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 другий (магістерський)
(рівень вищої освіти)

Моделі оптимізації коду в середовищі високорівневого синтезу при програмуванні SoC
(тема)

Виконав: Здобувач другого року навчання,
групи СКСм-23-2 Карманський Б.Ю.
(прізвище, ініціали)

Спеціальність *123 Комп'ютерна інженерія*

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

Освітня програма
Комп'ютерна інженерія
(повна назва освітньої програми)

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 другий (магістерський)

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 » 09 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Карманському Богдану Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи (проєкту) Моделі оптимізації коду в середовищі високорівневого синтезу при програмуванні SoC
затверджена наказом університету від « 08 » 11 2024 р. № 1189 Ст
2. Термін подання здобувачем роботи 30.12.2024

3. Вихідні дані до роботи (проєкту)
Налагоджувальна плата ZebBoard з кристалом ZYNQ-7000 Xilinx Inc.
Мови програмування C/C++
Мови опису апаратури VHDL, Verilog
САПР XILINX ISE, Vivado, Vitis

4. Перелік питань, що потрібно опрацювати у роботі
Моделі оптимізації коду
Кіберфізичні системи на платформі SoC
Взаємодія процесорного ядра та ПЛІС у складі SoC
Високорівневий синтез для плати ZebBoard з SoC ZYNQ-7000

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) 21 слайд_____
_____
_____
_____
_____

6. Консультанти розділів роботи (проєкту)

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проєкту)	Термін виконання етапів проєкту (роботи)	Примітка
1	Аналіз проблеми	09.09.2024 – 22.09.2024	
2	Огляд існуючих алгоритмів	23.09.2024 – 06.10.2024	
3	Визначення та реалізація основних моделей оптимізації коду	07.10.2024 – 04.11.2024	
4	Вибір технологічної платформи SoC	04.11.2024 – 10.11.2024	
5	Реалізація маршруту проєктування на платформі SoC ZYNQ-7000	11.11.2024 – 01.12.2024	
6	Оформлення пояснювальної записки	02.12.2024 – 22.12.2024	
7	Захист проєкту	13.01.2025 – 30.01.2025	

Дата видачі завдання 02.09.2024_____

Здобувач _____ Карманський Б.Ю._____
(підпис)

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

РЕФЕРАТ

Магістерська кваліфікаційна робота містить 59 сторінок, 9 рисунків, 10 таблиць, 9 джерел.

SOC, ZYNQ-7000, ARM, FPGA, ВИСОКОРІВНЕВИЙ СИНТЕЗ, ОПТИМІЗАЦІЯ, МОДЕЛІ ОПТИМІЗАЦІЇ, АВТОМАТИЗОВАНЕ ПРОЄКТУВАННЯ.

Мета роботи – дослідження моделей оптимізації коду в середовищі високорівневого синтезу для підвищення продуктивності платформ System-on-Chip (SoC).

Об’єкт дослідження – процес оптимізації коду для платформ SoC на базі Zynq-7000.

Предмет дослідження – моделі та техніки оптимізації коду в середовищі високорівневого синтезу з використанням інструментів Xilinx Vivado, Vitis.

Методи дослідження – аналіз архітектури SoC Zynq-7000, дослідження алгоритмів множення матриць, перетворення Фур’є та вейвлет-перетворення, тестування різних технік оптимізації: розгортання циклів, конвеєризації, зменшення кількості звернень до пам’яті.

Виконано детальний аналіз архітектури SoC Zynq-7000, зокрема взаємодії між програмованою логікою та процесорною системою. Проведено оптимізацію алгоритмів множення матриць, перетворення Фур’є та вейвлет-перетворення для підвищення продуктивності. Розроблено моделі оптимізації коду, які дозволяють скоротити час виконання. Проведено тестування та верифікацію розроблених моделей за допомогою інструментів Vivado та Vitis.

ABSTRACT

The master’s qualification thesis contains 59 pages, 9 figures, 10 tables, 9 references.

SOC, ZYNQ-7000, ARM, FPGA, HIGH-LEVEL SYNTHESIS, OPTIMIZATION, OPTIMIZATION MODELS, AUTOMATED DESIGN.

The purpose of the study is to investigate code optimization models in a high-level synthesis environment to enhance the performance of System-on-Chip (SoC) platforms.

The object of the study is the process of code optimization for SoC platforms based on Zynq-7000.

The subject of the study is models and techniques for code optimization in a high-level synthesis environment using Xilinx Vivado and Vitis tools.

Research methods include an analysis of the SoC Zynq-7000 architecture, a study of matrix multiplication algorithms, Fourier transform, and wavelet transform, as well as testing various optimization techniques such as loop unrolling, pipelining, and reducing memory accesses.

A detailed analysis of the SoC Zynq-7000 architecture, including the interaction between programmable logic and the processing system, was conducted. Optimization of matrix multiplication algorithms, Fourier transform, and wavelet transform was performed to improve performance. Code optimization models were developed to reduce task execution time. The developed models were tested and verified using Vivado and Vitis tools.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

SoC – System on Chip – система на кристалі

FPGA – Field Programmable Gate Array – програмована користувачем вентильна матриця

HLS – High-Level Synthesis – високорівневий синтез

PS – Processing System – обчислювальна система

PL – Programmable Logic – програмована логіка

AXI – Advanced eXtensible Interface – високопродуктивний інтерфейс передачі даних

ARM – Advanced RISC Machine – сімейство процесорів із розширеним набором команд

VHDL – VHSIC Hardware Description Language – мова опису апаратури для надвеликих інтегральних схем

ISE – Integrated Software Environment – інструмент для проєктування апаратного забезпечення від Xilinx

Vivado – Vivado Design Suite – програмне забезпечення для розробки цифрових систем

Vitis – Vitis Unified Software Platform – платформа для розробки програмного забезпечення на SoC

DDR3, DDR3L, LPDDR2 – типи оперативної пам’яті

FSBL – First Stage Bootloader – первинний завантажувач системи

OTA – Over-The-Air update – оновлення системи через бездротову мережу

NFS – Network File System – мережева файлова система

RTL – Register Transfer Level – рівень передачі регістрів

ASIC – Application-Specific Integrated Circuit – інтегральна схема спеціального призначення

DSP – Digital Signal Processor – процесор цифрової обробки сигналів

TDD – Test Driven Development – розробка, орієнтована на тестування

ВСТУП

Сучасний розвиток комп'ютерних технологій та електронних систем у різних галузях промисловості, науки та медицини вимагає постійного підвищення продуктивності та ефективності обчислювальних процесів. Зростання складності завдань обробки сигналів, зображень і великих обсягів даних стимулює пошук нових рішень для проєктування високопродуктивних обчислювальних платформ. Одним із ключових підходів до вирішення таких завдань є використання систем на кристалі (System-on-Chip, SoC), що об'єднують процесори загального призначення та програмовану логіку.

Серед сучасних платформ SoC особливої уваги заслуговує серія Zynq-7000 від компанії Xilinx, яка поєднує ARM-процесори з високою обчислювальною потужністю та гнучкість програмованої логіки FPGA. Використання таких платформ дозволяє не лише інтегрувати численні обчислювальні елементи на одному кристалі, але й досягати високої ефективності завдяки можливості апаратного прискорення критично важливих завдань.

Однією з ключових задач, висвітлених у роботі, є дослідження різних алгоритмів і методів, які можуть бути використані для апаратного прискорення. До таких алгоритмів належать множення матриць, перетворення Фур’є та вейвлет перетворення. Результати аналізу демонструють, як оптимізація коду та апаратна реалізація цих алгоритмів можуть покращити продуктивність системи.

Методологія роботи базується на використанні сучасних інструментів проєктування, таких як Xilinx Vivado та Vitis. Вони дозволяють інтегрувати високорівневий код з апаратною частиною системи, забезпечуючи ефективність та точність розробки. Крім того, у роботі приділено увагу тестуванню оптимізаційних моделей, зокрема, їх впливу на продуктивність та складність проєктування.

У роботі розглядаються різні техніки оптимізації, включаючи розгортання циклів, конвеєризацію, мінімізацію звернень до функцій та інші підходи, які дозволяють суттєво підвищити ефективність розробки та забезпечити продуктивність системи в обчислювально-інтенсивних задачах.

Об’єкт дослідження у роботі – процес автоматизованого проєктування та тестування цифрових пристроїв на технологічній платформі SoC.

Предмет дослідження – моделі, методи та техніки оптимізації для автоматизованого проєктування та тестування на цифрових пристроях, реалізованих на платформі SoC.

Мета роботи – впровадження моделей та методів оптимізації для автоматизованого проєктування, верифікації та тестування цифрових пристроїв на технологічній платформі SoC.

Для досягнення поставленої мети необхідно вирішити такі задачі:

– провести аналіз архітектури SoC Zynq-7000, зокрема взаємодії між програмованою логікою (PL) та процесорною системою (PS);

– дослідити існуючі алгоритми множення матриць, перетворення Фур’є та вейвлет перетворення для визначення їх придатності до оптимізації;

– розробити та впровадити моделі оптимізації коду в середовищі високорівневого синтезу для платформ SoC;

– протестувати різні методи оптимізації, такі як розгортання циклів, зменшення кількості звернень до функцій та оптимізація доступу до пам’яті;

– реалізувати та перевірити ефективність запропонованих моделей оптимізації на SoC Zynq-7000 та порівняти з результатом на інших сучасних платформах.

Таким чином, результати дослідження можуть бути корисними для розробки сучасних високопродуктивних систем, які базуються на платформі SoC, та сприяти подальшому розвитку інструментів високорівневого синтезу. Ця робота також надає рекомендації щодо практичного впровадження та тестування оптимізаційних моделей, що можуть стати основою для подальших досліджень у цій галузі.

1 АПАРАТНА ПЛАТФОРМА SOC ZYNQ-7000.

1.1 Модель архітектури Zynq-7000

Основна концепція SoC полягає в тому, що один кремнієвий чіп може бути використаний для реалізації функціональності цілої системи, замість використання кількох різних мікросхем. У минулому термін SoC зазвичай відносився до прикладної інтегральної схеми (ASIC), яка може містити цифрові, аналогові та радіочастотні компоненти, а також блоки змішаних сигналів для реалізації аналого-цифрових і цифро-аналогових перетворювачів (АЦП і ЦАП).

SoC може поєднувати в собі всі аспекти цифрової системи: обробку, високошвидкісну логіку, інтерфейс, пам'ять і так далі. Всі ці функції можуть бути реалізовані за допомогою фізично окремих пристроїв і об'єднані разом у систему на рівні друкованої плати. Рішення на основі SoC мають нижчу вартість, забезпечують швидшу та безпечнішу передачу даних між різними елементами системи, мають вищу загальну швидкість роботи системи, менше енергоспоживання, менші фізичні розміри та кращу надійність.

Основними недоліками SoC на базі ASIC є час і вартість розробки та відсутність гнучкості. Одноразові інженерні зусилля (і витрати) на розробку ASIC є значними, що робить цей тип SoC придатним лише для ринків з великими обсягами виробництва, де немає потреби в майбутніх модернізаціях. Характерними прикладами SoC на основі ASIC є інтегровані процесори, які використовуються в ПК, планшетах і смартфонах. Вони зазвичай складаються з щонайменше двох процесорних ядер, пам'яті, графіки, інтерфейсів та інших функцій, і виробляються у великих обсягах для продуктів з обмеженим терміном служби [1].

Zynq забезпечує платформу для реалізації гнучких SoC: Xilinx позиціонує пристрій як "повністю програмовану SoC" (APSoC). Високорівнева модель архітектури Zynq зображена на рисунку 1.1 [2]. Треба

зазначити, що Zynq складається з двох основних частин: обчислювальної системи (PS), сформованої навколо двоядерного процесора ARM Cortex-A9, та програмованої логіки (PL), яка є FPGA частиною SoC. Він також має інтегровану пам'ять, різноманітні периферійні пристрої та високошвидкісні комунікаційні інтерфейси.

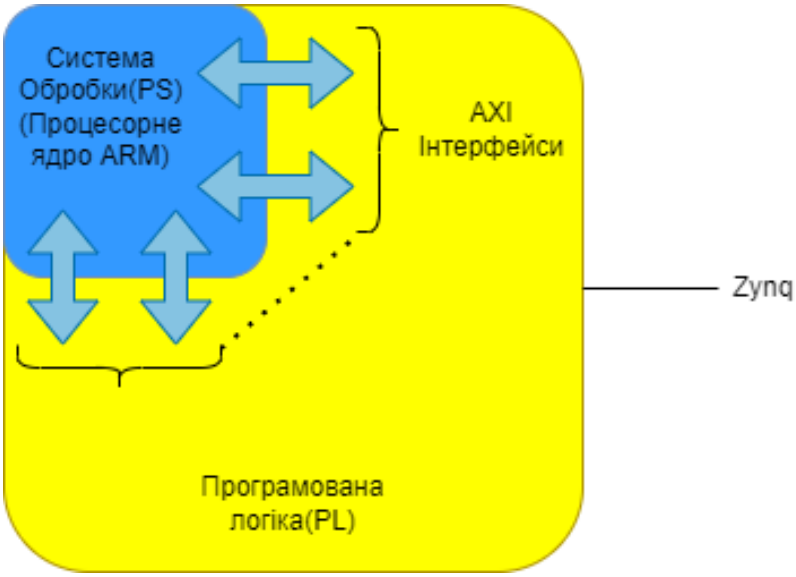

Рисунок 1.1 – Спрощена модель архітектури Zynq

1.2 Структура вбудованих SoC

Базову модель цифрових систем включають процесор, пам'ять і периферійні пристрої, а також шини, що з'єднують різні елементи між собою. Це і є апаратна система. Модель апаратної системи показана на рисунку 1.2.

Процесор можна вважати центральним елементом апаратної системи. На процесорі працює програмна система (програмний "стек"), що складається з застосунків (зазвичай на базі операційної системи (ОС)) і нижнього шару програмного забезпечення для взаємодії з апаратною системою. Зв'язок між елементами системи відбувається за допомогою з'єднань. Це можуть бути прямі з'єднання типу "точка-точка" або шини, що обслуговують декілька компонентів. В останньому випадку потрібен протокол для управління

доступом до шини. Треба зауважити, що хоча на рис. 1.2 показано одну шину з підключеними периферійними пристроями, процесор може обслуговувати декілька з'єднаних шин.

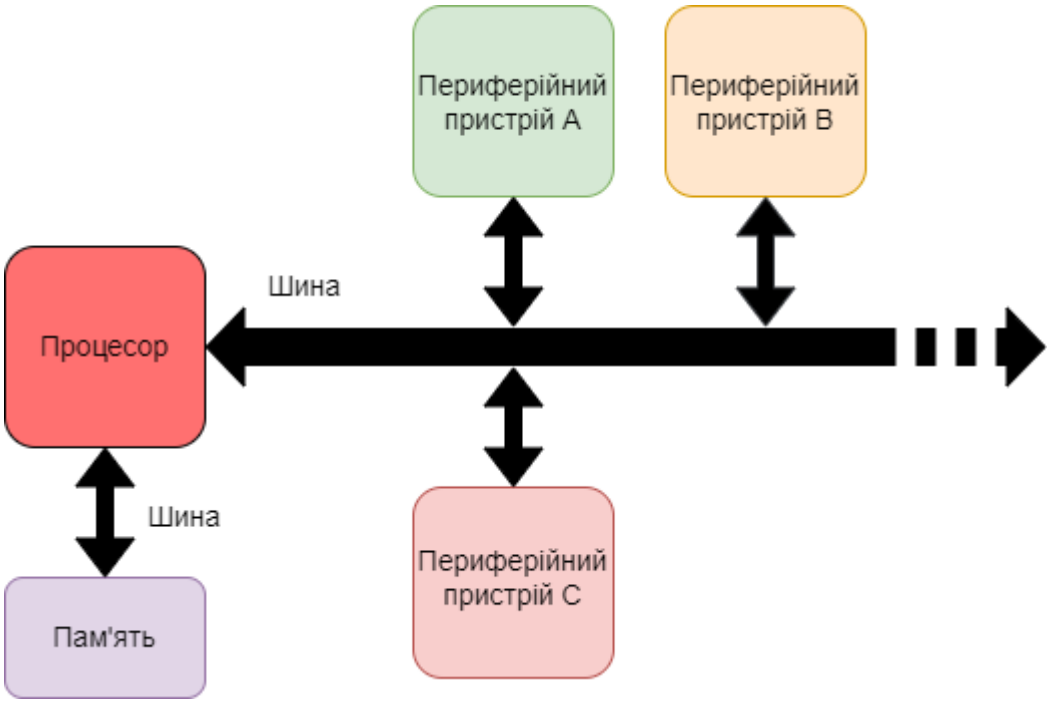

Рисунок 1.2 – Спрощена архітектура апаратної системи вбудованої SoC

Периферійні пристрої – це функціональні компоненти, розташовані поза процесором, і загалом вони виконують одну з трьох функцій: со-процесори – елементи, що доповнюють основний процесор, зазвичай оптимізовані для виконання певного завдання; ядра для взаємодії із зовнішніми інтерфейсами, наприклад, підключення до світлодіодів і перемикачів, перетворювачів(codecs) тощо; і додаткові елементи пам'яті. Периферійні пристрої можна розглядати як дискретні функціональні блоки, які можна спроєктувати, протестувати та інтегрувати в систему, а також "упакувати" для подальшого повторного використання.

На рисунку 1.3 показано вигляд апаратної системи, зображеної на рисунку 1.2, зіставленої з пристроєм Zynq (зображеним на рисунку 1.1). Архітектури обох пристроїв були суттєво спрощені, але на даному етапі метою

є надання детального пояснення того, як вбудовані мікропроцесори відображаються на пристрої Zynq. PS має фіксовану архітектуру і містить процесор та системну пам'ять, тоді як PL є повністю гнучкою, надаючи розробнику "чисте полотно" для створення власних периферійних пристроїв або повторного використання стандартних. Взаємозв'язок реалізується через інтерфейси AXI, що з'єднують PS і PL.

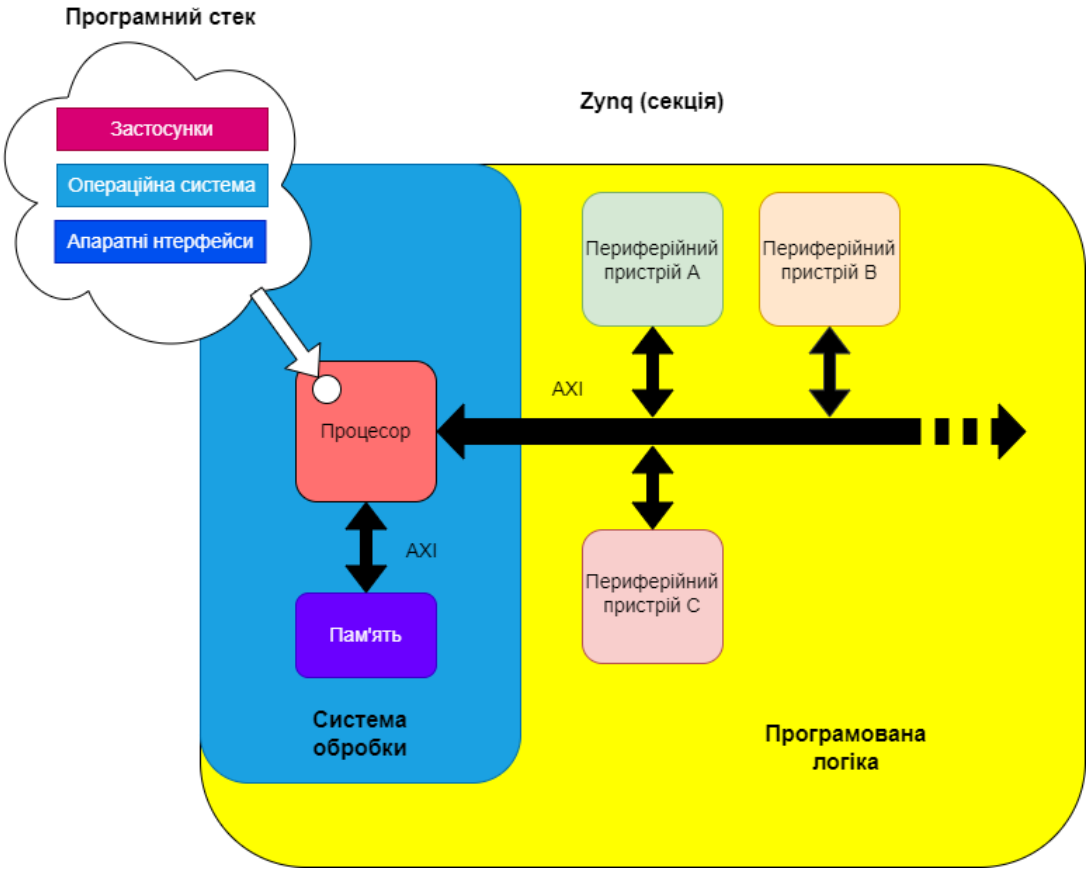

Рисунок 1.3 – Взаємозв’язок між програмною та апаратною частинами в архітектурі Zynq

1.3 Процесорна система обробки

Processing System (PS) в архітектурі Zynq-7000 є критично важливою складовою, яка забезпечує високу продуктивність та гнучкість для розробки вбудованих систем. Головною обчислювальною одиницею PS є двоядерний

процесор ARM Cortex-A9. Кожне ядро якого працює на частоті до 1 ГГц і забезпечує високу обчислювальну потужність при низькому енергоспоживанні. Процесор підтримує виконання інструкцій в режимі суперскалярного паралелізму, що дозволяє одночасно виконувати кілька інструкцій.

Крім обчислювальних ядер, PS містить кеш-пам'ять першого рівня (L1) і другого рівня (L2). Кеш L1 поділяється на кеш для інструкцій і кеш для даних, що дозволяє зменшити затримки доступу до часто використовуваних даних та інструкцій. Кеш L2 є загальним для обох ядер і забезпечує додаткову буферизацію даних між процесором та основною пам'яттю.

PS підтримує кілька типів оперативної пам'яті, включаючи DDR3, DDR3L та LPDDR2. Контролери пам'яті забезпечують високу пропускну здатність і низькі затримки, що є критично важливим для обчислювальних завдань, які потребують швидкого доступу до великих обсягів даних.

Інтерфейси вводу-виводу (I/O) у PS забезпечують підключення до зовнішніх периферійних пристроїв та комунікаційних каналів. Серед них є інтерфейси USB, Ethernet, SDIO, UART, SPI та I2C. Інтерфейси USB та Ethernet забезпечують підключення до мереж та зовнішніх пристроїв з високою пропускною здатністю, що є важливим для багатьох мережевих та мультимедійних застосувань.

PS також містить кілька додаткових функцій, таких як Direct Memory Access (DMA), що дозволяє швидко передавати дані між пам'яттю та периферійними пристроями без участі процесора. Це значно знижує навантаження на процесор та підвищує загальну продуктивність системи.

Безпека є ще одним важливим аспектом PS. Вона забезпечується за допомогою різноманітних механізмів, включаючи захищені зони пам'яті та функції шифрування даних. Це дозволяє захистити конфіденційні дані та забезпечити надійне виконання критично важливих завдань.

1.4 Програмована логіка

Програмована логіка (PL) базується на FPGA технології від Xilinx, яка дозволяє користувачам конфігурувати апаратну логіку для виконання спеціалізованих завдань. Архітектура PL містить велику кількість програмованих логічних блоків, пам'яті та блоків цифрової обробки сигналів (DSP). Логічні блоки складаються з комбінаційних логічних функцій та тригерів, що дозволяє виконувати різноманітні логічні операції. З’єднання між цими блоками забезпечують гнучкість конфігурації, дозволяючи створювати складні логічні схеми.

Блоки пам'яті (Block RAM) забезпечують високошвидкісну внутрішню пам'ять для зберігання даних та інструкцій. Ці блоки можуть бути конфігуровані для різних режимів роботи, включаючи одно- та двопортові конфігурації, що дозволяє оптимізувати використання пам'яті для конкретних завдань.

DSP блоки (DSP48E1) призначені для виконання швидких арифметичних операцій, таких як множення та додавання, що є критично важливим для цифрової обробки сигналів та інших обчислювально інтенсивних завдань. DSP блоки можуть працювати на високих частотах та забезпечувати обробку великих обсягів даних в реальному часі.

Конфігурація PL здійснюється за допомогою файлів конфігурації (bitstream), які створюються за допомогою інструментів розробки від Xilinx, таких як Vivado. Ці інструменти дозволяють розробникам описувати логічні схеми на мовах апаратного опису (HDL), таких як VHDL або Verilog, або використовувати високорівневий синтез (HLS) для створення апаратних прискорювачів з C/C++ коду.

Після створення файлу конфігурації, він завантажується в PL через інтерфейс конфігурації, що дозволяє швидко змінювати конфігурацію апаратної логіки залежно від вимог застосунку. Це забезпечує високу

гнучкість та можливість адаптації системи до змінюваних умов і також виконувати часткову реконфігурацію PL частини.

PL в архітектурі Zynq-7000 забезпечує широкий спектр застосувань, включаючи обробку зображень та відео, цифрову обробку сигналів, мережеві рішення, системи реального часу та багато інших. Завдяки гнучкості та високій продуктивності, PL дозволяє створювати інноваційні рішення, які можуть адаптуватися до специфічних вимог застосунку.

Однією з головних переваг PL є можливість створення апаратних прискорювачів, які можуть значно прискорити виконання обчислювально інтенсивних завдань порівняно з традиційними процесорними рішеннями. Це дозволяє зменшити затримки, підвищити продуктивність та знизити енергоспоживання системи.

1.5 Інтерфейси взаємодії між PL та PS

Основним способом взаємодії між PS та PL є використання інтерфейсів AXI (Advanced eXtensible Interface). Існує кілька типів AXI інтерфейсів, що забезпечують різні види комунікації між PS і PL. AXI Master інтерфейси дозволяють PL доступ до пам'яті PS, що дозволяє програмованій логіці читати та записувати дані безпосередньо в пам'ять, керовану процесором. Це дає змогу знизити затримку та підвищити швидкість обробки даних, особливо в завданнях, що потребують інтенсивного використання пам'яті.

AXI Slave інтерфейси, навпаки, дозволяють PS доступ до ресурсів PL. Це означає, що процесор може безпосередньо керувати програмованою логікою, передавати їй дані або отримувати результати обробки. Така гнучкість дозволяє ефективно розподіляти завдання між PS і PL, використовуючи сильні сторони кожної з цих частин.

AXI Stream інтерфейси використовуються для високошвидкісної передачі даних між PS і PL. Вони ідеально підходять для завдань, що потребують обробки потоків даних в реальному часі, таких як обробка відео

або аудіо. Інтерфейси AXI Stream забезпечують мінімальні затримки та високу пропускну здатність, що дозволяє виконувати складні обчислювальні завдання з максимальною ефективністю.

Важливою частиною інтерфейсів між PS і PL є система керування пам'яттю. PS має кілька контролерів пам'яті, що дозволяють ефективно організовувати доступ до оперативної пам'яті як для PS, так і для PL. Це забезпечує можливість швидкого обміну даними між процесором і програмованою логікою, що є критично важливим для завдань, які потребують інтенсивної обробки даних.

Завдяки використанню інтерфейсів AXI та ефективній системі керування пам'яттю, PS і PL можуть спільно використовувати ресурси пам'яті, що значно підвищує загальну продуктивність системи. Це дозволяє розробникам створювати ефективні та продуктивні рішення для вбудованих систем, використовуючи переваги як процесорної, так і програмованої логіки.

Інтерфейси між PS і PL також підтримують додаткові функції, такі як Direct Memory Access (DMA) та кеш-когерентність. DMA дозволяє виконувати швидкий обмін даними між пам'яттю та периферійними пристроями без участі процесора, що значно знижує навантаження на процесор та підвищує загальну продуктивність системи.

Кеш-когерентність забезпечує узгодженість даних між кеш-пам'яттю PS і PL, що дозволяє уникнути помилок при доступі до спільних даних. Це особливо важливо для завдань, які потребують одночасного доступу до одних і тих самих даних з боку процесора та програмованої логіки.

2 ТЕОРЕТИЧНІ ОСНОВИ ВИСОКОРІВНЕВОГО СИНТЕЗУ ТА ПРОГРАМУВАННЯ SOC

2.1 Маршрут проєктування SoC на базі Zynq

Звичайно, як і в будь-якому проєкті, першим етапом є визначення бажаної поведінки системи, тобто створення відповідної специфікації з набору вимог. Це зображено як відправна точка у верхній частині діаграми, і вона є основою для подальшої розробки дизайну системи.

Архітектура Zynq поєднує в собі процесор ARM (для програмних елементів проєктованої системи) та ПЛІС (переважно для апаратних елементів системи, хоча при бажанні тут також можуть бути реалізовані додаткові soft-процесори такі як MicroBlaze або RISK-V-based). Отже, ключовим елементом наступного етапу проєктування системи є належний розподіл передбачуваної функціональності між програмним і апаратним забезпеченням, а також визначення інтерфейсів між цими двома частинами. Звичайно, можливо, що цей розподіл згодом буде скориговано, оскільки розробники ітераційно наближатимуть систему до завершення.

Після розбиття системи на розділи розробка програмного та апаратного забезпечення може здійснюватися значною мірою паралельно. З погляду розробки апаратного забезпечення, завдання полягає у визначенні необхідних функціональних блоків для реалізації проєкту, а потім їх збірці за допомогою певної комбінації повторного використання проєкту і розробки нових IP ядер, а також встановлення відповідних зв'язків між блоками. Аналогічно, програмний аспект проєкту може бути реалізований шляхом розробки спеціального коду або повторного використання вже існуючого програмного забезпечення. Верифікація як програмного, так і апаратного забезпечення буде необхідною, і це є невід'ємною і важливою частиною процесу.

Нарешті, апаратні та програмні елементи системи повинні бути інтегровані відповідно до інтерфейсів, визначених на етапі специфікації, а

також проведене подальше тестування "всієї системи". Базова модель потоку проєктування для SoC на базі Zynq зображена на рисунку 2.1 [1].

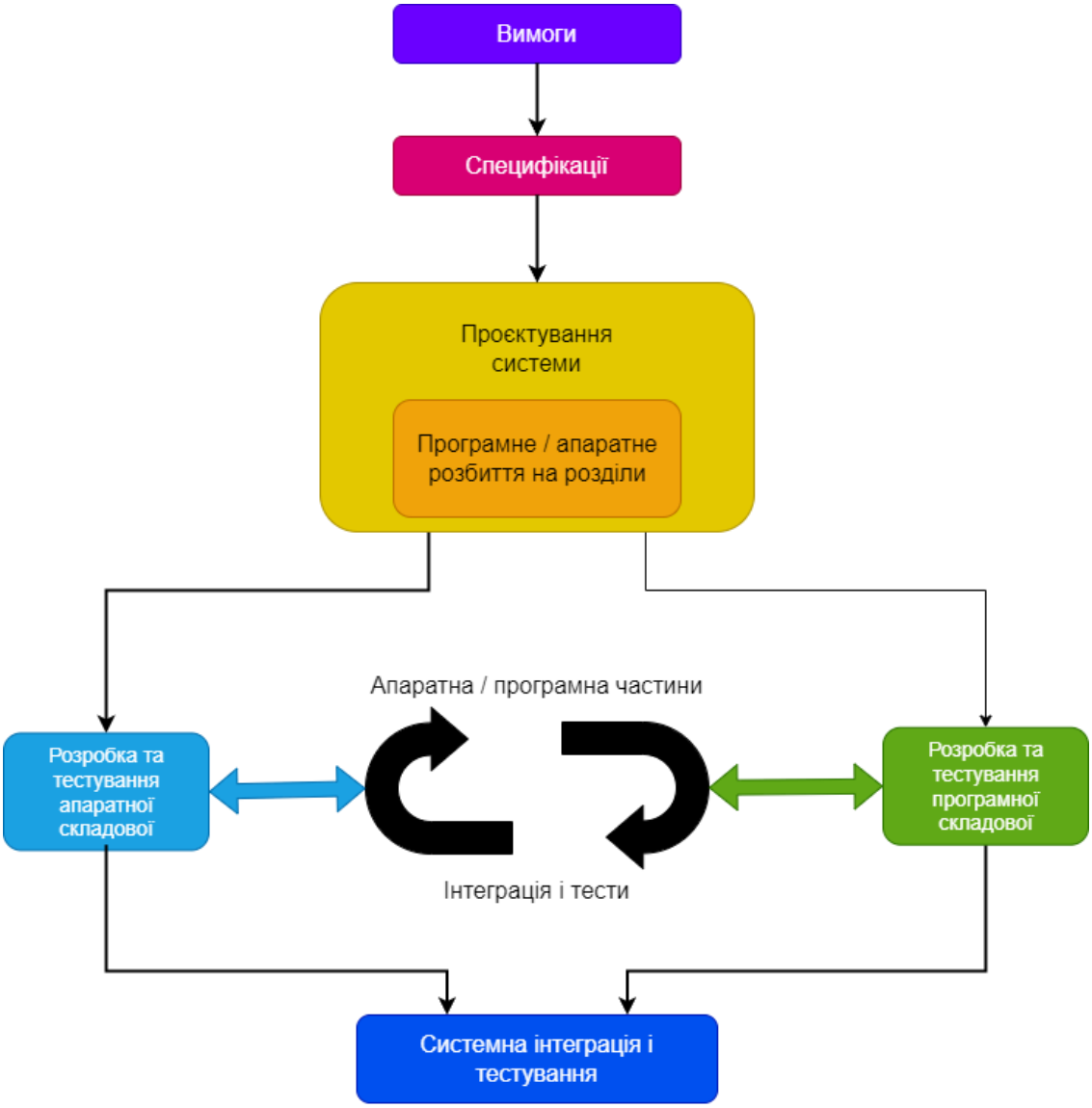

Рисунок 2.1 – Базова модель потоку проєктування для SoC на базі Zynq

2.2 Етапи проєктування SoC Zynq

Процес створення вбудованих систем на основі кристалів програмованої логіки з архітектурою FPGA та розширюваних обчислювальних платформ серії Zynq-7000 AP SoC включає такі етапи:

1. Розробка проєкту мікропроцесорної системи. На цьому етапі визначаються архітектура системи та розподіл функцій між апаратною та програмною частинами, використовуючи критерії та метрики, такі як енергоспоживання, швидкість реакції системи, оптимальність виконання блоків. Вибирається цільовий процесор або SoC для подальших етапів проєктування.

2. Проєктування апаратної платформи системи. Включає створення проєкту апаратної платформи та визначення необхідних IP-ядер для конкретної задачі. Реалізація апаратної платформи базується на обраному чіпі, а також проводиться її верифікація.

3. Підготовка системних програмних засобів нижнього рівня. Сюди входить розробка та налагодження первинного завантажувача системи (FSBL – first stage bootloader) і створення пакету BSP (board support package). Під час розробки FSBL можуть бути впроваджені базові засоби OTA (Over-The-Air update), що спрощує подальші етапи тестування. Розробка BSP включає створення пакету драйверів, їх тестування, інтеграцію та адаптацію для обраного набору периферійних пристроїв.

4. Розробка основного програмного забезпечення системи. Включає визначення архітектури програмного забезпечення, підготовку коду застосунків та виконання процесу розробки з тестуванням (TDD – Test Driven Development) для ітераційної розробки програмного забезпечення.

5. Комплексне моделювання та налагодження. На цьому етапі виконується інтеграція програмних та апаратних компонентів на цільовій платформі. Проводиться комплексне налагодження, яке включає перевірку інтеграції компонентів та налаштування граничних станів системи, таких як сплячий режим, вихід зі сплячого режиму, перехід у режим енергозбереження, навантажувальне тестування та перевірки безпеки системи.

6. Генерація завантажувального образу та розгортання системи. Для обраного варіанту завантаження системи створюється образ, який може включати компоненти для прошивки зовнішньої пам’яті eMMC, розгортання

системи за допомогою NFS boot (Network File System) для розробників, а також генерацію та прошивку fuse-конфігурації для захисту системи.[3]

2.3 Високорівневий синтез

Високорівневий синтез (HLS) є важливим етапом у розробці сучасних цифрових систем, зокрема систем на кристалі. HLS дозволяє автоматично трансформувати алгоритми, написані на мовах високого рівня, таких як C/C++, у апаратний опис на мовах низького рівня, як-от VHDL або Verilog. Це дозволяє зменшити час і складність проєктування апаратури, що є критично важливим у сучасних високотехнологічних галузях.

HLS забезпечує кілька ключових переваг. По-перше, він зменшує розрив між програмним і апаратним забезпеченням, дозволяючи програмістам і інженерам апаратного забезпечення працювати на одному рівні абстракції. Це дозволяє швидше і ефективніше реалізовувати складні алгоритми та оптимізувати їх для виконання на специфічному апаратному забезпеченні, такому як FPGA.

Однією з найпоширеніших платформ для розробки SoC є Zynq-7000 від компанії Xilinx. Ця плата поєднує в собі потужність програмованої логіки FPGA з продуктивністю процесорного ядра ARM, що забезпечує високу гнучкість та ефективність у реалізації складних систем. Архітектура Zynq дозволяє інтегрувати різноманітні периферійні пристрої та використовувати програмовану логіку для прискорення обчислень, що робить її ідеальною платформою для розробки вбудованих систем і прототипування нових технологій.

Основою успіху при використанні Zynq є ефективна взаємодія між програмованою логікою і процесорним ядром. Це досягається за рахунок високошвидкісних інтерфейсів, які дозволяють передавати дані між цими двома частинами системи з мінімальними затримками. Таке рішення дозволяє

реалізувати складні обчислювальні задачі, використовуючи апаратні прискорювачі, що значно підвищує продуктивність системи.

Одним із ключових інструментів для високорівневого синтезу є Vitis HLS, який надає розробникам можливість швидко і ефективно створювати апаратні прискорювачі з використанням мов програмування високого рівня. Vitis HLS автоматично перетворює код, написаний на C/C++, у апаратний опис, оптимізуючи його для виконання на FPGA. Це забезпечує високу продуктивність і ефективність розроблених систем, дозволяючи зосередитися на алгоритмах і функціональності, залишаючи оптимізацію апаратного рівня інструменту HLS.

Процес роботи з Vitis HLS включає кілька етапів. Спочатку розробник створює проєкт, визначаючи вихідні файли та налаштування. Далі виконується синтез коду і його симуляція для перевірки коректності роботи. Після цього проводиться аналіз результатів і застосування оптимізацій, які дозволяють підвищити продуктивність і зменшити споживання ресурсів. Завершальним етапом є генерація кінцевого апаратного коду, який завантажується на FPGA для виконання.

Застосування високорівневого синтезу має великий вплив на продуктивність і ефективність цифрових систем. Оптимізація коду, виконана за допомогою HLS, дозволяє значно зменшити кількість необхідних обчислень, скоротити час виконання задач і підвищити ефективність використання апаратних ресурсів. Це робить HLS незамінним інструментом для розробки сучасних високопродуктивних систем.

2.4 Причини використання високорівневого синтезу

Синтез високого рівня обіцяє потужну перспективу: можливість генерування реалізацій рівня передачі регістрів (RTL) виробничої якості на основі високорівневих специфікацій. Іншими словами, HLS автоматизує

ручний процес, усуваючи джерело багатьох помилок проєктування і прискорюючи дуже довгу та ітеративну частину циклу розробки[4].

2.4.1 Новий потік проєктування

Щоб повністю зрозуміти потенціал і переваги HLS, важливо розглянути все в перспективі процесу проєктування апаратного забезпечення. Сьогодні більшість проєктів починаються з певної форми специфікації. Іноді це простий письмовий документ, але досить часто створюється виконувана модель - зазвичай на ANSI C, C++ або SystemC. На цьому ранньому етапі специфікація є по суті функціональною: вона майже не містить деталей апаратної реалізації, і її основна мета – перевірити і точно налаштувати бажану поведінку. Після тестування ця поведінкова модель проходить кілька кроків, поки не набуває форми реальної апаратної реалізації. Першим кроком є визначення оптимальної архітектури для реалізації бажаної функціональності. Якщо функціональність визначає, "що" робить система, то архітектура визначає, "як" вона це робить, з прямими наслідками на продуктивність, площу та енергоспоживання. Після того, як архітектура визначена, команда розробників вручну кодує ці рішення у вигляді Verilog або VHDL RTL опису.

Саме тут і криється найбільша проблема. Знайти відповідну архітектуру – непросте завдання, а знайти оптимальну – ще складніше. Але фундаментальною проблемою є ручна природа всього цього підходу. Простіше кажучи, будь-яке ручне втручання є джерелом помилок. Раптом те, що спочатку було простим процесом від специфікації до реалізації, перетворюється на кошмарний ітеративний цикл. Написаний вручну RTL-дизайн тестується, повідомляється про помилки, витрачається час на їх пошук і виправлення – лише для того, щоб перейти до наступної помилки. Цей процес міг би бути нескінченним, якби він не повинен був закінчитися в певний момент, щоб вкластися у встановлені часові терміни [4].

2.4.2 Доцільність переходу до високорівневого синтезу

Проблема, звичайно, загострюється зі збільшенням розмірів конструкції. Чим більша система і складніший застосунок, тим більше шансів на помилки і тим важче дотримуватися графіка. На жаль, постійно зростаюча складність – один із небагатьох постійних факторів у розробці електронних систем.

В електроніці: технологічні вузли й геометрія процесів продовжують зменшуватися до нанометрових масштабів (наприклад, 5-нм і 3-нм технології), тактові частоти зростають, кількість вбудованих ядер множиться, а методології верифікації запозичують машинне навчання і штучний інтелект зі спільноти розробників програмного забезпечення.

Процес створення RTL залишається головною проблемою під час проектування систем у складових частинах яких є FPGA або ASIC. Реалізація високорівневих алгоритмів таких як декодери H.265 (HEVC) і VP9 за допомогою мов, що мають низькорівневі примітиви для опису цільової схеми. Складність створення високоефективної обчислювальної архітектури зростає разом із витратами на її тестування і верифікацію. На поточний момент верифікація є вузьким місцем у будь-якому проєкті ASIC.

Високорівневий синтез усуває першопричину цієї проблеми, забезпечуючи безпомилковий шлях від абстрактних специфікацій до RTL. Використовуючи HLS, команди розробників значно прискорюють час проєктування, а також зменшують загальні зусилля з верифікації.

2.4.3 Полегшення проєктування та верифікації з допомогою HLS

При роботі на високому рівні абстракції потрібно набагато менше деталей для опису. Наприклад, на функціональному рівні інженерам не потрібно турбуватися про деталі реалізації, такі як мікроархітектура, особливості синтезу блоків під конкретну платформу або технологічний

процес. Вони можуть зосередитися лише на бажаній поведінці. Це значно полегшує написання опису. З меншою кількістю рядків коду ризик помилок значно зменшується, а з меншою кількістю речей, які потрібно перевірити у вихідному коді, легше повністю перевірити модель [4].

Після того, як високорівнева модель написана і перевірена, HLS автоматизує процес реалізації RTL. Але якщо інструменти HLS усувають ручне втручання і помилки, вони не усувають інженерного втручання. Тобто, рішення все одно потрібно приймати. За допомогою високорівневого синтезу інженери залишаються під контролем; вони приймають рішення, а інструмент HLS їх реалізує. Вони просто мають більш ефективний і продуктивний спосіб виконання своєї роботи. Наприклад, проєктувальник вирішує, який рівень паралелізму потрібен для оптимальної архітектури, і відповідно задає ці параметри в інструменті HLS. У свою чергу, інструмент піклується про розподіл і планування необхідних апаратних ресурсів, побудову шляху передачі даних і структур управління, щоб створити повністю функціональну і оптимізовану реалізацію. За допомогою HLS правильний RTL отримується швидше, що скорочує фазу створення. Як результат, знижуються витрати на налагодження і зменшується навантаження на верифікацію за рахунок можливостей використання ко-симуляції та написання тестових сценаріїв на C/C++.

3 ОГЛЯД АЛГОРИТМІВ

Вибрані для оптимізації алгоритми (множення матриць, перетворення Фур'є, вейвлет перетворення) широко використовуються у високопродуктивних обчисленнях, обробці сигналів, аналізі зображень та наукових обчисленнях. Висока обчислювальна складність цих алгоритмів робить їх пріоритетними кандидатами для оптимізації.

Обчислювальна складність:

– алгоритми множення матриць мають обчислювальну складність $O(N^3)$ для традиційного множення та алгоритму Віноградa і $O(N^{2,807})$ для алгоритму Штрассена, що суттєво впливає на час виконання для великих розмірів матриць;

– складність обчислення FFT складає $O(NlogN)$, проте все ще має великий потенціал для апаратної оптимізації;

– вейвлет перетворення дозволяє отримувати частотні характеристики сигналу разом із часовою локалізацією, що робить його ефективним для аналізу нестаціонарних сигналів. Обчислювальна складність $O(2N)$, забезпечує високу швидкість обробки, однак реалізація алгоритмів, таких як Добеші, вимагає багаторівневих ітерацій, що може значно впливати на загальний час виконання.

Всі обрані алгоритми є інтенсивними з погляду використання ресурсів пам'яті та обчислювальної потужності. Оптимізація цих алгоритмів має прямий вплив на продуктивність платформи SoC, зменшуючи час виконання. Також вона дозволяє адаптувати їх для різних типів задач, включаючи наукові дослідження, обробку мультимедіа, машинне навчання та інші прикладні області. Це підвищує універсальність отриманих результатів.

Вибрані алгоритми легко порівнювати за часом виконання, що дозволяє об'єктивно оцінити ефективність оптимізацій.

3.1 Множення матриць

Розглянемо алгоритми множення матриць. В цих експериментах, для виконання множення використовуються процесори AMD Ryzen 9 7945HX (з операційною системою Windows 11) та Qualcomm Snapdragon 732G (з операційною системою Android). Квадратні матриці заповнюються довільними цілими числами формату int. Заміри швидкості виконання проводяться в мілісекундах (мс).

3.1.1 Традиційне множення

Нехай є дві матриці $A\ (m\ \times\ n)$ і $B\ (n\ \times\ p)$. Тоді добутком буде матриця $\mathrm{C}\ =\ A\ast\ B$ розміром $m\times\ p$.

Елемент C_{ij} матриці C обчислюється за формулою:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} * B_{kj} \tag{3.1}$$

Приклад реалізації розрахунку добутку матриць традиційним множенням мовою програмування C наведено в лістингу 3.1.

Лістинг 3.1 – Функція розрахунку добутку матриць

```
void base_multiply(int A[MATRIX_SIZE][MATRIX_SIZE], int B[MATRIX_SIZE][MATRIX_SIZE],
int C[MATRIX_SIZE][MATRIX_SIZE])
{
   for (int i = 0; i < MATRIX_SIZE; i++)
   {
      for (int j = 0; j < MATRIX_SIZE; j++)
      {
         C[i][j] = 0;
         for (int k = 0; k < MATRIX_SIZE; k++) {
            C[i][j] += A[i][k] * B[k][j];
         }
      }
   }
}
```

Таблиця 3.1 – Швидкість виконання множення матриць традиційним алгоритмом

	Процесор	
Розмір матриці	Ryzen 9	Snapdragon 732
16x16	0,01	0,02
32x32	0,08	0,17
64x64	0,59	1,33
128x128	5,5	10,93

Обчислювальна складність алгоритму $O(N^3)$.

3.1.2 Алгоритм Віноградa

Алгоритм Віноградa – це оптимізована версія стандартного алгоритму множення матриць. Основна ідея алгоритму полягає в тому, щоб зменшити кількість операцій множення, використовуючи попередньо обчислені результати для спрощення та прискорення обчислень у великих квадратних матрицях.

Нехай є дві матриці $A\ (n \times m)$ і $B\ (m \times p)$. Тоді добутком буде матриця C розміром $n \times p$.

План виконання алгоритму[5]:

1. Попередня обробка рядків і стовпців:
   – для кожного рядка матриці A провести обчислення:

$$rowFactor[i] = \sum_{k=1}^{m/2} A[i][2k-2] * A[i][2k-1] \quad (3.2)$$

   – для кожного стовпця матриці B провести обчислення:

$$colFactor[i] = \sum_{k=1}^{m/2} B[2k-2][j] * B[2k-1][j] \quad (3.3)$$

2. Після обчислення проміжних значень, виконуються основні операції множення та додавання:

$$\mathrm{C}[i][j] = -rowFactor[i] - colfactor[j] + \sum_{k=1}^{\frac{m}{2}}(A[i][2k-2] + \\ +B[2k-1][j]) * (A[i][2k-1] + B[2k-2][j]) \quad (3.4)$$

3. Якщо число стовпців m непарне, потрібно виконати додаткове коригування:

$$C[i][j] += A[i][m-1] + B[m-1][j] \quad (3.5)$$

Приклад реалізації алгоритму Віноградa мовою програмування C наведений в лістингу 3.2.

Лістинг 3.2 – Базова реалізація розрахунку добутку матриць алгоритмом Віноградa

```
void winograd_multiply(int A[MATRIX_SIZE][MATRIX_SIZE], int
B[MATRIX_SIZE][MATRIX_SIZE], int C[MATRIX_SIZE][MATRIX_SIZE])
{
    int rowFactor[MATRIX_SIZE];
    int colFactor[MATRIX_SIZE];
    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        rowFactor[i] = 0;
        for (int j = 0; j < MATRIX_SIZE / 2; j++) {
            rowFactor[i] += A[i][2 * j] * A[i][2 * j + 1];
        }
    }
    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        colFactor[i] = 0;
        for (int j = 0; j < MATRIX_SIZE / 2; j++) {
            colFactor[i] += B[2 * j][i] * B[2 * j + 1][i];
        }
    }
    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            C[i][j] = -rowFactor[i] - colFactor[j];
```

```
            for (int k = 0; k < MATRIX_SIZE / 2; k++) {
                C[i][j] += (A[i][2 * k] + B[2 * k + 1][j]) * (A[i][2 * k + 1] + B[2 * k][j]);
            }
        }
    }
    if (MATRIX_SIZE % 2 == 1)
    {
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                C[i][j] += A[i][MATRIX_SIZE - 1] * B[MATRIX_SIZE - 1][j];
            }
        }
    }
}
```

Таблиця 3.2 – Швидкість виконання множення матриць алгоритмом Віноградa

	Процесор	
Розмір матриці	Ryzen 9	Snapdragon 732
16x16	0,01	0,02
32x32	0,09	0,14
64x64	0,48	1,09
128x128	4,3	8,97

Обчислювальна складність алгоритму відносно традиційного множення зберігається – $O(N^3)$, але в реальних умовах він може працювати швидше за рахунок зменшення кількості множень, особливо на великих матрицях.

3.1.3 Алгоритм Штрассена

Замість традиційного розрахунку елементів матриці результату за допомогою трьох вкладених циклів, алгоритм Штрассена використовує рекурсивний підхід і метод розбиття на блоки для зменшення кількості множень, необхідних для обчислення добутку двох матриць. Він працює особливо добре на великих матрицях, розмір яких є степенем двійки, що дозволяє зручно розділяти їх на блоки.

Алгоритм працює на квадратних матрицях розміру $2^n \times 2^n$. Якщо вхідні матриці не є квадратними або розміром не є степенем двійки, до них можуть

бути додані нульові рядки і стовпці, щоб отримати необхідний розмір.

План виконання алгоритму:

1. Дві квадратні матриці A і B розміром $2^n \times 2^n$ розбиваються на підматриці кожна розміром $n/2 \times n/2$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},\ B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{3.6}$$

2. Обчислення семи допоміжних матриць $M_1 - M_7$ за допомогою рекурсивних множень:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\ M_2 &= (A_{21} + A_{22})B_{11}, \\ M_3 &= A_{11}(B_{12} - B_{22}), \\ M_4 &= A_{22}(B_{21} - B_{11}), \\ M_5 &= (A_{11} + A_{12})B_{22,} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}), \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned} \tag{3.7}$$

3. Використовуючи матриці $M_1 - M_7$ обчислюються чотири підматриці результату $C_{11}, C_{12}, C_{21}, C_{22}$:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7, \\ C_{12} &= M_3 + M_5, \\ C_{21} &= M_2 + M_4, \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned} \tag{3.8}$$

4. Отримані блоки об’єднуються в одну матрицю C, яка є добутком матриць A і B.[5]:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \quad (3.9)$$

Приклад реалізації алгоритму Штрассена мовою програмування С наведений в додатку Б.

Таблиця 3.3 – Швидкість виконання множення матриць алгоритмом Штрассена

	Процесор	
Розмір матриці	Ryzen 9	Snapdragon 732
16x16	0,51	1,22
32x32	3,51	9,31
64x64	24,7	61,7
128x128	182	436

В алгоритмі використовується 7 множень, через це він стає ефективнішим на великих матрицях. З мінусів можна виділити складну реалізацію, через яку він має малу ефективність на малих матрицях, а також може накопичувати похибку при роботі з числами з плаваючою точкою.

Обчислювальна складність алгоритму складає $O(N^{2,807})$, але в реальних умовах він виконується значно довше традиційного множення, або множення алгоритмом Віноградa. Можливо цей алгоритм може надати прискорення виконання множення тільки на матрицях астрономічного масштабу. В ході цього дослідження ми не будемо розглядати такі матриці, тому оптимізувати цей алгоритм не будемо.

3.2 Перетворення Фур’є

В цих експериментах, для розрахунку перетворення Фур’є використовуються процесори AMD Ryzen 9 7945HX (з операційною системою Windows 11) та Qualcomm Snapdragon 732G (з операційною системою

Android). Вхідний сигнал для перетворення Фур'є(прямого та зворотного) заповнюється довільними числами з рухомою комою одинарної точності формату float. Заміри швидкості виконання проводяться в мілісекундах (мс).

Швидке перетворення Фур'є (FFT, Fast Fourier Transform). Це ефективний алгоритм для обчислення дискретного перетворення Фур'є (DFT) і його зворотного перетворення (IDFT). Воно значно прискорює обчислення DFT шляхом зниження обчислювальної складності з $O(N^2)$ [6] до $O(NlogN)$.

Для розрахунку FFT використаємо алгоритм Кулі-Тьюкі(Cooley-Tukey). Цей алгоритм розбиває сигнал на парні і непарні індекси, застосовує DFT до кожної частини і комбінує результати. Алгоритм реалізовується за допомогою властивості симетрії і періодичності функцій [7].

Розглянемо DFT. Для дискретного сигналу $x[n]$ з N точок, DFT визначається за формулою:

$$X[k] = \sum_{n=0}^{N-1} x[n] * e^{-j\frac{2\pi kn}{N}} \quad (3.10)$$

де $X[k]$ – спектральні компоненти, $x[n]$ – вхідний сигнал, j – уявна одиниця ($j^2 = -1$), N – кількість точок у сигналі.

Розглянемо IDFT. Для відновлення сигналу з частотного спектру використовується IDFT:

$$x[n] = \frac{1}{N}\sum_{n=0}^{N-1} X[k] * e^{j\frac{2\pi kn}{N}} \quad (3.11)$$

де $X[k]$ – спектральні компоненти, $x[n]$ – вхідний сигнал, j – уявна одиниця ($j^2 = -1$), N – кількість точок у сигналі.

Виходячи з цього розглянемо FFT. Нехай $x[n]$ – сигнал довжини N, де N є степенем двійки. Тоді FFT можна записати як:

$$X[k] = E[k] + e^{-j\frac{2\pi k}{N}} O[k] \quad (3.12)$$

$$X\left[k+\frac{N}{2}\right]=E[k]-e^{-j\frac{2\pi k}{N}}O[k], \quad (3.13)$$

де $E[k]$ – DFT для парних індексів, $O[k]$ – DFT для непарних індексів.

Приклад розрахунку швидкого перетворення Фур'є мовою програмування С наведено в лістингу 3.3.

Лістинг 3.3 – Функція розрахунку перетворення Фур'є.

```
void fft(Complex *x, int N) {
    if (N <= 1) return;
    Complex even[N/2], odd[N/2];
    for (int i = 0; i < N / 2; i++) {
        even[i] = x[i * 2];
        odd[i] = x[i * 2 + 1];
    }

    fft(even, N / 2);
    fft(odd, N / 2);
    for (int k = 0; k < N / 2; k++) {
        Complex t = complex_mul(complex_exp(-2.0 * M_PI * k / N), odd[k]);
        x[k] = complex_add(even[k], t);
        x[k + N / 2] = complex_sub(even[k], t);
    }
}
```

Таблиця 3.4 – Швидкість виконання прямого та зворотного перетворення Фур'є

Розмір вхідного сигналу	Пряме перетворення		Зворотне перетворення	
	Процесор			
	Ryzen 9	Snapdragon 732	Ryzen 9	Snapdragon 732
2048	0,61	0,99	0,47	1,03
4096	1,03	2,14	0,97	2,23
8192	2,03	4,67	2,34	4,92
16384	4,41	10,08	4,53	10,49

Швидке перетворення Фур'є (FFT) є потужним і ефективним інструментом для аналізу частотних компонент сигналів. Його застосування

охоплює широкий спектр галузей, включаючи обробку сигналів, аналіз зображень, цифровий зв'язок, науку про дані та багато інших. Завдяки значному зниженню обчислювальної складності, FFT став невід'ємною частиною сучасної обчислювальної техніки і наукових досліджень.

3.3 Вейвлет перетворення

В цих експериментах, для розрахунку Вейвлет перетворення використовуються процесори AMD Ryzen 9 7945HX (з операційною системою Windows 11) та Qualcomm Snapdragon 732G (з операційною системою Android). Вхідний сигнал для Вейвлет перетворення(прямого та зворотнього) заповнюється довільними числами з рухомою комою одинарної точності формату float. Заміри швидкості виконання проводяться в мілісекундах (мс).

3.3.1 Алгоритм Гаара

Вейвлет-перетворення Гаара (Haar Wavelet Transform) – це одне з найпростіших і найдавніших дискретних вейвлет-перетворень, запропоноване Альфредом Гааром у 1910 році. Воно широко використовується завдяки своїй простоті, особливо для обробки сигналів та зображень.

Вейвлет Гаара – це функція, яка розкладає сигнал на низькочастотну та високочастотну складові. Функція використовує прості арифметичні операції для поділу сигналу на:

– апроксимацію (низькочастотні компоненти) – грубий вигляд сигналу;

– деталізацію (високочастотні компоненти) – швидкі зміни сигналу.

Для реалізації дискретного вейвлет перетворення Гаара використовується простий підхід [8]:

1. Розділити сигнал на парні та непарні індекси.
2. Обчислити середнє для низькочастотних елементів і різницю для

високочастотних компонентів:

$$\text{Апрокимація } (L) = \frac{x[2i]+x[2i+1]}{2} \quad (3.14)$$

$$\text{Деталізація } (H) = \frac{x[2i]-x[2i+1]}{2} \quad (3.15)$$

Цей процес виконується рекурсивно на апроксимації зменшуючи оброблювану частину сигналу в два рази кожну ітерацію доки не залишиться один елемент.

Зворотне перетворення виконується за наступними кроками:

1. Відновлення значень із апроксимації та деталізації. Початкові елементи сигналу обчислюються за формулами:

$$x[2i] = L[i] + H[i] \quad (3.16)$$

$$x[2i+1] = L[i] - H[i] \quad (3.17)$$

2. Рекурсивне відновлення. Якщо вейвлет-перетворення було виконано рекурсивно (на декількох рівнях), то відновлення починається з найвищого рівня (де залишилась одна апроксимація) і поступово відновлюється початковий сигнал.

Приклад розрахунку вейвлет перетворення Гаара мовою програмування С наведено в лістингу 3.4.

Лістинг 3.4 – Функція розрахунку вейвлет перетворення Гаара.

```
void simple_wavelet(float incArray[SIGNAL_LENGTH], int n, int isign, float
tempArray[SIGNAL_LENGTH])
{
   if(n < 2 || n > SIGNAL_LENGTH) return;
   int nh, i, j, next;
   nh = n >> 1;
   if (isign >= 0) {
      for (i = 0, j = 0; j < n - 1; j += 2, i++) {
         tempArray[i] = (incArray[j] + incArray[j + 1]) * 0.5f;
```

```
            tempArray[i + nh] = (incArray[j] - incArray[j + 1]) * 0.5f;
        }
        next = n / 2;
    }
    else {
        for (i = 0; i < nh; i++) {
            tempArray[2 * i] = incArray[i] + incArray[i + nh];
            tempArray[2 * i + 1] = incArray[i] - incArray[i + nh];
        }
        next = n * 2;
    }
    for (i = 0; i < n; i++) {
        incArray[i] = tempArray[i];
    }
    if(n >= 2 || n <= SIGNAL_LENGTH) simple_wavelet(incArray, next, isign, tempArray);
}
```

Таблиця 3.5 – Швидкість виконання прямого та зворотного вейвлет перетворення Гаара

Розмір вхідного сигналу	Пряме перетворення		Зворотне перетворення	
	Процесор			
	Ryzen 9	Snapdragon 732	Ryzen 9	Snapdragon 732
16384	0,09	0,2	0,09	0,2
32768	0,17	0,4	0,16	0,38
65536	0,32	0,82	0,31	0,77
131072	0,61	1,75	0,58	1,63

Вейвлет перетворення Гаара широко застосовується для стиснення даних, обробки сигналів і аналізу зображень. Наприклад, воно дозволяє локалізувати високочастотні компоненти сигналу, що дає змогу відкидати малозначущі деталізації без втрати важливої інформації. У цифровій обробці зображень цей підхід використовується для стиснення формату JPEG2000. Щодо переваг, перетворення Гаара вирізняється простою реалізацією, низькими обчислювальними витратами та здатністю локалізувати сигнал у часі й частоті. Водночас є певні недоліки. Базова функція Гаара не є гладкою, що обмежує її застосування для складних сигналів, а також алгоритм може втрачати точність при стисненні даних.

Крім того, цей метод часто слугує основою для побудови складніших вейвлетів, які враховують специфіку аналізованих сигналів. Завдяки цьому вейвлет Гаара є важливим інструментом для навчання основ вейвлет-аналізу. Також перетворення Гаара може використовуватись в реальному часі через низькі вимоги до апаратних ресурсів.

3.3.2 Алгоритм Добеші

Чотирьох коефіцієнтне вейвлет перетворення Добеші (DAUB4) – це популярний вейвлет, розроблений Інґрід Добеші, який є одним із класичних ортогональних дискретних вейвлетів. DAUB4 часто використовується в обробці сигналів і зображень завдяки своїй простоті, ефективності та здатності виявляти локалізовані характеристики сигналу [8].

Алгоритм починається з розрахунку коефіцієнтів фільтра. Для DAUB4 фільтри низьких частот (h) і високих частот (g) мають такі коефіцієнти:

$$h_0 = \frac{1+\sqrt{3}}{4\sqrt{2}}, h_1 = \frac{3+\sqrt{3}}{4\sqrt{2}}, h_2 = \frac{3-\sqrt{3}}{4\sqrt{2}}, h_3 = \frac{1-\sqrt{3}}{4\sqrt{2}} \quad (3.18)$$

$$g_0 = h_3, g_1 = -h_2, g_2 = h_1, g_3 = -h_0 \quad (3.19)$$

DAUB4 базується на дискретному вейвлет перетворенні (DWT), яке використовує згортку сигналу з фільтрами h і g, а потім виконує операцію субдискретизації.

Для прямого перетворення розраховуємо:

– апроксимацію (коефіцієнти низьких частот):

$$a[n] = \sum_k h[k]x[2n-k] \quad (3.20)$$

– деталі (коефіцієнти високих частот):

$$в[n] = \sum_k g[k]x[2n-k] \quad (3.21)$$

Щоб провести зворотне перетворення треба провести згортку та додавання апроксимацій і деталей які були отримані в прямому перетворенні.

Приклад розрахунку вейвлет перетворення Добеші DAUB4 мовою програмування С наведено в лістингу 3.5.

Лістинг 3.5 – Функція розрахунку вейвлет перетворення Добеші DAUB4.

```
void daub4(float dataInc[SIGNAL_LENGTH], int n, int isign, float wksp[SIGNAL_LENGTH]){
   int nh, nh1, i, j;
   if (n < 4) return;
   nh1 = (nh = n >> 1) + 1;
   if (isign >= 0) { //Apply filter.
      for (i = 1, j = 1; j <= n - 3; j += 2, i++) {
       wksp[i] = C0 * dataInc[j] + C1 * dataInc[j + 1] + C2 * dataInc [j + 2] + C3 * dataInc[j + 3];
       wksp[i + nh] = C3 * dataInc[j] - C2 * dataInc[j + 1] + C1 * dataInc[j + 2] - C0 * dataInc[j + 3];
      }
      wksp[i] = C0 * dataInc[n - 1] + C1 * dataInc[n] + C2 * dataInc[1] + C3 * dataInc[2];
      wksp[i + nh] = C3 * dataInc[n - 1] - C2 * dataInc[n] + C1 * dataInc[1] - C0 * dataInc[2];
   }
   else {
      wksp[1] = C2 * dataInc[nh] + C1 * dataInc[n] + C0 * dataInc[1] + C3 * dataInc[nh1];
      wksp[2] = C3 * dataInc[nh] - C0 * dataInc[n] + C1 * dataInc[1] - C2 * dataInc[nh1];
      for (i = 1, j = 3; i < nh; i++) {
         wksp[j++] = C2 * dataInc[i] + C1 * dataInc[i + nh] + C0 * dataInc[i + 1] + C3 * dataInc[i +
nh1];
         wksp[j++] = C3 * dataInc[i] - C0 * dataInc[i + nh] + C1 * dataInc[i + 1] - C2 * dataInc[i +
nh1];
      }
   }
   for (i = 1; i <= n; i++) dataInc[i] = wksp[i];
}
```

Таблиця 3.6 – Швидкість виконання прямого та зворотного вейвлет перетворення Добеші

Розмір вхідного сигналу	Пряме перетворення		Зворотне перетворення	
	Процесор			
	Ryzen 9	Snapdragon 732	Ryzen 9	Snapdragon 732
16384	0,06	0,14	0,06	0,14
32768	0,12	0,28	0,12	0,27
65536	0,17	0,56	0,19	0,55
131072	0,43	1,18	0,43	1,18

DAUB4 характеризується високою обчислювальною ефективністю завдяки коротким фільтрам, що робить його придатним для роботи в реальному часі. Він також забезпечує гарну якість стиснення, особливо в таких застосунках, як JPEG 2000. До його переваг належать простота реалізації як програмно, так і апаратно. Водночас DAUB4 має певні недоліки. Він забезпечує лише одну безперервну похідну, що обмежує його гладкість, тому для аналізу сигналів, які потребують більш гладких функцій, використовують інші вейвлети Добеші, такі як DAUB6 або DAUB8. Крім того, його асиметрія може іноді призводити до появи артефактів у сигналах.

DAUB4 широко застосовується в обробці звуку, аналізі електрокардіограм, стисканні зображень, а також у чисельних методах для розв'язання диференціальних рівнянь. Його популярність пояснюється поєднанням простоти реалізації, ефективності та здатності працювати з широким спектром використань.

4 ОГЛЯД СПОСОБІВ ОПТИМІЗАЦІЇ

4.1 Оптимізація доступу до пам’яті

Оптимізація доступу до пам’яті є ключовим аспектом у підвищенні ефективності програм, особливо в системах з обмеженими ресурсами або високими вимогами до швидкості обчислень. У мові програмування C, яка дозволяє безпосередньо працювати з пам’яттю через покажчики та масиви, ефективне управління пам’яттю стає критично важливим [9].

Доступ до пам’яті має значний вплив на продуктивність через особливості сучасної ієрархії пам’яті. Центральний процесор (CPU) працює значно швидше, ніж може забезпечити доступ до оперативної пам’яті. Тому оптимізація використання кешу стає важливою складовою роботи з пам’яттю. Оптимізації часто спрямовані на те, щоб максимально використовувати дані, які вже завантажені в кеш. У цьому контексті важливими є локальність даних, яка буває просторовою і часовою. Просторова локальність означає, що дані, які розташовані близько один до одного в пам’яті, ймовірно, будуть використані разом, тоді як часова локальність припускає, що ті самі дані можуть використовуватися кілька разів протягом короткого проміжку часу.

Оптимізація доступу до пам’яті може включати кілька ключових підходів. Один із них – це правильна організація даних. Наприклад, для роботи з матрицями доцільно використовувати послідовну пам’ять (row-major order), що відповідає способу їхнього зберігання в C. Це дозволяє уникати значних витрат часу на кеш-промахи під час ітерації по рядках або стовпцях. Переміщення по пам’яті в неправильному порядку, наприклад, по стовпцях замість рядків, може спричинити значне уповільнення.

Іншим важливим аспектом є зменшення кількості звернень до пам’яті. Наприклад, можна зберігати проміжні результати у змінних, щоб уникнути повторного доступу до одного і того ж елемента масиву. Це особливо корисно в обчислювально інтенсивних алгоритмах, таких як множення матриць або

обробка сигналів.

Важливим прийомом є також вирівнювання даних. Сучасні процесори можуть працювати з пам’яттю ефективніше, якщо дані вирівняні за межами кеш-лінійки. Невирівняний доступ до пам’яті може призвести до збільшення часу виконання інструкцій.

Розглянемо виконання оптимізації на прикладі традиційного алгоритму множення матриць. Стандартне виконання якого, зазвичай базується на постійному доступі до елементів масиву результуючої матриці, яка при великих розмірах матриць зберігається в кеші (лістинг 3.1). Щоб оптимізувати доступ до пам’яті можна створити змінну в тілі циклу, проводити розрахунки в ній, і вже після записати отримані розрахунки в результуючу матрицю. Приклад виконання такої оптимізації наведено в лістингу 4.1.

Лістинг 4.1 – Оптимізована функція розрахунку добутку матриць

```
void base_multiply_opti_memory(int A[MATRIX_SIZE][MATRIX_SIZE], int
B[MATRIX_SIZE][MATRIX_SIZE], int C[MATRIX_SIZE][MATRIX_SIZE]) {
   for (int i = 0; i < MATRIX_SIZE; i++) {
      for (int j = 0; j < MATRIX_SIZE; j++) {
         int sum = 0;
         for (int k = 0; k < MATRIX_SIZE; k++) {
            sum += A[i][k] * B[k][j];
         }
         C[i][j] = sum;
      }
   }
}
```

Ще однією поширеною технікою є використання блокового оброблення. У таких алгоритмах, як множення матриць, пам’ять може бути розділена на менші блоки, які повністю завантажуються в кеш, що дозволяє зменшити кількість звернень до оперативної пам’яті. Це може бути реалізовано за допомогою вкладених циклів, які ітеративно обробляють блоки даних.

Окрім того, в оптимізації доступу до пам’яті часто застосовуються специфічні директиви компілятора, наприклад, для вирівнювання структур або передвибірки даних у кеш. У мові C можна також використовувати

асемблерні вставки для точного управління кешем, хоча це й підвищує складність коду.

Таким чином, оптимізація доступу до пам’яті є багатогранним процесом, що включає як правильну організацію даних, так і використання алгоритмічних підходів для покращення локальності та мінімізації звернень до оперативної пам’яті. Ці техніки дозволяють не лише значно підвищити продуктивність програм, але й ефективно використовувати ресурси сучасних апаратних платформ.

4.2 Розгортання циклів

Розгортання циклів (loop unrolling) – це техніка оптимізації коду, яка полягає в збільшенні кількості обчислень, виконуваних у кожній ітерації циклу, шляхом часткового або повного дублювання тіла циклу. Ця оптимізація спрямована на зменшення кількості ітерацій циклу, що дозволяє знизити витрати на обчислення умов циклу та перехід між ітераціями. Розгортання циклів може також покращити ефективність використання кешу та підвищити продуктивність за рахунок зменшення пов'язаних із циклом накладних витрат [9].

Розглянемо виконання оптимізації на прикладі традиційного алгоритму множення матриць. Зазвичай в тілі циклу проводять один розрахунок (лістинг 3.1). Щоб зробити розгортання циклу зробимо в тілі циклу ще один розрахунок, попередньо перевіривши, щоб він не вийшов за розміри матриці. Приклад виконання такої оптимізації наведено в лістингу 4.2.

Лістинг 4.2 – Оптимізована розгортанням циклу функція розрахунку добутку матриць

```
void base_multiply_loop_unroll(int A[MATRIX_SIZE][MATRIX_SIZE], int
B[MATRIX_SIZE][MATRIX_SIZE], int C[MATRIX_SIZE][MATRIX_SIZE]) {
   for (int i = 0; i < MATRIX_SIZE; i++) {
      for (int j = 0; j < MATRIX_SIZE; j++) {
         C[i][j] = 0;
         for (int k = 0; k < MATRIX_SIZE; k += 2) {
```

```
            C[i][j] += A[i][k] * B[k][j];
            if(k + 1 < MATRIX_SIZE) {
               C[i][j] += A[i][k + 1] * B[k + 1][j];
            }
         }
      }
   }
}
```

Розгортання циклів зазвичай застосовується вручну або автоматично компілятором, якщо увімкнено відповідні опції оптимізації. Розгортання циклів має кілька переваг. По-перше, воно зменшує накладні витрати на управління циклом, що особливо важливо для внутрішніх (інтенсивно використовуваних) циклів. По-друге, розгортання покращує можливості процесора щодо оптимізації інструкцій, наприклад, завдяки паралельному виконанню команд або векторизації. Це дозволяє збільшити завантаження обчислювальних блоків процесора, зменшуючи час простою.

Однак у розгортання циклів є й недоліки. Основним є збільшення розміру коду (code bloat), оскільки тіло циклу дублюється кілька разів. Це може призвести до перевантаження кешу інструкцій або погіршення продуктивності через недостатню локальність коду. Ще однією проблемою є необхідність коректної обробки залишкових ітерацій, якщо кількість елементів не кратна кількості оброблюваних за одну ітерацію. У таких випадках потрібно додатково додавати код для обробки залишкових елементів.

Таким чином, розгортання циклів є потужною технікою, яка дозволяє підвищити продуктивність шляхом зменшення накладних витрат циклів і покращення використання апаратних ресурсів процесора. Проте її ефективність залежить від особливостей програми та апаратної платформи, і вона повинна застосовуватися з урахуванням можливих недоліків.

4.3 Зменшення кількості звернень до функцій

Зменшення кількості звернень до функцій, або інлайнінг функцій

(function inlining) – це техніка оптимізації, яка полягає в заміні виклику функції безпосереднім вставленням її тіла у місце виклику. Цей підхід дозволяє уникнути накладних витрат на виклик функції, таких як збереження контексту виконання, передача параметрів і перехід до іншої частини коду. Зменшення кількості звернень до функцій може суттєво покращити продуктивність, особливо для коротких функцій, які часто викликаються у внутрішніх циклах [9].

Розглянемо виконання оптимізації на прикладі алгоритму Штрассена для множення матриць. Неоптимізована версія алгоритму має чотири додаткові функції для додавання і віднімання матриць, а також функцію для розбиття і збирання проміжних матриць (додаток Б). Щоб зробити інлайнінг зробимо реалізацію цих функцій всередині алгоритму. Також через те, що функції вносяться в основний алгоритм можна поєднати деякі сумування і віднімання, а також розподіл і збирання підматриць в один цикл, таким чином провівши аналог розгортання циклів. Приклад виконання таких оптимізацій наведено в додатку В.

Інлайнінг має кілька переваг. По-перше, він скорочує час виконання програми шляхом усунення викликів функцій. По-друге, після інлайнінгу компілятор має більше контексту для подальших оптимізацій, таких як видалення мертвого коду, розгортання циклів або оптимізація доступу до змінних.

Однак інлайнінг має і свої обмеження. Головним недоліком є збільшення розміру коду (code bloat), оскільки тіло функції дублюється у кожному місці її виклику. Це може призвести до погіршення локальності коду та перевантаження кешу інструкцій, особливо якщо інлайнуються великі функції або функції викликаються багаторазово. Крім того, зменшення кількості звернень не завжди доцільне для функцій, які рідко викликаються або мають значний розмір.

Таким чином, інлайнінг функцій є ефективною технікою оптимізації, яка дозволяє зменшити накладні витрати викликів функцій і покращити

продуктивність. Однак її слід застосовувати обережно, оцінюючи компроміс між швидкістю виконання та збільшенням розміру коду.

4.4 Тестування оптимізацій

В експериментах використовуються процесори AMD Ryzen 9 7945HX (з операційною системою Windows 11) та Apple M1 (з операційною системою MacOS). Також експеримент проводився на платформах Raspberry Pi з ядрами Cortex-A53 та Zynq 7000 (PS частина з ядрами Cortex-A9). Для множення матриць використовуються квадратні матриці розміром 128x128, які заповнюються довільними цілими числами формату int. Вхідний сигнал для Вейвлет перетворення(прямого та зворотного для вейвлетів Гаара та Добуші) довжиною 131072 одиниць, заповнюється довільними числами з рухомою комою одинарної точності формату float. Вхідний сигнал для перетворення Фур'є(прямого та зворотного) довжиною 16384 одиниць, заповнюється довільними числами з рухомою комою одинарної точності формату float. Заміри швидкості виконання проводяться в мілісекундах (мс).

Введемо умовні позначення для таблиць у цьому розділі:

Б – Базовий алгоритм.

ОДП – Оптимізація доступу до пам'яті.

РЦ – Розкриття циклів.

І – Інлайнінг (зменшення кількості звернень до функцій).

ПО – Поєднання оптимізацій.

ТМ – Традиційне множення матриць.

В – множення матриць алгоритмом Віноградa.

ВГ – Вейвлет перетворення Гаара.

ВД – Вейвлет перетворення Добеші.

ШПФ – Швидке перетворення Фур'є.

п – Пряме перетворення.

з – Зворотне перетворення.

Представлення результатів у вигляді гістограм, включає алгоритми розділені на три групи, при чому кожна група буде представлена в своєму масштабі. Група «Множення матриць» включає в себе середнє значення для алгоритмів традиційного множення і Віноградa; група «Вейвлет» включає в себе середні значення вейвлетів Гаара та Добеші їх прямі та зворотні перетворення; група «Фур'є» включає в себе середнє значення для прямого та зворотного виконання швидкого перетворення Фур'є.

Проведемо тестування оптимізацій на прикладі процесора AMD Ryzen 9.

Таблиця 4.1 – Порівняння оптимізацій на прикладі процесора AMD Ryzen 9

	Б	ОДП	РЦ	І	ПО
ТМ	4,1	2,61	4,04	-	2,36
В	3,38	2,39	3,24	-	2,34
ВГп	0,58	0,58	0,43	0,55	0,47
ВГз	0,55	0,5	0,44	0,53	0,45
ВДп	0,42	0,41	0,48	-	0,33
ВДз	0,43	0,42	0,49	-	0,35
ШПФп	3,38	3,48	3,48	1,32	1,35
ШПФз	3,49	3,89	3,44	1,21	1,21

Відобразимо результат у вигляді гістограми.

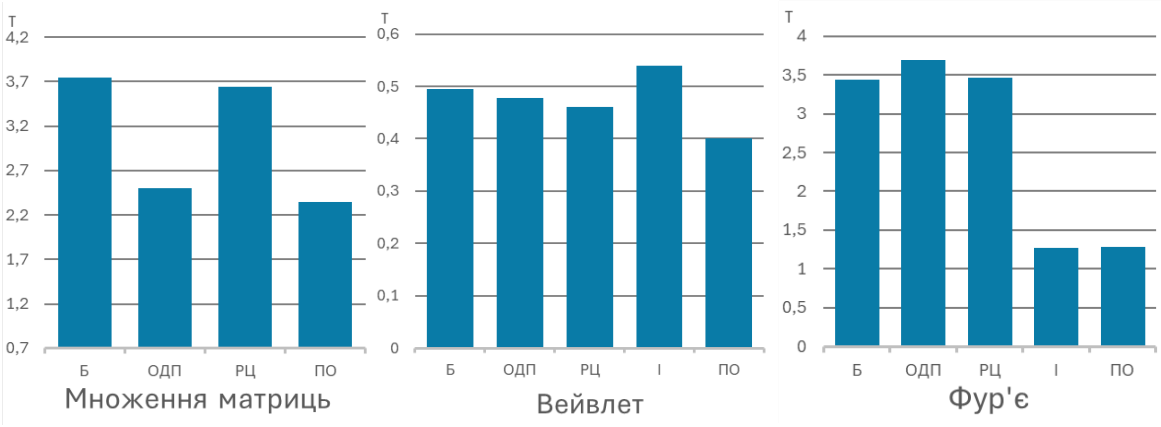

Рисунок 4.1 – Зведені результати для процесора AMD Ryzen 9 у вигляді гістограми

На платформі AMD Ryzen 9 для множення матриць помітно зменшився час виконання завдяки оптимізації доступу до пам'яті (2,61 мс для традиційного множення проти 4,1 мс базової реалізації). Алгоритм Віноградa також показує зменшення часу виконання. Вейвлети демонструють покращення часу виконання для алгоритму Добеші, особливо при поєднанні оптимізацій (0,33 мс для прямого та 0,35 мс для зворотного перетворення). Для Фур'є перетворення інлайнінг та поєднання оптимізацій істотно зменшили час виконання до 1,21 – 1,35 мс порівняно з 3,38 мс базової реалізації.

Проведемо тестування оптимізацій на прикладі процесора Apple M1.

Таблиця 4.2 – Порівняння оптимізацій на прикладі процесора Apple M1

	Б	ОДП	РЦ	І	ПО
ТМ	0,81	0,79	0,81	-	0,79
В	0,76	0,77	0,74	-	0,72
ВГп	0,06	0,07	0,1	0,06	0,09
ВГз	0,06	0,06	0,05	0,06	0,05
ВДп	0,1	0,08	0,19	-	0,1
ВДз	0,06	0,07	0,12	-	0,08
ШПФп	1,2	2,02	1,44	0,53	0,43
ШПФз	1,35	2,26	2,09	0,61	0,45

Відобразимо результат у вигляді гістограми.

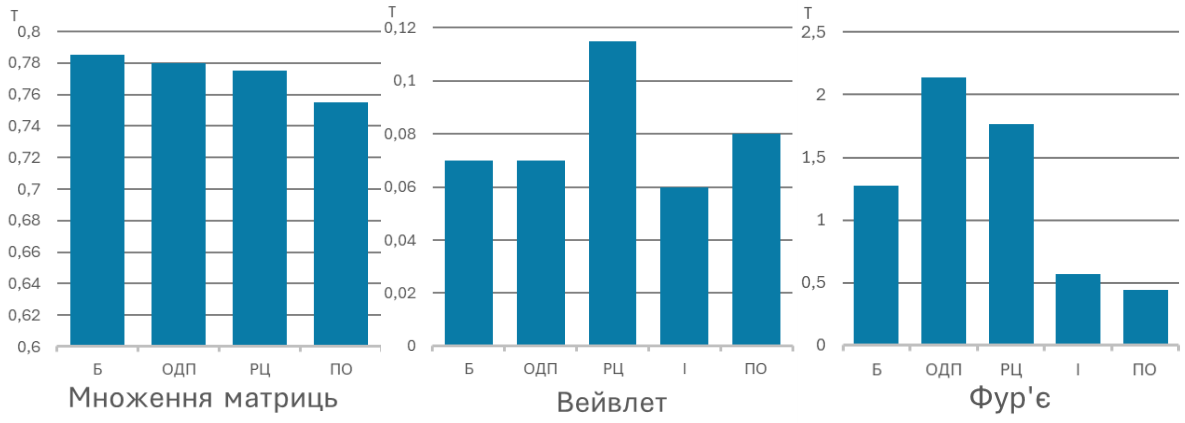

Рисунок 4.2 – Зведені результати для процесора Apple M1 у вигляді гістограми

На платформі Apple M1 всі методи оптимізації для множення матриць показують мінімальні покращення часу, оскільки M1 ефективно працює з кешем та доступом до пам'яті. Вейвлети демонструють покращення часу для прямого Вейвлета Добеші при оптимізації доступу до пам'яті (0,076 мс порівняно з 0,096 мс). Для Фур’є перетворення найбільше покращення часу спостерігається при поєднанні оптимізацій для зворотного перетворення (0,447 мс проти 1,347 мс оригіналу).

Проведемо тестування оптимізацій на прикладі платформи Raspberry Pi з ядрами Cortex A53.

Таблиця 4.3 – Порівняння оптимізацій на прикладі платформи Raspberry Pi

	Б	ОДП	РЦ	І	ПО
ТМ	31,74	25	30,92	-	24,33
В	25,82	22,56	26,94	-	22,5
ВГп	1,66	1,5	1,39	1,64	1,2
ВГз	1,24	1,24	1,46	1,24	1,22
ВДп	2,24	1,63	2,77	-	1,52
ВДз	1,35	1,36	2,25	-	1,58
ШПФп	30,55	31,16	30,9	3,86	3,96
ШПФз	31,03	31,63	31,03	4,1	4,19

Відобразимо результат у вигляді гістограми.

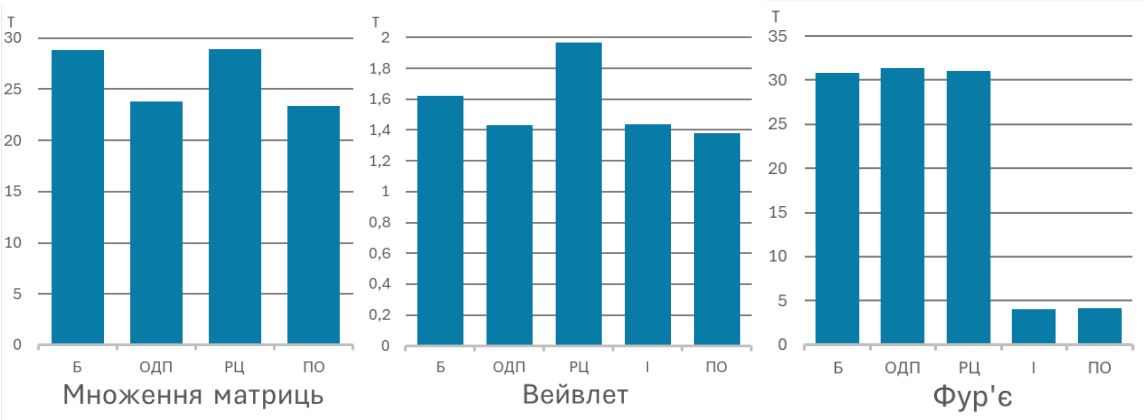

Рисунок 4.3 – Зведені результати для платформи Raspberry Pi у вигляді гістограми

На платформі Raspberry Pi поєднання оптимізацій показало найбільший ефект для традиційного множення матриць (24,33 мс проти 31,74 мс). Алгоритм Віноградa також отримав покращення часу виконання (22,5 мс). Найбільше зменшення часу для прямого Вейвлета Гаара спостерігається при поєднанні оптимізацій (1,196 мс). Для Фур'є перетворення оптимізації майже не вплинули на час виконання, але інлайнінг швидший завдяки уникненню викликів функцій, що знижує накладні витрати, а обчислення виконуються безпосередньо. Оптимізація використовує ітеративний підхід замість рекурсії, що ефективніше для платформи з обмеженим стеком, як Raspberry Pi.

Проведемо тестування оптимізацій на прикладі платформи Zynq-7000.

Таблиця 4.4 – Порівняння оптимізацій на прикладі платформи Zynq-7000

	Б	ОДП	РЦ	І	ПО
ТМ	45,52	40,96	44,03	-	36,74
В	38,69	34,24	35,25	-	31,45
ВГп	9	8,17	9	9	6,78
ВГз	6,8	6,8	6,8	6,81	8,11
ВДп	9,6	6,25	9,87	-	5,02
ВДз	9,41	6,17	9,68	-	6,01
ШПФп	54,45	57	53,49	9,48	8,91
ШПФз	55,94	58,59	54,93	10,36	9,8

Відобразимо результат у вигляді гістограми.

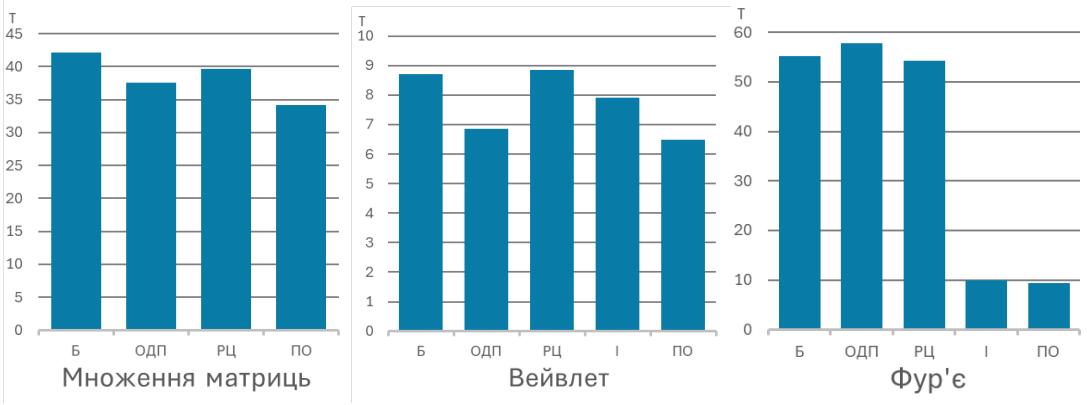

Рисунок 4.4 – Зведені результати для платформи Zynq-7000 у вигляді гістограми

Оптимізації, проведені на платформі Zynq-7000, показали різний ступінь ефективності залежно від типу алгоритмів. Оптимізація доступу до пам’яті виявилася однією з найефективніших, особливо для алгоритмів множення матриць і вейвлет перетворення Добеші. Для традиційного множення матриць і алгоритму Віноградa вона скоротила час виконання на 10 – 12%, що вказує на значний вплив на зменшення затримок при роботі з пам’яттю. У випадку вейвлет перетворення Добеші покращення було ще більш вираженим, досягаючи 35% для прямого перетворення.

Розкриття циклів, хоча і є популярною технікою оптимізації, не завжди виявлялося ефективним. Для множення матриць покращення часу було мінімальним, а для деяких алгоритмів вейвлет перетворення навіть спостерігалося незначне збільшення часу виконання. Ця оптимізація виявилася непередбачуваною, і її ефективність залежить від особливостей конкретного алгоритму.

Зменшення кількості звернень до функцій продемонструвало себе ефективним лише для алгоритмів швидкого перетворення Фур’є, де воно дозволило значно зменшити час виконання, скорочуючи виклики функцій і накладні витрати, пов’язані з ними. Це стало можливим завдяки тому, що обчислення виконувалися безпосередньо, що особливо важливо для алгоритмів, які часто звертаються до дрібних функцій.

Поєднання оптимізацій виявилося найбільш універсальним і ефективним підходом. Воно забезпечило значне скорочення часу виконання для всіх алгоритмів. Наприклад, у випадку множення матриць час скоротився на 19%, а для вейвлет і Фур’є перетворень покращення досягло 25 – 83%. Це підкреслює, що комбінування різних методів оптимізації дозволяє досягати максимального приросту продуктивності.

Загалом результати свідчать, що оптимізація доступу до пам’яті і поєднання оптимізацій є найефективнішими підходами, які забезпечують стабільний приріст продуктивності. Інші оптимізації, такі як розкриття циклів

та інлайнінг, можуть бути корисними в залежності від специфіки алгоритму, але не є універсальними рішеннями.

Тепер детальніше розглянемо по алгоритмах. Алгоритм традиційного множення на платформі Zynq-7000 в базовій реалізації виконується за 45,52 мс. Основною причиною затримки є інтенсивне звернення до оперативної пам’яті, яке на платформі з обмеженим кешем значно впливає на продуктивність. Оптимізація доступу до пам’яті зменшила час до 40,96 мс, завдяки більш ефективному використанню кешу даних. Розкриття циклів показало час 44,03 мс, оскільки збільшений обсяг інструкцій перевантажує кеш інструкцій, що нівелює переваги оптимізації. Найкращий результат було досягнуто при поєднанні оптимізацій – 36,74 мс, де оптимізація доступу до пам’яті доповнюється іншими техніками, що зменшують кількість звернень до пам’яті та покращують використання кешу даних.

Алгоритм Віноградa в базі виконується за 38,69 мс, де основна затримка пов’язана з доступом до пам’яті. Оптимізація доступу до пам’яті зменшила час до 34,24 мс завдяки кращому впорядкуванню інформації в кеші даних. Розкриття циклів покращило час до 35,25 мс, однак додаткові обчислення обмежили ефективність цієї оптимізації. Поєднання оптимізацій забезпечило найкращий результат – 31,45 мс, завдяки спільній дії оптимізацій, які мінімізують обсяг доступу до пам’яті та збільшують ефективність використання кешу даних.

Пряме перетворення вейвлета Гаара в базовій реалізації виконується за 9 мс. Цей алгоритм менш чутливий до оптимізацій через низьку обчислювальну складність. Оптимізація доступу до пам’яті покращила час до 8,17 мс, зменшуючи кількість звернень до оперативної пам’яті завдяки ефективнішому використанню кешу даних. Розкриття циклів погіршило час до 9 мс через збільшення обсягу інструкцій, що створило додаткове навантаження на кеш інструкцій. Інлайнінг дав схожий результат – 9 мс, оскільки функціональні виклики не були критичною частиною затримок.

Найкращий результат забезпечило поєднання оптимізацій – 6,78 мс, де вдалося зменшити загальне навантаження на пам’ять і кеш.

Зворотне перетворення Вейвлета Гаара має базовий час виконання 6,8 мс, який мало змінюється під дією більшості оптимізацій. Оптимізація доступу до пам’яті, розкриття циклів та інлайнінг показали час 6,8 мс, що свідчить про низьку чутливість алгоритму до цих змін. Поєднання оптимізацій несподівано збільшило час до 8,11 мс, що може бути спричинено конфліктами в кеші даних або взаємодією оптимізацій, яка погіршила продуктивність.

Пряме перетворення вейвлета Добеші в базі виконується за 9,6 мс і має вищу обчислювальну складність порівняно з Гааром. Оптимізація доступу до пам’яті значно зменшила час до 6,25 мс, забезпечуючи ефективну роботу з кешем даних. Розкриття циклів погіршило час до 9,88 мс через збільшення обсягу інструкцій і навантаження на кеш інструкцій. Поєднання оптимізацій забезпечило найкращий результат – 5,02 мс, завдяки синергії оптимізацій, які мінімізували звернення до пам’яті та покращили роботу з кешем даних.

Зворотне перетворення вейвлета Добеші в базовій реалізації виконується за 9,41 мс. Оптимізація доступу до пам’яті зменшила час до 6,17 мс, демонструючи сильний вплив ефективного управління кешем даних. Розкриття циклів збільшило час до 9,68 мс, оскільки перевантаження кешу інструкцій нівелює вигоди оптимізації. Найкращий результат досягнуто при поєднанні оптимізацій – 6,01 мс, завдяки взаємодії різних підходів, які мінімізують обчислювальне навантаження.

Пряме перетворення Фур’є в оригінальній реалізації виконується за 54,45 мс і є ресурсозатратним через велику кількість обчислень. Оптимізація доступу до пам’яті погіршила час до 57 мс, ймовірно, через збільшення кількості непрямих звернень до пам’яті, що перевантажило кеш даних. Розкриття циклів зменшило час до 53,49 мс, оскільки платформа змогла ефективно обробляти більші обсяги обчислень без перевантаження кешу інструкцій. Інлайнінг значно зменшив час до 9,48 мс, а поєднання оптимізацій

забезпечило найкращий результат – 8,91 мс, завдяки мінімізації викликів функцій і кращій роботі з кешем даних.

Зворотне перетворення Фур’є має базовий час виконання 55,94 мс. Як і у випадку прямого перетворення, оптимізація доступу до пам’яті погіршила час до 58,59 мс через подібні причини, пов’язані з перевантаженням кешу даних. Розкриття циклів дало покращення до 54,93 мс, що пов’язано з ефективнішою обробкою обчислень без перевантаження кешу інструкцій. Інлайнінг зменшив час до 10,36 мс, а поєднання оптимізацій забезпечило найкращий результат – 9,8 мс, завдяки оптимізації викликів функцій та зниженню обчислювальних витрат на кеш даних.

Відобразимо приріст швидкодії від оптимізацій у відсотках у вигляді гістограми

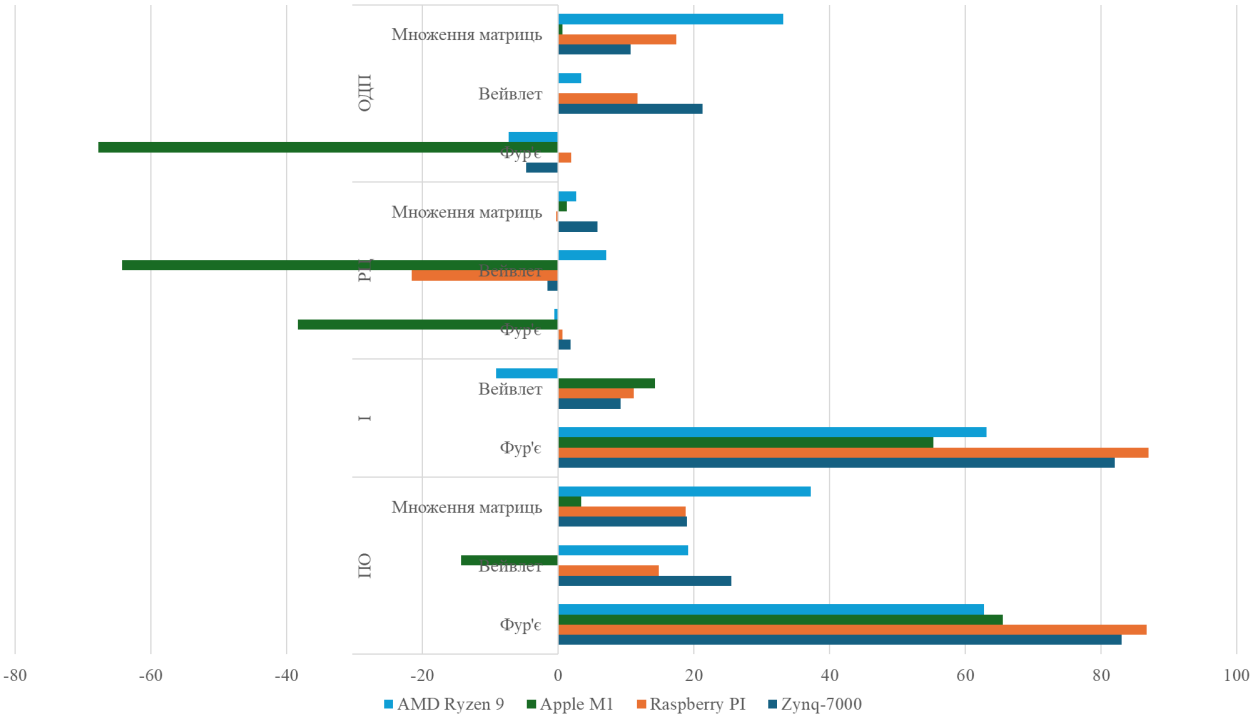

Рисунок 4.5 – Зведені результати ефективності оптимізацій у відсотках у вигляді гістограми

Загалом оптимізація доступу до пам'яті демонструє найбільше покращення для множення матриць, особливо на слабших процесорах, таких як Raspberry Pi та Zynq 7000. Розкриття циклів має непередбачуваний ефект залежно від платформи та алгоритму, і у випадку вейвлет і Фур’є перетворень на Raspberry Pi та Zynq ефективність не завжди підвищується. Інлайнінг найбільш ефективний для Фур’є перетворення, особливо на потужних процесорах AMD Ryzen 9 та Apple M1. Поєднання оптимізацій майже завжди забезпечує найкращий результат порівняно з іншими техніками оптимізації.

ВИСНОВКИ

У процесі виконання магістерської роботи було вирішено низку теоретичних і практичних задач, спрямованих на дослідження моделей оптимізації коду у середовищі високорівневого синтезу (HLS) для платформ System-on-Chip (SoC). Основною платформою для тестування та аналізу обрано Zynq-7000 від Xilinx, яка забезпечує поєднання потужних ARM-процесорів та програмованої логіки FPGA.

Проведений аналіз архітектури Zynq-7000 дозволив визначити ключові особливості взаємодії між процесорною системою (PS) та програмованою логікою (PL). Було розглянуто різні види інтерфейсів, зокрема AXI, які забезпечують ефективну передачу даних між компонентами SoC. Особливу увагу приділено управлінню пам’яттю, оптимізації кешування та зменшенню затримок під час обміну даними.

У межах роботи було детально вивчено теоретичні основи високорівневого синтезу, його переваги у порівнянні з традиційними методами проєктування апаратури, а також можливості використання мов програмування високого рівня, таких як C/C++, для створення апаратних прискорювачів. Проведено порівняння існуючих підходів до оптимізації коду, зокрема розгортання циклів, інлайнінг та оптимізації доступу до пам’яті.

На основі експериментального дослідження було розроблено, реалізовано та протестовано кілька моделей оптимізації, які спрямовані на покращення продуктивності алгоритмів для обчислювально-інтенсивних задач. Зокрема, було оптимізовано такі алгоритми, як множення матриць, перетворення Фур’є та вейвлет-перетворення. Отримані результати показали суттєве підвищення швидкості виконання алгоритмів завдяки використанню апаратних прискорювачів та зниженню кількості звернень до пам’яті.

Детальний аналіз результатів продемонстрував, що застосування високорівневого синтезу дозволяє не лише зменшити час розробки, але й

досягти значного покращення продуктивності системи.

Під час роботи було використано інструменти Xilinx Vivado та Vitis, які забезпечили інтеграцію високорівневого коду з апаратною частиною платформи. Це дозволило значно спростити процес розробки та тестування оптимізаційних моделей. Особливу увагу приділено аналізу продуктивності та складності розробки, що дозволило забезпечити збалансований підхід до оптимізації.

Результати роботи можуть бути використані для створення високопродуктивних систем у галузях, що потребують інтенсивної обробки даних, таких як обробка сигналів, аналіз зображень, телекомунікації та мультимедіа. Отримані рекомендації щодо оптимізації коду та використання апаратних прискорювачів можуть бути корисними для подальшого розвитку методологій високорівневого синтезу.

Таким чином, проведене дослідження підтвердило ефективність використання високорівневого синтезу у розробці систем SoC, забезпечило розробку та тестування оптимізаційних моделей, а також надало ґрунтовні рекомендації для майбутніх розробок у цій галузі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Crockett L. H., Elliot R. A., Enderwitz M. A. The zynq book: embedded processing with the ARM cortex-a9 on the xilinx zynq-7000 all programmable SoC. Strathclyde Academic Media, 2016. 468 с.

2. Карманський Б. Ю. Платформа Xilinx Zynq / Б. Ю. Карманський // Інформаційні радіотехнології та технічний захист інформації : Тези доп. 28-ого міжнародного молодіжного форуму конф., Харків, 16 квіт. 2024 р. – Харків, 2024. – С. 596-597.

3. Design and self-diagnostics of cyberphysical control devices on SOC platform / A. Shkil та ін. *Innovative technologies and scientific solutions for industries*. 2023. № 4(26). С. 122–132. URL: https://doi.org/10.30837/itssi.2023.26.122 (дата звернення: 17.05.2024).

4. Fingeroff M. High-Level synthesis blue book. Xlibris Corporation, 2010.

5. Cormen T. H., Stein C., Rivest R. L. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. 1332 с.

6. Schafer R., Buck J. R., Oppenheim A. V. Discrete-time signal processing. 2-ге вид. Englewood Cliffs, New Jersey: Prentice Hall, 1999. 870 с.

7. Burrus C. S. Fast fourier transforms. Lulu.com, 2012. 256 с. URL: https://archive.org/details/fast-fourier-transforms/page/n97/mode/2up (дата звернення: 19.05.2024).

8. Daubechies I. Ten lectures on wavelets. Philadelphia, Pa : Society for Industrial and Applied Mathematics, 1992. 357 с.

9. Seacord R. C. Effective C: An Introduction to Professional C Programming. No Starch Press, 2020. 272 с.