та зберігання даних до пристроїв, де вони збираються, а не надсилає дані у віддалене місце (зазвичай у хмару), яке може бути за тисячі кілометрів (рисунок 2.1). Ця парадигма допомагає чутливій до затримки програмі досягти продуктивності в реальному часі. Крім того, це може зменшити витрати, якщо обробка виконується локально (або майже локально), зменшуючи кількість даних, які необхідно обробити в централізованому або хмарному місці.

Експоненціальне зростання пристроїв IoT спонукало до розвитку парадигми периферійних обчислень. Ці пристрої зазвичай підключаються до Інтернету для отримання інформації з хмари або доставки даних назад у хмару.

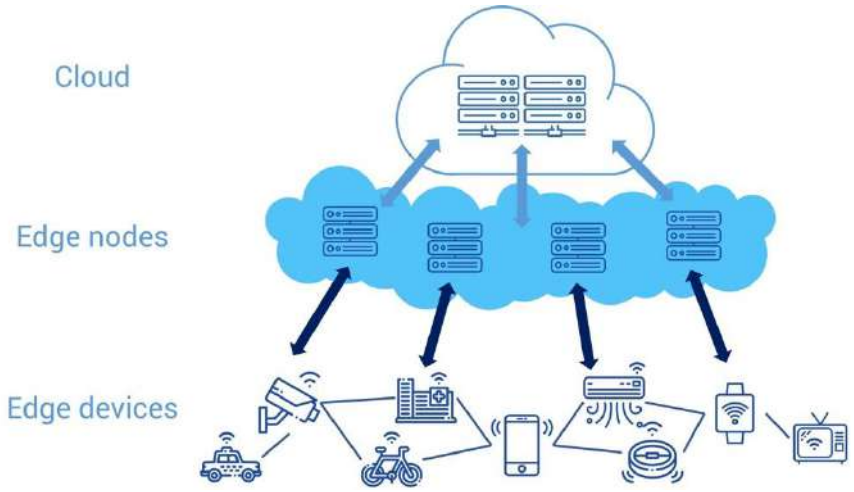


Рисунок 2.1 - Архітектура периферійних обчислень [3]

Прикладами є пристрої, які контролюють виробниче обладнання на заводі, або відеокамера CCTV, яка надсилає живі кадри з віддаленого геолокації. Ці сценарії можуть легко перевантажити ємність хмари та Інтернету, що

вимагає переосмислення обчислювальної парадигми. Архітектура периферійних обчислень може допомогти вирішити вищезгадану проблему, діючи як локальний шлюз обробки для багатьох випадків використання. Наприклад, локальний периферійний хмарлет може обробляти вхідне відео з вуличної камери для виявлення трафіку, а потім надсилати лише відповідні дані назад через хмару, зменшуючи пропускну здатність.

Граничні обчислення дають кілька переваг, які неможливо отримати за допомогою доступної парадигми хмарних обчислень. Тут ми пропонуємо ітерацію цих переваг: Чутливість до часу. Робочі навантаження периферійних обчислень зазвичай обслуговують неоднорідний набір додатків, які вимагають жорстко обмеженого часу відгуку. Одним з головних прикладів таких програм є пристрої IoT. Типові пристрої та датчики IoT, такі як камери, роботизовані руки, датчики температури тощо, постійно генерують дані та надсилають запити (разом із вхідними даними) постачальникам послуг. Ці постачальники запускають служби, які отримують ці запити, обробляють їх разом із вхідними даними та надсилають результат назад на пристрій. Більшість із цих пристроїв повинні реагувати на дії навколишнього середовища або користувача та виконувати швидкі та передбачувані дії відповідно. Таблиця 2.1 представляє середню прийнятну затримку для різних наборів програм. Зрештою, цільові служби повинні: (1) надавати відповіді в режимі реального часу (або майже в режимі реального часу) і (2) надсилати результат назад своєчасно з прогнозуванням. Слід зазначити, що типові пристрої Інтернету речей і користувальницькі пристрої не мають належних вимог до ресурсів (таких як процесор і введення/виведення) для локальної обробки запитів.

Характеристика чутливості до часу нових пристроїв і датчиків IoT відрізняє їх від традиційних додатків для хмарних обчислень. Традиційні хмарні сервіси в основному призначені для: виконання програм, незалежних від часу, у центрі тралізоване середовище; обслуговування інтерактивних запитів від кінцевих користувачів; і - обробка пакетів даних, що надходять з одного джерела. На відміну від цього, периферійні обчислювальні послуги мають

обслуговувати чутливі до часу програми з IoT, а також пристрої кінцевих користувачів. Ми дійшли висновку, що чутливість до часу є однією з важливих характеристик робочих навантажень периферійних обчислень, які мали обмежене існування в робочих навантаженнях попереднього покоління.

Поінформованість про місцезнаходження: розповсюдження IoT та пристроїв кінцевих користувачів запровадило неоднорідність як одну з основних характеристик нових периферійних обчислень.

Таблиця 2.1 - Допустимі затримки для різних послуг

Тип послуги	Прийнятна затримка
Онлайн ігри	< 1000 мс
Аудіо послуги	< 450 мс
Голос через IP	200 мс
Відеосервіси	< 150 мс
Відео через IP	70 мс
Дані	< 400 мс
Передача медичних даних	100 - 400 мс
Телехірургія	300 мс
Електрокардіограма	1000 мс
Послуги не в реальному часі	Кілька секунд

Типові робочі навантаження IoT покладаються на запити кінцевих пристроїв, а також на історичні/географічні відносні дані для підготовки найбільш правильної відповіді. Іншими словами, те саме робоче навантаження IoT, яке виконується за детермінованим алгоритмом для обслуговування запиту, може покладатися на динамічні вхідні дані, щоб надати правильну відповідь. Наприклад, система моніторингу трафіку може використовувати ту саму абстракцію, щоб виявити загальну версію інциденту, пов’язаного з вигулом сойки, але вона вимагає вибіркових даних із локального перерізу, щоб забезпечити максимально точну відповідь. Така характеристика дозволяє ро-

бочим навантаженням периферійних обчислень працювати об’єднаним способом, на відміну від традиційної централізованої моделі обчислень.

Чутливість робочих навантажень периферійних обчислень до найближчих історичних і географічних даних робить їх відмінними від застарілих робочих навантажень хмарних обчислень. Традиційні хмарні робочі навантаження мають доступ до всіх даних, що ускладнює адаптацію до невеликої групи запитів. Навпаки, парадигма периферійних обчислень допомагає обслуговувати запити щодо конкретної просторової та часової інформації. Хтось може заперечити, що традиційне хмарне рішення може надавати різні індивідуальні послуги на основі наборів конкретних історичних/географічних даних і кінцевих користувачів. На жаль, темпи зростання цих послуг виходять за межі можливостей хмари, яку важко підтримувати. Крім того, він не може задовольнити нові вимоги безпеки додатків периферійних комп’ютерів.

2.2 Огляд архітектури FPGA

Ранні програмовані пристрої мали регулярність архітектури та функціональну гнучкість, але до середини 1960-х років польове програмування, здатність змінювати логічну функцію чіпа після процесу виготовлення, було досягнуто завдяки впровадженню стільникових масивів точок. Функціональність кожної логічної комірки в масиві визначатиметься встановленням програмованих запобіжників за допомогою струмів або фотопровідного впливу, хоча всі з’єднання в масиві були зафіксовані. Таким чином, була впроваджена польова програмованість, яка дозволила спростити виготовлення масивів і розширити їх застосування.

У 1970-х роках були введені програмовані пристрої на основі постійної пам’яті (ПЗУ), хоча ПЗУ з можливістю програмування маски та запобіжника (ПЗУ) з n-вхідними адресами можна використовувати для реалізації логічних функцій n-входів, існували проблеми з площею -e ciency, які були представлені разом із цими пристроями. Таким чином, пізніші розробки були зроблені

для введення програмованих логічних масивів (PLA), де кожна площина в провідній структурі AND або провідній АБО разом з інверторами може створювати будь-який логічний термін AND або OR. Цей тип структури точно відповідає загальним логічним функціям і є більш ефективним.

Програмована вентильна матриця (FPGA) — це напівпровідниковий пристрій, побудований із базових компонентів конфігурованих логічних блоків (CLB). На рисунку 2.2 показано, що CLB з'єднані один з одним через з'єднання та програмовані комутатори. ПЛІС можна конфігурувати шляхом програмування їхніх CLB і з'єднань для виконання різних операцій і функцій. На відміну від графічних процесорів і ASIC, FPGA можуть приймати різні типи операцій, при цьому вони повністю незалежні один від одного, але вони можуть працювати одночасно. ПЛІС є дуже енергоефективними порівняно з типовими процесорами та графічним процесором загального призначення.

Порівняно з ASIC попередні FPGA були повільнішими, менш енергоефективними, і для тієї ж роботи FPGA потрібна більша площа. Пізніші FPGA таких великих галузей, як Xilinx і Altera (придбані Intel Inc. у 2015 році), наблизилися до ASIC, забезпечуючи значне зниження енергоспоживання, збільшення швидкості та зниження вартості матеріалів. Однак, у порівнянні з ASIC, FPGA можна реконструювати, і це робить їх придатними для створення прототипів, оскільки з їх використанням налагодження можливе і відносно просте, і часто їхній час виходу на ринок короткий. Порівняно з CPU і GPU, FPGA працюють з нижчими частотами.

Таким чином, щоб досягти підвищення продуктивності за допомогою FPGA, необхідно бути дуже обережним при розробці прискорювачів. По суті, завдяки незерновому паралелізму FPGA мають потенціал для значного підвищення продуктивності. Для підвищення продуктивності ПЛІС потрібні висококваліфіковані дизайнери.

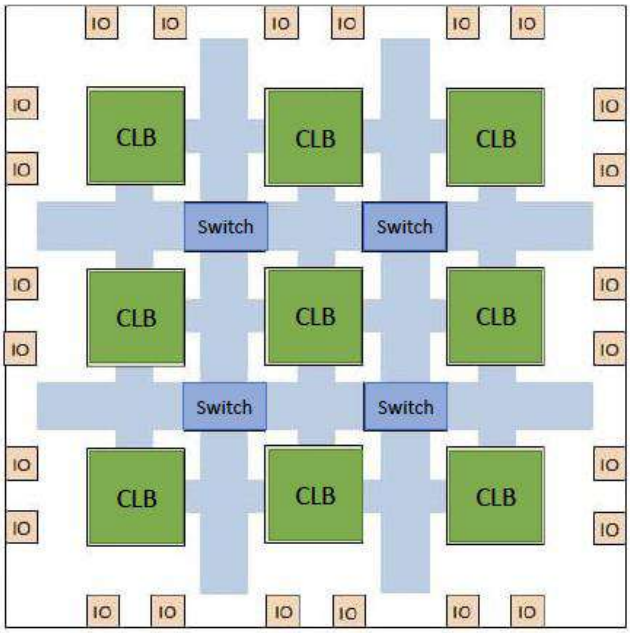


Рисунок 2.2 - Внутрішня архітектура типової FPGA

У доступних на даний момент системах і платформах існує дві основні платформи для взаємодії FPGA і багатоядерних процесорів:

- з використанням плат FPGA через PCIe.
- з використанням FPGA системи на кристалі (SoC).

Незважаючи на переваги SoC FPGA щодо їх упаковки, потужності та прямого доступу до спільної пам’яті, більшість доступних наразі комп’ютерів і серверів у хмарі та центрах обробки даних не підтримують SoC FPGA. Таким чином, інтерфейс між прискорювачами FPGA і багатоядерними процесорами в основному реалізується через інтерфейс вводу-виводу на основі PCIe. PCIe — це стандарт каналу передачі даних для підключення високошвидкісних компонентів. Кожен настільний комп’ютер має кілька слотів PCIe на материнській платі, які можна використовувати для додавання різних периферійних пристроїв, таких як графічні процесори, карти RAID, карти Wi-Fi, плати FPGA або карти SSD (твердотільний накопичувач). Слоти PCIe доступні в різних фізичних конфігураціях з різною кількістю смуг: x1, x4, x8, x16, x32. Наприклад, PCIe x4 має чотири фізичні смуги та може переміщувати дані зі швидкістю чотири біти за цикл. Таблиця 2.2 показує пропускну

здатність різних поколінь PCIe. Враховуючи високу пропускну здатність PCIe, це все ще дуже поширений інтерфейс у хмарі та центрах обробки даних.

Доріжка PCIe представляє два шляхи для кожного напрямку; таким чином, обидва напрямки незалежні один від одного і можуть відбуватися одночасно. Існують різні покоління стандарту PCIe, пропускна здатність кожного покоління приблизно вдвічі більша, ніж у попереднього.

Таблиця 2.2 - Пропускна здатність різних поколінь PCIe

PCI 1.0	1 ГБ/с
PCI 2.0	2 ГБ/с
PCI 3.0	4 ГБ/с
PCI 4.0	8 ГБ/с
PCI 5.0	16 ГБ/с

2.3 Програмні засоби FPGA

Хоча ПЛІС є апаратними платформами, для програмування яких спочатку потрібні апаратно-дружні інструменти низького рівня, було докладено величезних зусиль, щоб наблизити програмування ПЛІС до вищих рівнів абстракції. У цьому розділі наведено огляд програмування ПЛІС та досягнень у цьому напрямку.

2.3.1 Засоби розробки програмного забезпечення для FPGA

Рівень реєстру-передачі (Register-Transfer Level -RTL) — це абстракція дизайну, яка моделює цифрову схему. Проекти RTL для FPGA досягаються за допомогою мов опису обладнання (hardware description languages - HDL), таких як Verilog і VHDL. Концепції програмування HDL дуже відрізняються від програмного забезпечення. У той час як у програмному забезпеченні все є

послідовністю логіки та математики операцій, програмування HDL має природу паралелізму. Ця відмінність створює значні труднощі для звичайних програмістів програмного забезпечення, і їм потрібно глибоке розуміння дизайну апаратного забезпечення, щоб мати можливість відобразити свої знання в програмуванні HDL. Програмування HDL є найбільш зручною для апаратного забезпечення мовою для розробки прискорювачів.

Переходячи до програмно-подібного середовища для розробки прискорювачів FPGA, був представлений процес високорівневого синтезу (HLS). HLS, який також відомий як поведінковий синтез і алгоритмічний синтез, - це процес проектування, у якому високорівневий опис дизайну автоматично компілюється та перетворюється на дизайн RTL, який відповідає певним обмеженням дизайну, визначеним користувачем. Інструменти HLS, які виконують процес HLS і генерують проекти RTL з високорівневого опису, наприклад, C/C++, були комерційно доступні вже більше п’ятнадцяти років.

Незважаючи на те, що інструменти HLS позбулися складності програмування HDL, у використанні цих інструментів все ще є багато проблем. Інструменти HLS не обов'язково створюють найоптимальніший дизайн. Насправді через величезний доступний простір для проектування дуже складно досягти найкращого дизайну з високорівневого опису алгоритму. З іншого боку, програми рівня програмного забезпечення переважно не підтримують апаратне забезпечення, і використання різних конструкцій пам’яті та типів даних, які не підходять для апаратного забезпечення, може призвести до величезної продуктивності та неефективності використання ресурсів.

Кілька робіт у літературі пропонують автоматизовані методи та оптимізацію для використання інструментів HLS для прискорення дослідження простору дизайну. Вони пропонують оптимальний алгоритм для дослідження простору проектування повторного використання обчислень для трафаретних прискорювачів з операціями редукції..

Були докладені певні зусилля щодо оптимізації високорівневих інструментів синтезу для вкладених циклів, щоб знайти найкращі комбінації прагм

для досягнення максимальної продуктивності.

2.3.2 Схема інтерфейсу для прискорювачів на основі FPGA

У хмарі та центрах обробки даних прискорювачі FPGA в основному викликаються програмними додатками, які виконуються на головних комп'ютерах і серверах. Дуже важливою проблемою використання прискорення FPGA є інтеграція прискорювачів у програмні додатки. Було представлено кілька промислових і наукових робіт, які досліджують виклик прискорювача на основі FPGA і передачу даних між хостом і FPGA. Ці роботи в основному спрямовані на дві різні платформи SoC FPGA і FPGA на платі PCIe. Це рішення покращує продуктивність, запобігаючи надходженню додаткових даних копіювати з простору користувача на спеціальну адресу фізичної пам'яті, виділену прискорювачу. Однак для платформ на основі PCIe ця техніка не є привабливою, оскільки додаткове копіювання даних, природно, не відбувається за допомогою карти пам'яті чистого програмного забезпечення. SDSoC — це повністю автоматизований комерційний інструмент Xilinx для дослідження прискорювачів для SoC FPGA. Середовище SDSoC — це платформа на базі Eclipse для реалізації різнорідних вбудованих систем на платах Zynq. Intel також запропонувала відкриту програмну платформу під назвою OPAE для інтеграції прискорювача з багатоядерними процесорами. OPAE — це уніфікована програмна основа для прискорювачів FPGA.

Використання промислових платформ для впровадження та використання прискорювачів FPGA на основі PCIe є дуже поширеним підходом серед дослідників і дизайнерів. Платформи на основі OpenCL із основних галузей промисловості, такі як Xilinx SDAccel і Intel SDK, в основному розроблені для полегшення прискорення додатків на FPGA. Дотримуючись моделі програмування OpenCL для неоднорідних архітектур, ці платформи перетворюють визначені користувачем прискорювальні ядра в дизайни RTL, що супроводжують необхідні компоненти, включно з ієрархією пам'яті OpenCL,

щоб дозволити програмним програмам хоста викликати прискорювачі. Незважаючи на те, що ці промислові платформи від Xilinx і Intel забезпечують зручне середовище для прискорення FPGA, їх фундаментальна концепція контексту розглядає всю FPGA як найменший планований блок. Таким чином, вся FPGA може використовуватися лише одним додатком, що призводить до значного недовикористання FPGA [11]. З іншої точки зору, ієрархія пам'яті, яку використовують платформи на основі OpenCL, також дуже неефективна для потокових додатків через переміщення даних у кількох концентраторах, які не можуть перекриватися.

Щоб подолати звичайне використання FPGA для окремих додатків, було введено концепцію кількох орендарів для спільного використання простору FPGA між декількома прискорювачами, кожен з яких прив'язаний до певного користувача/додатку. Мульти - це багатокористувацький фреймворк, створений на основі Caribou- платформа обробки в сховищі для двигунів баз даних. Хоча Multes зосереджується на справедливому розподілі пропускної здатності між різними орендарями, він не розглядає прискорювачі/ресурси, які спільно/використовуються одночасно різними орендарями. Mbongue та ін. запропонували архітектуру «Мережа на кристалі» (NoC) для багатокористувацького використання FPGA у хмарі для призначення кількох регіонів FPGA користувачам. Однак, як і Multes, ця робота не підтримує спільний доступ до прискорювача між кількома користувачами.

BlastFunction [12] - це нещодавня багатокористувацька система для спільного використання FPGA як мікросервісів у хмарі. BlastFunction дозволяє декільком програмам одночасно виконувати ядра на одній FPGA. Однак BlastFunction не підтримує використання однакових прискорювачів різними програмами. Була запропонована структура виклику прискорювача, яка дозволяє лише різним потокам в одній програмі спільно використовувати прискорювач FPGA і не підтримує спільний доступ між декількома програмами.

Запропонована техніка вводить концепцію FPGA-as-a-Service (FaaS) для спільного використання прискорювачів FPGA, однак FaaS має деякі не-

доліки: виклик прискорювача блокує; використання однієї спільної області пам'яті для передачі даних запобігає накладанню переміщення даних і обчислень. З боку апаратного забезпечення потрібно виділяти великі буфери, тоді як ресурси BRAM обмежені на FPGA, тоді як для більшості проектів BRAM є ресурсами вузьких місць, які перешкоджають більшій кількості прискорювачів; І останнє, але не менш важливе, Кентавр [7] представляє середовище спільного використання прискорювача для спеціальної програми бази даних MonetDB на платформі Intel Xeon+FPGA. Centaur отримує переваги від спільного простору пам’яті між процесорними ядрами та FPGA для активації прискорювача. Centaur використовує механізм опитування для отримання різних завдань до FPGA. Використовуючи концепцію завдання, Centaur є найближчим до нашого фреймворком, який може динамічно керувати розподілом прискорювачів. Однак Кентавр страждає деякими недоліками. Він дозволяє обмінюватися лише кількома потоками в одній програмі та не підтримує прискорювачі спільного використання між кількома програмами. Крім того, обмеження спільної пам’яті для переміщення даних обмежує Centaur виконувати одночасне переміщення даних для кількох прискорювачів. Кентавр призначений тільки для інтегрованих Мікросхеми CPU+FPGA і не можуть бути легко розширені для підтримки PCIe; в той час як через наявні на даний момент платформи в хмарі та центрах обробки даних додавання FPGA через плати PCIe є більш поширеним. Оптимус [6] пропонує гіпервізор для спільної пам’яті на платформі спільної пам’яті FPGA Intel HARP. Використовуючи Memory Mapped I/O (MMIO) для виклику прискорювача та зберігаючи взаємодію ядра обробки з прискорювачами, Optimus не дозволяє надсилати одночасні запити для прискорювача FPGA. Ці структури або не підтримують, або не забезпечують безперебійний інтерфейс для кількох прискорювачів, до яких одночасно звертаються різні програми. З іншого боку, розгортання плат FPGA через PCIe є поширеним завдяки портативності, розширюваності та можливості легкого додавання до поточних систем.

Було розроблено деякі інші фреймворки з відкритим кодом, які дозво-

ляють програмам користувача передавати дані між головними комп'ютерами та прискорювачами FPGA [5-8]. Ці фреймворки потребують розробників програмного забезпечення для керування апаратними прискорювачами, які потребують розуміння апаратних розробок. Чорнило [5] і JetStream [5-7] забезпечують передачу даних datap-ath між хостом і FPGA покладається на комерційні IP-ядра Xilinx. Використання комерційних IP-ядер Xilinx зробило ці фреймворки неможливими для синтезу в поточних версіях інструментів синтезу Xilinx через численні зміни та оновлення IP-ядер.

Кожен прискорювач може бути підключений до цих буферів для передачі даних. Щоб використовувати прискорювач, програма користувача викликає функції send fpga() і receive fpga() у порядку на певному номері каналу, який передається як аргумент цим функціям. Вказівка цільового прискорювача програмою користувача суттєво погіршує продуктивність прискорювачів FPGA, коли прискорювачі спільно використовують декілька програм RIFFA та JetStream, усі вони страждають від підтримки одночасного багатоядерного доступу до прискорювачів. Завдяки керуванню кожним прискорювачем FPGA через певну область діапазону PIO, ці структури також мають обмеження на максимальну кількість прискорювачів, реалізованих на FPGA.

У цьому розділі ми надали огляд архітектури FPGA та їх використання для цілей прискорення. Ми досліджували найбільш пов'язані попередні роботи та дослідження в галузі прискорення FPGA, інтеграції додатків у прискорювачі та спільного використання прискорювачів FPGA. Підводячи підсумок цієї глави, можна сказати, що наявні в даний час інфраструктури для інтеграції користувальницьких додатків і прискорювачів FPGA не можуть забезпечити повне спільне середовище для доступу до прискорювачів FPGA. Повноцінне спільне середовище дозволяє різним програмам отримувати доступ до різних прискорювачів без блокування один одним або іншими зовнішніми обмеженнями. Відсутність такої структури спонукала нас запропонувати наші нові механізми, щоб уможливити спільне використання прискорювачів FPGA і розробити нашу повноцінну структуру UltraShare Express.

2.3.3 Апаратне прискорення та FPGA

Апаратне прискорення є вирішальним чинником високопродуктивних обчислень (High-Performance Computing - HPC). Через обчислювальну інтенсивність робочих навантажень HPC процесори не можуть забезпечити прийнятну продуктивність для програм, критичних до затримок. Ця проблема призводить до використання апаратних прискорювачів, таких як GPU, FPGA та ТПУ Jouppi та ін. (2017). На відміну від ЦП, апаратні прискорювачі можуть використовувати свій масивний паралелізм, щоб розділити основні функції на тисячі паралельних операцій і, зрештою, скоротити загальний час обчислень. Графічні процесори були ретельно вивчені та використані для прискорення робочих навантажень HPC. Хоча графічні процесори є високоефективними в обробці додатків із високим рівнем паралелізму та звичайними шаблонами доступу до пам’яті, вони недостатні для додатків із високим ступенем залежності та/або великою кількістю умовних розгалужень.

Доведено, що ПЛІС здатні забезпечувати набагато нижчу затримку порівняно з процесором і графічним процесором для додатків із критичними для затримки умовами. На відміну від широко поширених GPU і CPU, FPGA можуть прискорювати майже всі типи алгоритмів (незалежно від їх обчислювальної моделі) завдяки їх можливості реконгу рації. Крім того, вони можуть забезпечувати набагато кращу енергоефективність порівняно з процесорами та графічними процесорами, що має вирішальне значення для середовищ з обмеженим енергоспоживанням, таких як периферійні обчислення.

Останні досягнення в мовах високого рівня полегшили програмування та використання прискорювачів, особливо FPGA, для різних програм. Наприклад, розробники можуть використовувати C або C++, щоб описати свій алгоритм, скомпілювати та розгорнути його на цільовому прискорювачі. Постачальники апаратних прискорювачів інтегрували OpenCL, гетерогенну мову паралельного програмування, у свої платформи.

OpenCL має кілька переваг для розробників програмного забезпечення та системних дизайнерів. Це забезпечує легкість розробки збереження вищої абстракції ціною прийнятної втрати продуктивності. Крім того, це дозволяє розробникам програмного забезпечення скористатися перевагами максимальної продуктивності та енергоефективності доступної платформи. Використовуючи OpenCL, розробники можуть описувати свій алгоритм у стандартному представленні та націлюватися на всі доступні прискорювачі, такі як графічні процесори, центральні процесори та DSP. Щоб перенести OpenCL на різні платформи, розробнику потрібно лише внести незначні модифікації, щоб повністю використовувати унікальні функції цільової платформи.

2.3.4 Програмовані вентильні матриці

Програмовані вентильні матриці (FPGA) — це ферма логіки, обчислень і ресурсів зберігання, які можна динамічно конфігурувати (на рисунку 2.3 зображено зразок архітектури Intel Arria 10 FPGA). FPGA можуть бути перенаштовані для виконання алгоритму у визначеній формі. Вони успішно використовуються в багатьох областях застосування.

Незважаючи на вражаючу потужність прискорення, труднощі програмування та оптимізації були серйозними перешкодами для більш широкого впровадження FPGA. Нещодавні досягнення в підтримці високорівневого синтезу (HLS) зробили можливим програмування FPGA за допомогою мов високого рівня, особливо OpenCL, що зробило FPGA набагато простішим у використанні та набагато доступнішим для програм. Незважаючи на те, що програма на основі HLS може працювати не так добре, як ретельно створена вручну програма HDL.

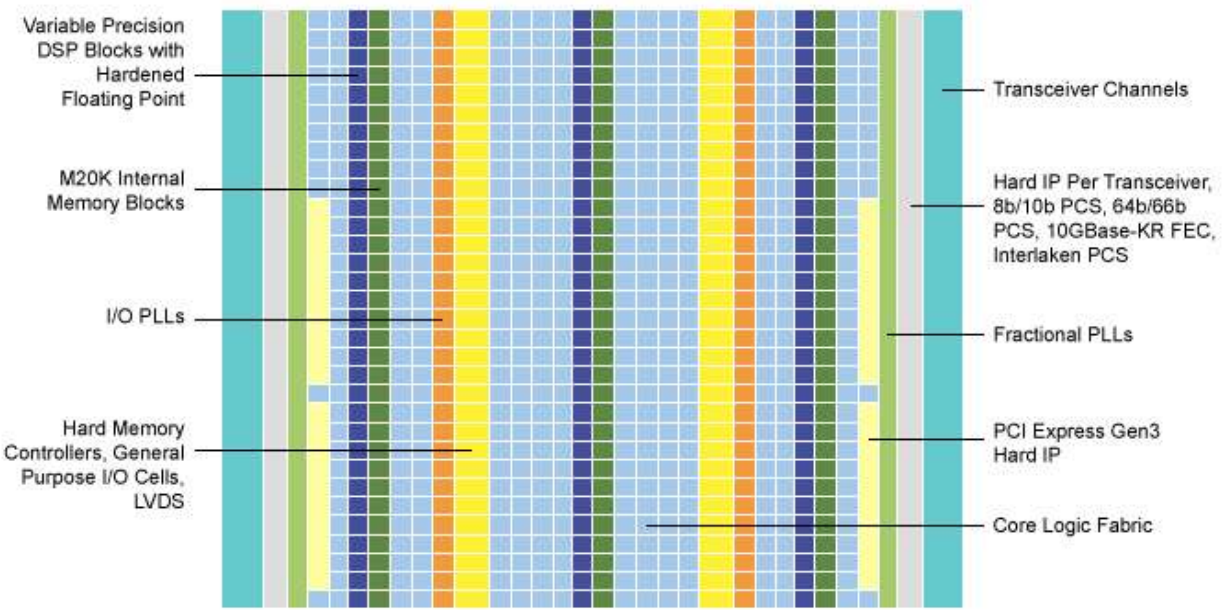


Рисунок 2.2 - Внутрішня архітектура Intel Arria 10 FPGA

Таблиця 2.3 представляє можливості різних прискорювачів у виконанні прикладів програм.

Таблиця 2.3 - Порівняння затримок між GPU, CPU та FPGA

Програма	Пристрій	Затримка (сек.)
Fractal Video Compression Chen and Singh (2013)	CPU	0,217
	GPU	0,018
	FPGA	0,013
Real-time Stereo Vision Kalarot and Morris (2010)	CPU	N/A
	GPU	0,05
	FPGA	0,033
Convolutional Neural Networks Zhang et al. (2018)	CPU	0,73
	GPU	0,023
	FPGA	0,025

Унікальні особливості ПЛІС роблять їх чудовим кандидатом для майбутніх периферійних обчислювальних платформ. Нижче ми повторюємо, як FPGA можуть задовольнити вимоги до робочого навантаження периферійних

обчислень:

- прискорення чутливих до часу програм: ПЛІС можна легко переналаштувати для прискорення певного алгоритму спеціальним способом. Ця унікальна функція може обслуговувати чутливу до часу програму з двох різних точок зору. По-перше, FPGA може конкретно відображати шлях виконання алгоритму на мікросхемі, що допомагає уникнути всіх накладних витрат, які існують у прискорювачах загального призначення, і забезпечує набагато меншу затримку. По-друге, це може гарантувати жорстку межу затримки для цільових IoT або пристроїв користувача, оскільки кількість циклів і довжина кожного циклу для завершення всього виконання на дошці відома заздалегідь. ПЛІС можуть забезпечити кращу затримку для обробки окремого фрагмента даних порівняно з ЦП і ГП:

- Конвеєрну потокову архітектуру з керуванням потоком даних можна легко побудувати на FPGA для обробки потоків даних і команд із каналів вводу/виводу та генерувати вихідні результати з постійною пропускною здатністю зі зменшеною затримкою.

- Адаптивність до характеристик алгоритму: платформи периферійних обчислень потрібні для обслуговування широкого спектру програм у хмарних програмах. ПЛІС можуть адаптуватися до будь-яких характеристик алгоритму завдяки своїй апаратній гнучкості. На відміну від процесорів і графічних процесорів, які здебільшого можуть використовувати просторовий паралелізм, ПЛІС можуть використовувати як просторовий, так і часовий паралелізм із меншою деталізацією у більшому масштабі. ПЛІС можуть створювати обидва типи паралелізму, використовуючи свої численні обчислювальні ресурси та конвеєрні регістри. Вища залежність даних між ітераціями призводить до меншої можливості для просторового паралелізму та більшої можливості для часового паралелізму. У своїх експериментах, споживання звичайних прискорювачів під час виконання широко використовуваних алгоритмів. Виходячи з цих результатів, ПЛІС споживають приблизно в 16 і 3,59 разів менше енергії порівняно з GPU і CPU. Ця перевага критично необхідна для

периферійних серверів, враховуючи їх обмежені форм-фактори.

- Енергоефективність: ПЛІС споживають значно менше електроенергії порівняно з ЦП і ГП, забезпечуючи порівнянну пропускну здатність, що забезпечує покращену термічну стабільність і зниження витрат на охолодження. У таблиці 2.4 представлена загальна потужність споживання звичайних прискорювачів під час виконання широко використовуваних алгоритмів.

Виходячи з цих результатів, ПЛІС споживають приблизно в 16 і 3,59 разів менше енергії порівняно з GPU і CPU. Ця перевага критично необхідна для периферійних серверів, враховуючи їх обмежені форм-фактори.

Таблиця 2.4 - Порівняння затримок між GPU, CPU та FPGA

Програма	Пристрій	Потужність (Вт)
Sliding-Window Application Fowers et al. (2012)	CPU	130
	GPU	274,5
	FPGA	20
Fractal Video Compression Chen and Singh (2013)	CPU	130
	GPU	215
	FPGA	25
Dense Linear Algebra Zohouri et al. (2016)	CPU	78,64
	GPU	184,41
	FPGA	29,48
Convolutional Neural Networks Zhang et al. (2016)	CPU	87,3
	GPU	328,3
	FPGA	19,1

Незважаючи на всі вищезазначені переваги, FPGA не мають певних переваг порівняно з GPU та CPU. Нижче ми повторюємо, чому FPGA не вистачає порівняно з GPU та ЦП, особливо на периферійних навантаженнях:

- низька тактова частота: робоча тактова частота FPGA зазвичай становить від 200 до 300 МГц. Це набагато менше, ніж тактова частота процесорів і графічних процесорів, яка може перевищувати 2 ГГц. Ця величезна різниця може спричинити недостатню продуктивність FPGA для певних програм із певними типами та розмірами вхідних даних. Наприклад, з однаковою кількістю паралелізму на FPGA та іншому прискорювачі вища тактова частота може призвести до значно вищої продуктивності. Як наслідок, FPGA може працювати погано в конкретних програмах порівняно з іншими прискорювачами.

- Обмежені апаратні ресурси: ПЛІС оснащені певними типами ресурсів, такими як цифрові сигнальні процесори (DSP), таблиці пошуку (LUT), тригери (FF) тощо. Загальна кількість цих ресурсів обмежена фізичною площею мікросхема FPGA. Порівняно з прискорювачами ASIC (CPU та GPU), FPGA може використовувати набагато меншу кількість елементів обробки. У результаті вони можуть забезпечити обмежену кількість паралелізму в просторовій області (просторовий паралелізм). Це обмеження робить ПЛІС непридатними для додатків із високим ступенем просторового паралелізму.

- Складне програмування: спочатку ПЛІС можна було програмувати за допомогою мов опису обладнання (HDL), таких як VHDL і Verilog. Останні досягнення покращили програмованість цих пристроїв, дозволивши розробникам використовувати мови програмування вищого рівня, такі як C/C++, для опису та компіляції своїх алгоритмів на цільових FPGA. На жаль, навіть за наявності цих мов розробникам апаратного забезпечення потрібно витратити значну кількість часу, щоб повністю налаштувати дизайн для FPGA, щоб отримати найвищу можливу продуктивність. Отже, ПЛІС доступні лише обмеженій кількості групи розробників обладнання, але не загальна спільнота програмного забезпечення. З іншого боку, графічні та центральні процесори підтримуються високоефективними компіляторами та великою кількістю бібліотек, що робить їх доступними для різноманітної групи користувачів.

2.4 SoC FPGA

Пристрої SoC FPGA інтегрують як процесор, так і архітектуру FPGA в одному пристрої. Об’єднання двох технологій забезпечує низку переваг, включаючи кращу інтеграцію, меншу потужність, менший розмір плати та більшу пропускну здатність зв’язку між процесором і FPGA. Найкращі в своєму класі пристрої використовують унікальні переваги об’єднаного процесора та системи FPGA, зберігаючи при цьому переваги автономного процесора та підходу FPGA [14].

Оскільки сигнали між процесором і FPGA тепер знаходяться на одному кремнії, зв’язок між ними споживає значно менше енергії порівняно з використанням окремих мікросхем. Інтеграція тисяч внутрішніх з’єднань між процесором і FPGA призводить до значно вищої пропускної здатності та меншої затримки порівняно з двочіповим рішенням.

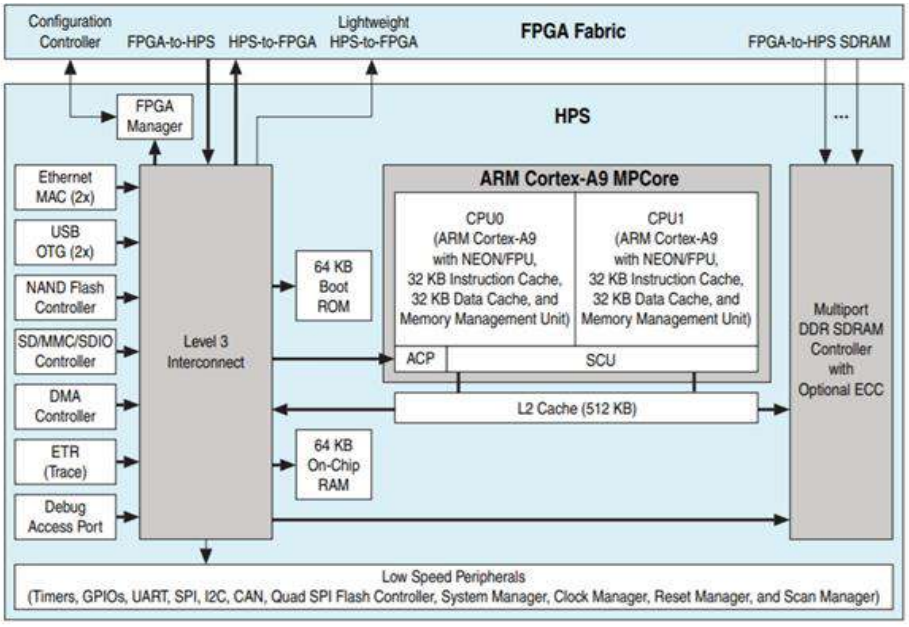


Рисунок 2.3- Cyclone V SoC FPGA - HPS з двоядерним процесором ARM Cortex A9 MPCore з'єднується з FPGA Fabric [9]

Рисунок 2.3 показує архітектуру Altera Cyclone V, SoC FPGA з HPS.

HPS складається з процесора ARM Cortex-A9 MPCore, багатого набору периферійних пристроїв і спільного багатопортового контролера пам’яті SDRAM. Існує три мости HPS-FPGA AXI, які підтримують специфікації передової архітектури шини мікроконтролерів (AMBA) і вдосконаленого розширюваного інтерфейсу (AXI): (a) Міст FPGA до HPS AXI — це високопродуктивна шина, що підтримує 32-, 64- та 128 -бітова ширина даних дозволяє FPGA-матеріалу бути головним для підлеглих на HPS.

HPS to FPGA AXI Bridge — це високопродуктивний 32-, 64- та 128-бітний модуль ширини даних, який дозволяє HPS бути основним для підлеглих мереж FPGA. (c) Легкий HPS до FPGA є низькопродуктивним 32-розрядним мостом AXI, де HPS є головним [12].

3 АНАЛІЗ ФУНКЦІОНУВАННЯ БАГАТОЧЕРГОВОГО FPGA - ПРИСКОРЮВАЧА

3.1 Сучасний стан проблеми

ПЛІС показали життєздатне рішення для вирішення вимогливої продуктивності та енергоефективності в епоху додатків з великими даними. Однак, незважаючи на багатообіцяючі покращення, яких було досягнуто в усіх дослідженнях і роботах для конкретних програм із використанням прискорювачів FPGA, отримати максимальну продуктивність від FPGA у хмарі та центрах даних є дуже складним завданням.

У найближчому майбутньому FPGA стануть звичайними платформами обробки в центрах обробки даних і хмарних обчисленнях. У хмарі та центрах обробки даних багато програм із різними обчислювальними ядрами потребують використання прискорювачів FPGA, щоб досягти своїх вимогливих обчислювальних можливостей. Щоб задовольнити цей попит, необхідна структура інтерфейсу для інтеграції програмних додатків у прискорювачі FPGA та керування запитами FPGA.

Менеджмент запитів FPGA ще більше ускладнюється збільшенням кількості паралельних потоків, що запитують доступ до прискорювачів, і експоненціальним зростанням обсягу даних для надсилання/отримання до/від прискорювачів FPGA. Як наслідок, розгортання прискорювачів на основі FPGA у великих програмних системах ускладнюється відсутністю масштабованої схеми від прикладного програмного забезпечення до інтерфейсу вводу-виводу, яка б дозволяла багатоядерним системам отримувати доступ до прискорювачів на основі FPGA з високою швидкістю передачі даних.

На відміну від графічних процесорів, доступ до кількох прискорювачів FPGA можна отримати з багатоядерного процесора (рисунок 3.1) які можуть бути розподілені між кількома потоками. Такі множинні прискорювачі при-

вабливі для додатків, оскільки це дозволяє уникнути тривалого часу очікування для потоків, щоб отримати доступ до прискорювачів. Однак керування кількома запитами на доступ до кількох прискорювачів є складнішим і складнішим як на стороні FPGA (контролер прискорювача), так і на стороні центрального процесора (програмний стек вводу-виводу). Отже, нам потрібна схема інтерфейсу, яка масштабується зі збільшенням кількості хост-ядер, прискорювачів і обсягу даних введення-виведення.

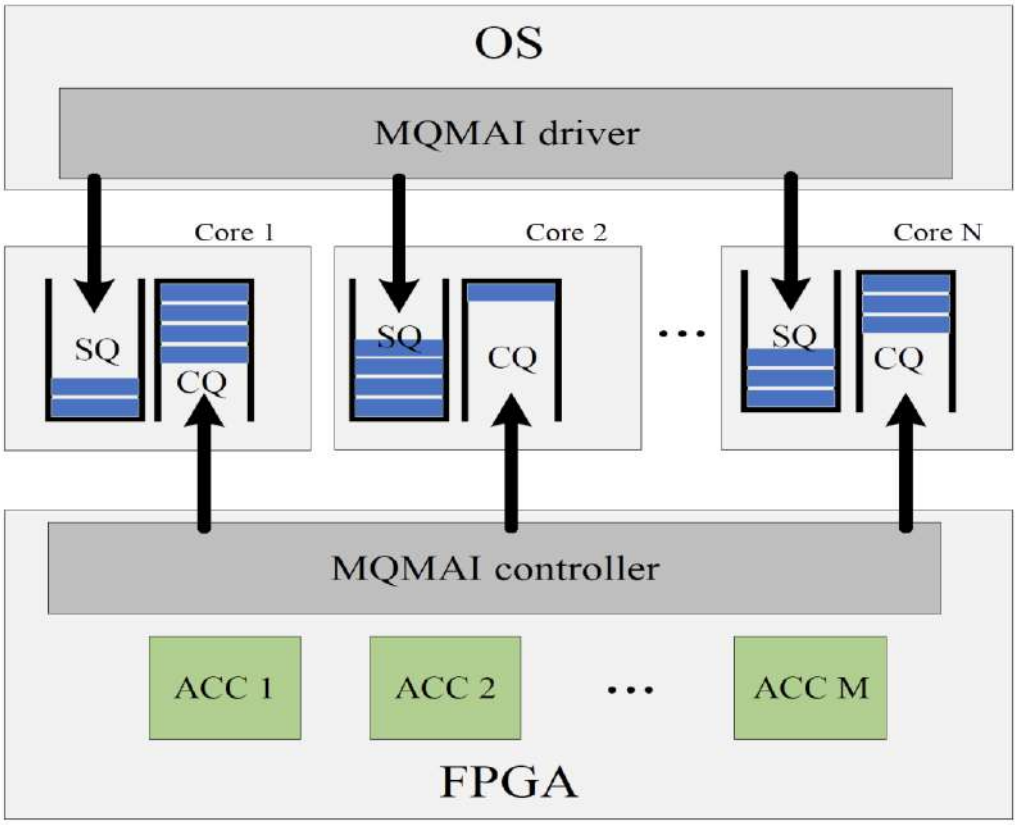


Рисунок 3.1 - Багаточергова структура MQMAI

Сучасний дизайн на основі FPGA, як-от Xilinx SDAccel і Intel Open Programmable Acceleration Engine (OPAE) не підтримують спільний доступ до прискорювачів між кількома програмами катіони/потоки. Спільне використання прискорювачів, як запропоновано в попередніх роботах [2-5] створює проблеми з конфліктом ресурсів у програмному стеку вводу/виводу, тобто потік очікує на прискорювач, коли його використовує інший потік. Щоб усу-

нути цей недолік, ми представляємо MQMAI, механізм на основі команд із кількома чергами, щоб зменшити суперечку за доступ до кількох прискорювачів FPGA. Запропонована нами техніка покращує паралелізм введення/виведення для доступу до кількох прискорювачів. Наскільки нам відомо, це перша спроба запровадити ефективний інтерфейс із кількома чергами, щоб дозволити доступ кільком потокам/додаткам і спільне використання кількох прискорювачів на основі FPGA.

Механізм передачі даних з кількома чергами в MQMAI (Multi-Queue Multi-Accelerator Interface) базується на неблокуючих командах і забезпечує ефективний і масштабований доступ до кількох прискорювачів на FPGA через PCIe. MQMAI складається з:

1) програмного стека на стороні хоста;

2) апаратного контролера на стороні FPGA.

З боку програмного забезпечення драйвер пристрою керує архітектурою програмного забезпечення з кількома чергами та подає команди, коли надходить запит на доступ. Він оновлює черги, коли хост отримує сповіщення про завершення команди з FPGA. Роль апаратного контролера починається з отримання команди. Контролер відповідає за переміщення даних і планування доступу до прискорювача без будь-якої взаємодії з головним ЦП, отже, зменшуючи час ЦП, який витрачається на керування прискорювачем і передачу даних.

У порівнянні з найсучаснішими фреймворками передачі даних, такими як RIFFA (Reusable Integration Framework for FPGA Accelerators), запропонований нами метод зменшує трафік зв’язку між центральним процесором і FPGA для переміщення даних, а також конкуренції за ресурси. Це призводить до значного покращення загальної затримки (загальний доступ і час процесу) паралельних потоків, які отримують доступ до одного або кількох прискорювачів на FPGA. Порівняно з RIFFA наші експериментальні результати показують, що зі збільшенням кількості паралельних потоків, до яких звертається один прискорювач, загальна затримка збільшується більш масш-

табовано за допомогою інфраструктури MQMAI. Наприклад, для 4 паралельних потоків, які отримують доступ до прискорювачів FPGA у системі з 4 прискорювачами на основі FPGA, загальна затримка в 18 разів нижча порівняно з RIFFA.

Підсумуємо наші подальші кроки щодо MQMAI наступним чином, це:

- розробка структури на основі команд із кількома чергами, що дозволяє зменшити кількість конфліктів у середовищі спільного використання прискорювача;
- надання бібліотеки C із зручним API для доступу до прискорювачів на основі FPGA;
- розробка контролера прискорювача на апаратному рівні для розподілу прискорювачів для кількох запитів від паралельних потоків/додатків;
- надання фреймворку з відкритим вихідним кодом MQMAI від програмного стека до проектування RTL контролера акселератора.

3.2 Структура та інтерфейс багаточергового прискорювача MQMAI

У багатих на прискорювачі архітектурах кілька потоків можуть одночасно запитувати доступ до прискорювачів. Спільне використання прискорювачів між потоками призводить до проблем із конфліктом ресурсів у програмному стеку вводу-виводу. Конкуренція за ресурс (тобто потік очікує на ресурс під час використання іншим потоком) є недоліком багатопоточних програм. Управління суперечками за ресурси є ключовим завданням для максимального використання ресурсів. Коли запит на доступ до ресурсу надсилається на рівні програми, він повинен пройти через стек системного програмного забезпечення, щоб отримати доступ до ресурсу. Відхилення запиту через суперечку призводить до зниження продуктивності, оскільки запиту необхідно повторно пройти через стек програмного забезпечення. Конкуренція за ресурси також різко погіршує продуктивність, оскільки або ядро обробки буде заблоковано,

Введення кількох черг у багатоядерні процесори є багатообіцяючим рішенням для керування конкуренцією за ресурси в додатках із інтенсивним введенням-виведенням. У цьому рішенні кожне ядро має власну виділену чергу для отримання доступу до ресурсу введення-виведення. Таким чином, кожне ядро може надсилати запит, не стикаючись із суперечками з точки зору програмного стеку.

Розгортання багаточергової архітектури для випуску запиту читання/запису на SSD-накопичувачі, такі як NVMe, значно покращили доступ до пам’яті через послідовний порт PCIe [3,13]. NVMe використовує кілька черг, щоб запропонувати вищий рівень паралелізму вводу-виводу. Після того як ЦП записує команду в чергу, ЦП завершує запит на введення/виведення і може перейти до іншого запиту. В іншому випадку ЦП відповідає за проходження запиту через кілька функцій до того, як команда буде остаточно видана пристрою. Таким чином, драйвер NVMe скорочує час процесора для надсилання запиту введення-виведення. NVMe є багатообіцяючим стандартом для забезпечення масштабованого доступу до SSD з високою продуктивністю та пропускною здатністю.

Порівняно з NVMe, кілька прискорювачів, доступ до яких використовується кількома паралельними потоками програм, стикаються з подібними проблемами. Кожен прискорювач на FPGA можна розглядати як блок пам’яті в SSD для паралельного доступу. Великий розмір даних вводу-виводу прискорювачів для додатків з великими даними та зростаюча кількість потоків, які отримують доступ до кількох прискорювачів, нагадують додатки з інтенсивним введенням-виведенням з боку програмного забезпечення вводу-виводу процесора. Прийняття стандартів, подібних до NVMe, і стека програмного забезпечення для доступу з декількома чергами до мультиприскорювачів у FPGA є масштабованою структурою, що забезпечує більший паралелізм і високу продуктивність; що було б багатообіцяючим рішенням для усунення відсутності доступу до багатоядерного прискорювача в попередніх структурах передачі даних.

У цьому розділі ми представляємо MQMAI, багаточерговий командний інтерфейс для прискорювача FPGA. Подібно до NVMe, MQMAI обходить рівень блоків і натомість розгортає механізм із кількома чергами. MQMAI дозволяє різним хост-додаткам безпосередньо викликати та спільно використовувати прискорювачі FPGA через шину PCIe (проте протокол не обмежується шиною PCIe) без суперечок. Цей інтерфейс визначає набір команд і реєстрів, які будуть використовуватися між драйвером MQMAI в головній операційній системі та контролером MQMAI на FPGA. Він має кілька черг вводу-виводу, які спільно використовуються між драйвером і контролером, як показано на рисунку 3.1 вище, який є масштабованим з точки зору кількості черг і кількості записів у черзі.

3.2.1 Платформа MQMAI

MQMAI складається з: програмного стека та апаратного контролера. Стек програмного забезпечення включає драйвер пристрою, який керує запитами викликів бібліотеки, командами, чергами підзадач/завершення та обробником переривань MSI від контролера MQMAI. Роль апаратного контролера починається з отримання сигналу (званого дверним дзвінком) від програмного забезпечення, яке показує, що команда доступна. Тоді весь процес керування даними виконується в апаратному забезпеченні без взаємодії з центральним процесором. Після зворотного запису результату запиту прискорювача в пам’ять хоста контролер сповіщає ЦП хоста про завершення команди через переривання MSI.

Стек програмного забезпечення на центральному процесорі.

Стек програмного забезпечення MQMAI складається з двох рівнів: рівня бібліотеки та рівня драйвера. Бібліотечний рівень також стискає та стандартизує аргументи викликів API і передає їх драйверу пристрою. Драйвер MQMAI здатний обробляти одночасний багатопоточний (ядерний) доступ, використовуючи черги кількох команд (подання/завершення). Ми пояснюємо

основні компоненти драйвера MQMAI наступним чином:

Команда подання/завершення: команда — це набір інформації, пов’язаної з кожним запитом/відповіддю. Команда надсилання генерується драйвером пристрою для вказівки запиту до апаратного контролера. Драйвер пристрою надсилає команду надсилання в чергу надсилання відповідного ядра. Таблиця 3.1 показує різні поля команди подання.

Таблиця 3.1 - Структура команди UltraShare

Ім'я	Номер біт	Пояснення
Cmd_id	10	Кожна команда має певний ідентифікатор команди
Core_id	5	Ідентифікатор ядра, яке надало команду
Тип_аккаунта	5	Запитаний прискорювач
In_sgl_addr	64	Адреса SGL для вхідних даних
In_sgl_len	10	Довжина вхідного SGL
Out_sgl_addr	64	Адреса SGL для вихідних даних
Out_sgl_len	10	Довжина вихідного SGL
Зарезервований	88	Зарезервований

Кожна команда надсилання має 256 біт, включаючи ідентифікатор команди та ідентифікатор ядра, яке виконує відповідний потік. Тип команди введення/виведення вказує, якщо адреса вводу/виводу відноситься до основних даних (сторінок фізичної області (PRP)) або списку розсіяних даних (SG). Якщо розмір даних перевищує одну сторінку пам’яті, операційна система зберігає кожну сторінку в іншому місці. Таким чином, для читання/запису даних у потрібне розташування апаратному забезпеченню потрібна адреса всіх сторінок, використовуючи список усіх адрес SG. Ідентифікатор прискорювача вказує тип прискорювача. Він керує контролером MQMAI, щоб призначити правильний тип прискорювача кожному запиту (у разі наяв-

ності кількох типів прискорювача). Довжина вводу/виводу вказує довжину даних/SG для передачі.

Команда завершення генерується апаратним контролером, щоб повідомити драйверу про виконаний запит. Апаратний контролер записує команду завершення в чергу завершення відповідного ядра через запит DMA. Команда завершення містить лише ідентифікатор команди та біт перевірки.

Черга надсилання/завершення: у драйвері пристрою для кожного ядра призначено одну чергу надсилання та одну чергу завершення. Команди, видані з кожного ядра, надсилаються до відповідної черги відправлення. Ці черги спільно використовуються між драйвером пристрою та апаратним контролером для обміну командами. Цей механізм із кількома чергами дає можливість обробляти паралельні запити введення/виведення для доступу до прискорювачів. Розмір черг залежить від навантаження і кількості паралельних запитів.

Дверний дзвінок: драйвер пристрою сповіщає апаратний контролер про існування команд, надсилаючи до контролера реєстр дверного дзвінка. Цей регістр містить кінцеву адресу відповідної черги подання через режим програмованого вводу-виводу (PIO). Кожна черга подання має свій окремий дверний дзвінок. Апаратний контролер MQMAI завжди відстежує головні адреси всіх черг подання. Таким чином, отримавши кінцеву адресу, апаратний контролер буде знати про доступні команди та надсилає запит на читання до DMA, щоб отримати команди. Передача реєстрів дверних дзвінків є єдиним режимом PIO, який розгортає MQMAI. Решта передачі даних здійснюється через DMA. Це робить центральний процесор від’єднаним від процесу передачі даних. Попередні інтерфейси з відкритим вихідним кодом, наприклад ink і RIFFA, тримає головний процесор зайнятим шляхом обміну інформацією про квитування між хостом і FPGA в режимі PIO під час передачі даних.

Обробник переривань: коли апаратний контролер записує команду завершення в чергу завершення, він сповіщає про це драйвер пристрою, видаючи повідомлення про переривання (MSI). Переривання MSI визначає іден-

тифікатор ядра, з якого було видано запит на доступ. Таким чином, драйвер пристрою починає зчитувати всі команди завершення з відповідної черги завершення.

Апаратна архітектура.

На рисунку 3.2 показано блок-схему апаратної архітектури MQMAI. Конструкція апаратного забезпечення складається з 3 основних модулів:

1) інтерфейсу PCIe;

2) контролер MQMAI;

3) канали інтерфейсу.

Для інтерфейсу PCIe ми використовували вбудований IP-блок Xilinx PCIe. Цей IP-блок має інтерфейс AXI головний-підлеглий, який можна налаштувати для надсилання та отримання даних як у режимі PIO, так і в режимі DMA [15].

Контролер MQMAI: містить 4 основні частини:

1) блок керування «doorbell registers unit»;

2) контролер команд;

3) блок керування SG (SGMU) ;

4) управління командою завершення.

Блок керування реєстраторами дверних дзвінків складається з регістрів дверних дзвінків і контролера дверних дзвінків. Для кожної черги подання на стороні хоста два регістри дверних дзвінків зберігають головну та кінцеву адреси DMA черги подання на стороні обладнання. Контролер дверного дзвінка починає надавати запит DMA через модуль RX MUX щоразу, коли головна адреса не дорівнює кінцевій адресі для будь-якої з черг.

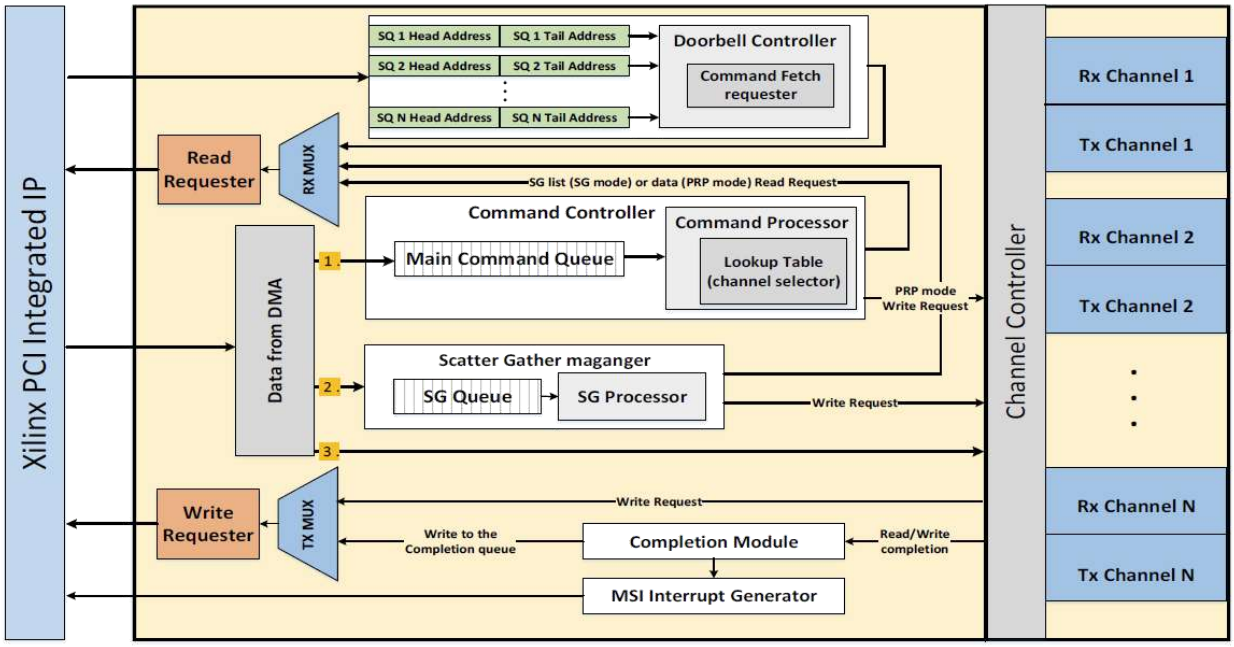


Рисунок 3.2 - Блок-схема апаратної архітектури MQMAI

Отримані команди будуть перенаправлені до контролера команд і збережені в черзі команд для обробки в блоці процесора команд. Контролер повинен призначити прискорювач для кожної команди. Якщо на FPGA є кілька ідентичних прискорювачів, планувальник має можливість призначати прискорювачі командам на основі наявності прискорювачів або в порядку критичності команд. У нашій поточній системі контролер має циклічний планувальник. Це ключова функція, яка дозволяє віртуалізувати цільові буфери на рівні програми та покращує використання прискорювача.

Інтерфейсні канали мають дві основні частини:

- канали Rx/Tx;
- контролер каналу.

MQMAI має один контролер каналів і стільки доступних каналів Rx/Tx прискорювачів. Канал Rx зберігає вхідні дані для прискорювача, а дані Tx зберігають результат роботи прискорювача. Контролер каналів керує всіма даними зчитування/запису з/до каналів Rx/Tx. Контролер каналу може обробляти одночасне читання та запис з/в пам'ять хоста. Контролер каналу також

отримує запити на запис у пам'ять від контролера команд і SGMU, зберігає запит, доки на відповідному каналі Tx не буде достатньо даних, а потім надсилає запит на запис у пам'ять через DMA.

3.2.2 Протокол MQMAI

MQMAI розгортає механізм на основі команд для надсилання запитів прискорювача від хост-додатку до відповідного прискорювача на FPGA. Будучи заснованим на одній команді, ви можете додавати багато функцій, включаючи повне спільне використання прискорювачів між декількома додатками.

Рисунок 3.3 показує основну часову діаграму протоколу виклику прискорювача Ultra-Share Express.

Як показано на рисунку 3.3, починається обробка запиту від головної програми шляхом перегляду стека програмного забезпечення UltraShare, тобто бібліотеки UltraShare та драйвера пристрою UltraShare. Бібліотека UltraShare - це інтерфейс для перекладу зручних API у команди I/O controller, які взаємодіють із драйвером пристрою. У драйвері пристрою для кожного запиту створюється команда, яка включає всю необхідну інформацію, яка представляє запит, який потрібно обробити в прискорювачі FPGA без додаткової взаємодії з головними процесорами. Згенерована команда надсилається в чергу подання (SQ). Кожен SQ має сигнал «дверного дзвінка», який сповіщає апаратний контролер UltraShare (реалізований на FPGA) про команди, які знаходяться в черзі в цьому SQ. З цього моменту процесорні ядра хоста не взаємодіють із виданим запитом до отримання переривання завершення та команди завершення від сторони FPGA.

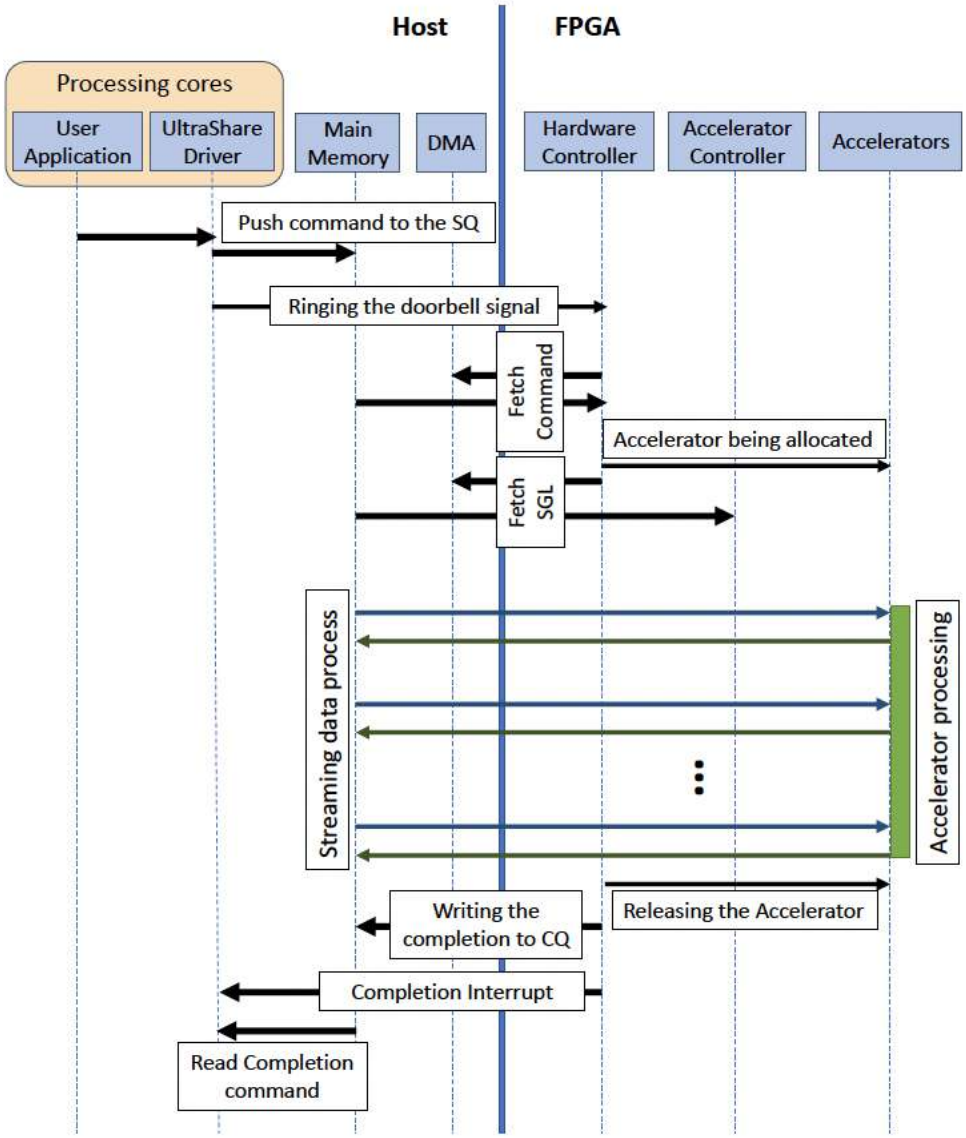


Рисунок 3.3 - Часова діаграма UltraShare Express від програми користувача, яка запитує процес прискорення, до моменту обробки запиту

Коли надходить сигнал «дверного дзвінка», апаратний контролер отримує команди з черги у відповідному SQ. Кожна отримана команда ставиться в чергу в апаратному контролері в очікуванні своєї черги для обробки відповідним прискорювачем. Коли прискорювач доступний для обслуговування команди, першим кроком є отримання пов’язаних вхідних і вихідних списків розкиду (SG) команди. Потім вхідні дані на основі вхідного SGL вибираються для подачі виділеного прискорювача. Після завершення процесу результат записується назад в основну пам’ять хоста на основі вихідного

SGL. Потім команда завершення записується в чергу завершення (QC) в основній пам'яті хоста, і видається переривання, щоб процесори хоста повідомили про завершення.

Модель програмування програмного забезпечення.

Щоб використовувати MQMAI, користувач повинен дотримуватися нашої моделі програмування у своїх кодах C/C++. Поточна версія MQMAI має 4 основні API: fpga open(), fpga close(), fpga acc() і fpga wait(). fpga open() і fpga close() використовуються для відкриття/закриття драйвера пристрою MQMAI, і почати/закінчити використання MQMAI. fpga acc() використовується для виклику прискорювача. Це неблокуючий API, який дозволяє програмам, які не залежать від даних, мати можливість надсилати багато запитів прискорювача або виконувати решту своїх обчислень. fpga wait() використовується для очікування виданих запитів прискорювача.

Апаратний інтерфейс.

Щоб інтегрувати потоковий прискорювач у MQMAI, прискорювач має використовувати сигнали нашого апаратного інтерфейсу. Апаратний інтерфейс включає 7 різних сигналів:

- Input signals: Reset, User clock, Tx data, Tx data valid, Rx data request.
- Output signals: Rx data, Rx data valid.

Щоб зчитувати дані з каналу Rx, прискорювач повинен встановити сигнал запиту даних Rx на 1 і дочекатися, поки сигнал дійсності даних Rx стане 1. Дійсні дані надходять на шину даних Rx, тоді як сигнал дійсності даних Rx дорівнює 1. записувати дані в канал Tx, прискорювач повинен лише помістити дані на шину даних Tx і підтримувати дані Tx дійсними на високому рівні, поки дані Tx несуть дійсні дані.

3.3 Експериментальні дослідження

Щоб оцінити MQMAI, ми використовували головний комп’ютер із чотирьохядерним процесором Intel i5-4590, що працює на частоті 3,3 ГГц, і

пам’яттю 8 ГБ під керуванням операційної системи Linux. Ми розробили бібліотеку на рівні користувача для програм і драйвер пристрою на рівні ядра в ядрі Linux, щоб отримати доступ до нашої плати FPGA через 4 смуги шини PCI Express 3.0. Це плата даних ADM-PCIE-7V3 alpha з Xilinx Virtex 7 FPGA, де ми програмуємо контролер MQMAI та деякі прискорювачі. Щоб створити бітові файли дизайну RTL, ми використали інструмент пакету проектування Xilinx Vivado 16.4. Для порівняння ми також запровадили RIFFA[46],яка є найсучаснішою платформою з відкритим кодом.

Розроблена платформа показана на рисунку 3.4. Щоб оцінити масштабованість продуктивності MQMAI, ми змінювали кількість ядер (C), кількість потоків на ядро (T) і кількість запитів на прискорювач, які ітеративно надсилаються як пакет у циклі for кожного потоку (K). На рисунку 3.5 показний псевдокод потоків. Різні потоки можуть викликати один і той самий прискорювач або різні прискорювачі відповідно, що може допомогти зрозуміти наслідки суперечок за ресурси.

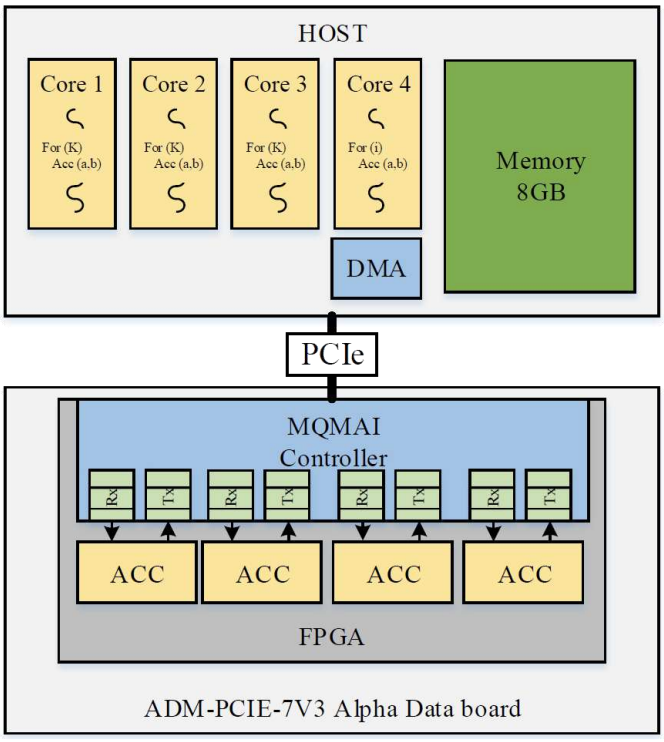


Рисунок 3.4 – Схема експериментальної платформи

Ми програмуємо плату FPGA за допомогою нашого контролера MQMAI та 4 функціонально ідентичних прискорювачів чорного ящика. Передбачається, що кожен прискорювач має конвеєрну архітектуру з потоковими даними введення/виведення. Це припущення стосується простої експериментальної установки, і їх можна запрограмувати на виконання різних функцій. Ми також припускаємо, що затримка, викликана кожним прискорювачем, становить 10 нс для обробки кожного запиту прискорювача розміром даних 1 Мбайт від потоку.

```
Void *thread_main () {
        buffer *A;
        buffer *B;
        …
        While () {
        // Start of measuring the delay
        For (i = 0, i < K, i ++)
                // A: input data      B: output data
                Ret = acc_fpga (fpga, acc_type, A, length, B, length);
        wait_fpga (fpga, acc_id, K);
        // End of measuring the delay
        }
}
```

Рисунок 3.5 - Псевдокод потоків

Кожен прискорювач зчитує дані зі свого виділеного каналу Rx і записує результат назад у свій канал Tx (рисунок 3.4). Зауважте, що наша оціночна метрика – це наскрізна затримка (або затримка), виміряна на рівні програми для кожної партії K запитів прискорювача. малюнок3.5показує дві точки відміток часу в потоці, щоб отримати наскрізну затримку.

3.3.1 Результати експерименту

Масштабованість продуктивності.

Однопотоковий одиночний прискорювач - рисунок 3.6 показує пропускну здатність, досягнуту MQMAI та RIFFA для різних розмірів запитів прискорювача від хоста до плати FPGA. У цьому випадку ми запускаємо один потік, який генерує запити через той самий прискорювач. Результати показують, що пропускна здатність зростає майже лінійно до розміру запиту прискорювача як для MQMAI, так і для RIFFA. Вони забезпечують однакову пропускну здатність для кожного розміру запиту, але MQMAI завжди трохи кращий за RIFFA. Для розміру запиту 1 Мбайт (МБ) MQMAI досягає 2803 МБ/с, а RIFFA — 2727 МБ/с. Це покращення здебільшого відбувається завдяки меншій кількості взаємодій між процесором і FPGA для обміну інформацією для налаштування передачі даних і сповіщення про завершення. Для решти експериментів ми встановили розмір запиту 1 Мб.

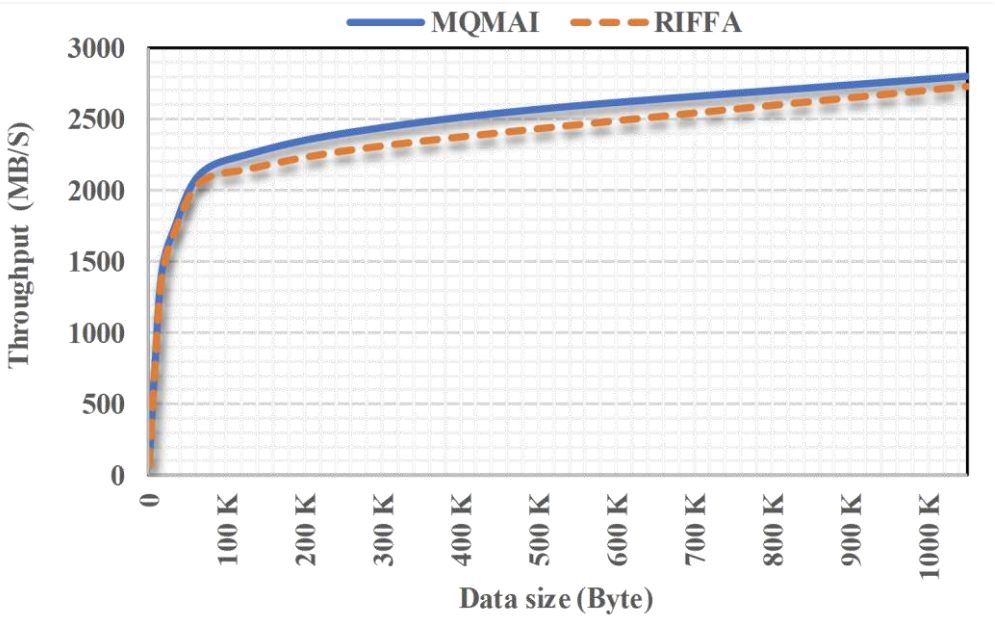


Рисунок 3.6 - Пропускна здатність для запитів різних розмірів у випадку однопотокового прискорювача

Випадок MTSA (багатопотокового одиночного прискорювача): щоб оцінити здатність до масштабування продуктивності, отриману від нашого \інтерфейсу на основі команд» MQMAI, ми змінюємо кількість потоків (T) на одному ядрі від 1 до 4 і число запитів на прискорювач, виданих як пакет від кожного потоку (K) від 1 до 20. Передбачається, що всі потоки мають доступ до одного прискорювача. Таблиця 3.2 показує середню наскрізну затримку, виміряну на рівні програми як для MQMAI, так і для RIFFA.

Таблиця 3.2 - Середня наскрізна затримка у випадку багатопотокового одиночного прискорювача

Framework	N потоків (T)	End-to-End затримка (секунди)				
		Кількість запитів у пакеті (K)				
		1	2	5	10	20
RIFFA	1	0,00211	0,00442	0,00905	0,01899	0,03720
	2	0,00882	0,03869	0,06087	0,10994	0,23968
	4	0,01728	0,02467	0,80174	1,6634	2,25089
MQMAI	1	0,00197	0,00368	0,00871	0,01721	0,03393
	2	0,00302	0,00667	0,01581	0,03035	0,06531
	4	0,00481	0,00996	0,02748	0,05536	0,11783

З таблиці ми бачимо, що MQMAI завжди дає меншу затримку, ніж RIFFA, для всіх комбінацій T і K, які ми тестуємо. Коли T = 4 і K = 20, затримка, отримана за допомогою MQMAI, становить лише 0,118 секунди, що в 18,5 разів швидше, ніж у RIFFA, тобто 2,25 секунди. Причина цього вдосконалення полягає в тому, що в MQMAI будь-який запит, можливо згенерований з різних потоків, не відхиляється в черзі подання, адресованої потоками, поки в ньому є доступні записи. Таким чином, можна отримати вигоду від ефекту конвеєрної передачі всіх таких невиконаних запитів між хостом і FPGA. Навпаки, RIFFA відхиляє будь-які запити, надіслані до прискорювача, який зайнятий обслуговуванням запиту, таким чином змушуючи програму

повторно надсилати відхилений запит і знову проходити весь стек програмного забезпечення, доки це не вдасться. Таким чином, у RIFFA затримка значно збільшується, коли виникають такі суперечки.

Нагадаємо, що числа в таблиці 3.2 є середньою затримкою. Для 100 зразків, отриманих для випадку T = 4 і K = 20, стандартне відхилення становить 1,633 для RIFFA і 0,017 для MQMAI. Тобто MQMAI дає набагато кращу продуктивність навіть з точки зору найгіршої затримки.

У випадку з MTMA ((Multi-Thread Multi-Accelerator): щоб оцінити здатність до масштабування продуктивності, отриману від нашої «multi-queue структури» MQMAI, ми розподіляємо потоки на чотири ядра.

Таблиця 3.3 показує середню наскрізну затримку, отриману з 4 потоками та 20 запитами в пакеті. У цьому налаштуванні ми припускаємо, що кожен потік працює на своєму виділеному ядрі як для MQMAI, так і для RIFFA. Однак у нашому MQMAI всі потоки мають доступ до всіх чотирьох прискорювачів одного типу (або функції), оскільки контролер MQMAI може динамічно планувати та відправляти кожен вхідний запит до доступного прискорювача.

Таблиця 3.3 - Середня наскрізна затримка в Multi-Thread Multi-Accelerator

Framework	N потоків для кожного прискорювача				Середня затримка (с)
	ACC1	ACC2	ACC3	ACC4	
RIFFA	1	1	1	1	0,03722
	2	1	1	0	0,09141
	2	2	0	0	0,24017
	3	1	0	0	1,16931
	4	0	0	0	2,26002
MQMAI	Динамічне планування в контролері				0,03485

Ця функція віртуалізації кількох прискорювачів одного типу в один віртуальний прискорювач звільняє розробників додатків від відповідальності

вирішувати, який прискорювач використовувати. Навпаки, RIFFA не дозволяє таку віртуалізацію. У RIFFA кожен потік має бути запрограмований вручну, щоб його можна було скерувати до певного прискорювача.

Згідно з таблицею 3.3, RIFFA досягає найкоротшої середньої затримки, тобто 0,0372 секунди, коли кожен потік спрямовується на один виділений прискорювач. Однак у нашому MQMAI такі визначені користувачем вказівки не потрібні, оскільки контролер MQMAI динамічно планує всі запити для всіх прискорювачів. У гіршому випадку спрямування запитів у RIFFA, тобто всі потоки спрямовують свої запити до одного прискорювача, RIFFA накладає 60-кратну затримку порівняно з MQMAI.

Накладні витрати на ресурси.

Таблиця 3.4 порівнює використання ресурсів FPGA MQMAI та RIFFA, коли розмір запиту для прискорювачів становить 1 МБ. Крім того, використання ресурсів JetStream наводиться на основі цифр у [11].

Таблиця 3.4 - Використання ресурсів на Xilinx Virtex 7 FPGA для 4 каналів введення/виведення

Блоки пам’яті	Загальні ресурси	JetStream	RIFFA	MQMAI
LUT	433200	8571 (1,97%)	26433 (6,1%)	7883 (1,82%)
LUTRAM	174200	392 (0,22%)	2464 (1,41%)	456 (0,26%)
FF	866400	6955 (0,8%)	39720 (4,58%)	12196 (1,41%)
BRAM	1470	NA	83 (5,6%)	95 (6,4%)
DSP	3600	0 (0%)	0 (0%)	0 (0%)

Порівняно з RIFFA, MQMAI використовує на 70% менше логіки (LUT), на 81% менше LUTRAM і на 69% менше IP-операцій. Порівняно з JetStream, MQMAI використовує трохи менше LUT, але використовує трохи більше тригерів і LUTRAM. Вважайте це, як ми вже згадували раніше, JetStream наразі недоступний, і представлені цифри дають певні ідеї. З іншого боку, ці

цифри будуть змінюватися через зміну максимально можливого обсягу передачі даних.

MQMAI використовує на 13% більше BRAM (блоків пам’яті) порівняно з RIFFA. MQMAI вимагає більше BRAM для реалізації черги команд. Як наслідок, з інфраструктурою MQMAI є більше невикористаних ресурсів FPGA, які можна використовувати для більш (складних) прискорювачів.

3.4 Висновки з розділу

У цьому розділі ми дослідили роботу багаточергового інтерфейсу даних на основі команд для доступу до мультиприскорювачів на FPGA. Як результат - запропонований фреймворк є неблокуючим інтерфейсом із зменшенням конкуренції за ресурси завдяки програмній архітектурі вводу-виводу з кількома чергами для збільшення доступу до паралельного вводу-виводу. Контролер прискорювача, реалізований на FPGA, керує командами та розподіляє прискорювачі за допомогою циклічної політики. Накладні витрати запропонованої структури як на програмний стек, так і на апаратний контролер невеликі.

Запропонована нами структура є більш масштабованою та забезпечує загальне покращення затримки у 18 разів порівняно з найсучаснішою RIFFA.

4 АПАРАТНЕ ПРИСКОРЕННЯ ОБЧИСЛЕНЬ НА FPGA З ВИКОРИСТАННЯМ OPENCL

4.1 Відкрита мова обчислень (OpenCL)

OpenCL (Open Computing Language) — стандартний фреймворк від Khronos Group дозволяє паралельно програмувати гетерогенні системи, які можуть включати центральні процесори, графічні процесори, DSP і FPGA. Мова програмування OpenCL заснована на C і C++. Він підтримує програмування пристроїв на гетерогенній платформі, а також інтерфейс прикладного програмування (API) для керування зв’язком між обчислювальними пристроями. OpenCL надає стандартний інтерфейс і функції API для підтримки паралельних обчислень за допомогою паралелізму на основі завдань і даних. У цьому розділі ми надаємо різні концепції та технологію, необхідні для розуміння моделі програмування OpenCL [2].

Архітектура OpenCL складається з ієрархії апаратних одиниць, узгодженої моделі виконання та моделі пам’яті, яка підтримує паралельне виконання коду в гетерогенних системах.

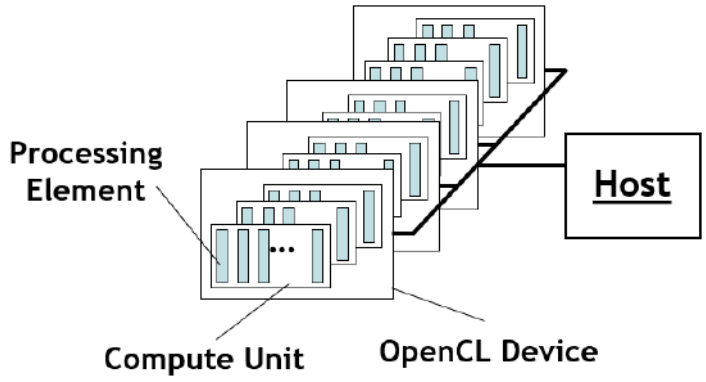


Рисунок 4.1 - Модель платформи OpenCL

4.1.1 Апаратна платформа OpenCL

OpenCL розглядає обчислювальну систему як ряд обчислювальних одиниць, керованих головним процесором. Апаратне забезпечення OpenCL поділяється на різні рівні абстракції (рис. 4.1). На верхньому рівні розташований центральний процесор на основі центрального процесора, який підключено до одного або кількох пристроїв OpenCL (наприклад, центрального процесора, графічного процесора або FPGA). Кожен пристрій OpenCL складається з одного або кількох обчислювальних блоків (CU), які далі поділяються на один або кілька елементів обробки (PE) [3].

Обчислення виконуються в елементах обробки. Кілька пристроїв PE, CU та OpenCL можуть працювати паралельно. Хост керує розподілом робочого навантаження та передачею даних між блоками за допомогою OpenCL API.

4.1.2 Модель виконання OpenCL

Програми OpenCL мають два блоки виконання: програму ядра та програму хоста. Хост-програма OpenCL виконується на хості. Хост-програма використовує функції OpenCL API, щоб запитувати статус пристроїв Compute і керувати робочими навантаженнями між блоками Compute. Тоді як програма ядра виконується на елементах обробки (PE). Програми ядра виконують обчислювальну частину програми OpenCL.

Весь простір даних, який потрібно виконати, називається NDRange, де N позначає кількість вимірів вхідних даних, які можуть бути одним, двома або трьома. NDRange розкладається на робочі групи та робочі елементи (рисунок 4.2). Кожен незалежний елемент виконання в усьому робочому навантаженні називається робочим елементом. Набір робочих елементів згруповано в робочу групу. Декомпозицію вказують змінні розміру (N), глобального розміру та локального розміру. Глобальний розмір визначається як загальна

кількість глобальних робочих елементів у NDRange для заданого розміру. Локальний розмір визначається як кількість локальних робочих елементів у робочій групі для заданого розміру. Рисунок 4.3 який ілюструє декомпозицію вигляду зверху 1-вимірного та 2-вимірного NDRange. Робочі групи можна виконувати паралельно та розподіляти між обчислювальними блоками. Усі робочі елементи, що належать одній робочій групі, обчислюються різними елементами обробки (PE) одного обчислювального блоку.

4.1.3 Модель пам'яті OpenCL

Ієрархія пам’яті OpenCL структурована для підтримки спільного використання даних і синхронізації робочих елементів. Модель пам’яті OpenCL складається з глобальної пам’яті, локальної пам’яті, приватної пам’яті та пам’яті хоста (показано на рисунку 4.4)

- Пам'ять хоста: пам'ять, безпосередньо доступна для хоста. Інші обчислювальні пристрої не можуть отримати доступ до пам'яті хоста.

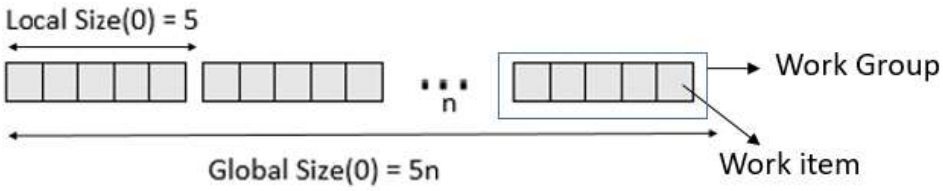
- Глобальна пам’ять: до глобальної пам’яті можуть отримати доступ усі обчислювальні блоки (compute units - CU), елементи обробки (processing elements -PE) і хост. Таким чином, він спільний для всіх робочих елементів і робочих груп. Глобальна затримка пам’яті може бути високою, тому бажано кешувати глобальну пам’ять у локальну пам’ять, коли це можливо. Частина глобальної пам’яті, яка є постійною під час виконання, називається постійною пам’яттю. Тут оголошено постійні змінні, оскільки це дозволяє доступ лише для читання для обчислювальних блоків.

- Локальна пам’ять: кожен обчислювальний блок має унікальну локальну пам’ять, до якої інші CU не можуть отримати доступ. Ця область пам’яті є локальною для робочої групи, і її можуть спільно використовувати робочі елементи в одній робочій групі. Змінні, визначені в локальній пам’яті, невидимі для інших робочих груп.

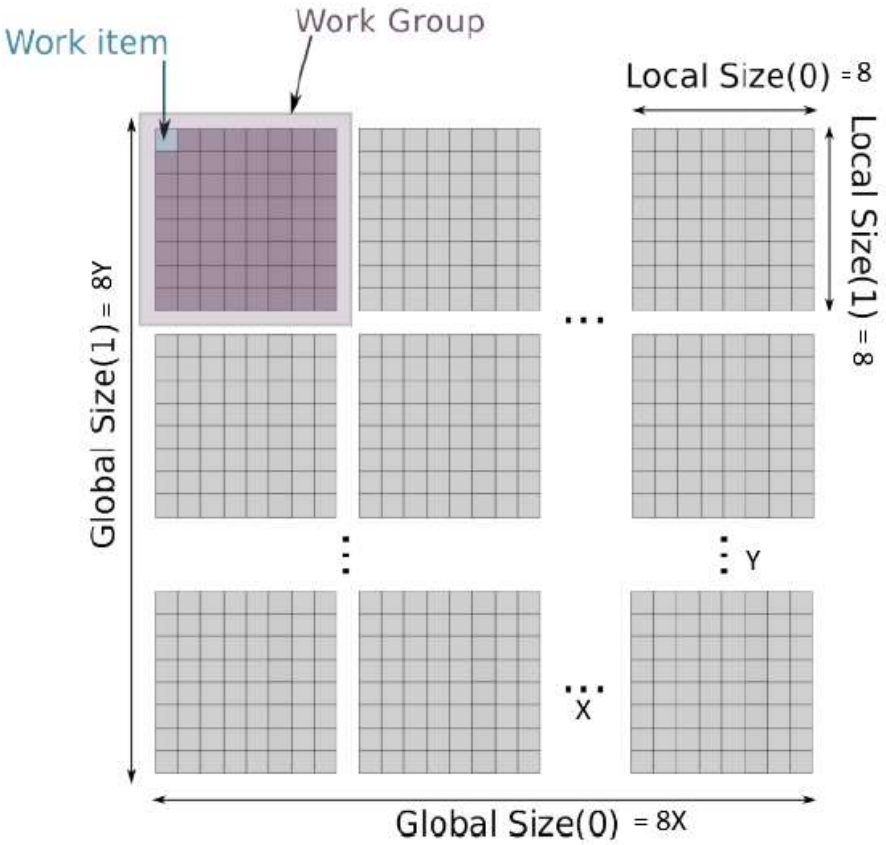
- Приватна пам’ять: кожен робочий елемент має приватну пам’ять і не

відображається для інших робочих елементів.

-



а) 1- вимирний



б) 2- вимірний

Рисунок 4.3 - (a) Приклад поділу робочої групи та робочого елемента для 1 виміру. (b) Приклад розподілу робочої групи та робочого елемента для дво-вимірного [5]

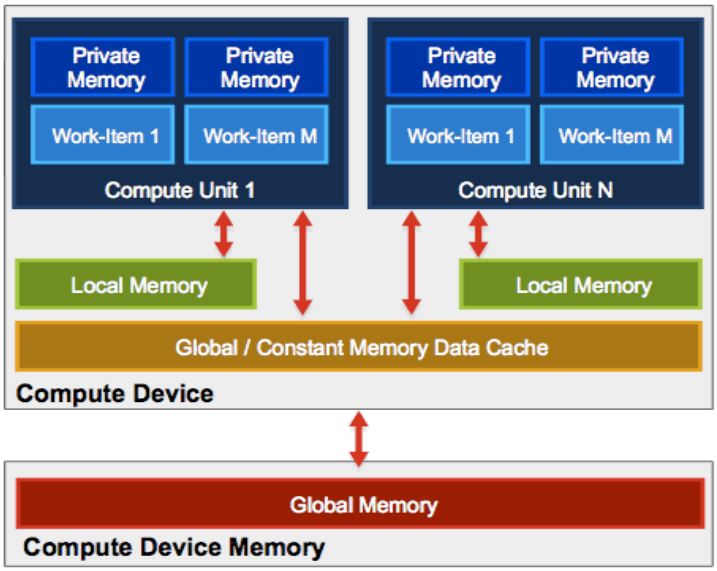


Рисунок 4.4 - Модель пам'яті OpenCL

Важливо, щоб операції читання та запису в локальній і глобальній пам’яті були синхронізовані, оскільки вони є спільною пам’яттю. Це робиться за допомогою функцій синхронізації пам’яті, таких як Barriers і Memory fences.

4.1.4 Програмування OpenCL

Для більшості паралельних додатків послідовна частина виконується в головній програмі, а частини паралельних обчислень спільно використовуються обчислювальними блоками, запрограмованими за допомогою програм ядра. Крім того, головна програма повинна керувати та координувати обчислювальні блоки, які працюють паралельно. Для цього використовуються функції OpenCL API (інтерфейс прикладного програмування).

Функції OpenCL API дозволяють користувачеві створювати та визначати такі об’єкти, як контекст, черга команд і буфери даних. Контекст визначає середовище, в якому виконується ядро. Черга команд визначається для кожного пристрою OpenCL. Ці черги дозволяють хосту віддавати команди та взаємодіяти з пристроєм. Буфери використовуються для передачі даних між

хостом і пристроєм OpenCL під час виконання.

4.2 Апаратне та програмне забезпечення системи

Усі експерименти проводилися на платі Terasic DE1-SoC, яка складається з апаратної платформи проектування, побудованої на Altera SoC FPGA Cyclone V. Cyclone V інтегрує жорстку процесорну систему (HPS) на базі ARM, що складається з процесора, периферійних пристроїв та інтерфейсів пам'яті, бездоганно пов'язаних із структурою FPGA за допомогою високосмугового з'єднання. Цей інтерфейс забезпечує зв'язок між процесором ARM і FPGA.

Система зображена на рисунку 4.5.

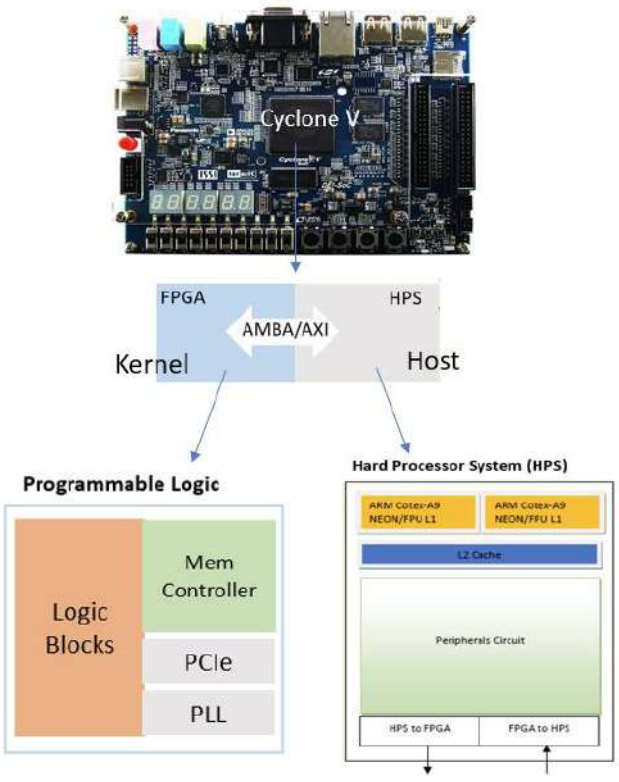


Рисунок 4.5 - Системне обладнання Terasic DE1-SoC з Altera Cyclone V

Ми використовували термінал Linux на комп'ютері для взаємодії з пла-

тформою FPGA. Команди до плати FPGA користувач може надавати з терміналу Linux на комп'ютері. Система об'єднана з картою micro SD, інстальованою з образом Linux, який дозволяє запускати команди Linux на платі FPGA. Кабель USB-UART використовується для зв'язку між платою та хостом Linux на комп'ютері. Ми використовували порт LAN на платі для передачі файлів між платою та комп'ютером.

Ми використовували таке програмне забезпечення для компіляції програм OpenCL і синтезу на основі FPGA:

- Altera FPGA SDK для OpenCL (AOCL). Altera FPGA OpenCL SDK (Software Development Kit) — це інструмент HLS, який передбачає складну процедуру перетворення високорівневого коду OpenCL у файли конфігурації обладнання. Він також дозволяє функціональну перевірку шляхом емуляції коду прискорювача OpenCL на хості x86. Інструмент надає звіт про оптимізацію з інформацією про залежності в конвеєрі, зупинки пам'яті та інформацію про час проекту. Цей звіт дає уявлення про синтезований дизайн, який допомагає модифікувати дизайн для продуктивності. Процеси створення звітів компіляції, OpenCL до HDL, емуляції та оптимізації можуть зайняти лише кілька секунд. Тоді як повний процес синтезу для створення файлу конфігурації апаратного забезпечення (.aocx файлу) з OpenCL може тривати 1-5 годин залежно від складності конструкції. Кінцевий файл процесу синтезу з розширенням .aocx використовується для конфігурації FPGA.

- Altera Quartus II. Інструмент Altera Quartus використовується для конфігурації цільового пристрою FPGA з HDL (Hardware Description Language), наданого користувачем. Цей інструмент дозволяє компілювати проекти, виконувати аналіз часу та конфігурувати цільовий пристрій. Цей інструмент потрібен компілятору AOCL для створення синтезованого файла .aocx

- Altera SoC Embedded Design Suite. Altera SoC FPGA Embedded Development Suite (SoC EDS) — це інструмент для розробки вбудованого програмного забезпечення на Intel SoC FPGA. Ми використовували цей ін-

струмент для компіляції хост-програми OpenCL і створення виконуваного файлу ARM. Цей виконуваний файл виконується вбудованим процесором ARM у системі.

4.2.1 Архітектура системної пам'яті

Система складається з трьох типів пам'яті (рисунок 4.6), кожен з них описаний нижче:

1) Глобальна пам'ять. Глобальна пам’ять — це пам’ять DDR (подвійна швидкість передачі даних) на мікросхемі. Передача даних між хостом і ядром може здійснюватися лише через глобальну пам'ять. Тому весь інтерфейс хост-ядро відбувається в глобальній пам’яті. Він пропонує високу місткість і високу затримку. Глобальна пам'ять доступна для всіх робочих груп.

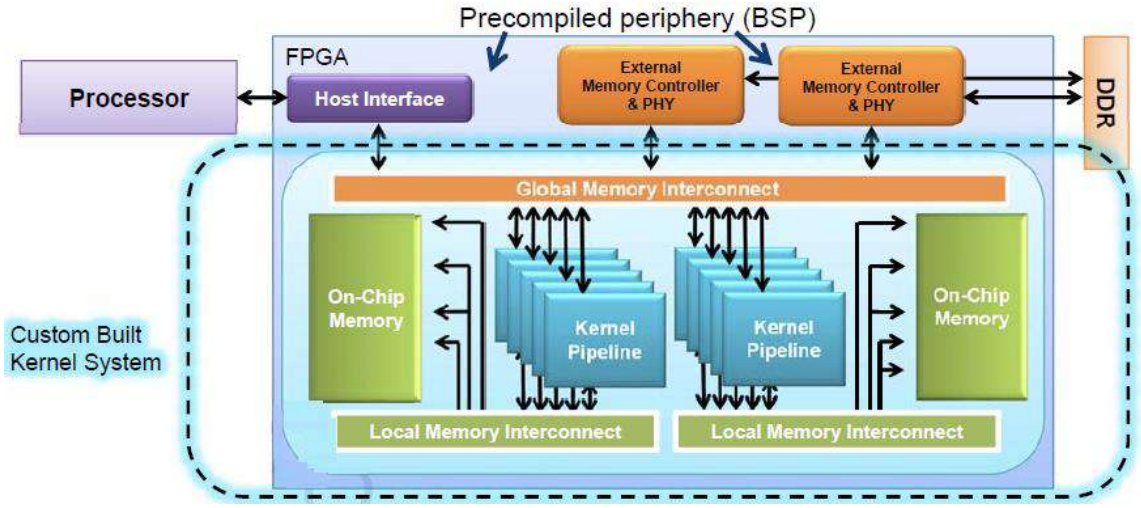


Рисунок 4.6 - Архітектура системної пам'яті: глобальна, локальна та приватна пам'ять

2) Локальна пам'ять. Локальна пам'ять - це пам'ять на чіпі, вона лежить на тому ж чіпі, що і HPS і FPGA. Це пропонує вищу пропускну здатність і меншу затримку порівняно з глобальною пам’яттю. Локальна пам'ять спільно використовується між робочими елементами однієї робочої групи.

3) Приватна пам'ять. Приватна пам'ять використовує регістри та блокову оперативну пам'ять на FPGA. Він пропонує продуктивність за ціною площі. Приватна пам’ять є унікальною для кожного робочого елемента.

4.3 Програмування системи

Більшість програм мають два типи сегментів коду. Сегменти, які потрібно виконати послідовно, і сегменти, які можна виконати паралельно.

У той час як хост-програма може бути скомпільована, а хост-виконуваний файл створений Altera SoC Embedded Design Suite. Хід системи програмування з використанням хосту та програм ядра OpenCL показано на рисунку 4.7.

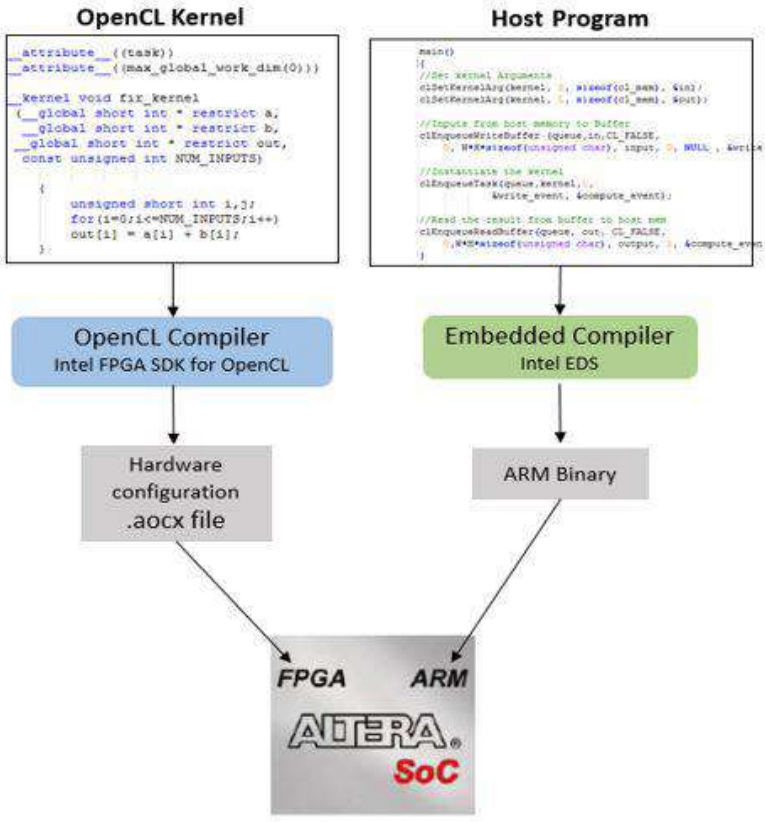


Рисунок 4.7 - Огляд налаштування системи з кодом OpenCL

Як правило, послідовні сегменти є ефективними для процесорів загаль-

ного призначення (тобто центрального процесора) з конвеєрністю, виконанням поза порядком, ієрархією пам’яті тощо.

Паралельні сегменти є ефективними для платформ, які підтримують паралельну обробку, таких як багатоядерні процесори, графічні процесори або FPGA. Не тільки паралельні завдання, FPGA також може досягти продуктивності в обчислювально інтенсивних послідовних завданнях, розробивши спеціальне обладнання для обчислення цих завдань.

SoC FPGA, будучи гетерогенною платформою, забезпечує продуктивність як процесорної системи (ARM), так і реконфігурованої структури (FPGA). Послідовні сегменти можуть бути виконані процесором ARM, тоді як паралельні сегменти даних і інтенсивні обчислювальні сегменти можуть бути виконані спеціальним апаратним забезпеченням на FPGA. Тут завдання полягає у встановленні комунікаційного інтерфейсу між HPS і FPGA. Крім того, нам потрібна спільна мова для програмування як систем, так і інтерфейсу.

Тому для задоволення цих вимог розроблено мову програмування OpenCL. OpenCL підтримує розробку програм, які виконуються на різнорідних одиницях платформи. OpenCL широко використовується для паралельної обробки, оскільки дозволяє налаштувати комунікаційний інтерфейс між блоками як для паралелізму завдань, так і для паралелізму на основі даних. Більшість з них - останні апаратні пристрої та інструменти компілятора від різних постачальників підтримують виконання OpenCL.

4.3.1 Хост і ядро

Програма OpenCL розглядає обчислювальну систему як комбінацію головного процесора (зазвичай ЦП) і набору прискорювальних пристроїв OpenCL, які можуть працювати паралельно, вони називаються ядрами. Головний пристрій відповідає за передачу функцій обчислювальним блокам. і керування інтерфейсом передачі даних.

У цьому випадку процесор ARM є головним процесором, а FPGA є пристроєм ядра. Кожна програма OpenCL має два набори програм, програму Host і програму OpenCL Kernel для програмування хоста та ядер відповідно. Цілком можливо, що пристрій ядра (FPGA) може мати кілька обчислювальних блоків, що працюють паралельно. Код ядра можна скомпільувати, емулювати та синтезувати за допомогою Altera FPGA OpenCL SDK.

4.3.2 Хост-програма та процес розробки ядра пристрою

Програма Host — це програма OpenCL із бібліотечними функціями на основі C/C++, скомпільована програмою Embedded Design Suite для створення виконуваного файла ARM. Код ядра містить стандартний OpenCL і скомпільований Altera OpenCL FPGA SDK для створення файлів конфігурації обладнання (.aocx), файлів RTL, звітів HTML, звітів про синтез та сценаріїв тощо.

Рисунок 4.8 показує метод розробки ядра для програми. Ці етапи включають функціональну перевірку, оптимізацію та покращення продуктивності ядра. Кожен етап надає різну інформацію про дизайн, яка допомагає оптимізувати апаратне ядро. Нижче наведено опис кожного кроку.

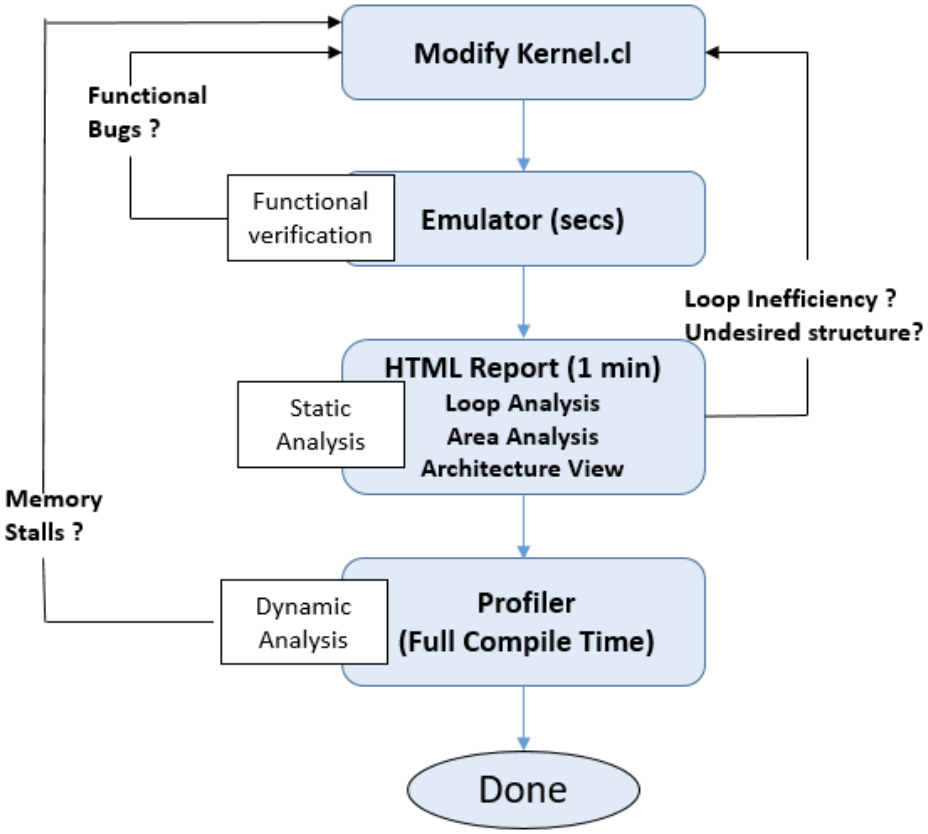


Рисунок 4.8 - Потік розробки функції ядра

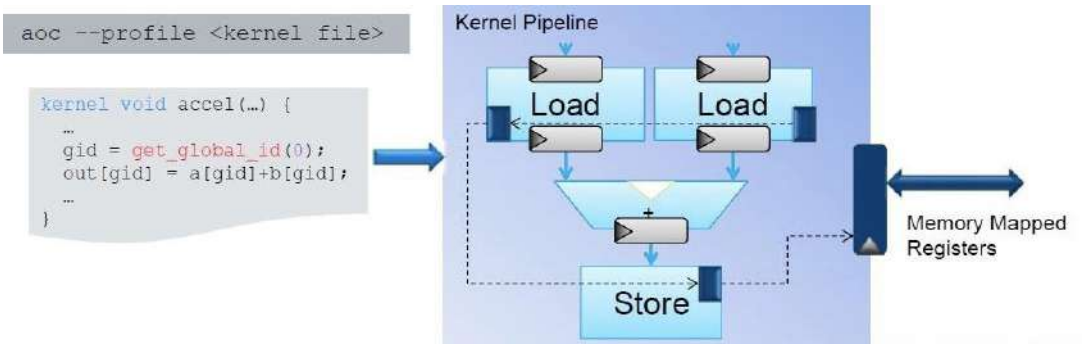


Рисунок 4.9 - Аналіз за допомогою Profiler, ядро, інтегроване з Flip-flops при завантаженні та збереженні транзакцій

1) Функціональна перевірка: емуляція та налагодження. Цей крок використовується для функціональної перевірки та налагодження програм хоста та ядра за короткий час перед початком повної компіляції. За допомогою емулятора AOCL за лічені секунди створюється файл .aocx, який емулює фу-

нкціональність ядра. Це файл .aocx може виконуватися на хості x86-64 Windows або Linux. Для підтримки налагодження оператори друку можна використовувати в ядрі, а також є інструменти налагодження, такі як Intel Code Builder, які дозволяють установлювати точки зупину, читати проміжні змінні тощо. Емуляція схожа на функціонально, але не забезпечує правильний час виконання. Насправді він може виконуватися з нижчою швидкістю, ніж на фактично оптимізованому обладнанні.

2) Статичний аналіз: аналіз звіту про оптимізацію. Компілятор автоматично створює файл звіту HTML, який дає огляд архітектури системи, яка буде синтезована. Він складається з аналізу циклу, звіту про зону та архітектурного вигляду системи. Інформацію зі звіту можна використати для зміни коду ядра для підвищення продуктивності.

- Звіт про аналіз циклу: цей звіт показує статус кожного циклу в додатку. Він надає статус розгортання циклу if, стан конвеєра та інтервал ініціації (II) кожного циклу конвеєра. Ініціальний інтервал визначається як кількість циклів, протягом яких усі нові дані надходять у конвеєр. Можна виявити будь-яке вузьке місце через залежність даних або зупинку пам’яті.

- Звіт про аналіз території: компілятор надає звіт про використання ресурсів з ієрархічною інформацією. Тобто кількість ресурсів, які використовуються кожним рядком у вихідному коді та кожним блоком у вихідному коді. Таким чином, дозволяє нам зобразити, яка частина коду використовує найбільше ресурсів (наприклад, LUT, пам’ять тощо). Як правило, складні операції з даними, операції з точкою вівса споживають більше логічних ресурсів, а розгорнуті цикли споживають більше пропускної здатності пам’яті.

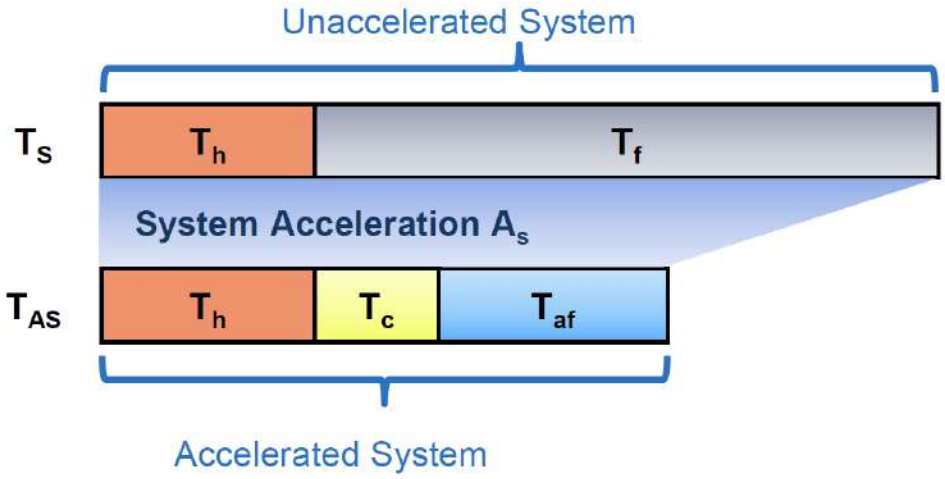


Рисунок 4.10 - Часова шкала неприскореної системи проти прискореної системи

- Перегляд системи: ця частина звіту відображає блоки в системі та реалізацію доступу до пам'яті. Він представляє інформацію про всі зупинки завантаження/зберігання та затримку як для глобальної, так і для локальної пам’яті.

3) Динамічний аналіз: Профайлер. Цей крок передбачає аналіз продуктивності ядра під час виконання. У цьому процесі додаткові лічильники продуктивності додаються при кожному завантаженні та зберігаються в дизайні, як показано на рисунку 4.9 вище. Ці лічильники збирають дані про продуктивність ядра та надсилають програмі хосту. Цю інформацію можна переглянути через GUI. Профілювання зазвичай використовується для визначення того, який канал у конвеєрі спричиняє зупинки пам’яті.

4.4 Оптимізація дизайну для прискорення

У цьому розділі ми обговорюємо методи та архітектури для підвищення ефективності обробки даних на FPGA та різні показники оптимізації, які впливають на продуктивність. Згідно з часовою діаграмою на рисунку 4.10 і позначення часу в табл. 4.1, обговорюються основні фактори, за допомогою

яких можна досягти прискорення.

1) Зменшити T_{AF}: збільшити кількість паралельних операцій. Цього можна досягти збільшенням кількості паралельних операцій.

Таблиця 4.1 - Позначення часу

Позначення часу	Опис
Ts	Загальний час, витрачений на систему без прискорення
T_h	Час, витрачений на неприскорену систему для головної частини
T_f	Час, витрачений у неприскореній системі для функції без прискорення
T_{AS}	Загальний час, витрачений на прискорену систему
T_C	Час, витрачений на спілкування між хостом і прискорювачем
T_{AF}	Час, витрачений на прискорену систему для прискореної функції
A_S	Системне прискорення = T_s/T_{AS}

- Паралелізм рівня даних: у програмах, де дані можна розділяти та обчислювати незалежно, можна розгорнути паралелізм рівня даних. Це робиться шляхом реплікації обчислювальних блоків, векторизації блоків обробки даних у ядрі та їх паралельного запуску. Подібним чином ітерації циклу можуть виконуватися паралельно розгортанням циклу.

- Паралелізм рівня інструкцій: паралелізм рівня інструкцій може бути досягнутий шляхом конвеєрної обробки програми ядра. Залежності між інструкціями можна обробляти за допомогою зворотного зв’язку даних.

- Паралелізм рівня завдання: набір незалежних завдань можна запускати паралельно на різних ядрах. Паралелізм рівня завдань досягається шляхом поділу великої проблеми на окремі пристрої ядра та їх паралельного запуску.

2) Зменшити T_c: Зменшення витрат на спілкування: щоб зменшити за-

тримку зв’язку між хостом і ядром, можна застосувати такі методи, щоб зменшити глобальний доступ до пам’яті. Система дозволяє зв'язок хоста з ядром лише через глобальну пам'ять. Через високу глобальну затримку пам’яті глобальний доступ до пам’яті з ядра має бути зменшений, коли це можливо:

- Шляхом переміщення часто використовуваних даних у локальну пам’ять або вбудовану пам’ять.

- Використання каналів і каналів для обміну даними між ядрами замість глобальної пам'яті для передачі даних між ними. Рисунок 4.11 порівнює зв’язок між ядром через глобальну пам’ять і канали/канали.

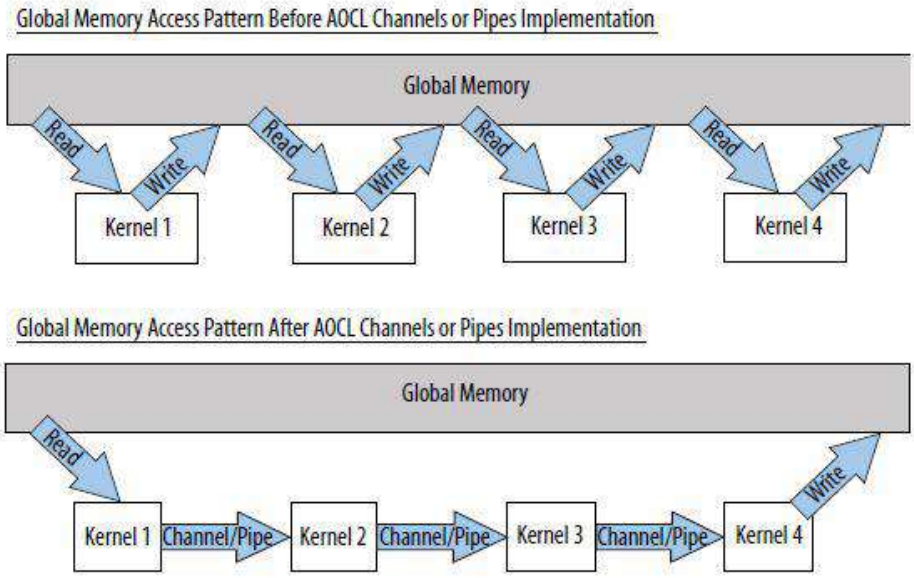


Рисунок 4.11 - Зв'язок між ядрами: використання глобальної пам'яті проти каналів [8]

4.4.1 Архітектури оптимізації

Компілятор Altera SDK для OpenCL (AOCL) пропонує дві різні архітектури обробки даних для ядер, це виконання одного робочого елемента та виконання ядра NDRange. Залежно від застосування, один або комбінація обох може бути обрана для дизайну ядра на FPGA. На ранній стадії планування

дизайну слід вирішити, чи створювати ядро OpenCL як ядро з одним робочим елементом або як ядро NDRange. Обидві архітектури детально описані нижче.

1) Ядра одного робочого елемента (Single Work Item Kernels).

Уся програма ядра виконується як один потік. У цьому режимі виконання одного робочого елемента ядро складається з одного обчислювального блоку, який виконує робочі елементи послідовно одна за одною. Якщо ядро визначено з атрибутом «завдання», то компілятор автоматично синтезує його як єдине ядро виконання робочого елемента.

У цих конструкціях пропускна здатність досягається за допомогою методу під назвою Loop Pipelining, показаного на рисунку 4.12. У цьому методі компілятор визначить конвеєрне паралельне виконання через ітерації циклу. Тобто структура циклу є конвеєрною, і кілька ітерацій циклу виконуватимуться одночасно в конвеєрі. Рисунок 4.13 показує приклад ядра OpenCL, реалізованого як ядро з єдиним робочим елементом, він показує, що коли робочий елемент 1 знаходиться на етапі додавання, робочий елемент 2 знаходиться на етапі завантаження в конвеєрі.

Для ядер з одним завданням пропускна здатність в основному залежить від фактора, який називається Ініціальний інтервал (II). Визначається як кількість циклів, протягом яких запускаються всі нові дані в pipeline. Він визначає частоту запуску ітерацій циклу. Тому мінімізація II є ключем до продуктивності ядра окремого робочого елемента (формула 4.1). Найкраща продуктивність досягається, коли нові дані запускаються кожного циклу (тобто II=1). Вищий II можна зменшити, пом’якшивши залежності, що передаються циклом, за допомогою таких методів, як перетворення вкладених циклів в одиночний цикл, використання локальних змінних замість глобальних змінних тощо.

$$\text{Exec time} = ((\text{Num iterations} * \text{II}) + (\text{Loop latency})) * \text{Time period} \qquad (4.1)$$

Залежності даних у послідовних ітераціях циклу будуть вирішені шляхом включення зворотного зв’язку даних на апаратне забезпечення FPGA, як показано на рисунку 4.13 тут, кожна ітерація залежить від значення C[i], записаного в попередній ітерації. Таким чином, обчислений C[i] передається через зворотний зв’язок до наступної ітерації. Тож наступну ітерацію можна запустити, щойно залежність буде очищено.

Цей режим є ідеальним, коли між робочими елементами існує залежність і між робочими елементами неможливий паралелізм на рівні даних. Тому в тих випадках, коли складно розділити дані між паралельними робочими елементами або якщо дані потрібно спільно використовувати робочі елементи, виконання окремого робочого елемента забезпечує кращу пропускну здатність.

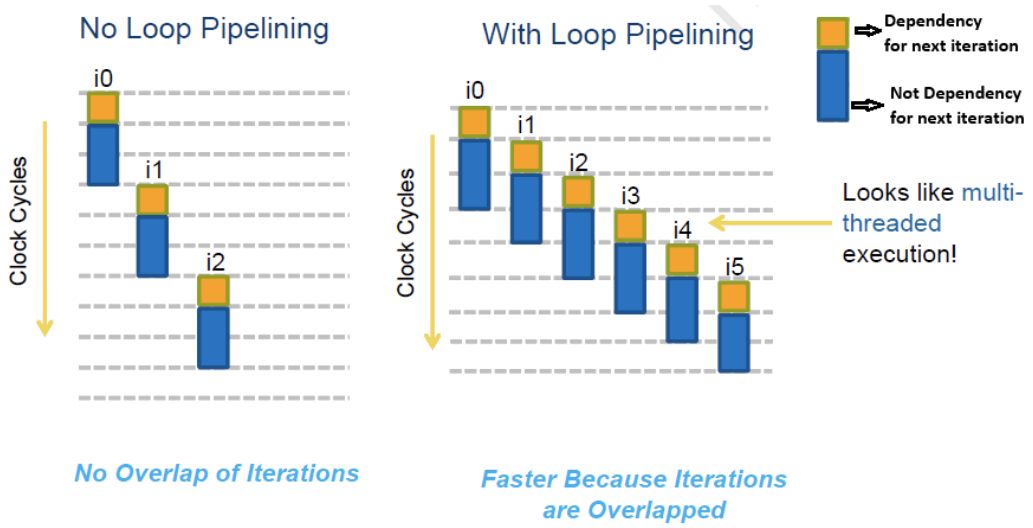


Рисунок 4.12 - Ітерації конвеєрної обробки без циклу проти конвеєрної обробки циклу

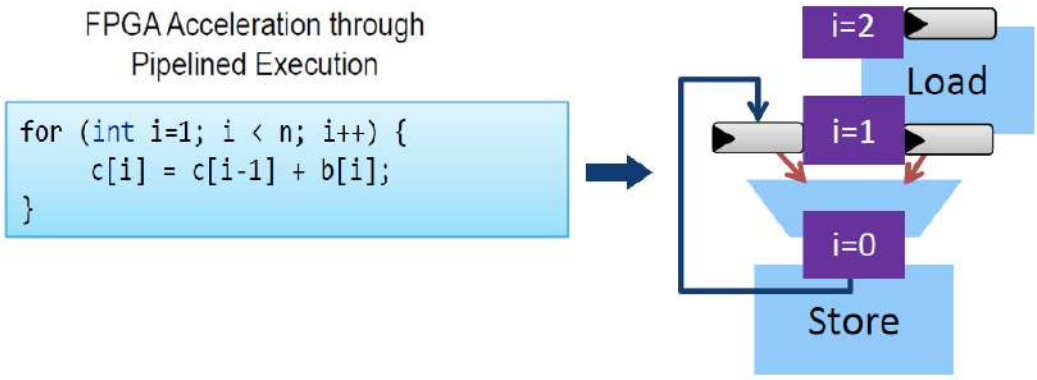


Рисунок 4.13 - Реалізація одного робочого елемента, що показує зворотний зв’язок даних для обробки залежності

Наприклад, FIR-фільтр, інтерполяційний фільтр, децимаційний фільтр добре використовувати як ядра окремого робочого елемента, оскільки нові виходи залежать від попередніх вхідних даних і вихідні дані.

2) Ядра NDRange.

У ядрі NDRange ефективність обробки даних досягається шляхом впровадження паралелізму рівня даних. Апаратне забезпечення копіюється, щоб забезпечити можливість паралельного виконання. Ядро NDRange може мати кілька обчислювальних блоків (CU) і кілька блоків обробки даних з однією інструкцією (SIMD), які одночасно працюють над різними робочими групами або паралельними робочими елементами.

Досягає пропускної здатності за рахунок паралельної обробки за рахунок пропускної здатності пам’яті та використання логіки.

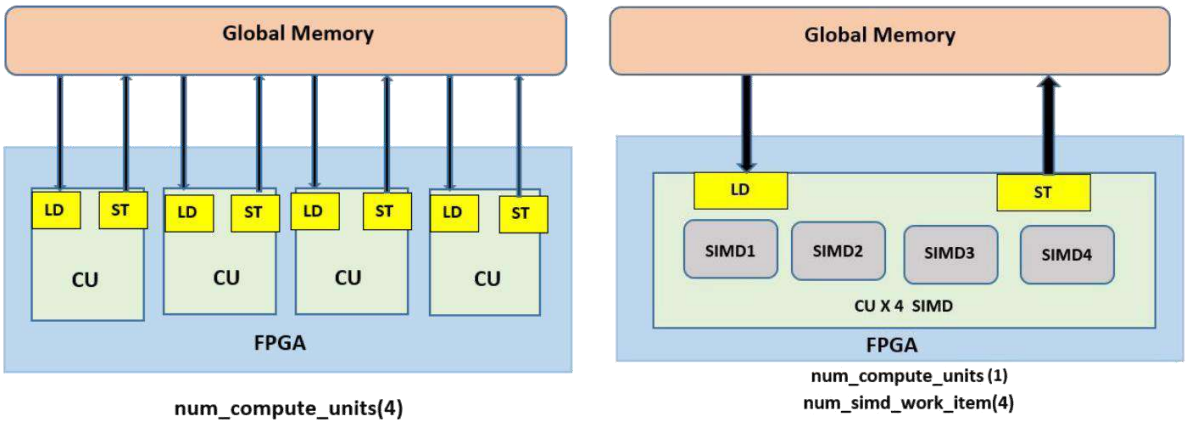
Програма може бути реалізована як ядро NDRange, якщо програма ядра не має циклу чи залежності пам’яті між робочими елементами. Наприклад, додатки шифрування AES, множення матриці тощо.

CU працюють над різними робочими групами паралельно, тоді як SIMD працюють з різними робочими елементами, які належать одній робочій групі. У дублюванні обчислювальної одиниці всі одиниці шляху даних, шляху керування та блоки доступу до пам'яті реплікуються. На відміну від цього, для векторизації SIMD дублюється лише шлях даних обчислювального бло-

ку, а логіка керування є спільною.

Рисунок 4.14а та 4.14б відображує різницю між реплікацією обчислювального блоку та векторизацією SIMD. Хоча обидва пропонують пропускну здатність, вони забезпечують різну ефективність. Кілька обчислювальних блоків споживають більше логічного використання та створюють небажані моделі доступу до пам'яті, коли вони працюють з даними з різних робочих груп. Векторизація SIMD дозволяє об'єднувати доступи до пам'яті та є ефективною для області.

Тому в більшості випадків векторизація SIMD є кращою перед кількома обчислювальними блоками. У кількох випадках, коли робочі елементи мають різну логіку керування та не можуть використовувати один елемент керування path кілька обчислювальних одиниць використовуються для паралелізму.



(а) (б)

Рисунок 4.14 - (а) Ядро з 4 обчислювальними блоками; (б) Ядро з 1 обчислювальним модулем і 4 модулями SIMD на обчислювальний блок

4.4.2 Прагми та атрибути оптимізації

Додавання наступних атрибутів і прагм до коду ядра OpenCL дозволяє компілятору відповідним чином змінювати мікроархітектуру ядра. Кожен

має свої переваги та недоліки над ефективністю. Змінюючи ці атрибути в різних комбінаціях, ми можемо створювати різні апаратні конфігурації. Ці атрибути по-різному впливають на використання логіки, пропускну здатність пам’яті, шаблони доступу до пам’яті та продуктивність.

1) num_compute_units(N): цей атрибут дублює N обчислювальних одиниць у ядрі. Це дублює шлях керування та шлях даних. Ці CU працюють паралельно в різних робочих групах, через що можливо, що вони мають ширші моделі доступу до пам’яті.

2) num_simd_work(N): це створює N кількість SIMD на кожну обчислювальну одиницю. Значення N обмежене 2, 4, 8 і 16. Це дублює лише шлях даних, шлях керування є спільним. SIMD працюють на робочих елементах однієї робочої групи. Таким чином, шаблони доступу до пам'яті можуть бути об'єднані.

Таблиця 4.2 - Опис системи бенчмарків

Еталон	Ядро	Pipeline статус	II	$F_{макс}$ (МГц)	Логіка (%)	Найвище прискорення
Sobel	Одиничне завдання	Так	2	136	20	7.3
FIR	Одиничне завдання	Так	1	168	20	0,95
ADPCM	Одиничне завдання	Так	40	158	20	2.6
Децимація	Одиничне завдання	Так	1	129	81	2.8
Інтерполяція	Одиничне завдання	Так	1	125	28	2.8
AES	NDRage CU=2, SIMD=2	Немає	N/A	135	84	6.9

3) #pragma unroll < N >: ця Pragma розгортає наступний цикл N разів. Збільшує вимоги до пропускної здатності пам’яті.

4) max_work_group_size(N): Цей атрибут N визначає максимальний розмір робочої групи, яку має виконувати ядро.

5) reqd_work_group size(x; y; z): цей атрибут визначає точний розмір робочої групи, яку має виконувати ядро у трьох вимірах.

4.5 Контрольні показники та результати прискорення

4.5.1 Тести OpenCL

Ми розробили шість тестів прискорення, включаючи фільтр Sobel, фільтр FIR, фільтр ADPCM, фільтр децимації, інтерполяцію та шифрування AES у OpenCL. Ці тести розроблено з набору S2CBenchmark, які створюються в SystemC. Усі контрольні тести написані на OpenCL за допомогою хост-програми OpenCL на основі C і стандартного OpenCL.

Діаграма на рисунку 4.15 відображає співвідношення між часом, витраченим на обчислення, і часом, витраченим на зв'язок між хостом і ядром. Для всіх тестів більша частина часу виконання витрачається на передачу даних між хостом (ARM) і ядром (FPGA).

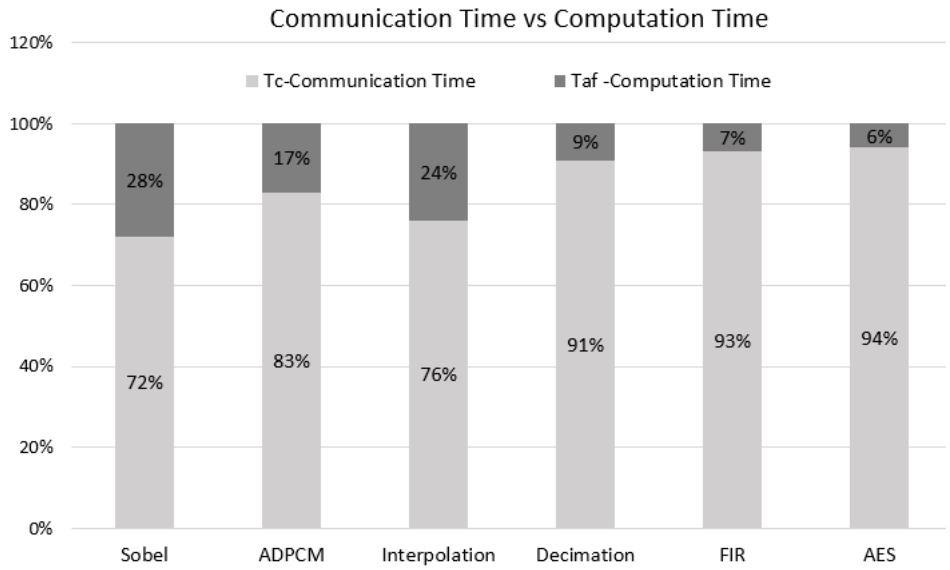


Рисунок 4.15 - Комунікація проти часу обчислень у прискореній системі (ARM+FPGA)

Ядро. Основною метою цих тестів є прискорення їх виконання на системах SoC FPGA. Ці тести переносяться на більшість Altera SoC-FPGA з незначними модифікаціями або без них. Ці контрольні тести можна використовувати для створення різних апаратних мікроархітектур шляхом зміни атрибутів і параметрів.

- Sobel: Sobel - це фільтр зображень 3X3 із розпізнаванням країв, розроблений для 8-розрядних зображень .bmp.
- FIR: Фільтр кінцевої імпульсної характеристики на 10 натискань.
- ADPCM - адаптивний кодер диференційної імпульсно-кодової модуляції для 16-бітових зразків імпульсно-кодової модуляції (PCM).
- Децимація - 5-ступеневий децимаційний фільтр. Він складається з 5 каскадних FIR-фільтрів, де вихід одного фільтра подається на наступний фільтр.
- Інтерполяція - 4-ступеневий інтерполяційний фільтр.
- Шифрування AES: 16-бітне шифрування даних і 128-бітне шифрування AES.

У таблиці 4.3 представлені значення прискорення, отримані для різних мікроархітектурних конфігурацій еталонного дизайну AES. AES розроблено як ядро NDRange, у якому робочі групи даних можуть виконуватися паралельно. Змінюючи атрибути, скажімо, кількість CU, SIMD і розмір робочої групи, продуктивність дизайну впливає на зміну ефективності даних, доступ до пам’яті, через що змінюється прискорення.

Ми зібрали кожне тестове ядро за допомогою компілятора AOCL і хост-програми за допомогою Altera EDS. Ми розгорнули кожен тест на платі Terasic DE1-SoC, яка має FPGA Altera Cyclone V. Еталонні тести розроблені таким чином, що кожен тест запускає програму у двох версіях: перша версія на неприскореній системі (ARM), а друга версія на прискореній системі (ARM + FPGA).

Таблиця 4.3 - Прискорення AES шляхом зміни кількості атрибутів CU та SIMD для різних розмірів даних, робочих груп = N і робочих елементів = (Кількість входів)/N, де N=2,4

Робочі групи	Обчислені одиниці	SIMD одиниць	256 Входів	512 Входів	1024 Входів
2	1	1	2.5	4.6	6.6
2	1	2	2.7	5.4	6.6
2	1	4	2.4	5.4	6.9
2	2	1	4.0	4.6	6.9
2	2	2	2.6	5.2	6.9
	(а) Кількість робочих груп = 2				
Робочі групи	Обчислені одиниці	SIMD одиниць	256 Входів	512 Входів	1024 Входів
4	1	1	0,7	2.6	6.2
4	1	2	1.0	2.7	4.6
4	1	4	3.8	5.5	5.7
4	2	1	1.3	2.1	5.7
4	2	2	1.2	2.0	6.5
	(б) Кількість робочих груп = 4				

Відповідний час виконання в обох системах записується. Час виконання в системі ARM+FPGA відносно часу виконання лише на процесорі ARM використовується для визначення отриманого прискорення системи. Таблиця 4.2, вище, показує системні параметри для різних тестів. Нижче наведені терміни, визначені контрольними тестами:

- час роботи (arm_time) = час обчислення програми лише на процесорі ARM,
- час запису (write_time) = час, витрачений на передачу вхідних даних від хоста до буфера вхідних даних,
- час читання (read_time) = час, витрачений на передачу вихідних да-

них із вихідного буфера на хост,

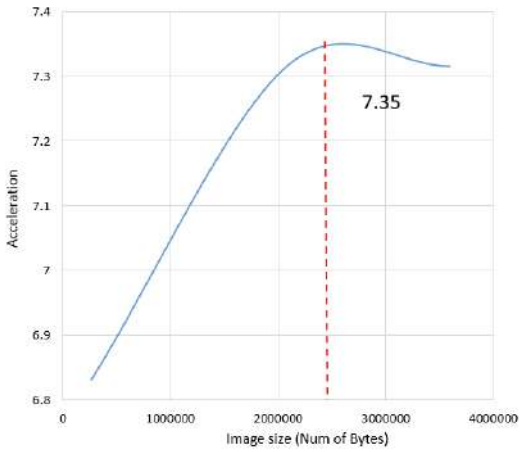
- api_time = Час передачі даних між хостом (ARM) та ядром (FPGA),
- comp time = час обчислення програми на ядрі FPGA,
- hw time = api time + comp time,
- acceleration = arm time/hw time,
- api% = (api time / hw time) *100.

4.5.2 Результати експерименту

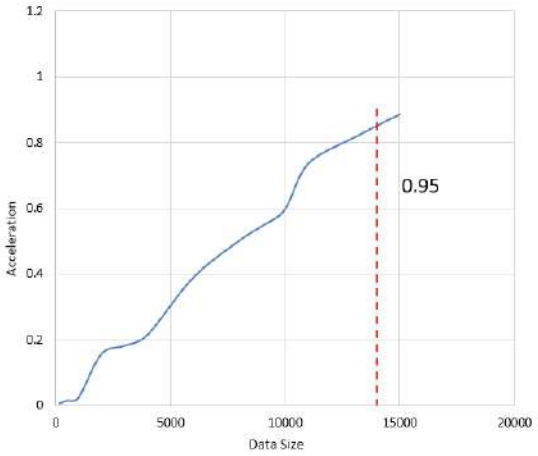
Для кожного тесту час виконання як неприскореного виконання (ARM), так і прискореного виконання (ARM+FPGA), а також прискорення визначається для кількох розмірів вхідних даних. Таким чином, результати показують прискорення системи ARM+FPGA відносно ARM.

Графіки на рисунку 4.16 представляють тенденції прискорення відносно розміру вхідних даних.

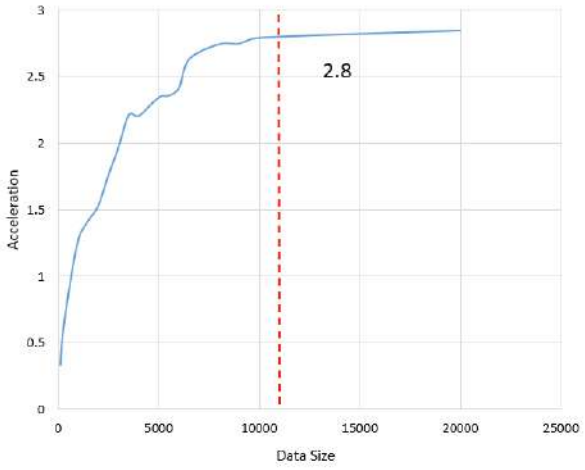
Можна помітити, що спочатку прискорення має тенденцію до збільшення розміру вхідних даних, оскільки зі зростанням обчислювальної складності результуюче прискорення обчислень долає накладні витрати на зв’язок між хостом і ядром. Як правило, для всіх тестів прискорення має тенденцію до насичення після досягнення певного вхідного розміру. Прискорення обчислень зависає за межею точки, оскільки немає миттєвих даних, доступних для обробки через накладні витрати на зв’язок і обмежений розмір буфера даних між хостом і ядром.



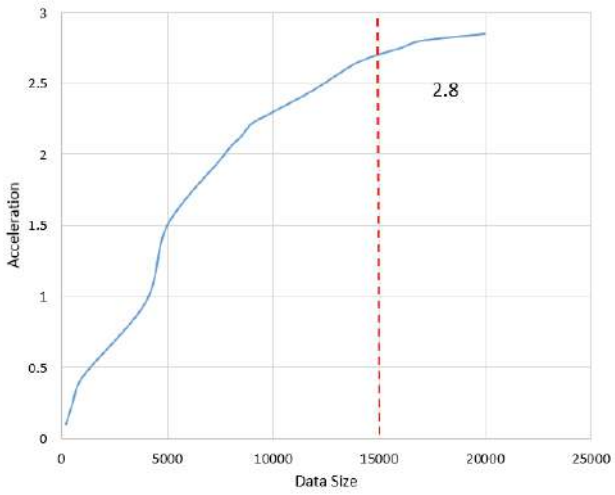
(а) Sobel



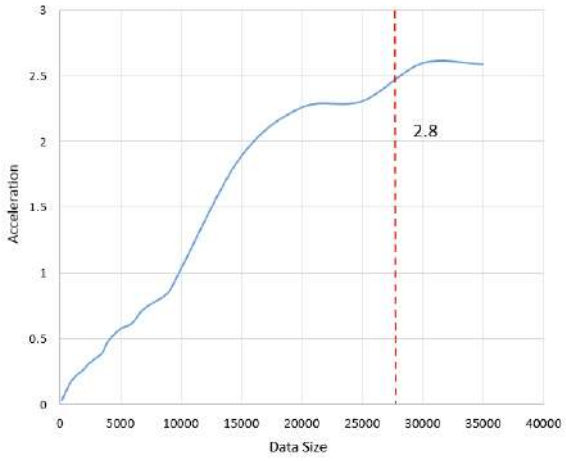
(б) FIR



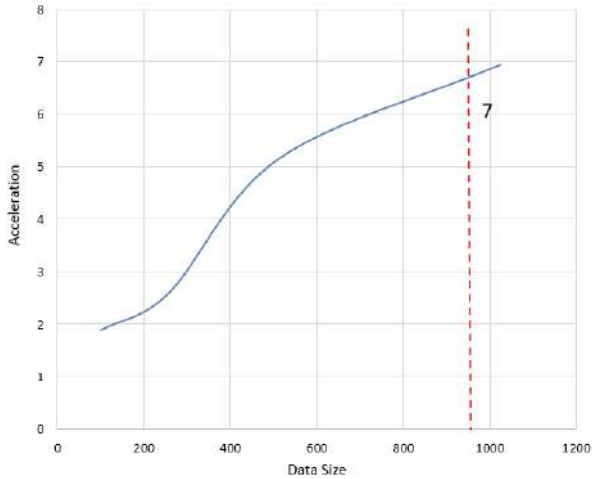
(в) Decimation



(г) Interpolation



(д) ADPCM



(е) AES, CU=2, SIMD=2

Рисунок 4.16 - Графіки прискорення та розміру вхідних даних для набору тестів

Найвища швидкість для FIR-фільтра становить 0,95 (рисунок 4.16(б)), прискорення не досягнуто через 93% витрат на зв’язок.

Вище були представлені архітектури та атрибути, які впливають на продуктивність системи. Ми представили кілька тестів OpenCL, які зіставлені з Cyclone V SoC-FPGA. Спостерігається середнє прискорення в 4 рази, і більша частина часу виконання витрачається на передачу даних між хостом і ядром. Результати показують, що при більшому розмірі вхідних даних прискорення насичується, оскільки прискорення обчислень обмежується накладними витратами на зв’язок.

ВИСНОВКИ

У кваліфікаційній роботі було зроблено огляд сучасного стану проблеми дослідження та придатність FPGA для прискорення периферійних робочих обчислень, вивчили загальну придатність FPGA для робочих навантажень IoT на периферійних серверах. З'ясовано, що ПЛІС є чудовим кандидатом для вирішення завдань периферійних обчислень, використовуючи можливості програмування та здатність обробляти потокові дані. ПЛІС можна розгортати разом із звичайними прискорювачами та створювати гетерогенність нового покоління для прискорення різних типів додатків IoT. З появою FPGA переваги гетерогенних систем стають більш значущими, оскільки FPGA можуть працювати з певним класом додатків як у хмарі, так і на периферії.

У хмарах і центрах обробки даних виконуються різноманітні додатки з різними ядрами з інтенсивними обчисленнями. На відміну від GPU, оскільки різні незалежні прискорювачі можуть бути реалізовані на FPGA, одна FPGA має потенціал відповідати на запити від кількох програм одночасно. Через обмежену кількість ресурсів FPGA дуже важливо спільно використовувати ресурси FPGA між різними програмами, щоб уникнути недовикористання ресурсів. Однак наявні в даний час інфраструктури для інтеграції програмних додатків і прискорювачів FPGA або не працюють, або не можуть повністю підтримувати ефективний доступ до прискорювачів FPGA.

У роботі представлені тести на OpenCL, розроблені для досягнення прискорення на архітектурах SoC-FPGA, і результуючі тенденції прискорення обговорюються на основі експериментів, проведених на Altera Cyclone V SoC FPGA. Експерименти показують, що прискорення насичується після певного розміру вхідних даних через витрати зв'язку між процесором і FPGA.

Також був розроблений інтерфейс багаточергового прискорювача (MQMAI), який зосереджений на вирішенні конфліктів доступу до FPGA між

кількома додатками, що працюють одночасно на головних комп'ютерах. Наші експериментальні результати показують значні покращення в прискоренні FPGA порівняно з найсучаснішими фреймворками в багатоядерних процесорах за наявності кількох прискорювачів FPGA.

Запропонований нами механізм групування прискорювачів у нашій роботі може відкрити цікавий шлях до інтерактивного використання прискорювачів FPGA у хмарах і центрах обробки даних. Ми запропонували та впровадили механізм на основі групування, який дозволяє хост-серверам спільно контролювати прискорювачі групування. Використовуючи переваги цієї нової функції, шляхом визначення областей прискорювача часткової реконструкції, можна керувати пропускною здатністю кожного типу прискорювачів на основі попиту.

Використання прискорювачів FPGA залишається складним процесом, доки потужні платформи не забезпечать ефективні автоматизовані механізми для впровадження достатньо повних бібліотек для підтримки широкого діапазону операцій. У цій роботі ми запропонували платформу з відкритим вихідним кодом. Розширення UltraShare Express для підтримки загальних прискорювачів, окрім прискорювачів потокового передавання, і надання багатої бібліотеки загальних IP-адрес прискорювачів із автоматизацією їхніх керуючих сигналів може призвести до величезних змін у сфері прискорення FPGA.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Коваленко А.А, Ільяшов О.А. Майбутнє FPGA апаратного прискорення обчислень у центрах обробки даних та хмарах: 2023.

2. Cong, J., Z. Fang, Y. Hao, P.Wei, C. H. Yu, C. Zhang and P. Zhou, \Best-effort FPGA programming: A few steps can go a long way", 2018.

3. Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. Foundations and Trends in Electronic Design Automation, 2(2):135{253, 2008.

4. INTEL. Intel fpga opencl sdk programming guide cyclone v. URL https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

5. Intel Altera. Intel opencl optimzation. https://www.altera.com/products/design-software/embedded-software-developers/opencl/developer-zone.

6. CoreTech. Fpga architectures overview, https://www.pdx.edu/nanogroup/sites/www.pdx.edu.nanogroup/files/FPGA-architecture.pdf.

7. Texas Instruments. Multicore socs stay a step ahead of soc fpgas. URL http://www.ti.com/lit/wp/spry296/spry296.pdf.

8. D.A Patterson, J.L. Hennessy, Computer organization and design: The hardware/software interface. 5th edition, Morgan Kaufmann, 2014.

9. Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. Person Education, ;№ 49, 2012.

10. Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries. XAPP1167. Rev. 3.0. Xilinx. June 2015.

11. Chapuis, G., S. Eidenbenz and N. Santhi, \Gpu performance prediction through parallel discrete event simulation and common sense", in \Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies

and Tools", pp. 204-211 (ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

12. Eyraud-Dubois, L. Marchal, O. Sinnen and F. Vivien, \Parallel scheduling of task trees with limited memory", ACM Transactions on Parallel Computing, 2015.

13. CoreTech. Fpga architectures comparision, . URL http://ee.sharif.edu/~asic/Docs/fpga-logic-cells_V4_V5.pdf.

14. Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., \Tensor ow: A system for large-scale machine learning", in \12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)", pp. 265-283 (2016).

15. Abbas, N., Y. Zhang, A. Taherkordi and T. Skeie, \Mobile edge computing: A survey", IEEE Internet of Things Journal 5, 1, 450-465 (2017).

16. INTEL. Intel fpga opencl sdk programming guide cyclone v. URL https: //www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

17. Intel Altera. Intel opencl optimzation, . URL https://www.altera.com/products/design-software/embedded-software-developers/opencl/developer-zone.html#design-examples.

18. Agullo, E., P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. l'Excellent and F.-H. Rouet, \Robust memory-aware mappings for parallel multifrontal factorizations", SIAM Journal on Scienti c Computing 38, 3, C256{C279 (2016).

19. The OpenCL Specification, Khronos OpenCL Working Group, rev. 19 , 2012.