Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Гетерогенна модель для глибокої нейронної мережі

(тема)

Виконав:

студент II курсу, групи СПзм-22-1

Доценко 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: ст.викл Знайдюк В.Г.
(посада, прізвище, ініціали)

Допускається до захисту

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

2024 р.

Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання

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)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Доценку Олексію Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Гетерогенна модель для глибокої нейронної мережі

затверджена наказом по університету від " 01 " квітня 2024 р. № 45 Стз

2. Термін подання студентом роботи до екзаменаційної комісії 15 червня 2024 р.

3. Вхідні дані до роботи Гетерогенні нейронні мережі
Глибоке навчання
Архітектура мережі
Тестування та оцінка результатів

4. Перелік питань, що потрібно опрацювати у роботі
Загальні положення
Огляд впроваджень SqueezeNet
Методологія та архітектура
Результати та оцінка ефективності
Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) 13

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

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд літературних джерел	01.04.24-08.04.24	
2	Вибір та обґрунтування методики та засобів	09.04.24-12.04.24	
	дослідження		
3	Вибір та обґрунтування методів	13.04.24-19.04.24	
4	Програмна реалізація нейронних мереж	20.04.24-09.05.24	
5	Проведення експериментальних досліджень	10.05.24-23.05.24	
6	Оформлення матеріалів кваліфікаційної роботи	24.05.24-03.06.24	
7	Подання кваліфікаційної роботи керівникові та	04.06.24-07.06.24	
	її попередній захист		
8	Подання кваліфікаційної роботи на	08.06.24-12.06.24	
	рецензування		
9	Захист	22.06	

Дата видачі завдання 01 квітня 2024 р.

Студент _____ (підпис)

Керівник роботи _____ Ст викладач Занйдюк В Г.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 60 с., 21 рис., 10 табл., 1 дод., 19 джерел.

ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ; ГЕТЕРОНЕНІ ОБЧИСЛЕННЯ; СИСТОЛІЧНІ МАСИВИ; FPGA

Метою кваліфікаційної роботи є розробка гетерогенної моделі глибокої нейронної мережі.

У ході виконання кваліфікаційної роботи було проведено аналіз та дослідження нових підходів у побудові глибоких нейронних мереж із використанням гетерогенних компонентів. Запропоновано методи інтеграції різних архітектур нейронних мереж, що дозволяють підвищити ефективність навчання та адаптивність моделей. Результати експериментів підтверджують покращення продуктивності та точності порівняно з традиційними методами, відкриваючи нові можливості для застосування глибокого навчання у різних галузях.

ABSTRACT

Master’s thesis: 60 pages, 21 figures, 10 tables, 1 appendices, 19 sources.

CONVOLUTIONAL NEURAL NETWORKS; HETEROGENEOUS COMPUTATION; SYSTOLIC ARRAYS; FPGA

The major goal of this thesis is to develop a heterogeneous model of a deep neural network.

In the course of the qualification work, the analysis and research of new approaches in the construction of deep neural networks using heterogeneous components was carried out. Methods of integration of different architectures of neural networks are proposed, which allow to increase the efficiency of learning and the adaptability of models. Experimental results confirm improved performance and accuracy compared to traditional methods, opening up new opportunities for deep learning applications in various industries.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ГНМ (гетерогенна нейронна мережа) – нейронна мережа, що поєднує різні архітектури і підходи для підвищення ефективності та адаптивності.

CNN (Convolutional Neural Network) – Згорткова нейронна мережа, що використовується для обробки зображень та відео.

RNN (Recurrent Neural Network) – Рекурентна нейронна мережа, що використовується для обробки послідовних даних.

DNN (Deep Neural Network) – Глибока нейронна мережа, що складається з багатьох шарів нейронів для вирішення складних задач.

GPU (Graphics Processing Unit) – Графічний процесор, який використовується для швидкої обробки великих обсягів даних у машинному навчанні.

API (Application Programming Interface) – Інтерфейс прикладного програмування, що дозволяє взаємодію між різними програмними компонентами.

LSTM (Long Short-Term Memory) – Архітектура рекурентної нейронної мережі, що дозволяє зберігати інформацію на довгий період.

ВСТУП

Штучний інтелект (ШІ) – одна з найперспективніших технологій, що базується на алгоритмах машинного навчання. У цій роботі ми пропонуємо робочий процес для реалізації глибоких нейронних мереж. Цей робочий процес намагається поєднати гнучкість мереж на основі компіляторів високого рівня (HLS – high-level compilers) з архітектурними можливостями управління потоками на основі мов опису апаратного забезпечення (HDL – hardware description languages). Архітектура складається зі згорткової нейронної мережі SqueezeNet v1.1 та твердотільної процесорної системи (HPS – hard processor system), яка співіснує з апаратними засобами прискорення, що розробляються. Ця методологія дозволяє порівнювати рішення, засновані виключно на програмному забезпеченні (PyTorch 1.13.1), і пропонувати гетерогенні рішення для виведення, використовуючи найкращі варіанти в рамках програмно-апаратного потоку. Запропонований робочий процес реалізовано на недорогій платформі польової програмованої логічної матриці на кристалі (ПЛІС SOC), а саме на розробницькій платі DE10-Nano. Ми надали систолічні архітектурні рішення, написані на мові OpenCL, які є дуже гнучкими і легко налаштовуються для повного використання ресурсів програмованих пристроїв і досягнення чудової енергоефективності при роботі з 32-бітною плаваючою комою. З точки зору верифікації, запропонований метод є ефективним, оскільки еталонні моделі у всіх тестах, як для окремих шарів, так і для всієї мережі, були легко доступні за допомогою пакетів, добре відомих у розробці, навчанні та виведенні глибоких мереж.

1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

Останнім часом стрімко розвиваються технології, засновані на штучному інтелекті та алгоритмах машинного навчання. Ці технології мають широкий спектр застосування – від класифікації зображень і розпізнавання об'єктів для медичної діагностики та контролю якості до генерації текстів.

Однак ці технології значною мірою покладаються на машини зі значною обчислювальною потужністю, тому обробка даних зазвичай здійснюється на хмарних серверах, щоб полегшити доступ до них кінцевому користувачеві. Якщо ми хочемо наблизити штучний інтелект до електроніки, такий підхід не підходить через затримку, енергоспоживання та обмеження в розмірах.

У відповідь на це і завдяки вдосконаленню новітніх процесорів з'явився новий підхід, відомий як штучний інтелект «на межі». Філософія цього підходу полягає в тому, що обробка даних, отриманих у вузлі, здійснюється всередині самого вузла, щоб він міг діяти після отримання результатів, дозволяючи електроніці бути незалежною [1].

Таким чином, можна знайти нові архітектури, метою яких є досягнення тієї ж функціональності, що і в інших технологіях, але в обмін на зменшення точності та складності результатів.

У цій роботі ми зосередимося на архітектурах на основі згортки, оскільки вони є чіткою тенденцією в сучасному ландшафті периферійних обчислень. Цей тип архітектури дозволяє нам витягувати особливості або характеристики з вхідних зображень, класифікувати їх або виявляти різні об'єкти.

Прикладами таких архітектур є Darknet або VGG, а також цікавий варіант, відомий як SqueezeNet v1.1. SqueezeNet – це нова версія [2], яка зберігає таку ж точність і зменшує обчислювальні витрати в 2,4 рази.

Ми можемо спостерігати використання SqueezeNet в таких додатках,

як: самокеровані автомобілі [3], розпізнавання обличчя [4], класифікація погоди [5], виявлення пожеж [6,7], розпізнавання садових комах [8] та класифікація перешкод в приміщенні [9]. У багатьох з цих застосувань ключовим елементом є можливість покладатися на рішення, яке може виконувати завдання прогнозування, розпізнавання, класифікації та виявлення в системах з обмеженими ресурсами, що забезпечують максимальну операційну автономність без шкоди для досягнутої точності. Ми реалізували згорткову нейронну мережу SqueezeNet v1.1 на недорогих SOC-платформах FPGA, таких як плата розробки DE10-Nano, що є нашим основним внеском. Ця система включає в себе процесор ARM HPS, який працює з апаратним прискоренням. Це дозволяє нам порівнювати рішення, засновані виключно на програмному забезпеченні (PyTorch 1.13.1), і створювати гетерогенні рішення для виведення, які використовують найкращі можливості як програмних, так і апаратних потоків.

Для скорочення часу проектування ми вдаємося до компіляторів високого рівня (HLS – high-level compilers), за допомогою яких підвищується рівень абстракції для генерування синтезу апаратного забезпечення за допомогою коду на мові C/C++. Однією з найцікавіших опцій від виробника, що дозволяє скористатися можливостями ПЛІС для оптимізації результатів з енергоефективністю та низькою латентністю, є Intel FPGA SDK для OpenCL. За допомогою цього інструменту ми можемо реалізувати наш алгоритм сортування зображень, використовуючи апаратне прискорення з можливістю паралелізму, як це передбачено стандартом OpenCL, на недорогій платі для розробки DE10-Nano.

Концептуальну схему, що узагальнює основну мету, показано на рисунку 1.1.

З методологічної точки зору, основний внесок цієї роботи полягає в тому, що весь процес верифікації базується на Python з використанням ноутбуків Jupyter, які отримують доступ до наших прискорювачів через PyopenCL. Ця технологія вже досить поширена в роботі з графічними

процесорами, але рідко використовується в недорогих ПЛІС, в яких роль хоста виконує жорсткий макрос ARM.

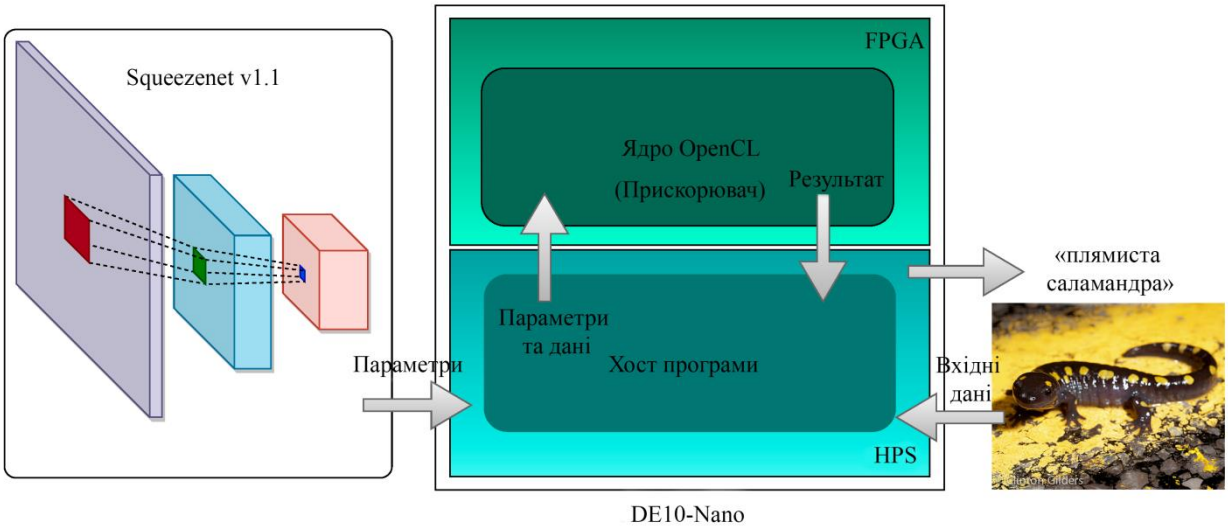

Рисунок 1.1 – Концептуальна схема основної мети

2 ОГЛЯД ВПРОВАДЖЕНЬ SQUEEZENET

У першому підрозділі ми розглядаємо основи згорткових шарів, які є фундаментальними для розуміння топології згорткової мережі SqueezeNet. Перш за все, ми закріплюємо термінологію, яка використовується в роботі, і встановлюємо теоретичні основи.

Також обговорюються інші типи шарів, які зазвичай використовуються у згорткових архітектурах, та обрана модель SqueezeNet V1.1.

Після того, як ми зібрали всі шматочки пазлу, що складають цей тип мережі, ми обговорюємо основний структурний елемент, який визначає суть цього типу мережі, – Fire модуль.

Потім ми розглянемо роботи, які працюють з цим типом мереж, з недорогими пристроями FPGA, які можуть розробляти ШІ на периферії.

Нарешті, ми обговоримо стандарт OpenCL і два його основні підходи до вирішення різних типів завдань з точки зору ПЛІС-рішень.

2.1 Шар згортки

Шар згортки зазвичай використовується для обробки вхідних даних або карт об'єктів, просторовий розподіл яких є двовимірним, як у випадку зображення, розмір якого зазвичай виражається шириною та висотою пікселів, H_{pixel}×W_{pixel}.

Однак зображення зазвичай мають різні канали, які містять інформацію про колір у форматі $HRed\text{-}Green\text{-}Blue$. Тому вхідні дані для шарів згортки зазвичай мають формат H_{in}×W_{in}×CH_{in}, які є висотою, шириною та глибиною каналу вхідної карти ознак відповідно.

Отже, 2D шари згортки формуються кількома фільтрами розмірності H_k×W_k×CH_{in} для забезпечення узгодженості з вхідними даними. Щодо розміру фільтрів, H_k×W_k, ми можемо знайти різні комбінації з поширеними

розмірами 1×1, 3×3 та 5×5. Ще один параметр, який є частиною згорток, називається зсувом. Він складається зі скалярного значення для кожного згорткового фільтра.

На рисунку 2.1 візуально представлено розміри з їх номенклатурою для карти ознак і фільтра, що використовуються у згортці 3×3.

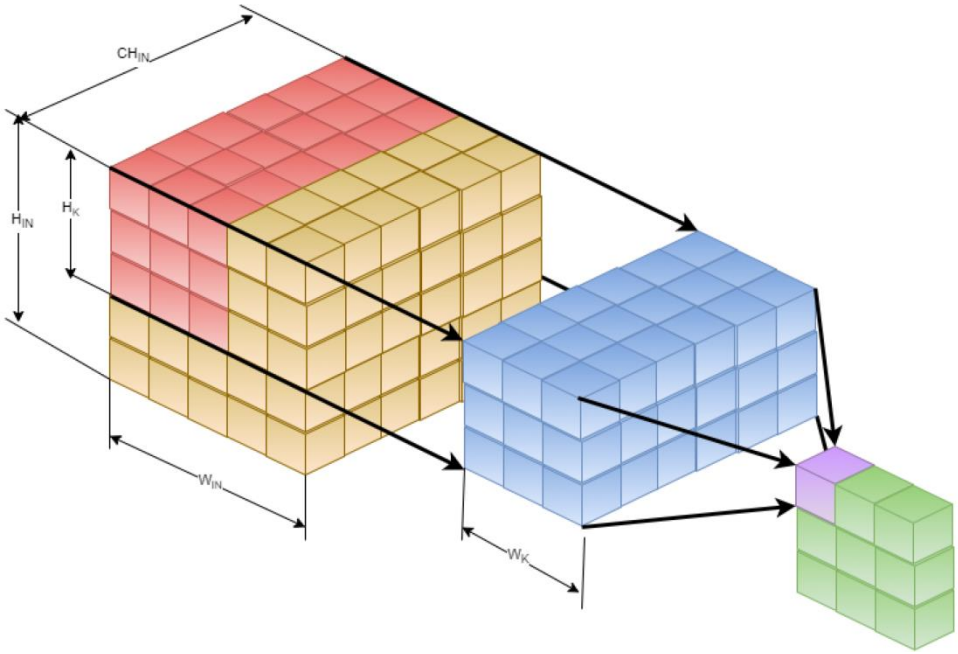

Рисунок 2.1 – Представлення CH_{in} карт об'єктів та CH_{in} ядра з номенклатурою його розмірностей

Результат згортки відповідає сумі добутків фільтра на вхідні дані елемент за елементом. Фільтр ковзає по розміру вхідних даних, створюючи вихідне значення для кожного зсуву. Якщо є більше одного каналу, фільтр застосовується до відповідного каналу, а результати всіх каналів підсумовуються.

Припустимо, що є вхідні дані H_{in}×W_{in}×CH_{in} і один фільтр H_k×W_k×CH_{in}. Перший елемент на виході відповідає наступному:

$$out\ (0,0) = bias + \sum_{c=0}^{CH_{in}} \sum_{i=0}^{W_k} \sum_{j=0}^{H_k} filter\ (c,i.j) \cdot feature(c,i,j). \quad (2.1)$$

Розмір вихідних даних залежить головним чином від розміру ядра, коефіцієнта ковзання або кроку, а також додаткового параметра, відомого як padding, який дозволяє додати рамку, зазвичай з нулів, навколо вхідних даних.

Якщо вхідні дані та фільтр мають квадратну форму з розмірами in_{size} та k_{size}, відповідно, ми можемо обчислити розмір вихідного сигналу out_{size} як:

$$out_{size} = \frac{in_{size} + 2 \cdot padding - k_{size}}{stride} + 1, \quad (2.2)$$

де padding – це розмір доданого кадру, а stride – це позиції, на які фільтр переміщується над вхідними даними.

Кількість каналів на виході CH_{out} залежить від кількості фільтрів, які застосовує згортка.

На рисунку 2.2 показано візуальний приклад роботи шару згортки. Ми бачимо вхідну карту ознак з розміром in_{size} 5 і фільтр з k_{size} 3, оскільки це згортка 3×3, з кроком і проміжком 1.

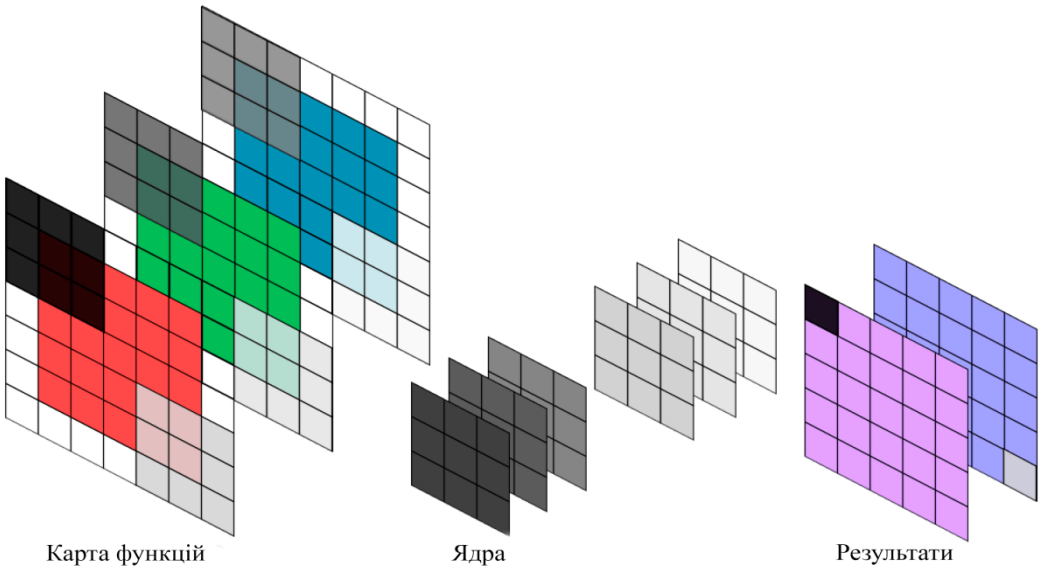

Рисунок 2.2 – Приклад згортки 3×3 з вхідним зображенням розміром 5×5×3 і двома фільтрами розміром 3×3×3 з відступом і кроком 1

Застосовуючи формулу (2.2), ми можемо перевірити, що вихідний розмір out_{size} дорівнює 5. Також, оскільки ми маємо два фільтри, є 2 вихідних канали (CH_{out}), отримуючи розмірність виходу 5×5×2.

На завершення необхідно зазначити, що поширеною практикою є застосування функції активації до результатів, отриманих в результаті згортки, та застосування нелінійностей до даних, що дозволяє мережі вирішувати більш складні задачі. Найпоширенішою є ReLU, кускова функція якої міститься в рівнянні (2.3).

$$ReLU(x) = \begin{cases} x & if\ x > 0 \\ 0 & otherwise \end{cases} \tag{2.3}$$

2.2 Шари пулу типів

Існує два основних типи шарів, max-pooling і average-pooling (обидва присутні в SqueezeNet v1.1), залежно від типу вилучення даних, яке вони виконують. Ці шари мають процедуру, подібну до згорток, що складається з фільтра заданого розміру, який ковзає по вхідних даних, виконуючи відповідну операцію. Крім того, можна налаштовувати як крок, так і прокладку. Однак вони не впливають на розмірність каналу вхідної карти ознак.

Шар max-pool витягує вхідні дані з найбільшим значенням у межах розміру ядра, тоді як шар average-pool обчислює середнє значення характеристик вхідних даних.

2.3 SqueezeNet

SqueezeNet – це архітектура ЗНМ (згорткової нейронної мережі), основною метою якої було зменшення кількості параметрів без суттєвого впливу на рівень точності класифікованих зображень. Зменшення кількості

параметрів у згортковій мережі робить її реалізацію можливою на ПЛІС або вбудованих системах. Крім того, це дозволяє зберігати параметри в пам'яті на кристалі, зменшуючи вузьке місце, яке створюється пропускною спроможністю доступу до пам'яті.

Для цього в [2] представлено модулі Fire, які складають глобальну архітектуру. Ці модулі складаються з шару стиснення, що відповідає згорткам 1×1, вихід якого безпосередньо з'єднується з шаром розширення, утвореним сумішшю згорток 1×1 і 3×3, конкатенація результатів яких формує вихід модуля.

Використання шару стиснення перед згорткою 3×3 дозволяє зменшити кількість вхідних каналів і кількість ядер кожного фільтра. На рисунку 2.3 показано розташування різних згорток у модулі Fire.

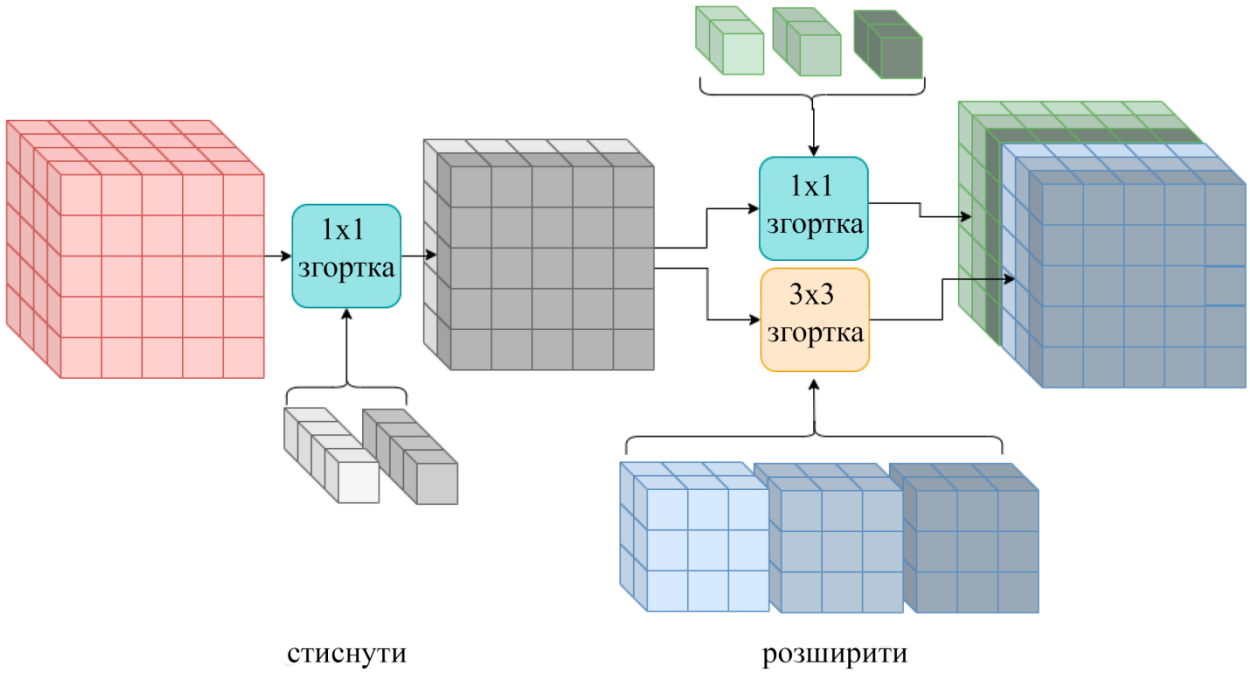

Рисунок 2.3 – Структура Fire модуля складається з різних фільтрів згортки

Згодом в офіційному репозиторії була випущена нова версія SqueezeNet, SqueezeNet v1.1 [2]. Ця версія модифікує оригінальну архітектуру, зберігаючи Fire модулі, дещо зменшуючи кількість параметрів

та отримуючи зменшення обчислювальних витрат у 2,4 рази без втрати точності.

Порівняння класифікацій помилок попередньо навченої моделі із зображеннями Imagenet можна знайти в документації бібліотеки машинного навчання з відкритим вихідним кодом PyTorch [10]. Результати наведено в таблиці 2.1.

Таблиця 2.1 – Порівняльна таблиця похибок прогнозування версій SqueezeNet з набором даних Imagenet [10]

Структура моделі	Помилка Топ-1	Топ-5 помилок
SqueezeNet V1.0	41.90%	19.58%
SqueezeNet V1.1	41.81%	19.38%

Структуру та конфігурацію SqueezeNet v.1.1 ми бачимо на рисунку 2.4 та в таблиці 2.2. Вона являє собою послідовність шарів (14). Якщо розбити кожен шар на складові елементи (рисунок 2.4), то нам потрібно реалізувати чотири фундаментальні елементи: згортку 1×1, згортку 3×3, а також операції max-pool та average-pool. Звичайно, повторне використання цих блоків вимагає, щоб реалізація була реконфігурованою, щоб адаптуватися до різних розмірів і параметрів, перелічених у таблиці 2.2.

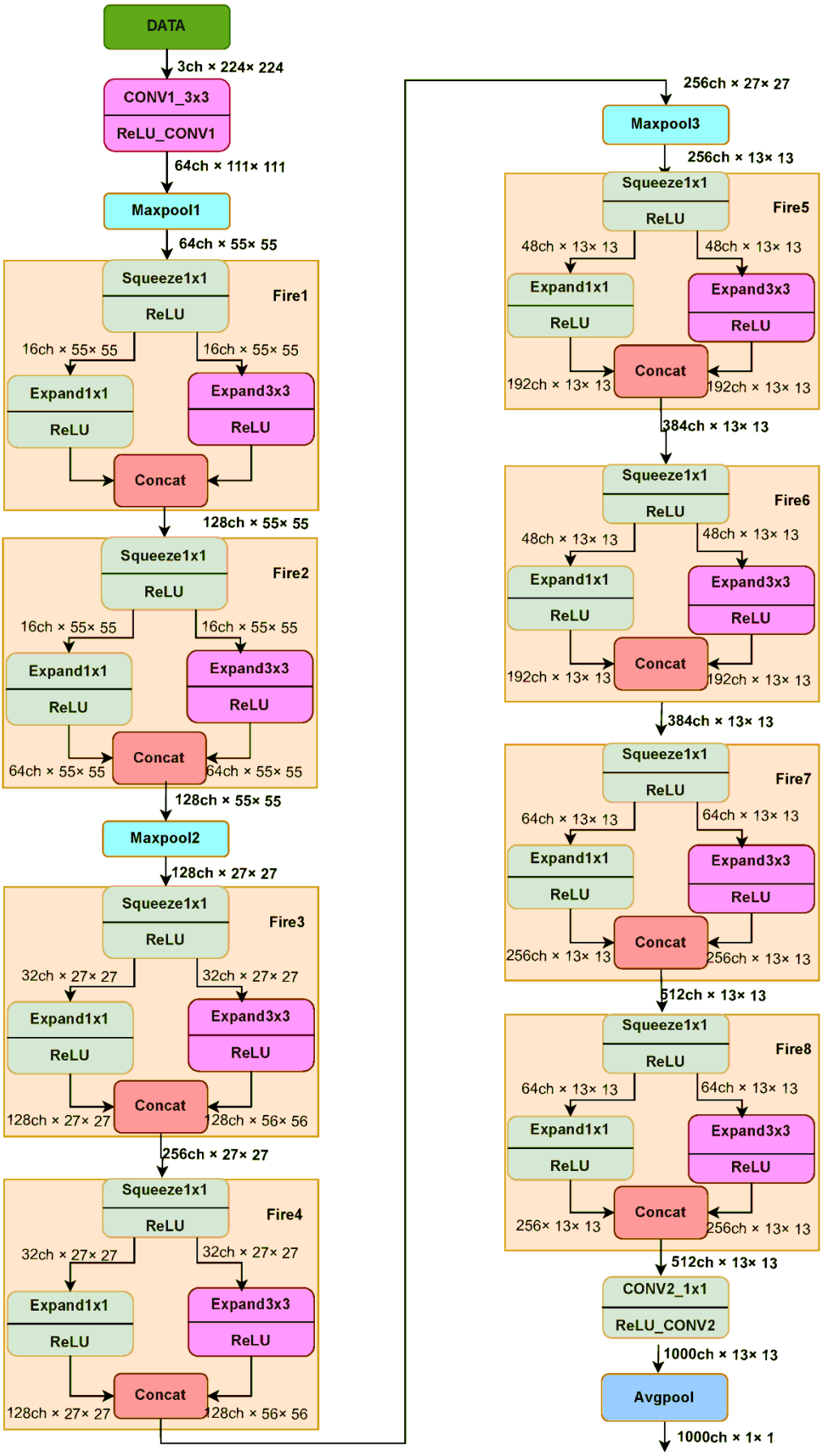

Рисунок 2.4 – Структура SqueezeNet v1.1

Таблиця 2.2 – Архітектура моделі SqueezeNet v1.1 [10].

Назва/тип шару	Вихідний розмір	Розмір/крок фільтра (без Fire модулів)	$S_{1\times1}$	$E_{1\times1}$	$E_{3\times3}$
Input image	224×224×3				
conv1	111×111×64	3×3/2 (× 64)			
maxpool1	55×55×64	3×3/2			
fire1	55×55×128		16	64	64
fire2	55×55×128		16	64	64
maxpool2	27×27×128	3×3/2			
fire3	27×27×256		32	128	128
fire4	27×27×256		32	128	128
maxpool3	13×13×256	3×3/2			
fire5	13×13×384		48	192	192
fire6	13×13×384		48	192	192
fire7	13×13×512		64	256	256
fire8	13×13×512		64	256	256
conv2	13×13×1000	1×1/1 (×1000)			
avgpool1	1×1×1000	13×13/1			

2.4 Пов'язані роботи

Невдовзі після появи моделі SqueezeNet Девід Гшвенд (David Gschwend) [11] реалізував її на недорогій платформі розробки Zynqbox, яка включала ПЛІС Xilinx Zynq-7000 у поєднанні з ARM Cortex-A9. Реалізоване

рішення отримало назву ZynqNet і було розділене на дві частини.

В оригінальну модель SqueezeNet були внесені зміни, щоб адаптувати її до вимог ПЛІС. Однією з найпомітніших модифікацій було усунення шарів з максимальним пулом, слідуючи філософії «all convolutional networks», де мережа складається лише зі згорткових шарів. Щоб отримати той самий результат, зменшивши вихідний розмір, вони додали крок у два шари до згортки одразу після цього. Однак у SqueezeNet цей шар завжди складається зі згортки 1×1, що призвело б до втрати інформації. Тому вони замінили цей перший шар фільтром 3×3. Вони також реалізували останній шар типу average pool, який можна розглядати як згортку, фільтри якої складаються зі значень $\mathrm{kernel}_{size}^{-1}$.

Ще одна модифікація, про яку варто згадати, – це зміна розмірів шарів, що робить їх висоту і ширину степенями двійки. Це примітно, оскільки такий тип змін є спільним у кількох реалізаціях, що зумовлює тип розробленої топології CNN, досягаючи в обмін на це кращих значень затримок. З іншого боку, при реалізації прискорювача автори вирішили використовувати плаваючу точку однієї точності для представлення даних, щоб зберегти сумісність з їхньою реалізацією ZynqNet на GPU.

Крім того, вони зробили ставку на використання кеш-пам'яті для полегшення читання і запису, а також вирішили розмотати цикли згорток 3×3, таким чином досягнувши помилки в топ-5 на Imagenet в 15,4% при використанні 32-бітового представлення даних з плаваючою комою. Підкреслимо, що прискорювач було оптимізовано для виконання згорток 3×3, і цілком можливо, що будь-яка інша конфігурація вимагала б додаткової логіки, що призвело б до низького використання операцій з декількома накопичувачами. Варто зазначити, що вони виконали свою реалізацію з використанням HLS від Vivado.

Робота, подібна до ZynqNnet, була проведена в [12], результатом якої стала EdgeNet. Запропонована згорткова мережа була реалізована на платі DE10-Nano, а запропонований прискорювач складався з конфігурованого

обчислювального блоку. Цей блок був розроблений для роботи в інверсному режимі, коли вхід прискорювача еквівалентний шару розширення, а вихід - шару стиснення. Це дозволило зменшити кількість вхідних каналів, оскільки на вхід обчислювального блоку надходив переважно результат згортки стисненого шару 1×1. На виході пристрою також був стиснений шар. Крім того, модуль мав доступ до шарів типу пулу залежно від конфігурації шляху даних.

Слід зазначити, що результати були отримані з представленням даних і параметрів з використанням 8-бітних плаваючих точок (п'ять біт експоненти і два біти мантиси) параметрів, отриманих в результаті навчання [11], що дозволило знизити точність класифікацій на 6% і досягти 51% точності топ-1 за допомогою Imagenet.

Продовжуючи тему квантифікації параметрів згорткових нейронних мереж, ми знову знаходимо в [13] реалізацію SqueezeNet v1.1, де перед розробкою прискорювача автори провели дослідження впливу квантифікації на точність моделі.

Вони помітили, що, використовуючи 8-бітний цілочисельний тип даних, вони значно зменшили як ресурси ПЛІС, так і доступ до пам'яті, отримавши втрату 0,69% і 0,72%, в результаті чого отримали 57,49% і 79,90%, перше і п'яте місце за точністю, відповідно.

Варто зазначити, що ця реалізація була виконана на недорогій платі DE10-Nano з використанням стандарту OpenCL та HLS.

Крім того, вони включили додаткову логіку, яка дозволила зчитувати вхідні дані, фільтрувати згортку і записувати результати в глобальну пам'ять. Крім того, оскільки функція активації ReLU генерує карти ознак з результатами, що дорівнюють нулю, вони ввели спеціальний контроль, за допомогою якого вони уникали виконання операцій з нульовими вхідними значеннями.

Продовжуючи тему реалізацій зі зниженою точністю з фіксованою комою, можна згадати роботу [14]. Вони використовували Zynq-7020 для

реалізації на основі плати розробки ZC702. Автори використали 8 біт для параметрів (ваг та зсуву) та 16 біт для активацій і виконували арифметику з фіксованою комою. Це дало час виконання 333 мс. Вимірювання енергоефективності було отримано за допомогою Xilinx Estimator Power (XPE) від Vivado і показало споживання 2275 Вт. Для проектування використовувався інструмент Xilinx HLS, для реалізації якого знадобилося два ядра. Зокрема, в його реалізації було використано 186 DSP.

Роботи, подібні до Zynq-7020, можна знайти в [15], де використовується HLS, але з 32-розрядною плаваючою комою. Вони досягли часу виконання 1 секунду, і що цікаво, при ступенях зв'язку, подібних до згаданих раніше, вони отримують 7,95 Вт енергоспоживання, знову ж таки використовуючи XPE.

Пізніше, в [16], дослідники розробили проект на платі SOC DE10-Nano, реалізацію SqueezeNet v1.1, що забезпечує паралелізм на багатопотоковому рівні, використовуючи HLS OpenCL.

Вони надали детальну документацію кроків, які були виконані для реалізації їх дизайну. Вони включили різні версії, де застосування методів оптимізації ядер, таких як коалесцентний доступ до пам'яті та розгортання циклу, зменшило час, необхідний для прогнозування мережі. Розглядаючи реалізацію, можна помітити, що використання коалесцентної пам'яті через структуру даних float4 обмежує кількість каналів кожного шару згортки, оскільки вони повинні бути кратними чотирьом через кількість елементів даних, які зчитуються коалесцентним способом.

3 МЕТОДОЛОГІЯ ТА АРХІТЕКТУРА

Ми розробили три архітектури. Ми розробили метод на основі Python з використанням блокнотів Jupyter для проектування та верифікації. Наші ядра, розроблені на OpenCl, були перевірені за допомогою хост-програм, розроблених на Python, які контекстуалізували та спілкувалися з ядрами через PyopenCL. Це дозволило нам спроектувати і перевірити нашу реалізацію OpenCL для використання в якості еталонної моделі в кожному з розроблених шарів і в кожній зі структур, які пов'язували їх (Fire, блоки і всю мережу) з рішенням PyTorch. Ця перевірка була виконана за допомогою тестових стендів, розроблених в ноутбуках Jupyter, як на рівні емуляції, так і на рівні косимуляції за допомогою HDL-симулятора, а також на фізичному рівні в самій ПЛІС.

Таким чином, ми могли налагоджувати виконання хосту та реалізацію ядра одночасно:

- модель SqueezeNet легко інстанціюється завдяки PyTorch, що робить її хорошою еталонною моделлю;

- результати кожного вузла моделі PyTorch можуть бути перетворені в масиви NumPy, що полегшує пошарове порівняння для налагодження проекту;

- реалізація на хості є гнучкішою та зручнішою для користувача завдяки PyOpenCL.

Перша описана архітектура дозволяє нам дізнатися базовий алгоритм, необхідний для різних рівнів SqueezeNet, і представити реалізацію, засновану на ядрах з простими завданнями. У наступних двох ми використовуємо систолічні реалізації для застосування концепції «паралелізму завдань», доступної в HLS, таких як OpenCL. Для кожної з цих архітектур ми порівнюємо використані ресурси та продуктивність з апаратними

архітектурами, які до цього часу пропонували найкращу продуктивність і які базуються на ядрах NDRange на тій самій платі. Ми також проводимо порівняння з рішеннями, розробленими на наноплаті DE10, але вдаючись лише до комбінованого використання двоядерного ARM Cortex-A9 плюс fpu NEON за допомогою пакету PyTorch Python.

3.1 Проста архітектура завдань

3.1.1 Згортка 1×1

В алгоритмі, зображеному на рисунку 3.1 ми бачимо рівняння, що беруть участь у згортці 1×1, а на рисунку 3.2 – графічну деталь для першої ітерації циклу, що міститься між рядками 3-13 цього алгоритму. Як бачимо, той факт, що вказівники не виконують операцій, значно прискорює час виконання. Більше того, єдиним методом, який дозволяє значно оптимізувати конвеєр циклу, є застосування регістра зсуву, причому версія 6 є найкраще оптимізованим ядром шару згортки 1×1.

Input: $in\_channels$ (parameter)
Input: in_size (parameter)
Input: $filte\_size$ (parameter)
Input: in_img (buffer read only)
Input: $filter\_weight$ (buffer read only)
Input: $filter\_bias$ (buffer read only)
Output: out_img (buffer)

```
1:  for filter_index ← 0 To filter_size do
2:      bias ← filter_bias[filter_index]
3:      for j ← 0 To (in_size × in_size) do
4:          tmp ← bias
5:          for k ← 0 To (in_channels) do
6:              tmp ← in_img[k × in_size × in_size + j] × filter_weight[k + filter_index × in_channels] + tmp;
7:          end for
8:          if tmp > 0 then
9:              out_img[ij + in_size × in_size × filter_index] ← tmp
10:         else
11:             out_img[ij + in_size × in_size × filter_index] ← 0
12:         end if
13:     end for
14: end for
```

Рисунок 3.1 – Згортка 1×1

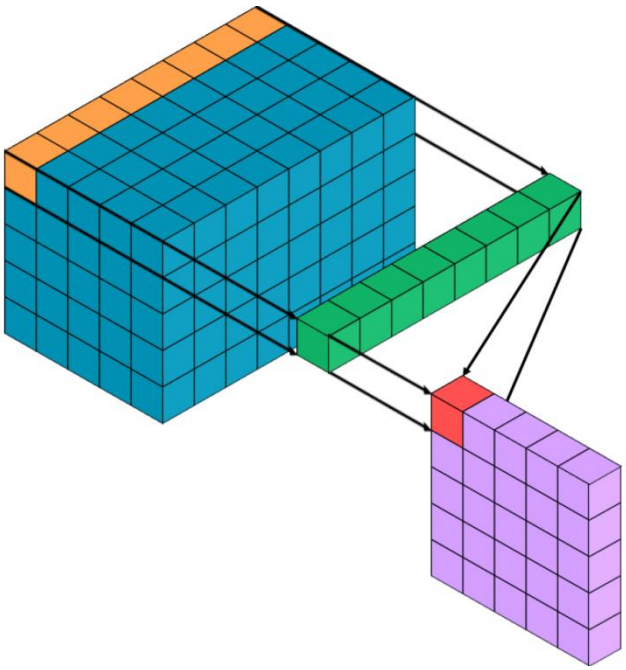

Рисунок 3.2 – Деталь згортки 1×1 з CH_{in}=8, CH_{out}=1 та in_{size}=5.

У таблиці 3.1 ми бачимо різні реалізації ядра, що відповідають згорткам 1×1. Ми відкинули версію 2 через ресурси, необхідні для її реалізації. Хоча це версія з найкоротшим часом виконання, вона не сумісна з синтезом ядра з усіма шарами SqueezeNet.

Таблиця 3.1 – Результати часу виконання та початкового інтервалу (ПІ), різних версій ядра conv1×1

	Затримка (с)	Початковий інтервал
V0	3,4381	16
V1	1,7096	16
V2	1,0423	31
V5	2,4772	16
V6	1,2000	1

У таблиці 3.2 наведено результати, отримані в результаті синтезу

апаратного забезпечення згортки 1×1 (V6) з виведенням регістра зсуву. Знову ж таки, ми отримуємо час виконання, необхідний для виконання згортки 1×1 з вхідною картою ознак 13×13×512 з 1000 фільтрів.

Таблиця 3.2 – Компіляція та результати виконання згортки 1×1 V6

	Єдине завдання	NDRange [16]	PyTorch
ALUTs	16421	11536	–
Реєстри	28536	21064	–
Логічні ресурси	12413/41910 (30%)	6962/41910 (24%)	–
Блоки DSP	9/112 (8%)	20/112 (18%)	–
Біти пам'яті	351936/5662720 (6%)	1092608/5662720 (19%)	–
Блоки оперативної пам'яті	101/553 (18%)	168/553 (30%)	–
F_{max} (МГц)	116,13	120,81	–
Затримка (с)	1,2000	0,3689	0,1823

3.1.2 Згортка 3×3

В алгоритмі 2 на рисунку 3.3 ми бачимо рівняння, що беруть участь у згортці 3×3, а на рисунку 3.4 – графічну деталь для першої ітерації циклу, що міститься між рядками 5-23 цього алгоритму.

Перше, що відрізняє цю реалізацію від Ndrange [16], з якою її порівнюють, – це те, що вони є реалізаціями з фіксованим розміром фільтра (3×3), а також з фіксованими значеннями padding (1) і stride (1).

Результати синтезу апаратного забезпечення згортки 3×3 з виведенням зі зсувним регістром наведено у таблиці 3.3, хоча її немає у фінальному коді OpenCL. Також показано час виконання, необхідний для виконання згортки 3×3 з вхідною картою ознак 13×13×64 на 256 ядрах. Це відповідає

одношаровій конфігурації розширення з 7 або 8 Fire модулів.

Input: *in_channels* (parameter)
Input: *in_size* (parameter)
Input: *pad* (parameter)
Input: *stride* (parameter)
Input: *out_size* (parameter)
Input: *filter_size* (parameter)
Input: *in_img* (buffer read only)
Input: *filter_weight* (buffer read only)
Input: *filter_bias* (buffer read only)
Output: *out_img* (buffer)

1: $out\_img \leftarrow start\_channel \times out\_size \times out\_size + out\_img$
2: **for** $filter\_index \leftarrow 0$ To $filter\_size$ **do**
3: $bias \leftarrow filter\_bias[filter\_index]$
4: **for** $i \leftarrow 0$ To (out_size) **do**
5: **for** $j \leftarrow 0$ To (out_size) **do**
6: $tmp \leftarrow bias$
7: **for** $k \leftarrow 0$ To $(in\_channels)$ **do**
8: **for** $l \leftarrow 0$ To 3 **do**
9: $h \leftarrow i \times stride + l - pad$
10: **for** $m \leftarrow 0$ To 3 **do**
11: $w \leftarrow j \times stride + m - pad$
12: **if** $(h >= 0)\&(h < in_size)\&(w >= 0)\&(w < in_size)0$ **then**
13: $tmp \leftarrow in\_img[k \times in\_size \times in\_size + h \times in_size + w] \times filter\_weight[9 \times k + 3 \times l + m] + tmp$
14: **end if**
15: **end for**
16: **end for**
17: **end for**
18: **if** $tmp > 0$ **then**
19: $out\_img[i \times out\_size + j] \leftarrow tmp$
20: **else**
21: $out\_img[i \times out\_size + j] \leftarrow 0$
22: **end if**
23: **end for**
24: **end for**
25: $filter\_weight \leftarrow input\_channels \times 9 + filter\_weight$
26: $out\_img \leftarrow out\_size \times out\_size + out\_img$
27: **end for**

Рисунок 3.3 – Згортка 3×3

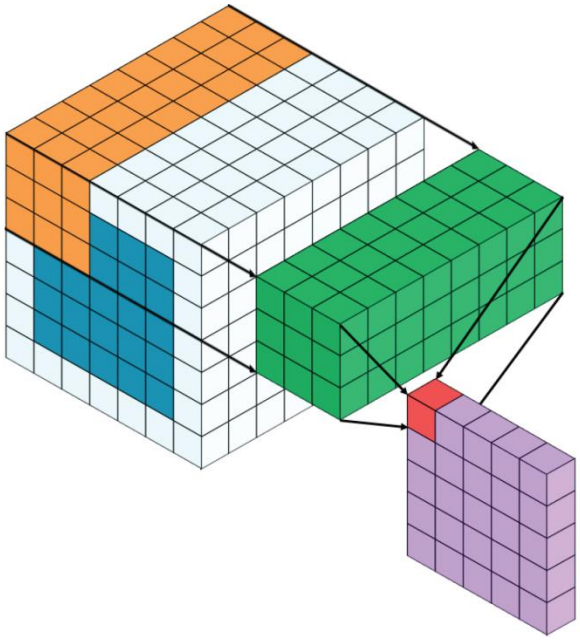

Рисунок 3.4 – Деталь згортки 3×3 з CH_{in}=8 , CH_{out}=1 , in_{size}=5, pad =1 та stride = 1.

Таблиця 3.3 – Компіляція та результати виконання згортки 3×3

	Єдине завдання	NDRange [16]	PyTorch
ALUTs	30109	255353	–
Реєстри	52305	38286	–
Логічні ресурси	23699/41910 (57%)	20028/41910 (48%)	–
Блоки DSP	29/112 (26%)	29/112 (26%)	–
Біти пам'яті	982288/5662720 (17%)	2831104/5662720 (50%)	–
Блоки оперативної пам'яті	202/553 (37%)	403/553 (73%)	–
F_{max} (МГц)	104,85	113,23	–
Затримка (с)	0,0723	0,0149	0,1795

3.1.3 Максимальний пул

В алгоритмі 3 на рисунку 3.5 ми бачимо рівняння, задіяні в шарі з максимальним пулом, а на рисунку 3.6 – графічну деталь для першої ітерації циклу, що міститься між рядками 3-14 цього алгоритму в двох різних каналах. Ця реалізація працює з фіксованим розміром фільтра (3×3), з прокладкою 0 і кроком 1. Отже, вона має працювати з out_{size} відповідно до розмірів in_{size} і цих префіксних параметрів.

У цьому розділі візуалізовано ресурси DE10-nano після компіляції за допомогою OpenCL HLS. Результати, представлені у таблиці 3.4, відповідають виконанню ядра, що реалізує шар max-pool з конфігурацією maxpool1 архітектури SqueezeNet v1.1.

Input: *in_size* (parameter)
Input: *out_size* (parameter)
Input: *channel_size* (parameter)
Input: *in_img* (buffer read only)
Output: *out_img* (buffer)

```
1:  for channel_index ← 0 To channel_size do
2:      for i ← 0 To (out_size) do
3:          for j ← 0 To (out_size) do
4:              tmp ← 0
5:              for l ← 0 To 3 do
6:                  for m ← 0 To 3 do
7:                      value ← in_img[(i × 2 + l) × in_size + j × 2 + m]
8:                      if value > tmp then
9:                          tmp ← value
10:                     end if
11:                 end for
12:             end for
13:             out_img[i × out_size + j] ← tmp
14:         end for
15:     end for
16:     in_img ← in_img × in_img + in_img
17:     out_img ← out_size × out_size + out_img
18: end for
```

Рисунок 3.5 – Max-pool

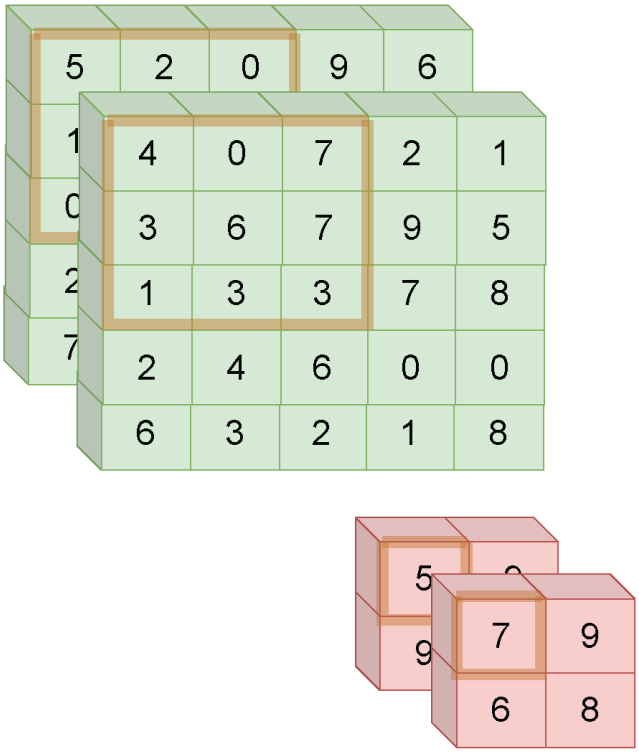

Рисунок 3.6 – Деталь шару max-pool з CH_{size}=2 , in_{size}=5 та out_{size}=2 pad = 0 та stride = 2

Таблиця 3.4 – Компіляція та результати виконання згортки 3×3

	Єдине завдання	NDRange [16]	PyTorch
ALUTs	6274	6318	–
Реєстри	10757	12317	–
Логічні ресурси	5432/41910 (13%)	6032/41910 (14%)	–
Блоки DSP	8/112 (7%)	12/112 (11%)	–
Біти пам'яті	183644/5662720 (3%)	234864/5662720 (4%)	–
Блоки оперативної пам'яті	68/553 (12%)	58/553 (10%)	–
F_{max} (МГц)	122,24	118,51	–
Затримка (с)	0,0308	0,0642	0,0947

3.1.4 Середній пул

В алгоритмі 4 на рисунку 3.7 ми бачимо рівняння, задіяні в шарі середнього пулу, а на рисунку 3.8 – графічну деталізацію для двох ітерацій циклу, що міститься між рядками 1-14 цього алгоритму. У таблиці 3.5 наведено результати синтезу коду шару середнього пулу на мові OpenCL, враховуючи, що розмір фільтра фіксований (13×13) і збігається з in_{size}. Цей шар відповідає останньому етапу обраної архітектури і дає нам результат класифікації.

Input: *in_img* (buffer read only)
Output: *out_img* (buffer)
1: **for** $class\_index \leftarrow 0$ To 1000 **do**
2: $tmp \leftarrow 0$
3: **for** $i \leftarrow 0$ To 169 **do**
4: $tmp \leftarrow bias$
5: **for** $i \leftarrow 0$ To $(in\_channels)$ **do**
6: $tmp \leftarrow in\_img[169 \times class\_index + i] + tmp$;
7: **end for**
8: $out\_im[class\_index] \leftarrow tmp/169$
9: **end for**
10: **end for**

Рисунок 3.7 – Середній пул

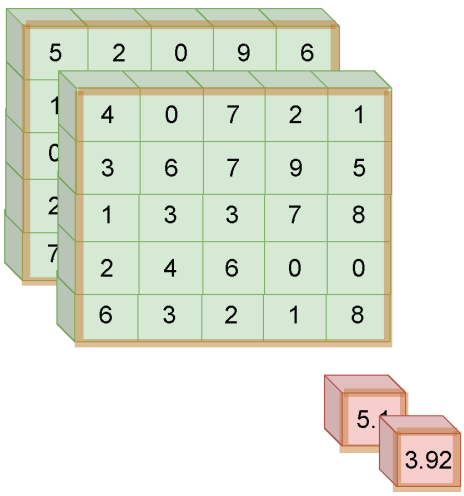

Рисунок 3.8 – Деталізація середнього пулу з CH_{size}=2, in_{size}=5 та out_{size}=2

Таблиця 3.5 – Компіляція та результати виконання згортки 3×3

	Єдине завдання	NDRange [16]	PyTorch
ALUTs	6178	6045	–
Реєстри	9303	8492	–
Логічні ресурси	4736/41910 (11%)	4633/41910 (11%)	–
Блоки DSP	4/112 (4%)	4/112 (4%)	–
Біти пам'яті	182798/5662720 (3%)	209162/5662720 (4%)	–
Блоки оперативної пам'яті	59/553 (11%)	49/553 (9%)	–
F_{max} (МГц)	133,16	117,28	–
Затримка (с)	0,0028	0,0034	0,0057

3.1.5 Повна модель

Якщо ми об'єднаємо чотири ядра, описані вище, і реалізуємо повну мережу, описану на рисунку 2.4, ми можемо побачити в таблиці 3.6, як нижча ефективність, досягнута в згортках 1×1 і 3×3, призводить до того, що постійне повторне використання цих ядер призводить до набагато нижчої продуктивності, ніж інші згадані апаратні рішення і наші програмні реалізації, засновані на бібліотеках PyTorch.

З точки зору використаних ресурсів, рішення можна порівняти з іншими рішеннями з продуктивністю в чотири рази вищою. Таким чином, незважаючи на роботу з рішенням для простих задач, яке, в принципі, є більш ефективним на пристроях FPGA, реальність така, що результати є невтішними.

Таблиця 3.6 – Результати компіляції та виконання повного OpenCL коду, що реалізує модель SqueezeNet v1.1

	Єдине завдання наше	PyTorch наше	NDRange [16]	ZynqNet [11]	Edgenet [12]	Int8 [13]	SqJ [14]
ПЛІС	Cyclone V	Cyclone V	Cyclone V	Zynq 7000	Cyclone V	Cyclone V	Zynq 7000
ALUTs	46615	–	446951	–	–		
Реєстри	826347	–	736337	137k	–	–	306554
Логічні ресурси	37k/42k (87%)	–	37k/42k (87%)	154k/218k (70%)	–	110k	
Блоки DSP	50/112 (45%)	–	65/112 (58%)	739/900 (82%)	–	–	
Біти пам'яті	1503k/ 5662k (27%)	–	4096k/ 5662k (72%)	–	–	–	186/200 46%
Блоки оперативної пам'яті	383/553 (69%)	–	553/553 (100%)	996/1090 (91%)	–	–	
F_{max} (МГц)	106,6	–	103,07	200	100	101,7	100
Затримка Fire (мс)	389	140	74	–	–	–	
Затримка блоку 3 (мс)	1203	379	331	–	–	–	
Глобальна затримка (мс)	4484	1231	1012	977	110	121	333

3.2 Архітектура систолічного розгортання

Фундаментальні внески в цих двох сферах вже були зроблені нашою групою в цій архітектурі наступним чином.

Систолічна генерація:

- перший внесок – це реалізація шарів згортки в парі, а не окремо. За допомогою цієї техніки ми можемо покращити зв'язок між шарами, виконуючи його внутрішньо в розробленому IC через канали. Блок обробки вогню SqueezeNet (SFPU – SqueezeNet fire processing unit) призначений для полегшення інтеграції шару стиснення і шару розгортання як єдиного цілого. Ця реалізація має бути достатньо універсальною, щоб її можна було багаторазово використовувати для реалізації різних блоків обробки Fire, які існують у згортковій мережі (таблиця 2.2);

- SFPU має бути фракціонованим, щоб ми могли реалізувати згортки 3×3 і 1×1 ізольовано, коли це необхідно (див. шари conv1 і conv3 в таблиці 2.2). Така реалізація має розширити можливості, доступні в опублікованих рішеннях. Вона має підтримувати розміри фільтрів, відмінні від 3×3, а також з конфігурованими прокладками та кроком;

- систолічна архітектура також повинна бути достатньо гнучкою, щоб реалізувати шари Max-pool та Average-pool з конфігурованими розмірами та характеристиками фільтрів;

- нарешті, вона повинна бути сумісною з можливістю реалізації щільних шарів, навіть якщо SqueezeNet не має такого типу шарів. З таким підходом ми могли б вирішити останні етапи створення CNN (LeNet, AlexNet, VGGNet).

Реалізація систолічної архітектури з використанням OpenCL:

- використання системи реалізації завдань, в якій проектовані обчислювальні елементи передбачаються функцією з властивістю autorun ядер. Відтепер цей елемент називатимемо проектованим процесорним блоком (PE – processing units);

- використання функцій для введення та виведення даних на основі двох технологій: буферів та черг;

- використання двох блоків керування, з яких він розподіляє свої керуючі сигнали решті ядер;

- використання потоку керування та потоку даних через канали (ексклюзивні для IntelFPGA OpenCL), які відіграють ключову роль у синхронізації всіх обчислювальних блоків.

Архітектура реалізації показана на рисунку 3.9. Вертикальні процесорні блоки в основному відповідають за стадію стиснення, а горизонтальні процесорні блоки – за стадію розширення. Однак останній має вищу складність і може реалізовувати шари з середнім та максимальним пулом.

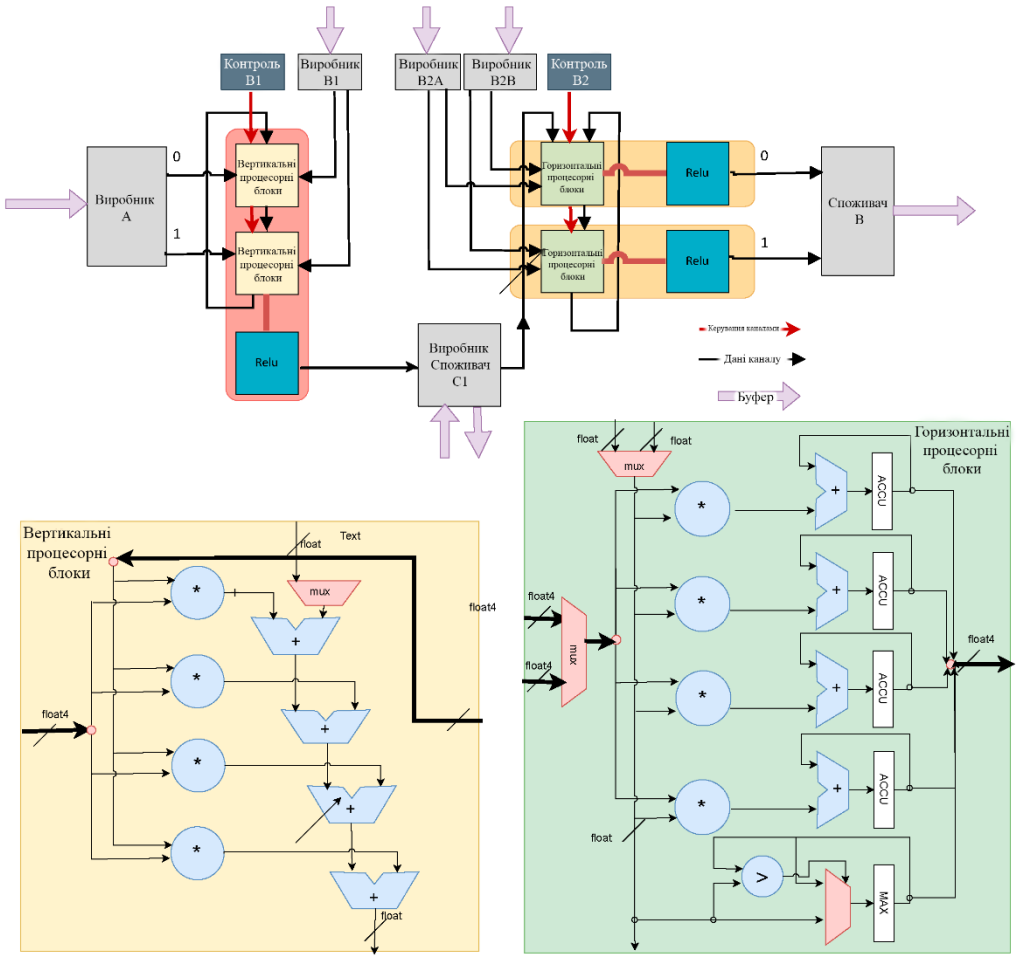

Рисунок 3.9 – Архітектура систолічного розгортання

Ця структура повторно використовується 13 разів для повної реалізації мережі (рисунок 3.10). Буфери (червоні блоки), які взаємодіють з блоками виробника та споживача, показані на рисунку 3.9, є фундаментальними. Ефективність зв'язку між ними та ядрами пристроїв і глобальною оперативною пам'яттю DE10-nano є критично важливою.

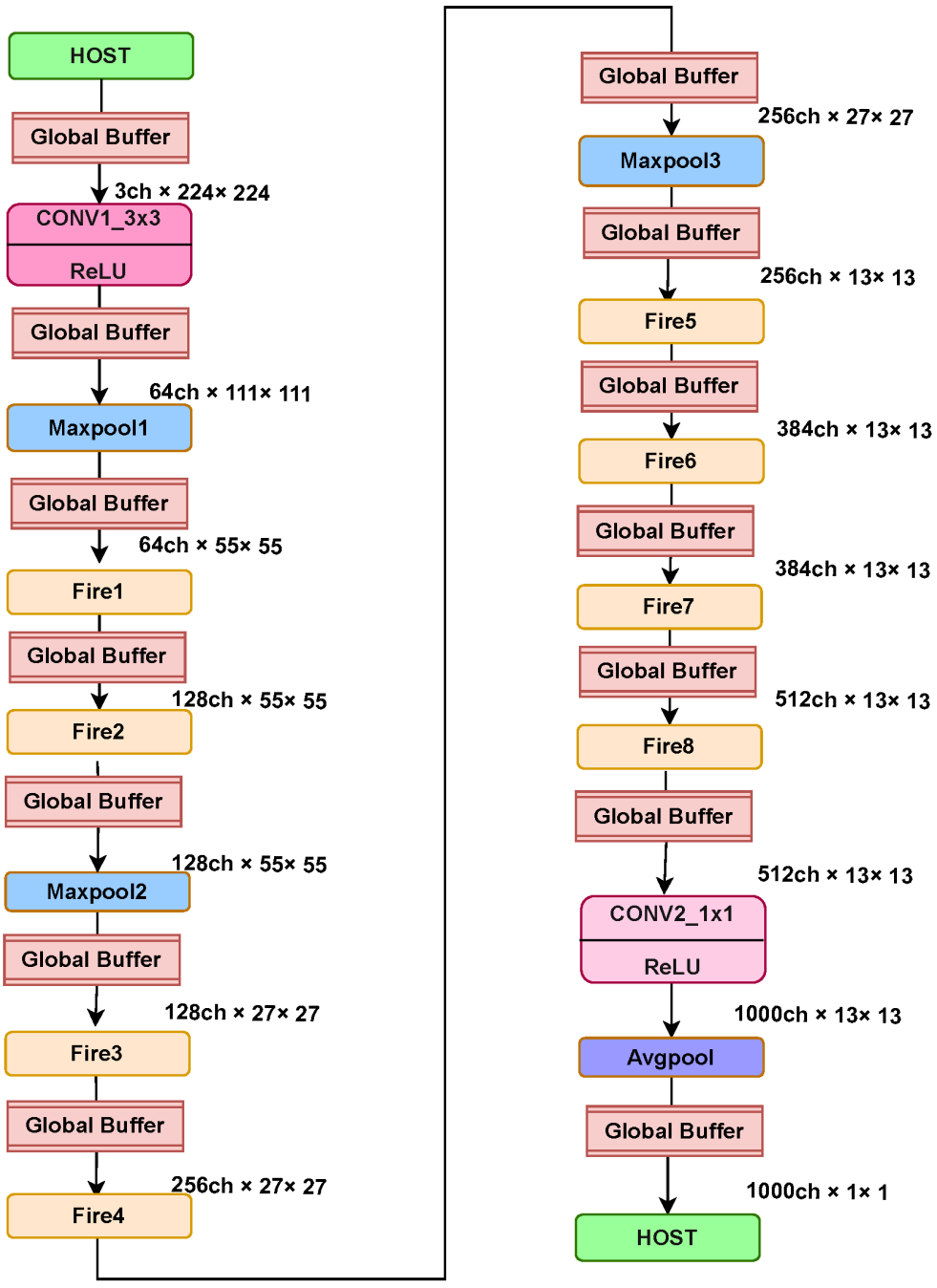

Рисунок 3.10 – Організація повторного використання архітектури розгортання для SqueezeNetv1.1.

3.3 Архітектура систолічного розгортання

Складена архітектура, яку ми описуємо на рисунку 3.11, дуже схожа на попередню, хоча краща ефективність досягається при комунікації між Fire1-Fire2, між Fire3-Fire4, між Fire5-Fire6 і між Fire7-Fire8, оскільки тепер це робиться через канали між ядрами. Звісно, контроль ускладнюється, але такий тип згортання може дозволити нам в майбутньому мати більше можливостей з реалізацією щільних шарів. Ми можемо чітко бачити, як покращується зв'язок на рисунку 3.12. З іншого боку, архітектура розгортання є більш адаптивною, ніж архітектура згортання, оскільки не обов'язково, щоб кількість вертикальних і горизонтальних обробних блоків була однаковою. Це пояснюється тим, що вона не передбачає зациклення між виходами та входами.

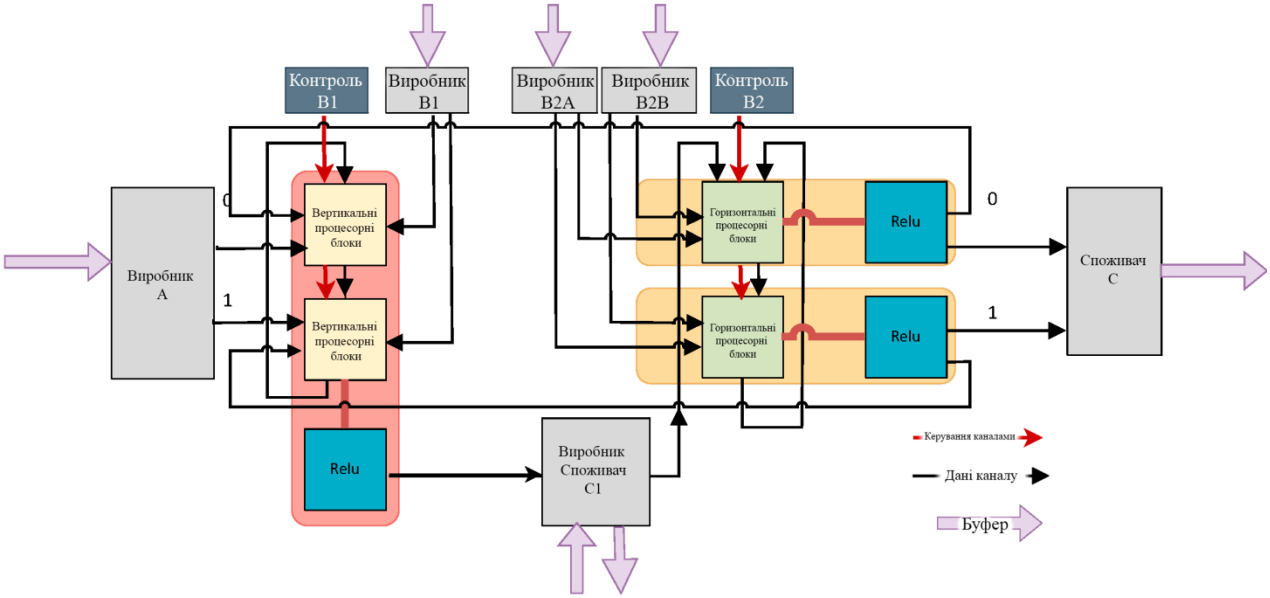

Рисунок 3.11 – Архітектура систолічного валика

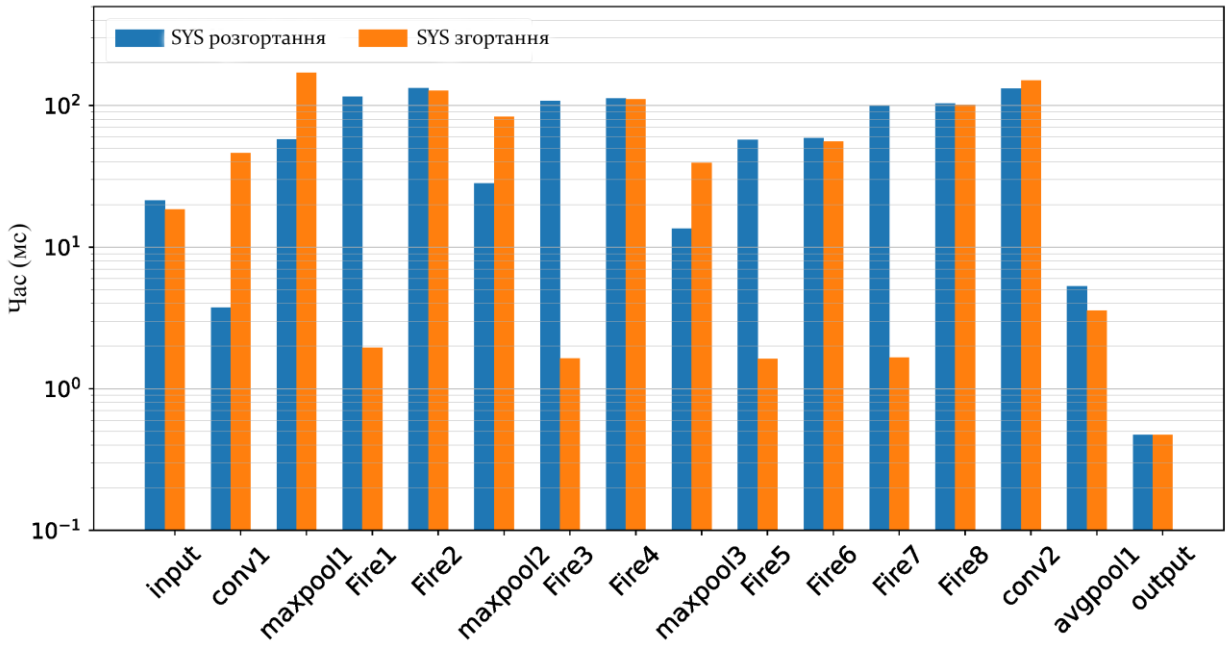

Рисунок 3.12 – Порівняння рівня Fire між версіями згортання та розгортання

3.3.1 Оптимізація: використання бібліотек RTL

Було розроблено власні бібліотеки для реалізації базових операцій (output = A×B + C та Accu = A×B + Accu) для різних технологій (Cyclone V та Arria 10). Ці бібліотеки виявилися дуже ефективними у заміні реалізації за замовчуванням компілятором OpenCL ([17,18]). Тому ми також використали їх у цій реалізації SqueezeNet. Результати можна побачити на рисунку 3.13, і немає сумнівів, що наші низькорівневі реалізації значно покращують результати при застосуванні в глибоких мережах. Цей рисунок зосереджено на реалізації згорток, в яких ми використовували ці функції. Зі збільшенням глибини мережі ця оптимізація чітко відображає досягнуте покращення, як показано на рисунку 3.14.

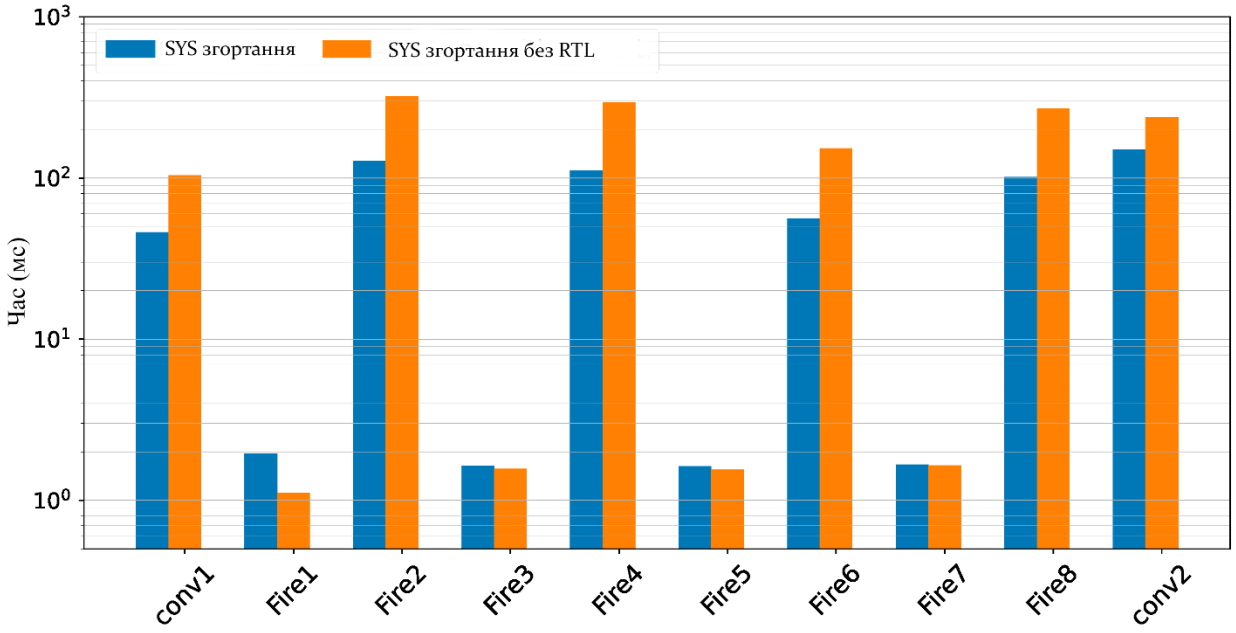

Рисунок 3.13 – Порівняння рівня Fire між версією згортання без RTL та версією згортання з RTL

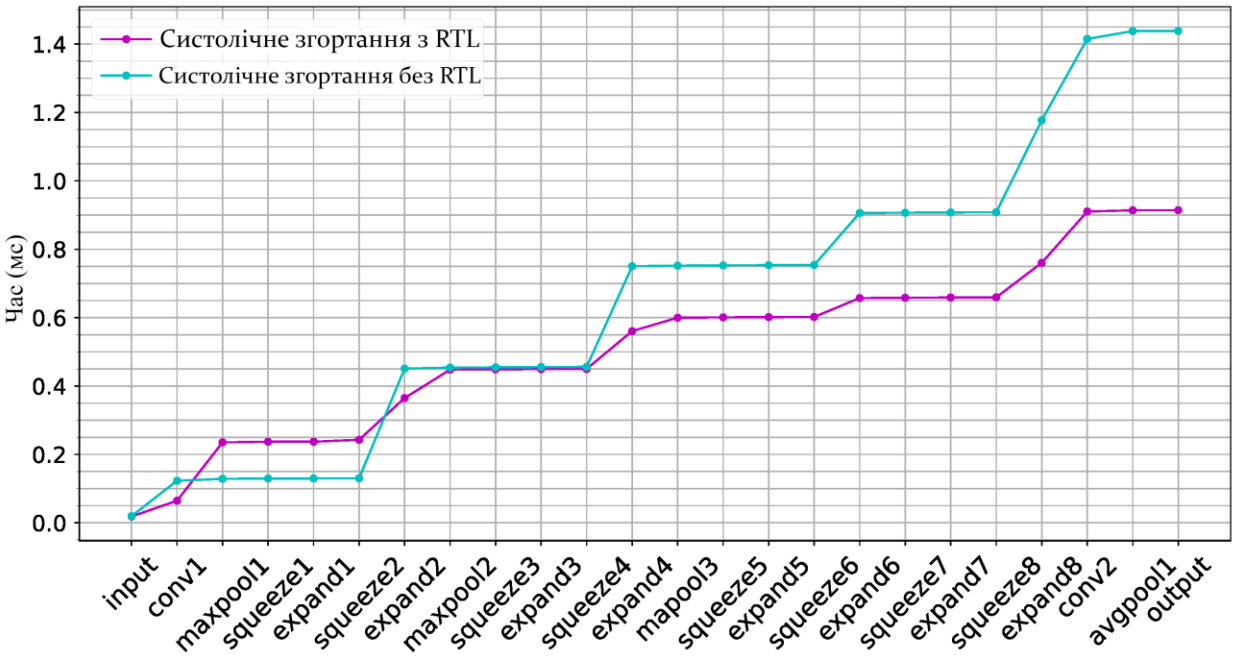

Рисунок 3.14 – Порівняння затримок при використанні RTL бібліотек та без використання RTL бібліотек

3.3.2 Оптимізація: використання гетерогенних рішень

Більш уважний погляд на рисунок 3.12 показує погіршення реалізації шарів max-pool при зміні архітектури між версією згортання та версією розгортання. Ця обставина, разом з необхідністю зменшити кількість ресурсів, що використовуються у ПЛІС, призвела до реалізації шару max-pool за допомогою простого ядра, а реалізацію середнього пулу хостом через PyTorch. Передавання між ядрами хоста та ПЛІС не є можливим рішенням всередині глибокої мережі, але воно можливе на початку та в кінці мережі, оскільки обидва кінці знаходяться в межах досяжності хоста. Наслідком такої оптимізації є змішана архітектура, яка відображається в результатах.

3.3.3. Оптимізація: передача даних

Ми виконали всі комунікації між хостом та ядром (через буфери) та між ядрами (через канали) з даними у форматі IEEE 754 одинарної точності. Якщо ми замінимо цю передачу даних на формат IEEE 754 з половинною точністю, ми зможемо значно підвищити ефективність нашої реалізації, оскільки ми зможемо подвоїти блоки множення та додавання наших процесорних блоків. Скажімо, ми можемо збільшити нашу обчислювальну потужність (однаково реалізовану з форматом IEEE 754 одинарної точності) за рахунок точності передачі даних.

4 РЕЗУЛЬТАТИ ТА ОЦІНКА ЕФЕКТИВНОСТІ

4.1 Оцінка ефективності

У застосунках, що використовують ШІ на передовій, затримка системи і, як наслідок, пропускна здатність програми зазвичай є найбільш затребуваними показниками продуктивності.

Не дивно, що кількість кадрів в секунду (FPS – frames per second) є широко використовуваною метрикою для оцінки ефективності додатків комп'ютерного зору. Цей показник використовувався в цьому дослідженні для порівняння наших результатів з результатами інших досліджень. Тим не менш, важливо пам'ятати, що цей параметр сильно залежить від топології мережі, яка включає вхідний шар, що має розмір вихідного зображення.

Більш інтригуючим і не залежним від топології є визначення пропускної здатності як функції кількості мережевих параметрів, що обробляються за секунду, оскільки прийнято вважати, що кожне синаптичне з'єднання вимагає одного параметра (однієї MAC-операції).

Слід уточнити, чи виконуються ці операції з фіксованою або плаваючою комою, і який саме тип представлення використовується. Ми можемо оцінити вплив на точність на різних рівнях, але це завжди буде залежати від використовуваної програми.

Важливість енергоефективності особливо актуальна, коли ШІ-обробка виконується на вбудованих пристроях з обмеженою ємністю акумулятора [19]. Зазвичай цей показник вимірюється в кількості операцій на джоуль, але в цій роботі використовувався показник мВт/Мпс, або енергія на операцію. Це важлива величина при порівнянні різних виробників. Єдиною проблемою з цією метрикою є визначення того, як була оцінена потужність (розрахована або виміряна); значення, знайдені в багатьох дослідженнях, є оцінками

проектних середовищ з ПЛІС (Vivado або Quartus), і часто буває важко встановити, що було зроблено з потужністю, яку споживає процесорна система (PS – processor system) в системах на кристалі (SOC – systems-on-chips).

Нарешті, важливо враховувати гнучкість та архітектурну адаптацію нашої реалізації. Інші реалізації можуть бути кращими з точки зору метрик, але вимагати трудомісткої апаратної перекомпіляції для будь-якої зміни топології. Крім того, реалізації, які залежать від розміру групи (наприклад, реалізації OpenCL на основі ядра NDrange), можуть бути ефективними з пакетом вхідних векторів, але це не є реалістичною ситуацією при виведенні. Тому наша реалізація має перевагу з точки зору гнучкості та архітектурної адаптації.

4.2 Ресурси

У таблиці 4.1 ми показуємо ресурси, що використовуються в кожній з розроблених архітектур. Вона не є важливою як елемент порівняння, оскільки насправді важливо те, що ми можемо реалізувати її на пристрої DE10-nano. Однак, вона корисна як довідковий елемент, коли ми хочемо порівняти продуктивність кожного рішення в наступному підрозділі.

4.3 Продуктивність

На цьому етапі ми покажемо результати швидкості, досягнуті запропонованими нами систолічними архітектурами. Для цього ми показуємо повний час роботи мережі SqueezeNet версії v1.1 (рисунок 4.1) і деталізацію часу виконання кожної фази мережі (рисунок 4.2). На кожному з двох рисунків ми також включили для порівняння чисте програмне рішення, розроблене за допомогою PyTorch (двоядерний ARM Cortex-A9 плюс fpu NEON) на тому ж пристрої, і апаратне рішення на основі NDrange з тих, що

згадуються в літературі і використовують той же пристрій.

Таблиця 4.1 – Результати компіляції повного OpenCL коду, що реалізує модель SqueezeNet v1.1

	Розгортання архітектури (Розділ 3.2)	Згортання архітектури (Розділ 3.3.1)	Змішана архітектура (Розділ 3.3.2)	Мікс-гібридна архітектура (Розділ 3.3.3)
ALUTs	48454	42878	44596	58680
Реєстри	75051	66240	69582	72497
Логічні ресурси	37463/41910 (89%)	31682/41910 (87%)	32426/41910 (77%)	37656/41910 (90%)
Блоки DSP	57/112 (51%)	61/112 (54%)	61/112 (54%)	77/112 (69%)
Біти пам'яті	3670872/ 5662720 (65%)	3627956/ 5662720 (64%)	2795292/ 5662720 (49%)	1764372/ 5662720 (31%)
Блоки оперативної пам'яті	553/553 (100%)	553/553 (100%)	502/553 (91%)	393/553 (71%)
F_{max} (МГц)	108,31	98,39	97,05	95,43

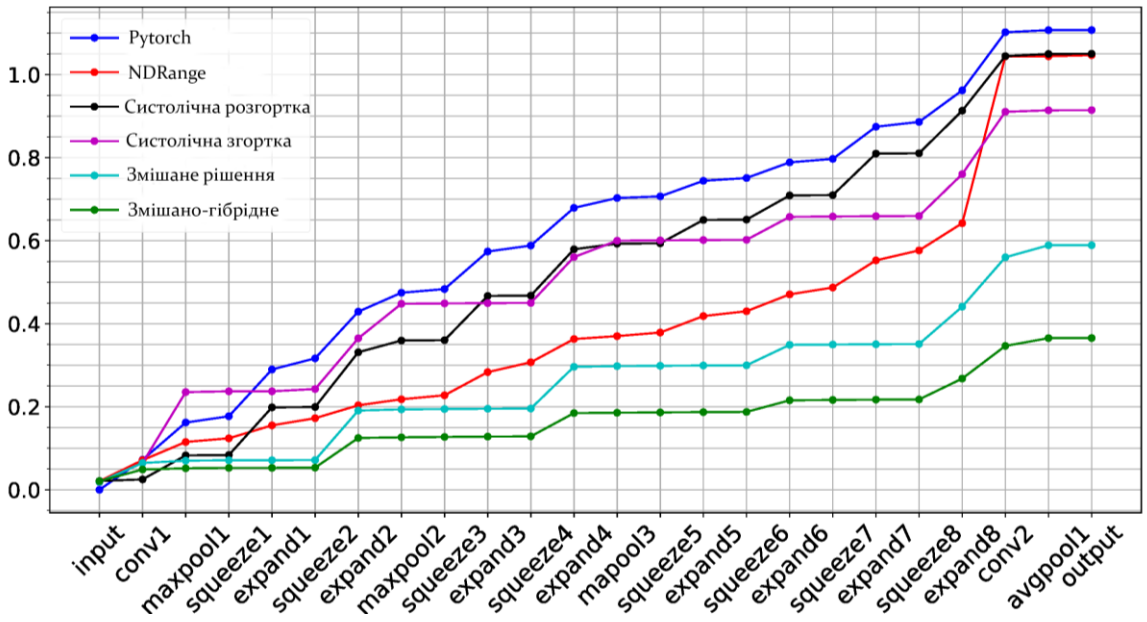

Рисунок 4.1 – Порівняння повного мережевого виводу з накопиченим часом

Для кожної із запропонованих архітектур можна побачити, де знаходяться можливі точки покращення. Наприклад, у нашій найкращій реалізації (змішане рішення) вузькі місця знаходяться в шарах розширення рівних Fire блоків, а також у фінальній згортці 1×1 (яку ми називаємо conv2).

Також можна помітити, що фінальна згортка 1×1 також може бути передбачена програмним забезпеченням за допомогою PyTorch. Розглядаючи графіки (рисунок 4.1 та рисунок 4.2), видно, що наш підхід призвів до значного покращення порівняно з попередніми рішеннями.

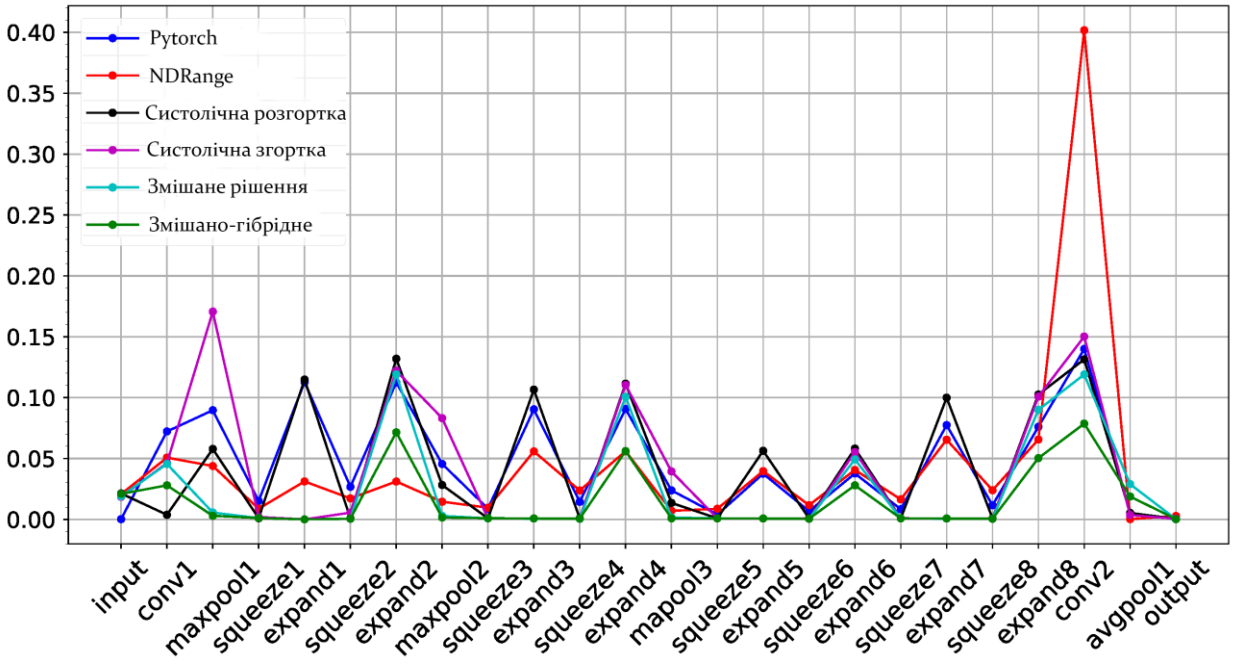

Рисунок 4.2 – Порівняння повного мережевого виводу з фазовим часом

4.4 Енергоефективність

Як ми вже говорили раніше, важливо зробити таке ж порівняння з точки зору енергоефективності. Для цього ми додаємо таблицю 4.2.

Таблиця 4.2 – Результати компіляції повного OpenCL коду, що реалізує модель SqueezeNet v1.1

	Одиничне завдання	ND Range	PyTorch	Розгортання архітектури	Згортання архітектури	Змішана архітектура	Мікс-гібридна архітектура
Енергія на вихід (Дж)	32,4	10,8	3,6	7,2	10,8	5,04	2,88
Потужність (мВт)	7523,55	10641,33	3716,07	6418,91	7928,37	7626,50	7447,42
мВт/Мп	27427,31	8780,49	2926,83	5853,65	8780,48	4097,56	2341,46

Очевидно, що остання з наших архітектур є більш енергоефективною, ніж реалізація алгоритму виключно за допомогою програмного рішення (PyTorch). Однак, з точки зору застосування, час виконання має велике значення, і наші систолічні архітектури, реалізовані на OpenCL, дозволили нам скоротити цей час виконання. Це дослідження енергоефективності ставить під сумнів придатність багатьох програмно-апаратних рішень з точки зору енергоефективності в недорогих пристроях.

4.5 Підсумки

Процесори, доступні в пристроях з обмеженими ресурсами, забезпечують дуже високу продуктивність завдяки використанню векторних прискорювачів (в даному випадку Neon) і навряд чи виправдовують використання частини ПЛІС для додаткового прискорення, за винятком випадків, коли нам потрібно звільнити мікропроцесор для виконання інших завдань. Нам вдалося підвищити енергоефективність наших ядер, але ціною втрати ресурсів ПЛІС, які могли б знадобитися для генерації та збору сигналів, необхідних для спеціалізованого приладу.

Хотілось би підкреслити ефективність компіляторів HLS. Ми виявили, що включення в них наших власних бібліотек RTL може значно підвищити їх продуктивність. Крім того, ми помітили, що чим більш деталізованою є система (що призводить до більшої кількості ядер, які необхідно з'єднати каналами), тим більше знижується ефективність систем потоку даних. Рішення може полягати в системі зв'язку між ядрами (канали, труби або потокові інтерфейси); однак не схоже, що вони можуть бути адаптовані або спроектовані на рівні RTL.

ВИСНОВКИ

У цій роботі представлено робочий процес для реалізації глибоких нейронних мереж, який поєднує гнучкість мереж на основі HLS з архітектурними можливостями управління потоками на основі HDL. OpenCL був основним інструментом, який використовувався в цьому робочому процесі, оскільки він забезпечує структурний підхід і високий рівень апаратного контролю, що особливо важливо при роботі з систолічними архітектурами.

З точки зору верифікації, запропонований метод на основі Pyopencl-Jupyter Notebook є ефективним, оскільки еталонні моделі у всіх тестах, як для окремих шарів, так і для всієї мережі, були легко доступні за допомогою пакетів, добре відомих у розробці, навчанні та виведенні глибоких мереж (PyTorch).

З точки зору результатів, нам вдалося зменшити час виведення SqueezeNet v1.1 на 0,4 с, працюючи над усіма обчисленнями з плаваючою комою одинарної точності (32 біти), і нам вдалося перевершити в енергоефективності результати, виміряні в рішеннях, які не використовували інфраструктуру програмованої логіки, доступну в пакеті.

Звичайно, результати легко перевершити при роботі з більш ресурсоємними пристроями, але вони стають нездійсненними в рішеннях AI-at-the-edge. Ми побачили, що в глибоких мережах єдиним рішенням є безперервне повторне використання гнучкої архітектури, і проміжний крок через глобальну пам'ять абсолютно необхідний. Таке повторне використання, разом з втручанням глобальної пам'яті, значно знижує результати реалізації глибоких мереж у порівнянні з поверхневими мережами.

Важливість взаємозв'язку між шарами в глибоких нейронних мережах очевидна і повинна бути ретельно вивчена. Наші наступні кроки будуть

спрямовані на покращення міжшарових передач шляхом покращення взаємозв'язку між FPGA та SDRAM, підключеними до системи жорсткого процесора (HPS).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Singh, R.; Gill, S.S. Edge AI: A survey. Internet Things Cyber-Phys. Syst. 2023, 3, 71–92.

2. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5 MB model size. arXiv 2016, arXiv:1602.07360. Available online: http://arxiv.org/abs/1602.07360.

3. Lee, H.J.; Ullah, I.; Wan, W.; Gao, Y.; Fang, Z. Real-Time Vehicle Make and Model Recognition with the Residual SqueezeNet Architecture. Sensors 2019, 19, 982.

4. Kwaghe, O.P.; Gital, A.Y.; Madaki, A.; Abdulrahman, M.L.; Yakubu, I.Z.; Shima, I.S. A Deep Learning Approach for Detecting Face Mask Using an Improved Yolo-V2 With Squeezenet. In Proceedings of the 2022 IEEE 6th Conference on Information and Communication Technology (CICT), Gwalior, India, 18–20 November 2022; pp. 1–5.

5. Fang, C.; Lv, C.; Cai, F.; Liu, H.; Wang, J.; Shuai, M. Weather Classification for Outdoor Power Monitoring based on Improved SqueezeNet. In Proceedings of the 2020 5th International Conference on Information Science, Computer Technology and Transportation (ISCTT), Shenyang, China, 13–15 November 2020; pp. 11–15.

6. Zhang, J.; Zhu, H.; Wang, P.; Ling, X. ATT Squeeze U-Net: A Lightweight Network for Forest Fire Detection and Recognition. IEEE Access 2021, 9, 10858–10870.

7. Tsalera, E.; Papadakis, A.; Voyiatzis, I.; Samarakou, M. CNN-based, contextualized, real-time fire detection in computational resource-constrained environments. Energy Rep. 2023, 9, 247–257.

8. Yang, Z.; Yang, X.; Li, M.; Li, W. Automated garden-insect recognition using improved lightweight convolution network. Inf. Process. Agric. 2023, 10,

256–266.

9. Huang, Q. Weight-Quantized SqueezeNet for Resource-Constrained Robot Vacuums for Indoor Obstacle Classification. AI 2022, 3, 180–193.

10. Team, P. Pytorch Vision SQUEEZENET Model. Available online: https://pytorch.org/hub/pytorch_vision_squeezenet.

11. Barkovska, O., Filippenko, I., Semenenko, I., Korniienko, V., & Sedlaček, P. (2023). Адаптація FPGA архітектури для прискорення алгоритмів обробки зображень. Радіоелектронні і комп'ютерні системи, 0(2), 94-106. doi:https://doi.org/10.32620/reks.2023.2.08.

12. Pradeep, K.; Kamalavasan, K.; Natheesan, R.; Pasqual, A. EdgeNet: SqueezeNet like Convolution Neural Network on Embedded FPGA. In Proceedings of the 2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Bordeaux, France, 9–12 December 2018; pp. 81–84.

13. Zhao, J.; Yin, Z.; Zhao, Y.; Wu, M.; Xu, M. Scalable FPGA-Based Convolutional Neural Network Accelerator for Embedded Systems. In Proceedings of the 2019 4th International Conference on Computational Intelligence and Applications (ICCIA), Nanchang, China, 21–23 June 2019; pp. 36–40.

14. Mousouliotis, P.G.; Petrou, L.P. SqueezeJet: High-Level Synthesis Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the Applied Reconfigurable Computing. Architectures, Tools, and Applications, Santorini, Greece, 2–4 May 2018; Voros, N., Huebner, M., Keramidas, G., Goehringer, D., Antonopoulos, C., Diniz, P.C., Eds.; Springer Nature: Cham, Switzerland, 2018; pp. 55–66.

15. Vyshnivskyi, D., Liashenko, O. і Yeromina, N. 2023. Система оцінка пози людини з використанням алгоритмів глибокого навчання. Системи управління, навігації та зв’язку. Збірник наукових праць. 2, 72 (Чер 2023), 75-79. DOI:https://doi.org/https://doi.org/10.26906/SUNZ.2023.2.075.

16. Ilyashov, O., Pokora, K., Diachenko, V. і Kovalenko, A. 2023. Класифікація даних апаратними прискорювачами FPGA у центрах обробки

даних та хмарах. Системи управління, навігації та зв’язку. Збірник наукових праць. 2, 72 (Чер 2023), 106-112. DOI:https://doi.org/https://doi.org/10.26906/SUNZ.2023.2.106.

17. Gadea-Gironés, R.; Fe, J.; Monzo, J.M. Task parallelism-based architectures on FPGA to optimize the energy efficiency of AI at the edge. Microprocess. Microsyst. 2023, 98, 104824.

18. Gadea-Gironés, R.; Herrero-Bosch, V.; Monzó-Ferrer, J.; Colom-Palero, R. Implementation of Autoencoders with Systolic Arrays through OpenCL. Electronics 2021, 10, 70.

19. Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. How to Evaluate Deep Neural Network Processors: TOPS/W (Alone) Considered Harmful. IEEE Solid-State Circuits Mag. 2020, 12, 28–41.