Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет __Комп'ютерної інженерії та управління_____
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти____перший (бакалаврський)_____
(рівень вищої освіти)

Апаратний прискорювач математичних операцій на платформі SoC
(тема)

Виконав:
здобувач ____4_____ року навчання,
групи КІУКІ-21-8

_____Караченцев К.В._____
(прізвище, ініціали)

Спеціальність *123 _ Комп'ютерна інженерія*

Тип програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма_____

_____*Комп'ютерна інженерія*_____
(повна назва освітньої прграми)

Керівник _доц. Шкіль О.С._
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ _Чумаченко С.В
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління

Кафедра Автоматизації проектування обчислювальної техніки

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123_Комп'ютерна інженерія
(шифр і назва)

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

Освітня програма Комп'ютерна інженерія
(повна назва

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«__02__»___05_____ 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Караченцеву Костянтину Володимировичу
(прізвище, ім'я, по батькові)

1. Тема роботи (проекту) Апаратний прискорювач математичних операцій на платформі SoC

затверджена наказом по університету від від " 21 "_____05_____2025 р. № 403 Ст

2. Термін подання студентом роботи до екзаменаційної комісії _____10.06.2025_____

3. Вихідні дані до роботи (проекту)

Налагоджувальна плата ZedBoard з кристалом ZYNQ-7000 фирмы Xilinx Inc.

Мова програмування С

Мови опису апаратури VHDL, Verilog

САПР XILINX ISE, Vivado, Vitis

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):

Матричні операції лінійної алгебри

Технології проектування SoC

Інструментальні засоби проектування SoC

Реалізація апаратного прискорювача математичних операцій на платі ZedBoard з кристалом ZYNQ-7000

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) _____
14 слайдів

6. Консультанти розділів роботи (п.6 включається до завдання 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 (проекту)	Термін виконання етапів проекту (роботи)	Примітка
1	Видача теми проекту, узгодження і затвердження	02.05.2025 -05.05.2025	
2	Аналіз проблемної галузі, постановка задачі, вибір інструментальних засобів	05.05.2025 -10.05.2025	
3	Аналіз технологій проектування та діагностування систем на кристалі	10.05.2025 -17.05.2025	
5	Вибір налагоджувальної плати та інструментарію для реалізації проекту	18.05.2025 -25.05.2025	
5	Реалізація проекту математичного сопроцесора	25.05.2025 -30.05.2025	
6	Фізична реалізація проекту в налагоджувальній платі	31.05.2025 -05.06.2025	
7	Оформлення пояснювальної записки	05.06.2025 -10.06.2025	
8	Перевірка виконаного проекту керівником,	10.06.2025 -12.06.2025	
9	Захист проекту	12.06.2025 -20.06.2025	

Дата видачі завдання 02.05.2025

Здобувач _____
(підпис)

Керівник роботи (проекту) _____ доц. Шкіль О.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Записка пояснювальна: 50 сторінок, 21 рисунок, 11 джерел за переліком посилань.

МНОЖЕННЯ МАТРИЦЬ, АПАРАТНИЙ ПРИСКОРЮВАЧ, СИСТЕМА НА КРИСТАЛІ ZYNQ-7000, ПЛАТА РОЗРОБКИ ZEDBOARD, САПР VIVADO/VITIS, МОВА ПРОГРАМУВАННЯ С

Метою кваліфікаційної роботи є реалізація апаратного прискорювача обчислювальних алгоритмів множення матриць з використанням SoC сімейства ZYNQ-7000 на налагоджувальній платі ZedBoard.

Проведена оцінка обчислювальної складності різних алгоритмів прискорення множення матриць та способів їх апаратної реалізації. Реалізація апаратного прискорювача виконана на базі стеку інструментальних засобів САПР Vivado/Vitis HLS з використанням мови програмування С. Показані переваги у швидкодії апаратного прискорювача на програмованій логіці перед програмною реалізацією обчислювальних алгоритмів на програмних компонентах вбудованих систем.

ABSTRACT

Explanatory note: 50 pages, 21 figures, 11 sources according to the list of links.

MATRIX MULTIPLICATION, HARDWARE ACCELERATION, ZYNQ-7000 SYSTEM ON CHIP, ZEDBOARD DEVELOPMENT BOARD, VIVADO/VITIS CAD, C PROGRAMMING LANGUAGE

The purpose of the qualification work is to implement a hardware accelerator for matrix multiplication computational algorithms using the ZYNQ-7000 SoC family on the ZedBoard debug board.

The computational complexity of various matrix multiplication acceleration algorithms and methods for their hardware implementation were assessed. The hardware accelerator was implemented based on the Vivado/Vitis HLS CAD tool stack using a C programming language. The advantages in performance of a hardware accelerator on programmable logic over software implementation of computational algorithms on software components of embedded systems are shown.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПК – персональний комп’ютер;

ПЛІС – програмована логічна інтегральна схема;

ЦП – центральний процесор (CPU);

AMBA – Advanced Microcontroller Bus Architecture;

ARM – Advanced RISC Machine (32-бітна процесорна архітектура);

AXI – Advanced eXtensible Interface (розширений розширюваний інтерфейс);

CUDA – Compute Unified Device Architecture (програмно-апаратна архітектура паралельних обчислень);

FPGA – field-programmable gate array (різновид ПЛІС);

GPU – graphics processing unit (графический процессор);

HLS – High-Level Synthesis (вісокорівневий синтез);

HW – hardware (апаратне забезпечення);

HDL – hardware description language (мова опису апаратури);

IDE – Integrated Development Environment (інтегроване середовище розробки);

IP-core – intellectual property core (IP-ядра, готові блоки для проектування SoC);

PL – programmable logic (блок програмованої логіки SoC);

PS – processing system (процесорний блок SoC);

RAM – Random Access Memory (оперативна пам’ять);

ROM – Read Only Memory (постійний запам’ятовуючий пристрій);

RTL – Register Transfer Level (рівень регістрових передач);

SoC – System on Chip (система на кристалі);

SW – software (програмне забезпечення);

VHDL – very high speed integrated circuits HDL (одна з мов опису апаратури);

ВСТУП

У своєму безперервному розвитку ринок мікроелектроніки постійно висуває все нові і більш жорсткі вимоги до виробів, що з'являються. Споживач хоче отримувати швидкодіючу, надійну і водночас малогабаритну продукцію, що мало споживає. Це вирішується за рахунок проектування вбудованих систем, що реконфігуруються.

У програмах, що розробляються сьогодні, для вбудованих електронних систем потрібно отримувати, обробляти і передавати складні набори даних. Вони можуть надходити з різних джерел, таких як датчики довкілля, камери нерухомого зображення та відеокамери. Після отримання та збереження в електронній пам'яті до даних здійснюється доступ та їх обробка з використанням відповідних математичних алгоритмів. Те, як дані зберігаються, яких здійснюється доступ, обробляються і передаються, вплине вартість обробки даних. Такі алгоритми традиційно реалізуються з використанням звичайних програмних програм, які виконуються на універсальному процесорі. Однак можна розглянути різні підходи до створення архітектури цифрової системи, яка б складалася з пам'яті, підсистем обробки та комунікаційної логіки. При розгляді математики, що лежить в основі процесів проектування, це призводить до створення системних архітектур, які можуть бути оптимізовані для реалізації необхідного алгоритму чи груп алгоритмів. Математика масивів – це клас операцій, який підтримує обчислення в n-вимірних масивах з використанням форм масиву та індексації значень, що зберігаються в масиві. Найчастіше мова йде про двомірні масиви, яки прийнято називати матрицями.

Такі завдання вирішуються за рахунок застосування процесорів, що конфігуруються. Конфігурований процесор реалізує пристрій, який може бути спрямоване для конкретного використання у вирішенні конкретного класу завдань. Наприклад, змінний набір інструкцій процесорного ядра,

додавання/виключення апаратного множення, кількість станів внутрішнього конвеєра, що програмується, тощо – все це може бути оптимізовано для кожної кінцевої програми. Результатом є оптимізоване, високопродуктивне та дешеве рішення для конкретного завдання.

Математичний співпроцесор – співпроцесор для розширення командної множини центрального процесора, він забезпечує його функціональністю модуля операцій з плаваючою комою, матричних операцій, перетворень Фур'є, нейронних мереж тощо. Він може являти собою пристрій в окремому корпусі, так знаходитися разом з центральним процесором на одному кристалі в якості вбудованої системи.

Вбудована система (embedded system) – це кінцева система певного виду, яка має одну або кілька спеціалізованих функцій в об'ємній системі, на відміну від компонентів загального призначення; тут всі вузли функціонують разом, згідно з набором правил, на підставі узгодженого та затвердженого плану. Технологічною платформою для побудови вбудованих систем є системи на кристалі. (System-on-Chip, SoC). SoC – надвелика інтегральна схема, що містить на кристалі різні складні функціональні блоки (СФ-блоки), які утворюють закінчений виріб для автономного застосування в електронній апаратурі. СФ-блоки, призначені для використання у різноманітних проектах, часто називають IP-модулями (Intellectual Property – модулі). При цьому до складу СФ-блоків входить мікропроцесорне ядро з периферійними пристроями у різних поєднаннях.

Таким чином, розробка апаратних прискорювачів математичних операцій на технологічній платформі SoC є актуальним завданням.

1 МАТЕМАТИЧНІ ОСНОВИ АПАРАТНОЇ ТЕАЛІЗАЦІЇ МАТРИЧНИХ ОПЕРАЦІЙ

1.1 Математичні основи алгоритмів множення матриць

Матриця - це прямокутна таблиця, що складається з рядків та стовпців, що містять числа. Загальний вигляд матриці має такий вигляд (рис. 1.1):

$$A = \begin{matrix} \\ \text{column} \downarrow \end{matrix} \overset{\text{row} \rightarrow}{\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & . & . & a_{1,m} \\ a_{2,1} & a_{2,2} & a_{2,3} & . & . & a_{2,m} \\ a_{3,1} & a_{3,2} & a_{3,3} & . & . & a_{3,m} \\ . & . & . & . & . & . \\ a_{n,1} & a_{n,2} & a_{n,3} & . & . & a_{n,m} \end{pmatrix}}$$

Рисунок 1.1 – Загальний вигляд матриці

Елементи матриці позначаються $a_{n,\,m}$,, де m – номер рядка, а n – номер стовпця, на перетині яких знаходиться даний елемент. Приклади матриць наведені на рис. 1.2

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 3 & 1 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 5 \\ 8 & 7 \\ 3 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 5 & 2 \\ 8 & 7 & 3 \\ 3 & 1 & 4 \end{pmatrix} \quad D_l = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Рисунок 1.2 – Приклади матриць

А - матриця з 2 рядків та 3 стовпців, в якій число 2 стоїть у першому рядку, третьому стовпці. В - матриця з 3 рядків та 2 стовпців, в якій число 8 стоїть у другому рядку, першому стовпці. Матриця з рівним числом рядків та стовпців називається квадратною матрицею. С - матриця з 3 рядків та 3

стовпців

D – загальний вигляд квадратної матриці (рис. 1.3). Елементи головної діагоналі виділені червоним кольором, але в побічної діагоналі - синім. Квадратна матриця, у якої всі елементи головної діагоналі дорівнюють 1, а всі інші елементи дорівнюють 0, називається одиничною матрицею (D_1).

$$D = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdot & \cdot & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdot & a_{2,n-1} & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdot & \cdot & a_{3,n} \\ \cdot & a_{n-1,2} & \cdot & \cdot & \cdot & \cdot \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdot & \cdot & a_{n,n} \end{pmatrix}$$

Рисунок 1.3 – Загальний вигляд квадратної матриці

Транспонована матриця виходить, якщо у вихідній матриці замінити рядки на стовпці. Якщо дана матриця A, транспонована матриця A позначається A^T (рис. 1.4).

$$A = \begin{pmatrix} 1 & 3 \\ 5 & 9 \end{pmatrix} \quad \text{тоді} \quad A^T = \begin{pmatrix} 1 & 5 \\ 3 & 9 \end{pmatrix}$$

Рисунок 1.4 – Транспонування матриці

Результатом множення матриць $A_{m\times n}$ і $B_{n\times k}$ буде матриця $C_{m\times k}$ така, що елемент матриці C, що стоїть у i-му рядку і j-му стовпці (c_{ij}), дорівнює сумі добутків елементів i-того рядка матриці A на відповідні елементи j-того стовпця матриці B: $c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \ldots + a_{in} \cdot b_{nj}$, або у загальному вигляді:

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = 1, 2, \ldots k;\ j = 1, 2, \ldots n).$$

Дві матриці можна перемножити між собою тоді і лише тоді, коли

кількість стовпців першої матриці дорівнює кількості рядків другої матриці.

Властивості множення матриць

$(A \cdot B) \cdot C = A \cdot (B \cdot C)$ – добуток матриць асоціативний;

$(z \cdot A) \cdot B = z \cdot (A \cdot B)$, де z – число;

$A \cdot (B + C) = A \cdot B + A \cdot C$ – добуток матриць дистрибутивний;

$D_n \cdot A_{nm} = A_{nm} \cdot D_m = A_{nm}$ – множення на одиничну матрицю D_n;

$A \cdot B \neq B \cdot A$ – у загальному випадку добуток матриць не коммутативний.

Добутком двох матриць є матриця, у якої стільки рядків, скільки їх у лівого множника, і стільки стовпців, скільки їх у правого.

Алгоритм множення матриць наступний.

1. Множимо елементи першого рядка на елементи першого стовпця.

– помножуємо перший елемент першого рядка на перший елемент першого стовпця.

– помножуємо другий елемент першого рядка на другий елемент першого стовпця.

– робимо те саме з кожним елементом, поки не дійдемо до кінця як першого рядка першої матриці, так і першого стовпця другої матриці.

– складаємо отримані добутки.

– отриманий результат буде першим елементом першого рядка добутку матриць.

2. Множимо елементи першого рядка першої матриці на елементи другого стовпця другої матриці.

– помножуємо перший елемент першого рядка на перший елемент другого стовпця.

– помножуємо другий елемент першого рядка на другий елемент другого стовпця.

– робимо те саме з кожним елементом, поки не дійдемо до кінця як першого рядка першої матриці, так і другого стовпця другої матриці.

– складаємо отримані добутки.

– отриманий результат буде другим елементом першого рядка добутку матриць.

3. Застосовуючи той же алгоритм, множимо елементи першого рядка першої матриці на елементи інших стовпців другої матриці. Отримані числа складуть перший рядок матриці, що обчислюється.

4. Другий рядок матриці, що обчислюється, знаходиться аналогічно множенням елементів другого рядка першої матриці на елементи кожного стовпця другої матриці: результати записуються в нову матрицю після кожного підсумовування.

5. Робимо це з кожним рядком першої матриці, доки всі рядки нової матриці не будуть заповнені.

На рис. 1.5 наведені приклади множення матриць.

$$A \cdot B = \begin{pmatrix} 1 & 2 & 2 \\ 3 & 1 & 1 \end{pmatrix} \begin{pmatrix} 4 & 2 \\ 3 & 1 \\ 1 & 5 \end{pmatrix} = \begin{pmatrix} 1\cdot 4+2\cdot 3+2\cdot 1 & 1\cdot 2+2\cdot 1+2\cdot 5 \\ 3\cdot 4+1\cdot 3+1\cdot 1 & 3\cdot 2+1\cdot 1+1\cdot 5 \end{pmatrix} = \begin{pmatrix} 12 & 14 \\ 16 & 12 \end{pmatrix}$$

$$B \cdot A = \begin{pmatrix} 4 & 2 \\ 3 & 1 \\ 1 & 5 \end{pmatrix} \begin{pmatrix} 1 & 2 & 2 \\ 3 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 4\cdot 1+2\cdot 3 & 4\cdot 2+2\cdot 1 & 4\cdot 2+2\cdot 1 \\ 3\cdot 1+1\cdot 3 & 3\cdot 2+1\cdot 1 & 3\cdot 2+1\cdot 1 \\ 1\cdot 1+5\cdot 3 & 1\cdot 2+5\cdot 1 & 1\cdot 2+5\cdot 1 \end{pmatrix} = \begin{pmatrix} 10 & 10 & 10 \\ 6 & 7 & 7 \\ 16 & 7 & 7 \end{pmatrix}$$

Множення на одиничну матрицю

$$A \cdot B = \begin{pmatrix} 1 & 4 & 3 \\ 2 & 1 & 5 \\ 3 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} 1\cdot 1+4\cdot 0+3\cdot 0 & 1\cdot 0+4\cdot 1+3\cdot 0 & 1\cdot 0+4\cdot 0+3\cdot 1 \\ 2\cdot 1+1\cdot 0+5\cdot 0 & 2\cdot 0+1\cdot 1+5\cdot 0 & 2\cdot 0+1\cdot 0+5\cdot 1 \\ 3\cdot 1+2\cdot 0+1\cdot 0 & 3\cdot 0+2\cdot 1+1\cdot 0 & 3\cdot 0+2\cdot 0+1\cdot 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 3 \\ 2 & 1 & 5 \\ 3 & 2 & 1 \end{pmatrix}$$

Рисунок 1.5 – Приклади множення матриць

1.2 Прискорені алгоритми множення матриць та їх обчислювальна складність

Оцінюванню алгоритмічної складності порядку виконання процедури множення матриць присвячено багато досліджень. Відомо, що при використанні «класичного» варіанту множення двох матриць M_1 *і* M_2 з відповідними розмірностями $(p \times q)$ *і* $(q \times r)$ необхідно виконати порядку p*r*q операцій (множення і складання), тобто $O(prq)$ Також відомо, що операція множення матриць є асоціативною. Припустимо, що для розв'язання деякого завдання необхідно знайти добуток деякої множини матриць $M_1, M_2, \dots, M_n$ з розмірностями, які можуть бути заданими у вигляді наступної послідовності $m_0, m_1, m_2.$ m_n так, що розмірність кожної *i*–ої матриці визначається показниками $(m_{i-1} \times m_i), i = 1,2, \dots, n$. Тоді саме порядок визначення добутків кожної наступної операції множення матриць може мати суттєвий вплив на зниження загальної трудомісткості. Тобто, «розумне» виконання послідовності операцій множення матриць (розстановки дужок у послідовності $M *M_2* \dots * M_n$) сприяє зниженню алгоритмічної складності виконання завдання. В той же час завдання правильної розстановки дужок характеризується як NP-складне завдання, трудомісткість його реалізації експоненційна, а кількість варіантів правильної розстановки дужок оцінюється значенням функції

$$f(n) = \frac{1}{n} C_{2n-2}^{n-1} = \frac{(2n-2)!}{\left((n-1)!\right)^2}, n = 2,3, \dots. \tag{1.1}$$

Існують прискорені алгоритми процесу множення матриць. При використанні класичного підходу до множення двох квадратних $(m \times n)$ матриць (рядок на стовпець) необхідно виконати n^3 множень і $n^2(n-1)$ додавань, тобто трудомісткість процедури оцінюється з порядком $O(n^3)$. Також проблемою застосування класичної схеми є неможливість ефективної

організації паралельної процедури обчислень у разі великих розмірностей початкових матриць ($n \to \infty$). В той же час відомі алгоритми обчислення добутку матриць, які можуть бути легко зорієнтовані на паралельну реалізацію. До таких алгоритмів можна віднести, наприклад, алгоритм, заснований на підході Карацуби до множення двох n-розрядних чисел, що дозволило зменшити трудомісткість множення чисел з оцінки $O(n^2)$ до $O(n^{log_2 3})$. Розповсюдження цього підходу на множення матриць приводить до наступної процедури. Нехай є дві квадратні матриці A і B з розмірностями ($m \times n$) кожна. Якщо n не є ступенем 2, то матриці розширюються до розмірності найменшого цілого зверху 2^k для n, так, що в знову утворених A^* і B^* початкові матриці A і B займають позиції, починаючи з північно-західного кута. Після цього кожна нова утворена матриця розрізається на чотири частини у такий спосіб

$$A^* = \begin{pmatrix} A^*_{11} & A^*_{12} \\ A^*_{21} & A^*_{22} \end{pmatrix}, \qquad B^* = \begin{pmatrix} B^*_{11} & B^*_{12} \\ B^*_{21} & B^*_{22} \end{pmatrix}. \tag{1.2}$$

В результаті множення A* і B* утворюється матриця C*

$$C^* = A^* \cdot B^* = \begin{pmatrix} C^*_{11} & C^*_{12} \\ C^*_{21} & C^*_{22} \end{pmatrix}. \tag{1.3}$$

Елементи матриці C* формуються за наступними правилами

$$\begin{aligned} C^*_{11} &= A^*_{11} \cdot B^*_{11} + A^*_{12} \cdot B^*_{21}, \\ C^*_{12} &= A^*_{11} \cdot B^*_{12} + A^*_{12} \cdot B^*_{22}, \\ C^*_{21} &= A^*_{21} \cdot B^*_{11} + A^*_{22} \cdot B^*_{21}, \\ C^*_{22} &= A^*_{21} \cdot B^*_{12} + A^*_{22} \cdot B^*_{22}. \end{aligned} \tag{1.4}$$

З отриманої матриці C* відокремлюється у розмірах $(m \times n)$ результуюча матриця C=A*B такий спосіб для множення двох матриць розмірності 2^k необхідно знайти 8 добутків і 4 суми матриць з розмінностями 2^{k-1}. Окрім того, що алгоритм спрямований на рекурсивне зменшення

розмірності знову утворених матриць, всі обчислення з такими матрицями можна проводити незалежно (паралельно).

На поточний час існують алгоритми множення матриць, які мають обчислювальну складність $O(n^{log_2 7})$ і, навіть, менше. До таких можна віднести, наприклад, алгоритм Штрассена. Основна відмінність такого алгоритму, від попереднього, полягає в тому, що після зменшення розміру кожної з матриць A і B , які беруть участь у множенні, процес обчислень зводиться до наступної послідовності перетворень з допоміжними матрицями

$$\begin{aligned} D_1^* &= (A_{11}^* + A_{22}^*) \cdot (B_{11}^* + B_{22}^*), \\ D_2^* &= (-A_{11}^* + A_{21}^*) \cdot (B_{11}^* + B_{12}^*), \\ D_3^* &= (A_{12}^* - A_{22}^*) \cdot (B_{21}^* + B_{22}^*), \\ D_4^* &= A_{11}^* \cdot (B_{12}^* - B_{22}^*), \\ D_5^* &= (A_{11}^* + A_{12}^*) \cdot B_{22}^*, \\ D_6^* &= A_{22}^* \cdot (-B_{11}^* + B_{21}^*), \\ D_7^* &= (A_{21}^* + A_{22}^*) \cdot B_{11}^*. \end{aligned} \tag{1.5}$$

Отримані допоміжні матриці надалі доводяться до вигляду (1.4) у такий спосіб:

$$\begin{aligned} C_{11}^* &= D_1^* + D_3^* - D_5^* + D_6^*, \\ C_{12}^* &= D_6^* + D_7^*, \\ C_{21}^* &= D_4^* + D_5^*, \\ C_{22}^* &= D_1^* + D_2^* + D_4^* - D_7^* . \end{aligned} \tag{1.6}$$

Можна побачити, що замість 8 множень у (1.4) завдання зводиться до виконання 7 множень у (1.5). Незважаючи на те, що при цьому значно зростає кількість складань, ці операції мають набагато менший час виконання в порівнянні з множенням матриць. Отже, при зростанні розмірності матриць, множення за Штрассеном набуває перевагу.

Алгоритм Штрассена покращує просте множення матриць за допомогою підходу «розділяй і володарюй». Ключове зауваження полягає в тому, що множення двох матриць 2 × 2 можна виконати лише 7 множеннями замість звичайних 8 (за рахунок 11 додаткових операцій додавання та віднімання). Це означає, що розглядаючи вхідні $n \times n$ матриці як блочні 2×2 матриці, задачу множення $n \times n$ матриць можна звести до 7 підзадач множення n/2×n/2 матриць. Застосування цього рекурсивно дає необхідний алгоритм зі складністю $O(n^{\log_2 7}) \approx O(n^{2.807})$ простих операцій.

На відміну від алгоритмів з більш швидкою асимптотичною складністю, на практиці найчастіше використовується алгоритм Штрассена. Чисельна стабільність знижена порівняно з простим алгоритмом, але він швидший у випадках, коли n > 100 або близько того. Алгоритми швидкого множення матриць не можуть досягти покомпонентної стабільності, але деякі з них демонструють нормовану стабільність. Це дуже корисно для великих матриць у точних областях, таких як кінцеві поля, де чисельна стабільність не є проблемою.

В 1990 Копперсміт і Виноград опублікували алгоритм, асимптотична складність якого становила $O(n^{2.3755})$. Цей алгоритм використовує ідеї, схожі на алгоритм Штрассена. Сьогодні модифікації алгоритму Копперсмита – Винограда є найбільш асимптотично швидкими. В останній модифікації (2024) складність алгоритму становить $O(n^{2.371552})$. Відомо, що широкий клас модифікацій цього алгоритму у принципі неспроможна досягти складність краще, ніж $O(n^{2.3078})$. Алгоритм Копперсміту-Винограда ефективний тільки на матрицях астрономічного розміру і на практиці застосовуватися не може.

У загальному вигляді показник обчислювальної складності множення матриці, зазвичай позначається ω, є найменшим дійсним числом, для якого будь-які два $n \times n$ матриць можна помножити за допомогою $n^{\omega+o(1)}$ простих операцій. Ця нотація зазвичай використовується в дослідженні обчислювальних алгоритмів, тому алгоритми, які використовують множення

матриці, мають обмеження на час виконання, які можуть оновлюватись у міру покращення обмежень на ω.

Використовуючи нижню межу та множення матриці шкільного підручника для верхньої межі, можна прямо зробити висновок, що $2 \leq \omega \leq 3$. Чи можна досягнути $\omega = 2$, є головним відкритим питанням у теоретичній інформатиці та чисельних методах. Існує ряд досліджень, які розробляють алгоритми множення матриць. щоб отримати покращені межі ω. Усі останні алгоритми в цьому напрямі досліджень використовують лазерний метод, що є узагальнення алгоритму Копперсміта-Віноградa, на рис. 1.6 наведені останні дані по розробці обчислювальних алгоритмів множення матриць [3].

Timeline of matrix multiplication exponent

Year	Bound on omega	Authors
1969	2.8074	Strassen
1978	2.796	Pan
1979	2.780	Bini, Capovani [it], Romani
1981	2.522	Schönhage
1981	2.517	Romani
1981	2.496	Coppersmith, Winograd
1986	2.479	Strassen
1990	2.3755	Coppersmith, Winograd
2010	2.3737	Stothers
2012	2.3729	Williams
2014	2.3728639	Le Gall
2020	2.3728596	Alman, Williams
2022	2.371866	Duan, Wu, Zhou
2024	2.371552	Williams, Xu, Xu, and Zhou

Рисунок 1.6 – Обчислювальна складність прискорених алгоритмів множення матриць

1.3 Застосування множення матриць у машинному навчанні

В основі машинного навчання лежить використання нейронних мереж. Нейронна мережа є системою з'єднаних і взаємодіючих між собою простих процесорів (штучних нейронів). Такі процесори досить прості (особливо у порівнянні з процесорами, що використовуються у персональних комп'ютерах). Кожен процесор подібної мережі має справу лише з сигналами, які він періодично отримує, та сигналами, які він періодично надсилає іншим процесорам. І, тим не менш, будучи з'єднаними в досить велику мережу з керованою взаємодією, такі прості процесори разом здатні виконувати досить складні завдання [4].

Можливість навчання – одна з головних переваг нейронних мереж перед традиційними алгоритмами. Технічно навчання полягає у знаходженні коефіцієнтів зв'язків між нейронами. У процесі навчання нейронна мережа здатна виявляти складні залежності між вхідними та вихідними даними, а також виконувати узагальнення. Це означає, що у разі успішного навчання мережа зможе повернути правильний результат на підставі даних, які були відсутні у навчальній вибірці, а також неповних та/або «зашумлених», частково спотворених даних.

Кожен нейрон (або вузол) приймає сигнали від вузлів попереднього шару та передає на наступний. Кожен зв'язок між нейронами має власну вагу. Таким чином, вхідний сигнал вузла 1 шару 1 передається на вузол 1 шару 2 з коефіцієнтом передачі на вузол 2 шару 2 з коефіцієнтом передачі і так далі. Усі сигнали, отримані вузлом 2 рівня складаються. Це його вхідний сигнал. Таким чином, сигнали передаються з рівня на рівень, до виходу. Таким чином, нейрон підсумовує вхідні сигнали, помножені на ваги зв'язків, бере сигмоїду від результату та подає на вихід (рис. 1.7).

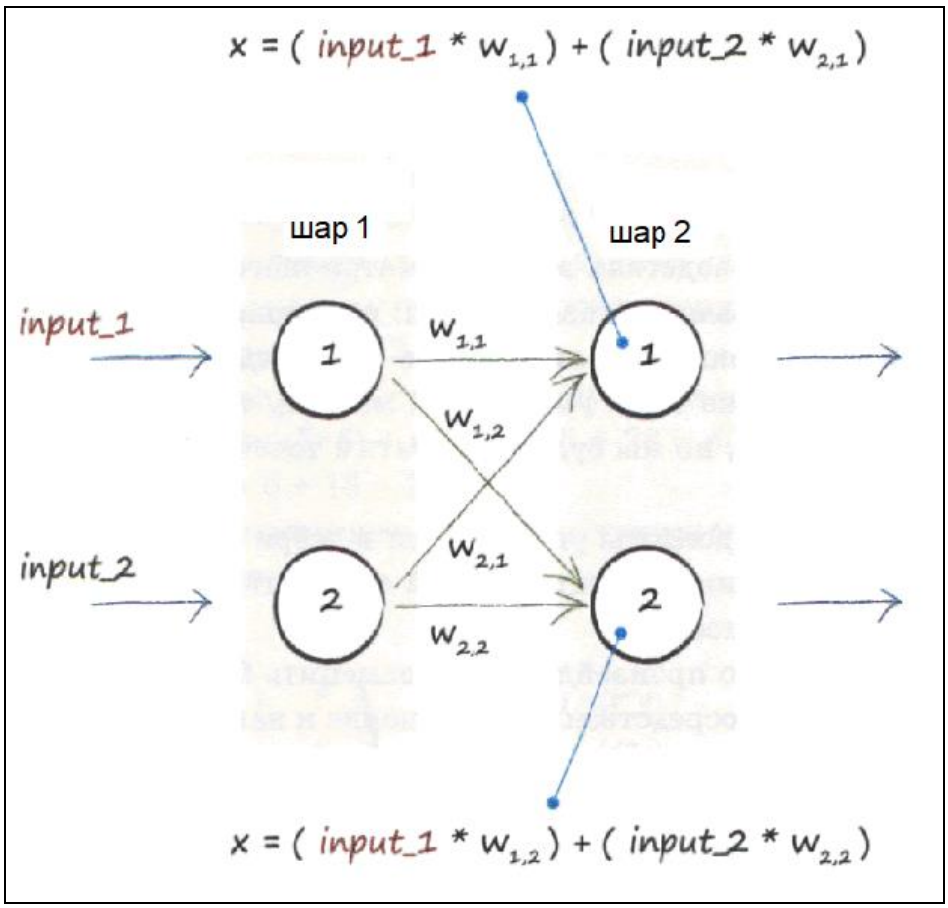


Рисунок 1.7 – Графічна інтерпретація отримання сигналу наступного рівня за формулою X = W*I

Якщо уявити правила поширення сигналу мовою математики, то щоб отримати сигнали нового шару, необхідно "помножити вагу кожного вузла шару 1 з його вихідний зв'язок, що веде до вузлу шару 2 і скласти" , що дуже підходить опис множення матриць. Справді, розташуємо в кожному стовпці матриці ваги зв'язків, що виходять з одного вузла, і помножимо праворуч на стовпець вхідних сигналів, то отримаємо вихідний сигнал цього шару в стовпці матриці, що вийшла (рис. 1.8).

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} * \begin{pmatrix} input\_1 \\ input\_2 \end{pmatrix} = \begin{pmatrix} (input\_1 * w_{1,1}) + (input\_2 * w_{2,1}) \\ (input\_1 * w_{1,2}) + (input\_2 * w_{2,2}) \end{pmatrix}$$

Рисунок 1.8 – Отримання сигналу наступного рівня за формулою X = W*I

У рядках ж матриці ваг будуть ваги зв'язків, що ведуть в один вузол нового шару, кожен з яких множиться на вагу вузла, що його породжує. Зрозуміло, через правила перемноження матриць висота кінцевого стовпця дорівнюватиме висоті матриці ваг, а висота матриці вхідних сигналів - ширині матриці ваг.

1.4 Апаратне прискорення обчислювальних алгоритмів

Апаратне прискорення, термін, який поступово стає повсюдним у різних технічних сферах, позначає техніку делегування певних завдань – що виконуються центральним процесором (ЦП) – спеціалізованим апаратним компонентам у межах комп'ютерної системи. Ця стратегічна перерозподіл як підвищує ефективність і швидкість виконання завдань, а й забезпечує збереження пропускну здатність ЦП інших обчислювальних функцій.

У своїй основі апаратне прискорення полягає в оптимізації обчислювальних завдань шляхом спрямування їх до апаратних засобів, спеціально розроблених для більш ефективного виконання цих завдань. Ця спеціалізація може змінюватись від графічних процесорів (GPU) для рендерингу зображень та відео до цифрових сигнальних процесорів (DSP) для керування аудіо та голосовими командами і навіть до спеціальних інтегральних схем (ASIC) для конкретних додатків, таких як майнінг криптовалют або виконання алгоритмів глибокого навчання .

У комп'ютерних технологіях під апаратним прискоренням розуміють застосування апаратного забезпечення виконання деяких функцій швидше проти виконання програм процесором загального призначення. Прикладами апаратного прискорення може бути блокове прискорення виконання у графічному процесорі та інструкції комплексних операцій на микропроцессоре. Обычно процесори виконують роботу послідовно, а інструкції виконуються по черзі. Для покращення продуктивності застосовуються різні способи, і апаратне прискорення – один із них.

Основна відмінність апаратного прискорення від програмного полягає в паралельності, дозволяючи апаратному забезпеченню бути набагато швидше, ніж програмному. Апаратні прискорювачі спеціально спроектовані для програмного коду, що створює високе обчислювальне навантаження. Залежно від ступеня деталізації, апаратне прискорення може змінюватись від невеликої функціональної одиниці до великого функціонального блоку, як, наприклад, відеообробка MPEG2. Апаратне забезпечення, що виконує прискорення у вигляді окремої одиниці центрального процесора, називається апаратним прискорювачем,

До теперішнього часу створено значний арсенал апаратних рішень для забезпечення обчислень з високою пропускною здатністю, а також програмних бібліотек, що надають інтерфейси прикладного програмування для використання різних апаратних платформ при виконанні прикладних завдань. Паралельні обчислення стали домінуючою парадигмою в архітектурі комп'ютерів.

У силу цього універсальним способом підвищення продуктивності обчислень є використання паралельних алгоритмів, реалізованих і виконуваних на паралельних обчислювальних системах, побудованих з урахуванням гетерогенних обчислювачів [5]. У таких обчислювальних системах звичайні центральні процесори (CPU) доповнюються спеціалізованими прискорювачами, що підвищують продуктивність та енергоефективність розв'язання різноманітних ресурсомістких завдань, вже звичних у світі. Як такі прискорювачі обчислень, крім графічних процесорів (GPU) і спеціалізованих інтегральних схем спеціального призначення (ASIC), все ширше застосування знаходять прискорювачі обчислень, побудовані на базі програмованих логічних інтегральних схем (ПЛІС). До основних переваг прискорювачів на ПЛІС на відміну від прискорювачів, заснованих на CPU або GPU, слід віднести більш високу продуктивність у додатках реального часу, скорочення затримок виведення машинного навчання, найкраще співвідношення продуктивності на ват споживаної потужності, а також

високий рівень гнучкості порівняно з ASIC .

Крім GPU, є спеціалізовані апаратні прискорювачі, розроблені для певних завдань. Наприклад, криптографічні співпроцесори сприяють більш швидкій та безпечній шифрації та розшифрації даних, що важливо у сучасну епоху, коли безпека даних має першорядне значення.

Як основа прискорювачів на ПЛІС використовуються мікросхеми програмованої логіки типу FPGA. Обчислювальна система, що включає в себе прискорювачі на FPGA – це, як правило, класична серверна платформа, що містить процесори і кілька карт прискорювачів на FPGA, або спеціалізовані промислові сервери. У таких системах прискорювач приймає виконання основних ресурсоємних завдань, у яких потрібні паралельні обчислення, підвищують продуктивність всієї системи загалом.

Технологія ПЛІС забезпечує створення цифрових електронних схем, що гнучко конфігуруються. Апаратні ресурси мікросхеми FPGA програмуються користувачем безпосередньо під саме завдання, що дозволяє реалізувати програму користувача як імплементацію алгоритму в кристалі, використовуючи як базові елементи (тригери, логічні елементи «І», «АБО», «НІ» тощо), так і спеціалізовані апаратні ядра (помножувачі, блокову пам'ять, комунікаційні інтерфейси та ін.). Це дозволяє досягти тим самим високої швидкодії за рахунок високого паралелізму та тактових частот [6].

Традиційний шлях розробки під FPGA передбачає написання програмного коду кожного з модулів пристрою, що проектується, з використанням мов опису апаратури інтегральних схем (HDL), таких як VHDL або Verilog, застосування готових стандартних програмних та апаратних модулів (IP-ядер), що входять до складу САПР. Використовуючи ці інструменти, розробник змушений витрачати значну частину часу на написання та налагодження низькорівневого програмного коду кожного функціонального модуля проекту, а також на створення і налаштування всієї периферії, необхідної для інтеграції розробленого програмного коду і забезпечення взаємодії із зовнішніми периферійними пристроями. Цей шлях

виправданий у разі виконання нескладних завдань, що потребують невеликих за обсягом мікросхем FPGA. Сучасні мікросхеми програмованої логіки останніх поколінь мають величезну кількість апаратних ресурсів, що надаються користувачеві, і призначені для вирішення ресурсомістких завдань і побудови складних програмних алгоритмів, наприклад, таких, як множення матриць

У сучасних реаліях традиційний шлях розробки йде на другий план і стає лише допоміжним. На перший план виходить нова парадигма – високорівневий синтез (HLS), вона полягає у розробці програмних алгоритмів мовами високого рівня, таких як C, C++ [7], що дуже спрощує та прискорює розробку та налагодження програмного коду. Компілятор HLS перетворює програмний код на синхронну цифрову схему на рівні регістрових передач, який згодом буде виконуватися на FPGA. Слід зазначити, що компілятор має низку обмежень порівняно з стандартними компіляторами коду C/C++, такі як відсутність системних викликів, пам'яті, що динамічно виділяється, рекурсивних функцій тощо.

Переваги апаратного прискорення такі.

Ефективність продуктивності. Завдання виконуються швидше та ефективніше за рахунок використання апаратних засобів, призначених для конкретних функцій.

Енергоефективність. Перерозподіляючи завдання до відповідного обладнання, часто спостерігається значне зниження енергоспоживання, що важливо для пристроїв з батарейним живленням.

Покращений досвід користувача. Програми, що використовують апаратне прискорення, можуть запропонувати більш чуйний та захоплюючий досвід завдяки плавній графіці та швидшому часу обробки.

У міру продовження технологічного прогресу область застосування та можливостей апаратного прискорення розширюватиметься. Нові технології, такі як штучний інтелект , віртуальна реальність та Інтернет речей, матимуть величезну користь із цієї парадигми, а спеціалізовані прискорювачі вже

розробляються для задоволення унікальних вимог цих областей.

Крім того, інтеграція апаратного прискорення в хмарні обчислення та центри обробки даних підкреслює його зростаючу важливість, пропонуючи значні покращення продуктивності для складних обчислень та обробки даних у великих масштабах.

У результаті апаратне прискорення є важливим кроком у розвитку обчислювальної техніки, дозволяючи більш ефективно, швидше і з меншими енергетичними витратами виконувати завдання за рахунок використання спеціалізованого обладнання для конкретних функцій. У міру розвитку технологій та збільшення попиту на обчислювальну потужність, роль апаратного прискорення у підвищенні продуктивності та поліпшенні користувальницького досвіду буде тільки зростати, знаменуючи собою значну епоху у розвитку синергії апаратного та програмного забезпечення.

1.5 Технічне завдання на проектування

Множення матриць – це один з базових алгоритмів, які широко застосовуються в різних методах, зокрема у алгоритмах машинного навчання. Багато реалізації прямого та зворотного поширення сигналу в перетворювальних шарах нейронної мережі засновані на цій операції. Так іноді до 90-95% всього часу, який витрачається на машинне навчання, припадає саме на цю операцію. Чому так відбувається? Відповідь у дуже ефективній реалізації цього алгоритму для процесорів, графічних прискорювачів (останнім часом і спеціальних прискорювачів матричного множення). Матричне множення – один із алгоритмів, що дозволяє ефективно задіяти всі обчислювальні ресурси сучасних процесорів та графічних прискорювачів. Тому не дивно, що багато алгоритмів цього типу ведуть до матричного множення – додаткові витрати, пов'язані з підготовкою даних, зазвичай з надлишком окупаються загальним прискоренням алгоритму.

Матричне множення виходить за рамки базової арифметики та є наріжним каменем багатьох алгоритмів машинного навчання. Йдеться не про множення відповідних елементів, замість цього він включає скалярний добуток і комбінацію рядків і стовпців. Множення матриць відіграє ключову роль у перетворенні даних, застосуванні ваг до функцій і обчисленні прогнозів.

Мета роботи – реалізація апаратного прискорювача обчислювальних алгоритмів множення матриць з використанням SoC сімейства ZYNQ-7000 на налагоджувальній платі ZedBoard.

Для досягнення поставленої мети необхідно вирішити такі задачі:

– розглянути та проаналізувати види алгоритмів множення матриць та їх обчислювальну складність;

– розглянути процедури проектування програмно-апаратних систем на основі SoC;

– обрати технологічну платформу реалізації та виконати програмування апаратного прискорювача математичних операцій;

– виконати макетування розробленого прототипу апаратного прискорювача та провести оцінку його апаратних та часових ресурсів.

2 ТЕХНОЛОГІЧНА ПЛАТФОРМА РЕАЛІЗАЦІЇ

2.1 Плата розробки ZedBoard для сімейства ZYNQ-7000 AP SoC

Стрімке впровадження нових технологій у процесі виробництва кристалів програмованої логіки та розширюваних процесорних платформ, що відбувається останнім часом, дозволило суттєвоним чином збільшити обсяг їх ресурсів.

Програмовані системи на кристалі All Programmable System-On-Chip (AP SoC) сімейства Zynq-7000 принципово відрізняються від ПЛІС з апаратними мікропроцесорними ядрами PowerPC. Об'єднання в одному кристалі апаратного двоядерного процесорного блокс архітектурою ARM Cortex-A9 і програмованої логіки FPGA (Field Programmable Gate Array) останнього покоління надає розробникам широкі можливості для проектування систем різного призначення з гнучкою і динамічно конфігурованою структурою.

У кристалах обчислювальних платформ сімейства Zynq-7000 AP SoC, що розширюються, апаратний процесорний блок є основним компонентом, який доступний до конфігурування програмованої логіки і може використовуватися незалежно від логічних ресурсів. Крім того, у складі процесорної системи цих кристалів представлені апаратні контролери найбільш затребуваних інтерфейсів, деякі з яких складно реалізувати на базі стандартних логічних ресурсів ПЛІС.

Активному поширенню програмованих систем на кристалі сімейства Zynq-7000 AP SoC сприяє також надання фірмою Xilinx широкого спектру інструментальних налагоджувальних комплектів і комплексів засобів автоматизованого проектування, що дозволяють в рекордно короткі терміни виконувати всі етапи розробки вбудованих мікропроцесорних систем, включаючи налагодження апаратної частини та апаратної частини.

Одним із прикладів програмованої системи на кристалі є FPGA сімейства Zynq-7000 фірми Xilinx Inc. Для побудови SoC різного призначення платформа Zynq-7000 EPP містить наступні функціональні блоки: процесорну підсистему, що включає процесорний модуль, інтерфейси пам'яті, периферійні інтерфейси, міжблокові інтерфейси та інтерфейси до програмованої логіки, а також програмовану логіку. При цьому процесор ARM Cortex A9 MPCore має вбудовану пам'ять, багатий набір периферійних пристроїв, інтерфейси до зовнішньої пам'яті. Взаємодія між двома процесорами може здійснюватися за допомогою міжпроцесорних переривань через область загальної пам'яті шляхом передачі повідомлень [8, 9].

Інтегральна схема Zync-7000 EPP (Extensible Processing Platform) виконана за технологічним процесом SRAM 28 нм і є FPGA з впровадженими додатковими функціональними блоками. Крім 350 тисяч логічних блоків (для FPGA Z-7055) кристал включає багатоядерний блок обчислювачів, побудований на базі двох ядер процесора ARM Cortex-A9, 2 Мбіт вбудованої багатопортової пам'яті, контролери зовнішніх динамічних оперативних запам'ятовуючих пристроїв (DDR2 і DDR3), контролери пристроїв flash-пам'яті (NAND і NOR), два вбудовані блоки високошвидкісних 12-розрядних аналого-цифрових перетворювачів, вбудований контролер інтерфейсу PCIe, вбудовані контролери спеціалізованих інтерфейсів таких, як I^2C, USB 2 Ethernet, UART, CAN, SPI тощо.

Програмована логіка містить конфігуровані логічні блоки (CLB), конфігуровані двопортові блоки пам'яті (BRAM), комірки цифрової обробки сигналів з 25×18-бітним помножувачем, 58-бітним акумулятором і суматором (DSP58E1), АЦП XADC, керовані блоки формування, конфігурований блок шифрування та автентифікації, конфігурований блок вводу-виводу (Select IO). (рис. 2.1).

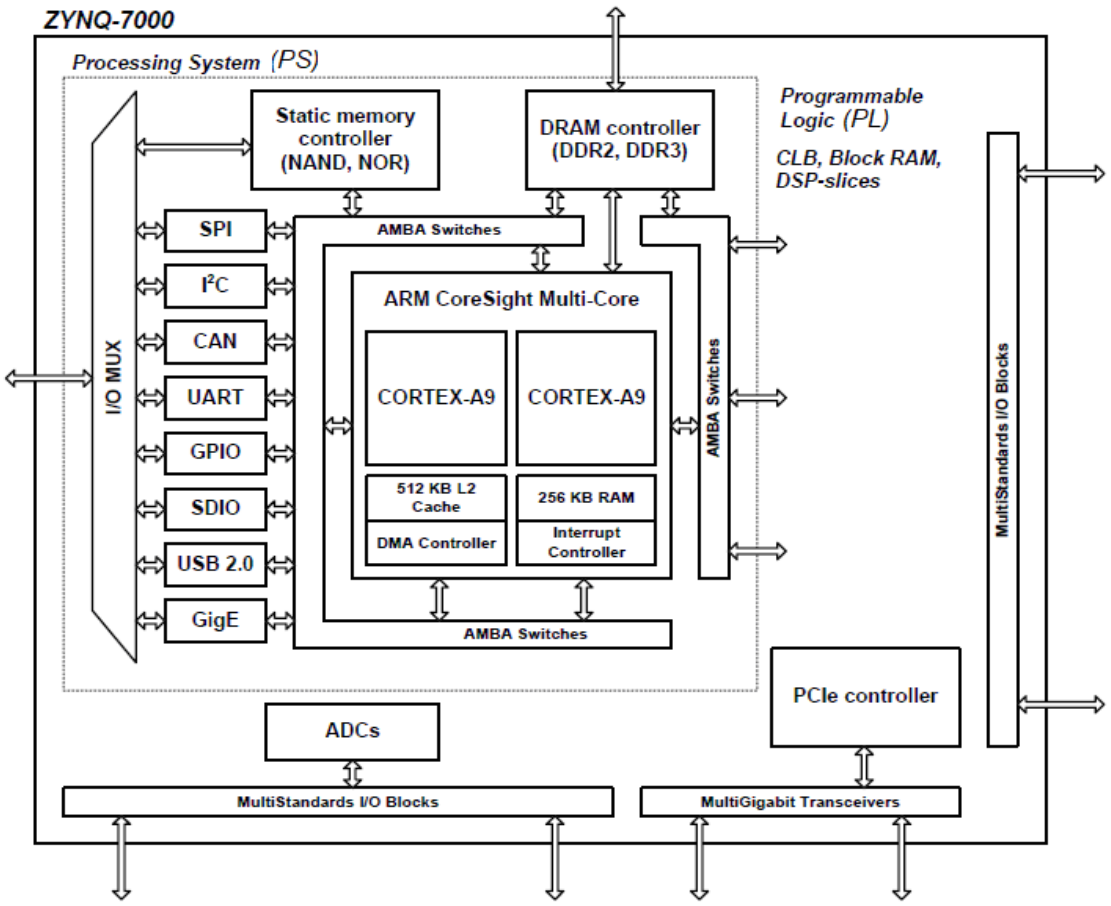


Рисунок 2.1 – Архітектура SoC Xilinx ZYNQ-7000

Xilinx ZYNQ-7000 платформа має архітектуру, де поєднано PS (processing system) – процесорну систему обробки даних та PL (programmable logic) - програмовану логіку. Взаємодія двох систем виконується через інтерфейс AMBA на основі протоколу Advanced eXtensible Interface (AXI).

Однією з популярних платформ розробки для роботи з Zynq є ZedBoard. Назва «Zed» означає Zynq Evaluation and Development. Загальний вигляд плати ZedBoard з елементами інтерфейсу представлений на рис. 2.2.

ZedBoard – це недорога загальнодоступна плата, оснащена пристроєм XC7Z020 Zynq. Це спільне підприємство Xilinx, Avnet (дистриб'ютор) і Digilent (виробник плат). Незважаючи на те, що ZedBoard є платформою розробки для промисловості, він також орієнтований на студентів, вчених і любителів, та пропонує спеціальні навчальні матеріали для нових користувачів Zynq [8, 9].

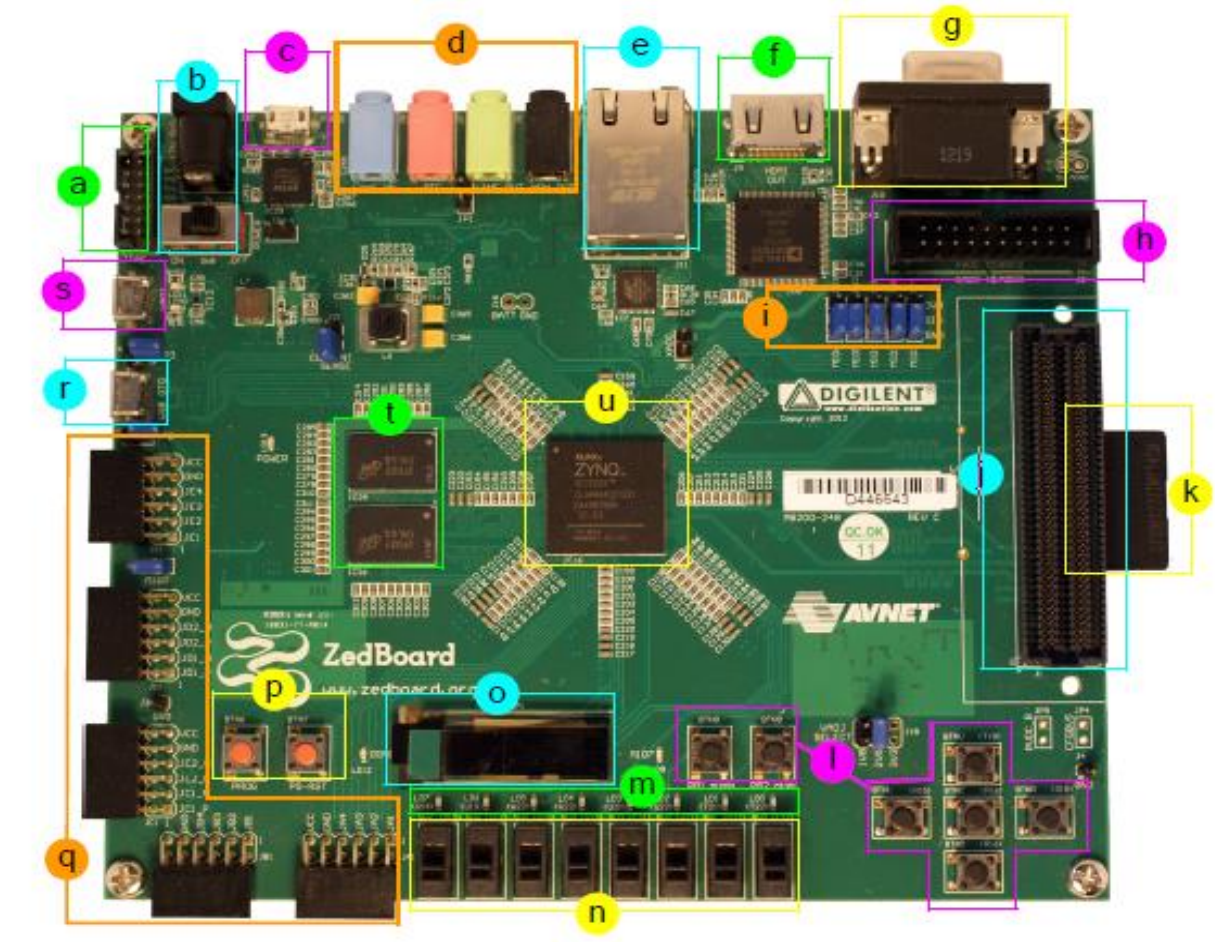


Рисунок 2.2 – Загальний вигляд плати ZedBoard

2.2 Реалізація обчислювальних алгоритмів в SoC

Одним із способів прискорення обчислень є використання спеціалізованих апаратних засобів, що мають значні обчислювальні ресурси.

Як популярний останнім часом приклад таких пристроїв можна навести графічні процесори (GPU), але основною проблемою при їх використанні є неможливість ефективної реалізації скільки-небудь широкого класу обчислювальних алгоритмів.

Сучасні графічні процесори дуже ефективно обробляють та відображають комп'ютерну графіку, завдяки спеціалізованій конвеєрній архітектурі вони набагато ефективніші в обробці графічної інформації, ніж типовий центральний процес (ЦП).

Відмінними рисами в порівнянні з ЦП архітектура, максимально націлена збільшення швидкості розрахунку текстур і складних графічних об'єктів та обмежений набір команд.

Висока обчислювальна потужність GPU пояснюється особливостями архітектури. Сучасні CPU містять невелику кількість ядер (порівняно з графічними процесорами), тоді як графічний процесор спочатку створювався як багатопотокова структура з множиною ядер. Різниця в архітектурі зумовлює і різницю у принципах роботи. Якщо архітектура CPU передбачає послідовну обробку інформації, то GPU історично призначався для обробки комп'ютерної графіки, тому розрахований на потужно паралельні обчислення.

Важливим етапом розвитку GPU стала поява обчислень загального призначення на графічних процесорах. Вони розширили роль GPU за межі рендерингу графіки до загальних обчислювальних завдань. Завдяки появі таких мов програмування, як CUDA від Nvidia, а потім OpenCL, розробники змогли використати паралельну обчислювальну потужність GPU для широкого спектру програм – від обчислювальних алгоритмів та наукового моделювання до машинного навчання.

Графічні процесори відіграють важливу роль у багатьох хмарних сервісах, але особливо в тих, які вимагають високопродуктивних обчислень або працюють з більшими обсягами даних. Наприклад, хмарні платформи машинного навчання часто надають екземпляри GPU для навчання моделей,

використовуючи переваги здатності GPU виконувати багато обчислень одночасно. Аналогічно хмарні сервіси візуалізації використовують GPU для візуалізації складних 3D-сцен в режимі реального часу.

Більш низькорівневим у порівнянні з GPU способом апаратного прискорення виконання алгоритму є створення обчислювальної логіки для конкретного алгоритму з використанням програмованих логічних інтегральних схем (ПЛІС), що відкриває перспективи апаратної апаратної реалізації різноманітних обчислювальних алгоритмів. Застосування ПЛІС найбільш зручне у випадках, коли потрібна швидка обробка та фільтрація великого потоку даних у реальному часі, наприклад, від експериментальної установки або у пристроях цифрової обробки звукових сигналів. Однак програмування ПЛІС є складним завданням, що вимагає наявності спеціальних знань та великих трудовитрат на реалізацію та налагодження.

Програмування ПЛІС значно спростилося після появи інструментів алгоритмічного синтезу Mentor Graphics C Synthesis, Cadence C-to-silicon compiler, Synopsis Synphony HLS, Xilinx Vivado/Vitis. Використання алгоритмічного синтезу дозволяє задавати логіку алгоритму у зручній формі, наприклад, мовою С, і дозволяє суттєво підвищити гнучкість реалізації конкретного алгоритму.

Алгоритмічний синтез – це спосіб розробки інтегральних схем (ASIC) або створення прошивок для ПЛІС (FPGA) з максимальним, на сьогоднішній день, рівнем абстракції опису функціональної частини схеми. Такий опис виконується алгоритмічною мовою, наприклад С, після чого програмне забезпечення алгоритмічного синтезу, керуючись заданими розробником обмеженнями (використовується технологія, бажана частота роботи електронної схеми, обмеження алгоритму), створює RTL-опис логічної схеми на мові Verilog або VHDL.

Генерована логічна схема може бути оптимізована за конкретними параметрами (кількість елементів або швидкість роботи). Завдяки застосуванню інструментів алгоритмічного синтезу значно скорочується час

розробки та налагодження алгоритму. Крім того, знижується обсяг спеціальних знань, необхідних фахівцям, які виконують проектування алгоритму.

Однією з основних завдань інструментів алгоритмічного синтезу є ретельний аналіз алгоритму з метою встановлення послідовності виконання операцій. Це, у свою чергу, дозволяє визначити необхідну кількість ресурсів та скласти розклад їхнього використання. Відповідно до розкладу задається прив'язка використання різних логічних елементів схеми на час виконання операцій, відсутня у формальному описі алгоритму.

Необхідність визначення кількості необхідних ресурсів на етапі синтезу призводить до вимог детермінованості алгоритму на етапі компіляції. Зокрема, з цього випливає:

– неможливість використання в алгоритмі динамічного відведення пам'яті;

– неможливість застосування рекурсії необмеженої глибини;

– небажаність застосування циклів із невідомою на момент компіляції кількістю ітерацій.

У пакеті Mentor Graphics як мови опису алгоритмів застосовується підмножина мови C++, що з мови C, до якого додано можливість використання шаблонів (templates). Наявність шаблонів дозволяє створювати алгоритми із досить гнучкими можливостями параметризації.

Шаблони дозволяють легко змінювати структуру алгоритму без зміни програмного коду. Для зміни кількості однотипних блоків, що використовуються в конвеєрі, або глибини кінцевої рекурсії, можна використовувати рекурсивні шаблони. Для кожної ітерації рекурсивного шаблону будуть побудовані окремі логічні елементи.

Крім шаблонів управління синтезом також застосовуються директиви компілятора (#pragma). За допомогою директив можна керувати розбиттям коду на ієрархічні блоки, розгортанням циклів (unroll) та побудовою конвеєрів (pipeline). Правильне застосування останніх двох директив

необхідне ефективного використання апаратного паралелізму.

З метою максимально ефективного використання дорогих ресурсів ПЛІС, таких як помножувачі або блоки цифрової обробки сигналів, оптимізатор Catapult C може генерувати розклад і схему таким чином, що той самий ресурс буде використовуватися при виконанні різних частин алгоритму. Наприклад, якщо виконання алгоритму складається з двох циклів, у кожному з яких використовується операція множення, Catapult C використовує один фізичний помножувач, дані до якого подаватимуться з потрібного циклу за допомогою мультиплексорів.

Проте чи об'єднання використання ресурсів призводять до безумовного виграшу. При реалізації складних алгоритмів може виявитися, що об'єднання ресурсів призвело до зниження максимальної частоти, на якій може працювати схема, або на наступних фазах оптимізатор взагалі не зміг впоратися із завданням генерації вихідного файлу. Один із способів контролю таких оптимізації – це завдання ієрархічних блоків. Ієрархічний блок - це самостійний елемент схеми, що має інтерфейс, що використовує пам'ять або канали (pipes) обмінюватись даними з іншими блоками. Будь-які оптимізації відбуваються лише усередині ієрархічних блоків. Ієрархічним блоком може бути будь-яка функція (метод класу).

Цикл синтезу алгоритму для ПЛІС. Створення алгоритму для застосування в ПЛІС (прошивки) із застосуванням інструментів алгоритмічного синтезу можна розбити на наступні стадії:

– опис алгоритму мовою С із зазначенням обмежень; завдання додаткових умов на блоки алгоритму, функціональне тестування реалізації на С, вибір використовуваної ПЛІС;

– синтез електронної схеми за допомогою програмного забезпечення алгоритмічного синтезу – створення RTL-опису електронної схеми мовою Verilog;

– перевірка результату синтезу за допомогою програмного забезпечення моделювання роботи електронної схеми на основі RTL-опису

(Mentor ModelSim), синтез рівня логічних елементів (Gate Level); наприклад, за допомогою програмного комплексу Mentor Precision Synthesis; за необхідності перевірка отриманої схеми за допомогою моделювання;

– завдання апаратної специфікації дизайну (які ніжки ПЛІС для яких сигналів використовуються, яка напруга подається на вхід тощо);

– створення прошивки для ПЛІС (place & route) за допомогою програмного забезпечення від виробника ПЛІС, у випадку ПЛІС Xilinx за допомогою програмного комплексу Xilinx ISE.

2.3 Інструментальні засоби автоматизованого проектування для SoC Zynq-7000

Програмні засоби серії Xilinx ISE Design Suite є системою наскрізного проектування, яка реалізує повний цикл розробки цифрових пристроїв і вбудованих мікропроцесорних систем на основі ПЛІС та розширюваних обчислювальних платформ, що включає етапи створення вихідних описів проекту, синтезу, моделювання, розміщення та трасування, програмування кристалів, а також внутрішньокристального налагодження [10].

Процес проектування вбудованих систем, що реалізуються на базі кристалів програмованої логіки з архітектурою FPGA і розширюваних обчислювальних платформ сімейства Zynq-7000 AP SoC, в загальному випадку включає наступні етапи [11].

1. Розробка проекту мікропроцесорної системи. На цьому етапі визначається архітектура системи та розподіл функцій по виконанню обчислень між апаратною та програмною частиною, використовуючи обрані критерії та метрики (енергоспоживання, швидкість реакції системи, оптимальність виконання блоків). Також додається необхідність вибору цільового процесора або SoC для подальших етапів проектування.

2. Проектування апаратної платформи системи, що розробляється, до якого входить формування проекту апаратної платформи та визначення IP-

ядер, необхідних для конкретної задачі. Наступними задачами тут є реалізація апаратної платформи на базі обраного чіпу та її верифікація.

3. Підготовка системних програмних засобів нижнього (апаратного) рівня. До цього етапу входить розробка та налагодження первинного завантажувача системи (First Stage BootLoader, FSBL) та формування пакету підтримки плати (Board Support Package, BSP). Під час розробки FSBL можуть бути розгорнуті базові засоби для оновлення (Over-The-Air update, OTA), що може спростити подальші етапи тестування системи. Розробка BSP включає в себе формування необхідного пакету драйверів системи та їх тестування/інтеграцію/адаптацію для обраного набору периферії.

4. Формування основного програмного забезпечення системи, що розробляється. На цьому етапі формується архітектура ПЗ системи та пишеться код застосунків разом з виконанням типової ітераційної розробки ПЗ, орієнтованої на тестування (Test Driven Development, TDD).

5. Комплексне моделювання та налагодження. Під час цього етапу виконується інтеграція розроблених програмних та апаратних компонентів на цільовій платформі. Комплексне налагодження містить як перевірку інтеграції компонентів, так і налагодження граничних станів системи (сплячий режим, вихід зі сплячого режиму, перехід у режим енергозбереження, навантажувальне тестування та перевірки безпеки системи в цілому).

6. Генерація завантажувального образу та розгортання розробленої системи. Для обраного варіанту завантаження системи генерується образ, який може містити такі компоненти, як образ для прошивки енергонезалежної вбудованої пам’яті (Embedded Multimedia Memory Card, eMMC), образ для розгортання системи, використовуючи завантажувач NFS boot (Network File System) для розробників, генерацію та прошивання fuse-конфігурації для захисту системи.

Після вибору варіанта конфігурації процесорного блоку, що забезпечує необхідну продуктивність при виконанні функцій вбудованої системи, можна

перейти безпосередньо до проектування її апаратної платформи.

Розробка вбудованого програмного забезпечення здійснюється з використанням Vitis IDE. Узагальнений процес проектування ПЗ на базі Vitis IDE являє собою типовий цикл розробки для вбудованої платформи. Основна частина – це конфігурування платформи для цільової плати та генерація Software Development Kit (SDK

Вбудоване середовище розробки Vitis IDE містить готові базові приклади роботи з Zynq-7000, серед яких додаток тесту DDR пам’яті, перевірка LwIP мережевого стеку. Після створення апаратної частини у Vivado та генерації необхідних файлів є доцільним пройти перевірку цілісності пам’яті та виконати додаток memory_test, результати якого будуть доступні у консолі послідовного порту, до якого підключено плату. Зазначимо, що для програмування цільової платформи може бути використано як офіційний Xilinx Platform Cable, так і Xilinx Virtual Cable Protocol, що дозволяє виконувати прошивку та налагодження віддаленого пристрою. Всі моделі логіки роботи SoC, як блоку PS, так блоків PL та AXI, виконуються мовою програмування C.

Узагальнений процес проектування IP-ядер виконується у середовищі Vivado IDE з використанням як готових блоків, так і користувацьких ядер. Після створення IP-ядра є можливість виконувати як контроль версій і відповідно оновлювати блоки, так і інтегрувати ядра через процес System Block Designer, де можна налаштувати параметри блоку та інтегрувати його разом до PL частини. Одним з доступних варіантів генерації IP-ядер є використання Vitis HLS для роботи з високорівневим синтезом. Також може бути використаний класичний підхід з Verilog/VHDL описом і відповідним testbench. Після створення IP-ядра необхідно під’єднати основні комунікаційні інтерфейси, а саме AXI шину та вхідні/вихідні порти. Для цього необхідно модифікувати згенерований шаблон на мові опису апаратури Verilog/VHDL.

Процес верифікації IP блоку включає в себе написання відповідного

testbench та інтеграцію у System Block Design. Зазначимо, що після проходження процесу верифікації та збирання IP блок стає доступним у репозиторії проекту, звідки може бути доданий до системи та налаштований. Основними параметрами налаштування можуть бути як константи, що можуть бути змінені для конкретного дизайну, так і параметри AXI шини.

У типовому процесі проектування для платформи ZYNQ є доцільним використовувати інструментарій як Vitis HLS для синтезу IP блоків, так і Vivado для проектування системи. Написання програмного забезпечення для ARM (PS частини) виконується у середовищі Vitis IDE. На рисунку 2.8 зображено типовий маршрут використання інструментів Xilinx. Цей процес являє собою реалізацію окремих IP блоків у середовищі Vitis HLS та написання відповідних testbench для них, після чого імпорт до середовища Vivado, де інтегруються блоки IP з PS частиною ZYNQ. Після синтезу та отримання бітстриму для конфігурації FPGA-частини ZYNQ виконується операція Export Hardware including bitstream для отримання xsa-файлу. Далі, згідно до створеної архітектури системи, є можливим або згенерувати образ Linux з використанням petalinux на базі Yocto Framework та xsa-файлу, або імпортувати отриманий xsa-файл до Vitis IDE та використовувати baremetal SDK від Xilinx [11, 12].

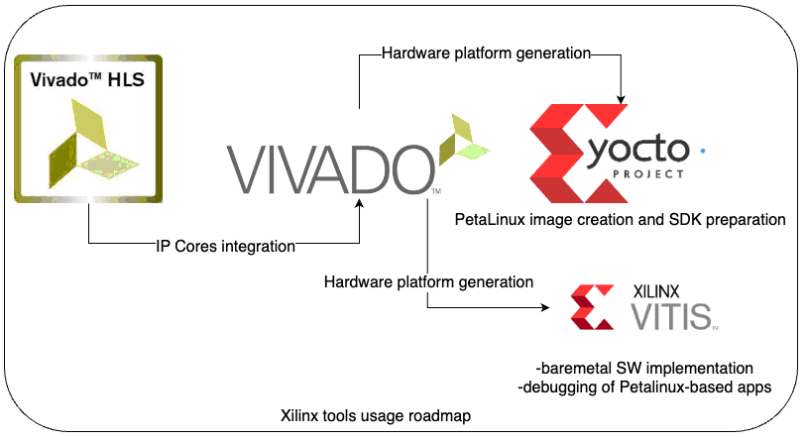


Рисунок 2.8 – Маршрут використання інструментів автоматизованого проектування Xilinx для платформи Zynq

3 РЕАЛІЗАЦІЯ АПАРАТНОГО ПРИСКОРЮВАЧА МНОЖЕННЯ МАТРИЦЬ НА ПЛАТФОРМІ ZEDBOARD

3.1 Схемна реалізація апаратного прискорювача множення матриць

Для розробки апаратного прискорювача множення матриць було обрано середовище Vitis Unified IDE де поєднано як компоненти Vitis IDE, так і Vivado IDE. У Vitis Unified IDE було розроблено IP ядро прискорювача з використанням високорівневого синтезу. Архітектуру розробленої системи наведено на рис. 3.1.

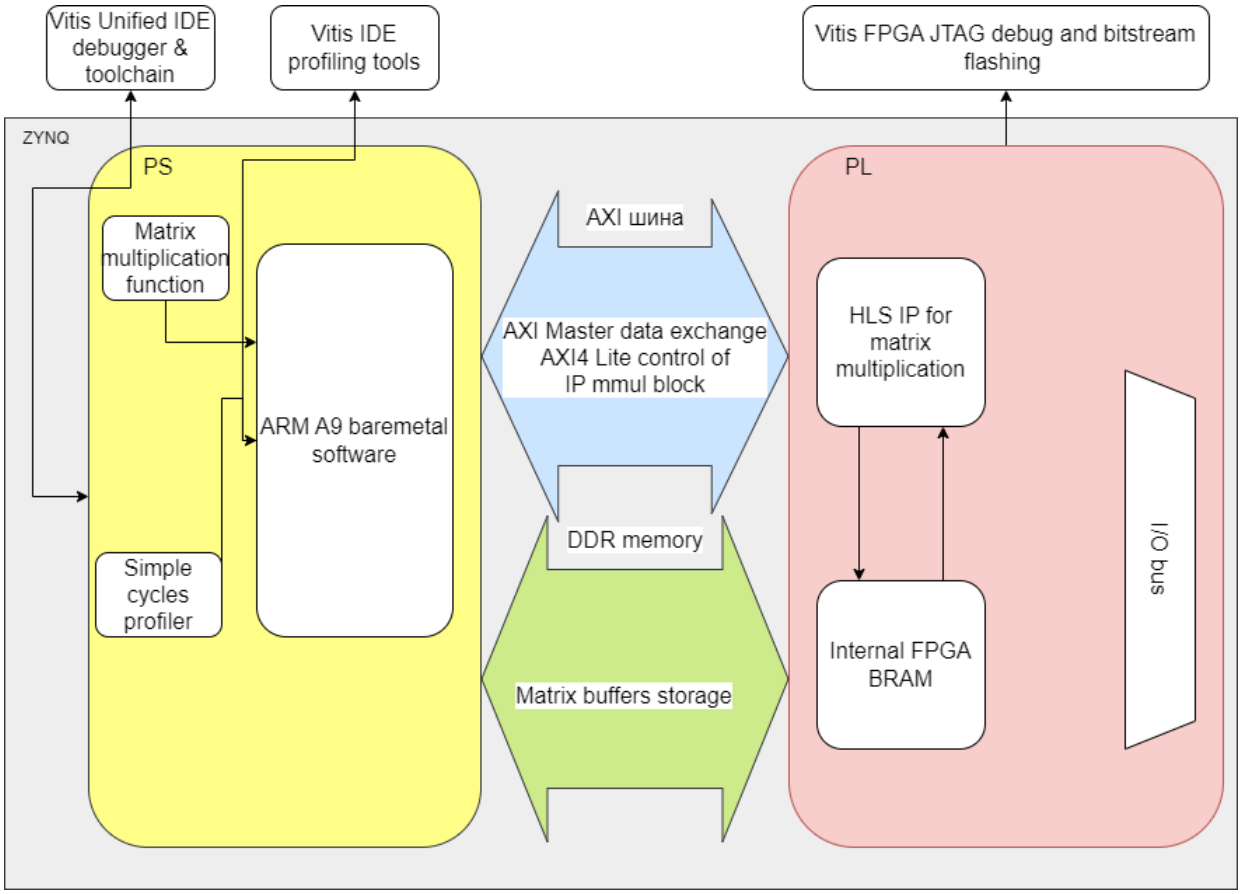


Рисунок 3.1 – Архітектура апаратного прискорювача матриць на на SoC Zynq 7000

Основними компонентами розробленої архітектури є IP ядро множення матриць, яке реалізоване на PL частині SoC, програмний модуль вимірювання часу виконання алгоритму, алгоритм множення матриць який реалізований на PS і PL частині. Процес обміну між синтезованим IP ядром та ARM частиною виконується за допомогою AXI-HP шини та спільного поля пам'яті між PS та PL. Таким чином, дані з матриць, що мають бути помножені, передаються до PL частини шляхом вказування початкових адресів матриць у пам'яті, після чого виконується flush data cache ліній для завантаження даних і запуск обробки даних на IP ядрі. З метою отримання часу виконання кожного з множень, використовується системний таймер SoC ZYNQ.

Діаграму реалізованої системи наведено на рис. 3.2.

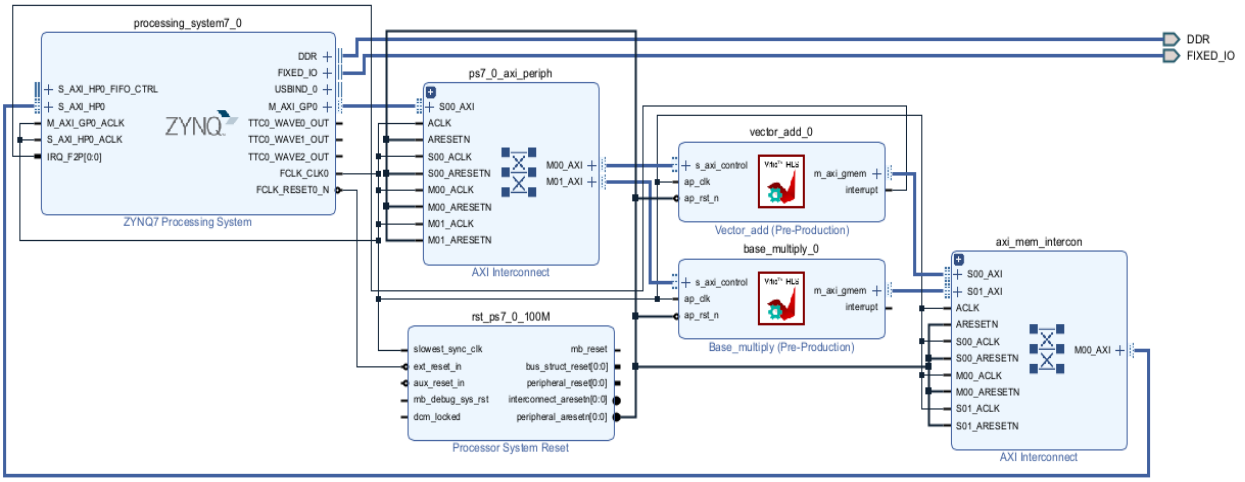


Рисунок 3.2 – Block Diagram архітектури прискорювача множення матриць у середовищі Vivado IDE

Розроблений IP блок для множення матриць містить у собі роботу з AXI-HP шиною для передачі даних з частини PS на PL. На рис. 3.3 зображено загальний вигляд розробленого IP блоку.

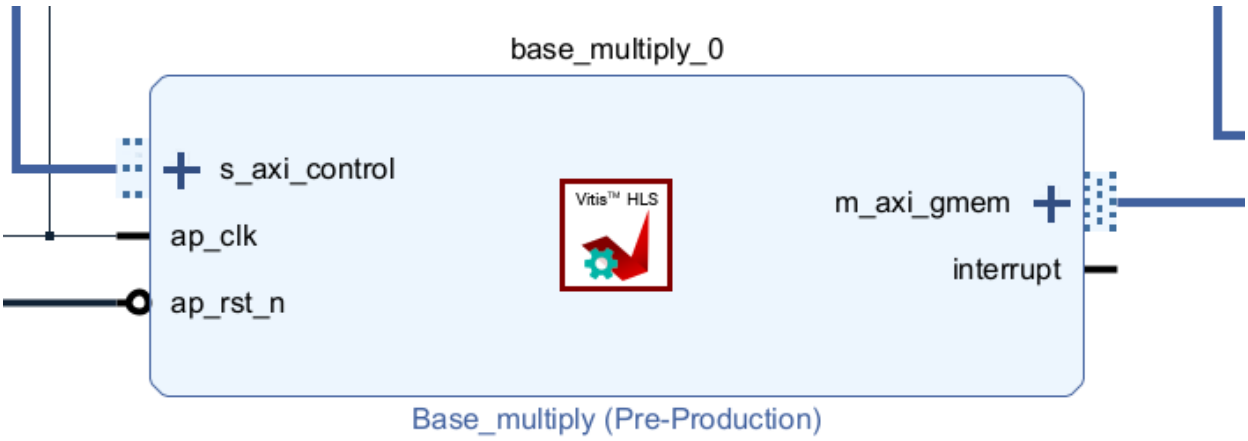


Рисунок 3.3 –Загальний вигляд IP блоку прискорювача множення матриць у середовищі Vivado

Для коректного визначення використаних апаратних витрат для реалізації алгоритму множення з різними розмірами матриць було використано вбудований аналізатор середовища Vitis Unified IDE. Для кожної варіації прискорювача множення був отриманий результат у вигляді апаратних витрат для SoC XC7Z020-CLG400, що встановлений на ZedBoard. Вигляд результатів використаних апаратних витрат у середовищі Vitis Unified IDE наведено на рис. 3.4.

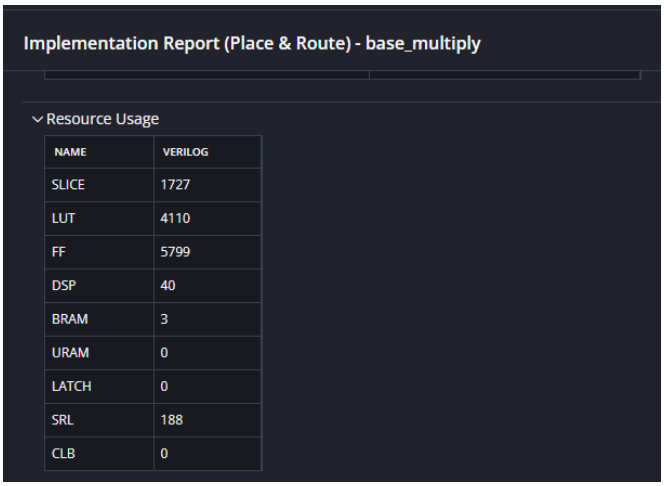


Рисунок 3.4 –Загальний вигляд результатів апаратних витрат IP ядра у середовищі Vitis Unified IDE

3.2 Програмна реалізація проекту

З метою визначення продуктивності розробленого акселератора є доцільним отримати параметри часу виконання множення на різних розмірах матриць. Був взятий класичний алгоритм множення матриць з обчислювальною складністю $O(n^3)$ з метою отримання часу виконання множення матриць розміром 8x8, 16x16, 32x32. Реалізація драйверів верхнього рівня у системі є частково автоматично згенерованою, але залишається необхідним коректно ініціалізувати буфери пам’яті та вказати знаходження матриць у спільному адресному просторі з PL частиною. Також, є необхідним реалізувати логіку обчислення часу виконання кожного множення шляхом використання вбудованого таймеру та макросів.

Для реалізації високорівневого синтезу необхідно вказувати тип інтерфейсу, що використовується для зв’язку PL та PS частин. У середовищі Vitis це робиться шляхом вказування pragma-директив відповідно до вхідних і вихідних параметрів. Реалізацію IP ядра для прискорення множення матриць наведено у лістингу 3.1.

Лістинг 3.1 – Реалізація алгоритму множення матриць з використанням високорівневого синтезу

```
#include <cstring>
#include "mmul.h"

void base_multiply(float *A, float *B, float *C) {
#pragma HLS INTERFACE m_axi port=A depth=32 offset=slave
#pragma HLS INTERFACE m_axi port=B depth=32 offset=slave
#pragma HLS INTERFACE m_axi port=C depth=32 offset=slave
#pragma HLS INTERFACE s_axilite port=return

        float a_buff[MATRIX_SIZE][MATRIX_SIZE];
        float b_buff[MATRIX_SIZE][MATRIX_SIZE];
        float c_buff[MATRIX_SIZE][MATRIX_SIZE];

   float* a_ptr = (float*)a_buff;
```

```
   float* b_ptr = (float*)b_buff;
   float* c_ptr = (float*)c_buff;

         // Load A & B
         memcpy(&a_buff[0][0], const_cast<float*>(A),
                        sizeof(float) * MATRIX_SIZE * MATRIX_SIZE);
         memcpy(&b_buff[0][0], const_cast<float*>(B),
                        sizeof(float) * MATRIX_SIZE * MATRIX_SIZE);

   // Matrix Multiplication
 for (int m = 0; m < MATRIX_SIZE; m++) {
   for (int o = 0; o < MATRIX_SIZE; o ++) {
     c_buff[m][o] = 0;
     for (int n = 0; n < MATRIX_SIZE; n++) {
       c_buff[m][o] += a_buff[m][n] * b_buff[o][n];
     }
   }
  }
         // Store C
         memcpy(C, const_cast<float*>(&c_buff[0][0]),
                        sizeof(float) * MATRIX_SIZE * MATRIX_SIZE);
}
```

Для налаштування адресів, за якими зберігаються вхідна та вихідна матриці, використовуються функції, що генеруються під час компіляції platform-target у середовищі Vitis Unified IDE. Також є необхідним виконати процедуру data cache flush для коректного відображення даних в IP ядро. Фрагмент встановлення адресів матриць та ініціалізації IP ядра наведено у лістингу 3.2.

Лістинг 3.2 – фрагмент налаштування адресів буферів, що використовуються для зберігання матриць у пам’яті та data cache flush

```
      XBase_multiply instance { };
        static bool is_initalized = false;


        if (!is_initalized) {
                XBase_multiply_Initialize(&instance, XPAR_BASE_MULTIPLY_0_DEVICE_ID);
                is_initalized = true; }
        XBase_multiply_Set_A(&instance, reinterpret_cast<UINTPTR>(a.matrix.data()));
        XBase_multiply_Set_B(&instance, reinterpret_cast<UINTPTR>(b.matrix.data()));
```

```
Xil_DCacheFlushRange(reinterpret_cast<UINTPTR>(a.matrix.data()),
                MATRIX_SIZE * MATRIX_SIZE * sizeof(float));
Xil_DCacheFlushRange(reinterpret_cast<UINTPTR>(b.matrix.data()),
                MATRIX_SIZE * MATRIX_SIZE * sizeof(float));
XBase_multiply_Set_C(&instance, reinterpret_cast<UINTPTR>(c.matrix.data()));
```

Для запуску IP ядра на виконання та отримання результатів з нього виконується процедура виклику відповідних функцій та інвалідації data cache з метою перевичитування даних контролером пам’яті з IP ядра. Фрагмент запуску IP ядра для виконання множення матриць та інвалідації data-cache лінії наведено у лістингу 3.3.

Лістинг 3.3 – Фрагмент запуску IP ядра для виконання множення матриць та інвалідації data cache

```
XBase_multiply_Start(&instance);
  while (!XBase_multiply_IsDone(&instance)) {
  }
  Xil_DCacheInvalidateRange(reinterpret_cast<UINTPTR>(c.matrix.data()),
                MATRIX_SIZE * MATRIX_SIZE * sizeof(float));
```

Реалізацію профілювання обчислення часу виконання множення матриць було реалізовано з використанням системного таймеру SoC Zynq. На лістингу 3.4 наведено фрагмент макросу, що використовується для профілювання виклику функції множення.

Лістинг 3.4 – Фрагмент реалізації макросу для отримання часу виконання обраної функції

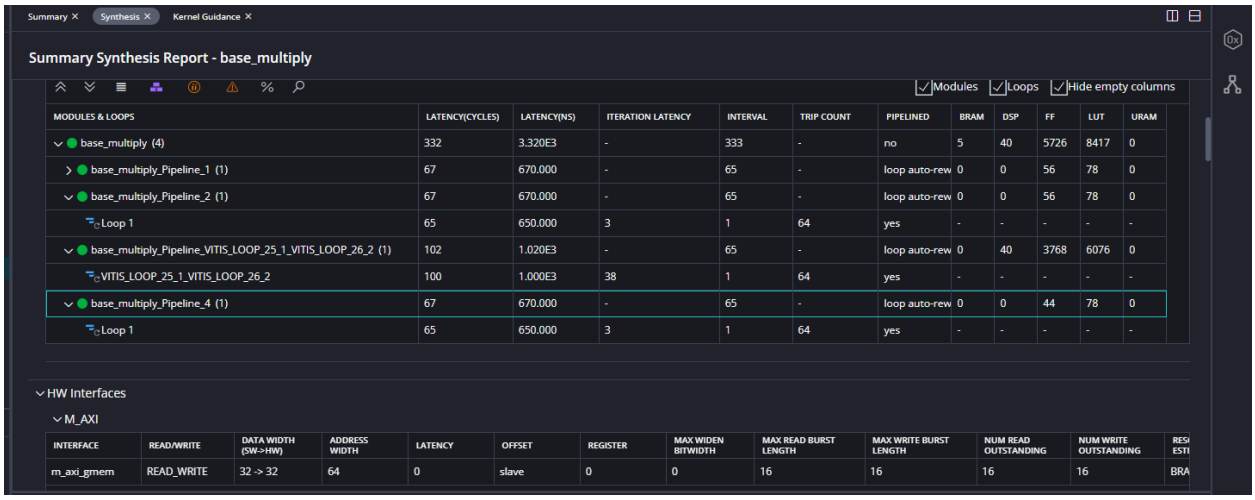
```
#ifndef SIMPLE_PROFILER_H
#define SIMPLE_PROFILER_H
#include <xtime_l.h>
#include <stdio.h>
#ifdef PROFILING

#define PRINT_EXECUTION_TIME(FUNC) \
   do { \
      XTime start, end; \
```

```
        XTime_GetTime(&start); \  FUNC; \
        XTime_GetTime(&end); \
        printf("Execution time of %s: %.2f us\n", #FUNC, \
              1.0 * (end - start) / (COUNTS_PER_SECOND / 1e6)); \
    } while (0)
#else
#define PRINT_EXECUTION_TIME(FUNC) \
 do { \  FUNC; \ } while(0)
#endif
#endif
```

3.3 Оцінка апаратних та часових ресурсів апаратного прискорювача

З метою отримання результатів апаратних витрат та часу виконання множення матриць на реалізованому IP ядрі прискорювача було використано плату налагодження ZedBoard. Для реалізації обраний найпростіший алгоритм множення матриць з обчислювальною складністю $O(n^3)$. В якості прикладу обрані матриці розміром (8x8, 16x16 та 32x32), елементи яких є знакові числа з плаваючою точкою (float data type). Результати апаратних витрат були отримані шляхом аналізу вихідних звітів високорівневого синтезатора. На рисунку 3.5 наведено приклад одного з отриманих звітів у Vitis Unified IDE по апаратним витратам реалізації прискорювача на PL частині SoC ZYNQ-7000.



MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM	DSP	FF	LUT	URAM
base_multiply (4)	332	3.320E3	-	333	-	no	5	40	5726	8417	0
base_multiply_Pipeline_1 (1)	67	670.000	-	65	-	loop auto-rew	0	0	56	78	0
base_multiply_Pipeline_2 (1)	67	670.000	-	65	-	loop auto-rew	0	0	56	78	0
Loop 1	65	650.000	3	1	64	yes	-	-	-	-	-
base_multiply_Pipeline_VITIS_LOOP_25_1_VITIS_LOOP_26_2 (1)	102	1.020E3	-	65	-	loop auto-rew	0	40	3768	6076	0
VITIS_LOOP_25_1_VITIS_LOOP_26_2	100	1.000E3	38	1	64	yes	-	-	-	-	-
base_multiply_Pipeline_4 (1)	67	670.000	-	65	-	loop auto-rew	0	0	44	78	0
Loop 1	65	650.000	3	1	64	yes	-	-	-	-	-

INTERFACE	READ/WRITE	DATA WIDTH (SW->HW)	ADDRESS WIDTH	LATENCY	OFFSET	REGISTER	MAX WIDEN BITWIDTH	MAX READ BURST LENGTH	MAX WRITE BURST LENGTH	NUM READ OUTSTANDING	NUM WRITE OUTSTANDING	RES ESTI
m_axi_gmem	READ_WRITE	32 -> 32	64	0	slave	0	0	16	16	16	16	BRA

Рисунок 3.5 – Результати синтезу апаратного прискорювача множення матриць

Для аналізу апаратних витрат використано порівняння апаратних витрат PL частини SoC ZYNQ-7000 на реалізацію прискорювача для матриць різного розміру (8x8, 16x16 та 32x32) при використані одного виду алгоритму множення. Результати наведені у табл. 3.1, яка містить зведену інформацію апаратних витрат на реалізацію розподіленого прискорювача матриць з використанням PS-PL частин SoC ZYNQ-7000.

Таблиця 3.1 – Зведена таблиця апаратних витрат реалізації прискорювача на платформі Xilinx ZYNQ 7000

Розмір матриці	Апаратні витрати на реалізацію прискорювача множення матриць
N1(8x8)	FF 5799, LUT 4110, DSP 40/220, BRAM 3
N2(16x16)	FF 9421, LUT 6784, DSP 80/220, BRAM 3
N3(32x32)	FF 15920, LUT 27523, DSP 160/220, BRAM 70

З наведених результатів видно, що при зростанні кількості комірок матриці у 16 разів (від 8x8 до 32x32) апаратні витрати PL частини SoC ZYNQ-7000 зростають в середньому в 3-6 разів. Це підтверджує ефективність обраної архітектури апаратного прискорювача множення матриць з точки зору апаратних витрат.

Швидкодія апаратного прискорювача перевірялася на алгоритмі множення матриць з обчислювальною складністю $O(n^3)$ для матриць (8x8, 16x16 та 32x32). Порівняльний аналіз виконувався між PS частиною з ARM Cortex A9 ZYNQ-7000 (а), PL частиною ZYNQ-7000 з обміном даними з PS частиною через AXI HP (б), одноплатним комп'ютером Cortex A53 Raspberry Pi Zero 2W з архітектурою процесора ARM Cortex A53 (в), мікроконтролером STM32G474RGT6 з ядром ARM Cortex M4 (г) і персональними комп’ютерами (д) Apple Mac M1 (8 ядер, тактова частота 3,2

GHz, оперативна пам'ять 8 GB) та (е) x86 Ryzen 5 3600 (6 ядер, тактова частота 3,6 GHz, оперативна пам'ять 16 GB).

При проведенні експерименту по перевірці швидкодії доцільно порівнювати аналогічні за призначенням пристрої, які використовуються у вбудованих системах. У таблиці 3.2 наведено результати швидкодії кожної з реалізацій для розглянутих пристроїв та розмірів матриць.

Таблиця 3.2 – Зведена таблиця швидкодії різних пристроїв апаратної реалізації множення матриць

Платформа → / Розмір матриці	а PS	б PL	в Raspberry	г STM	д x86	е Mac
N1(8x8)	5.24 µs	7.63 µs	10 µs	560 µs	5 µs	3 µs
N2(16x16)	36.20 µs	20.38 µs	20 µs	4400 µs	10 µs	8 µs
N3(32x32)	268.10 µs	70.8µs	260 µs	35000 µs	20 µs	17 µs

Найменшу швидкодію показує мікроконтролер STM32G474RGT6. Це обумовлено виключно програмною реалізацією прошивки мікроконтролера. Одноплатний комп’ютер Raspberry Pi та реалізація на PS частині ZYNQ-7000 показують приблизно однакову швидкодію. Найбільшу швидкодію серед компонентів вбудованих систем показує реалізація на PL частині ZYNQ-7000. Часові витрати для реалізації на PL частині зростають приблизно лінійно (від N2(16x16) до N3(32x32) обсяг даних, що обробляються зростає в 4 рази і часові витрати зростають приблизно в тому ж обсязі).

Стаціонарні ПК з великою оперативною пам’яттю, високошвидкісним процесором та спеціальними математичними бібліотеками задають верхню межу швидкодії, до якої має наближатися ідеальний апаратній прискорювач на SoC, що проектується. Саме його реалізація на PL частині ZYNQ-7000 наближається до показників ПК по швидкодії (збільшення часу виконання всього в 2 – 4 рази).

ВИСНОВКИ

В ході виконання кваліфікаційної роботи розроблений прототип апаратного прискорювача множення матриць з використанням SoC сімейства ZYNQ-7000 на налагоджувальній платі ZedBoard .

Реалізація апаратного прискорювача виконана на базі стеку інструментальних засобів САПР Vivado/Vitis HLS з використанням мови програмування С.

Виконано макетування розробленого апаратного прискорювача математичних операцій та проведений демонстраційний експеримент, який підтвердив працездатність розробленого проекту. Проаналізовані апаратурні витрати та швидкодія прискорювача для множення матриць різних розмірів.

Аналіз апаратурних витрат на реалізацію прискорювача на PL частині SoC ZYNQ-7000 показав, що апаратурні витрати в основному залежать від розміру матриці, але не напряму. Наприклад, при зростанні кількості комірок матриці у 16 разів (від 8x8 до 32x32) апаратні витрати PL частини зростають в середньому в 3-6 разів. Це підтверджує ефективність обраної архітектури апаратного прискорювача множення матриць з точки зору апаратних витрат.

При проведенні експерименту по перевірці швидкодії порівнювалися аналогічні за призначенням пристрої, які використовуються у вбудованих системах та стаціонарні комп’ютери. Найменшу швидкодію показує мікроконтролер STM32G474RGT6. Це обумовлено виключно програмною реалізацією прошивки мікроконтролера. Одноплатний комп’ютер Raspberry Pi та використання PS частини ZYNQ-7000 показують приблизно однакову швидкодію. Найбільшу швидкодію серед компонентів вбудованих систем показує реалізація на PL частині ZYNQ-7000. Саме ця реалізація наближається до показників стаціонарних ПК по швидкодії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Дмитрієва О.А. Спеціальні розділи обчислювальної математики. Комп'ютерний практикум: навч. посіб. / О. А. Дмитрієва. – Київ: КПІ ім. Ігоря Сікорського, 2023. – 110 с.

2. Шахно С.М., Дудикевич А.Т., Левицька С.М. Практична реалізація чисельних методів лінійної алгебри: навч. посібник. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2009. – 137 с.

3. Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, Renfei Zhou. New Bounds for Matrix Multiplication: from Alpha to Omega. Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 3792–383. DOI:10.1137/1.9781611977912.134.

4. Shkil, A.S., Filippenko, O.I., Rakhlis, D.Y., Filippenko, I.V., Parkhomenko, A.V. and Korniienko, V.R. 2024. Adaptive filtering and machine learning methods in noise suppression systems, implemented on the SoC . Radio Electronics, Computer Science, Control. 4 (Dec. 2024), 163–174. DOI:https://doi.org/10.15588/1607-3274-2024-4-16

5. Reich O., Ozkan M.A., Hannig F., Teich J., Schmid M. Loop parallelization techniques for FPGA accelerator synthesis. Journal of Signal Processing Systems, 2018, vol. 90, no.1, P. 3–27. DOI: 10.1007/s11265-017-1229-7.

6. Bogaraju V.C.S., Paul K., Lavenier D., Balakrishnan M. Hardware acceleration of de novo genome assembly. IJES, 2017, vol. 9, no. 1, P. 74–89. DOI: 10.1504/IJES.2017.081729.

7. Fang J., Mulder Y.T.B., Hidders J., Lee J., Hofstee P.H. In-memory database acceleration on FPGAs: a survey. The VLDB Journal, 2020, vol. 29, pp. 33–59. DOI: 10.1007/s00778-019-00581-w.

8. The Zynq Book. Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC [Electronic resource] – Zynqbook.com – Mode of access: http://www.zynqbook.com/ – Date of access:

17.09.2023.

9. AXI Basics 1 - Introduction to AXI [Electronic resource] – Adaptive SoC & FPGA Support – Mode of access: https://support.xilinx.com/s/article/1053915? – Date of access: 10.09.2023.

10. Advanced Micro Devices - Vitis HLS [Electronic resource] – AMD Xilinx – Mode of access: https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html – Date of access: 20.09.2023.

11. Шкіль О. Проєктування та самодіагностика кіберфізичних пристроїв керування на платформі SoC / О. Шкіль, Д. Рахліс, І. Філіпенко, В. Корнієнко // Сучасний стан наукових досліджень та технологій в промисловості. – 2023. – № 4 (26). – С. 122-134. DOI: https://doi.org/10.30837/ITSSI.2023.26.122.

12. Шкіль О.С., Автоматизоване проектування цифрових фільтрів на платформі SoC / О.С. Шкіль, В.Р. Корнієнко, Д.В. Карась // Проблеми інформатизації: тези доп. 12-ї міжнар. наук.-техн. конф., 21-22 листопада 2024 р., Баку – Харків – Бельсько-Бяла: . Т. 3 : секція 5 / Інститут систем управління МНО Азербайджанської республіки [та ін.]. – Харків : ХНУРЕ, 2024. – С. 11.