

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка

до дипломного проекту (роботи)

магістра

(ступінь вищої освіти)

на тему FRGA СИСТЕМА ДЛЯ ХЕШУВАННЯ БЛОКЧЕЙНУ

Виконав: студент 2 курсу, групи КНТ-612м
спеціальності _____

123 Комп'ютерна інженерія

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Спеціалізовані комп'ютерні системи

ПИЛИПЕНКО П. А.

(ПРИЗВИЩЕ та ініціали)

Керівник ГРУШКО С.С.

(ПРИЗВИЩЕ та ініціали)

Рецензент СТЕПАНЕНКО О.О.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій
 Кафедра «Комп'ютерні системи та мережі»
 Ступінь вищої освіти магістерський
 Спеціальність 123 Комп'ютерна інженерія
 (код і найменування)
 Освітня програма (спеціалізація) «Спеціалізовані комп'ютерні системи»
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Зав. кафедри Кудерметов Р.К.

“24” жовтня 2023 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

ПИЛИПЕНКА Павла Анатолійовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) FPGA система для хешування блокчейну
 керівник проєкту (роботи) к. т. н., доцент, Грушко Світлана Сергіївна
 (науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)
 затверджені наказом вищого навчального закладу від “24” жовтня 2023 року № 400
2. Строк подання студентом проєкту (роботи) 10 грудня 2023 року
3. Вихідні дані до проєкту (роботи) алгоритм хешування функцій Poseidon, наявні апаратні платформи, підвищення швидкості роботи наявного алгоритму
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Теоретичний огляд;
2) Розробка архітектури системи;
3) Розробка FPGA системи для хешування блокчейну
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

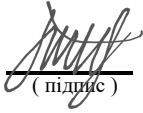
6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	ГРУШКО С. С., доцент		
нормоконтроль	ЩЕРБАК Н.В., ст. викл.		

7. Дата видачі завдання 01.10.2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз технології блокчейну, огляд апаратних платформ і алгоритмів хешування функцій	10.10.2023 р.	
2	Розробка архітектури системи	15.10.2023 р.	
3	Налагодження програмного забезпечення	20.10.2023 р.	
4	Реалізація архітектури системи	05.10.2023 р.	
5	Емуляція апаратного забезпечення	20.11.2023 р.	
6	Оформлення отриманих результатів у ПЗ	25.11.2023 р.	
7	Оформлення графічного матеріалу	01.12.2023 р.	
8	Оформлення допоміжного матеріалу	10.12.2023 р.	

Студент  Павло ПИЛИПЕНКО
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник проєкту (роботи)  Світлана ГРУШКО
(підпис) (ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

ПЗ: 92 сторінки, 18 рисунків, 2 таблиці, 44 джерела

FPGA, БЛОКЧЕЙН, ДЕЦЕНТРАЛІЗАЦІЯ, МАЙНІНГ, ХЕШУВАННЯ

Об'єкт розробки – система для хешування у блокчейні.

Предмет розробки - FPGA система для ефективного хешування в контексті блокчейн технологій.

Метою магістерської роботи є створення високоефективної і надійної системи на базі FPGA для хешування блокчейну з метою забезпечення безпеки, конфіденційності і цілісності даних в блокчейн мережах.

В першому розділі підкреслюється значущість технології блокчейн у сучасному світі, з акцентом на важливість хешування для забезпечення безпеки, конфіденційності та надійності системи.

В другому розділі описується використання AMD Varium C1100 і AMD Ryzen 3700x, розглядається Ubuntu 20.04, AMD Vitis Unified Software Platform, і AMD Xilinx AMD-Xilinx Run Time (XRT) для Ubuntu 20.04, підсумовується аналіз апаратного та програмного забезпечення для системи.

В третьому розділі визначаються ключові вимоги до системи хешування, описується процес створення дизайну, розглядається процес реалізації запропонованої архітектури, описується процес емуляції програмного і апаратного забезпечення, а також аналіз та редагування та синтез кернела.

Подаються висновки щодо розробки FPGA системи для хешування блокчейну.

ABSTRACT

EN: 92 pages, 18 figure, 2 tables, 44 sources

FPGA, BLOCKCHAIN, DECENTRALIZATION, MINING, HASHING

Object of development – a system for hashing in blockchain.

Subject of development - FPGA system for efficient hashing in the context of blockchain technologies.

The purpose of the master's thesis is to create a highly efficient and reliable FPGA-based system for blockchain hashing with the aim of ensuring the security, confidentiality, and integrity of data in blockchain networks.

The first chapter emphasizes the significance of blockchain technology in the modern world, with a focus on the importance of hashing for ensuring the security, confidentiality, and reliability of the system.

The second chapter describes the use of AMD Varium C1100 and AMD Ryzen 3700x, considers Ubuntu 20.04, AMD Vitis Unified Software Platform, and AMD Xilinx AMD-Xilinx Run Time (XRT) for Ubuntu 20.04, and summarizes the analysis of hardware and software for the system.

In the third chapter, the key requirements for the hashing system are defined, the design process is described, the implementation process of the proposed architecture is considered, the emulation process of software and hardware is described, as well as the analysis, editing, and synthesis of the kernel.

Conclusions are presented regarding the development of an FPGA system for blockchain hashing.

ЗМІСТ

Вступ.....	8
1 Теоретичний огляд	10
1.1 Огляд технології блокчейну	10
1.2 Апаратні платформи для хешування.....	16
1.2.1 Огляд апаратних платформ	16
1.2.2 Графічні процесори.....	16
1.2.3 Спеціалізовані ASIC-пристрої	18
1.2.4 FPGA.....	20
1.3 Алгоритми хешування функцій	24
1.3.1 SHA-256 (Secure Hash Algorithm 256-bit)	24
1.3.2 Кецсак	25
1.3.3 Ripemd-160	26
1.3.4 Blake2.....	27
1.3.5 Scrypt.....	29
1.3.6 Ethash	30
1.3.7 X11	31
1.3.8 Poseidon	32
1.4 Висновки до розділу 1	34
2 Розробка архітектури системи	36
2.1 Апаратне забезпечення системи	36
2.1.1 AMD Varium C1100 Blockchain Accelerator Card.....	36
2.1.2 AMD Xilinx AMD Ryzen 3700x.....	39
2.2 Програмне забезпечення системи.....	41

	7
2.2.1 Ubuntu	41
2.2.2 AMD Vitis Unified Software Platform	42
2.2.3 AMD Xilinx AMD-Xilinx Run Time (XRT) for Ubuntu 20.04.....	46
2.3 Висновки до розділу 2	47
3 Розробка FPGA системи для хешування блокчейну	49
3.1 Вимоги до системи.....	49
3.2 Створення дизайну проєкту	50
3.3 Реалізація архітектури системи	53
3.4 Емуляція	59
3.4.1 Програмна емуляція.....	59
3.4.2 Емуляція апаратного забезпечення	59
3.4.3 Аналіз.....	60
3.4.4 Редагування та синтез кернела	60
3.5 Висновки до розділу 3	68
Висновки	70
Перелік джерел посилання	72
Додаток А.....	77

ВСТУП

Сучасний світ переживає період інтенсивних змін, де інновації та технологічний прогрес стають визначальними чинниками. Серед усіх новаторських рішень, технологія блокчейн безсумнівно видається однією з найбільш захоплюючих та актуальних. Вона об'єднує у собі потужну комбінацію децентралізації, безпеки та інноваційного підходу до обробки даних.

Завдяки своїй безпрецедентній можливості створювати децентралізовані системи, технологія блокчейн революціонізує різні галузі. Вона надає змогу створювати надійні та безпечні системи обліку, які звільняють користувачів від необхідності довіряти посередникам. Це особливо важливо в фінансовій сфері, де блокчейн став основою для створення криптовалют та нових способів фінансового взаємодії.

Технологія блокчейн також привнесла новий підхід до укладання угод завдяки смарт-контрактам. Ці автоматизовані угоди можуть виконуватися безпосередньо між сторонами без участі посередників. Вони знайшли застосування в багатьох галузях, зокрема в логістиці, юридичній сфері та інтернеті речей.

Блокчейн також допомагає вирішувати проблеми прозорості та трасованості. Він дозволяє детально відстежувати всі транзакції та події в реальному часі. Це особливо важливо в контексті відстеження походження товарів, боротьби з підробкою та підвищення рівня довіри споживачів.

Більше того, технологія блокчейн вже знайшла застосування в індустрії, де вона допомагає оптимізувати процеси виробництва та покращувати якість продукції.

Хешування є однією з основних складових технології блокчейн та є важливим елементом, що гарантує безпеку та надійність цієї інноваційної системи.

По-перше, хешування використовується для створення унікальних ідентифікаторів для блоків даних в ланцюгу блоків.

По-друге, хешування використовується для захисту конфіденційності даних. У технології блокчейн, особисті дані зазвичай не зберігаються у відкритому вигляді.

По-третє, хешування грає важливу роль у процесі майнінгу, де комп'ютери в мережі конкурують за право додавати нові блоки до ланцюга. Для отримання права на майнінг, вони повинні розв'язати складну математичну задачу, яка базується на хеш-кодів попереднього блоку. Цей процес називається "доказом роботи" і вимагає значної обчислювальної потужності, що робить маніпулювання блоками практично неможливим.

Отже, хешування в технології блокчейну є фундаментальним елементом, який забезпечує надійну безпеку, конфіденційність та цілісність системи. Без цього компонента блокчейн став би невиправдано вразливим перед різними загрозами, і важко уявити, як сучасний світ міг би використовувати цю технологію без такого важливого елементу, як хешування.

1 ТЕОРЕТИЧНИЙ ОГЛЯД

1.1 Огляд технології блокчейну

Блокчейн - це інноваційна технологія, яка дозволяє створювати безпечні та децентралізовані системи обліку та обміну даними. Ця технологія ґрунтується на декількох ключових принципах, що роблять її унікальною та надійною [1-4, 23, 36].

Одним з основних принципів блокчейну є відсутність централізованого управління. Замість цього, дані зберігаються на різних комп'ютерах, які називаються вузлами мережі [2]. Це означає, що немає одного основного сервера чи центральної влади, яка могла би контролювати всю мережу.

Децентралізація - це одна з головних характеристик технології блокчейн, і вона грає важливу роль у роботі цієї інноваційної системи. Суть децентралізації полягає в тому, що вона відмовляється від ідеї централізованого управління чи контролю і дає кожному учаснику мережі рівні права та можливості. Ось детальніше про децентралізацію в блокчейні [5].

У мережі блокчейну зазвичай існують тисячі і навіть мільйони вузлів, розподілених по всьому світу. Кожен з цих вузлів має однакові права та можливість брати участь у процесі обробки транзакцій та створення нових блоків. Важливо відзначити, що немає центрального сервера або організації, яка контролювала б цю мережу. Кожен вузол є рівноправним гравцем.

Вся інформація, що зберігається у блокчейні, розподілена між всіма вузлами мережі. Кожен вузол зберігає повну копію ланцюга блоків, і немає одного пункту, де може втратитися вся інформація. Це робить систему стійкою до втрат даних через збої або кібератаки.

Процес підтвердження транзакцій та створення нових блоків у блокчейні також децентралізований. Учасники мережі взаємодіють, щоб досягти консенсусу щодо стану системи. Це може включати в себе такі механізми, як Proof of Work (PoW) чи Proof of Stake (PoS), і вони всі гарантують, що жоден вузол не має переваги.

Децентралізація в блокчейні робить цю технологію відкритою, надійною та стійкою до цензури [6-7]. Кожен користувач має можливість приєднатися до мережі, брати участь у голосуванні та зберігати копію даних, що робить її відмінним інструментом для створення децентралізованих додатків та послуг.

Блокчейн складається з послідовності блоків, де кожен блок містить певну кількість транзакцій або інших даних. Кожен новий блок посиляється на попередній блок у ланцюгу, утворюючи таким чином послідовний ланцюг, що надійно захищає дані від змін.

У світі технології блокчейн ланцюг блоків (або просто "блокчейн") виступає як фундаментальна складова, яка забезпечує надійну і послідовну зберіганість інформації. Поняття ланцюга блоків може здатися складним, але в реальності воно легко пояснюється.

Ланцюг блоків представляє собою послідовну систему блоків, кожен з яких містить певну кількість інформації. Кожен новий блок у ланцюгу посиляється на попередній, утворюючи таким чином незмінний ланцюг.

Коли нова інформація або транзакція генерується у мережі блокчейн, вона не негайно додається до загального ланцюга. Замість цього, вона спершу зберігається у новому блоку. Кожен блок містить, окрім іншого, унікальний код (хеш) попереднього блоку.

Цей посиляльний код настільки складний і унікальний, що будь-яка навіть найменша зміна в даних буде негайно виявлена. Це робить блоки відмінною системою для зберігання та передачі даних, оскільки вони міцно захищені від фальсифікації.

Коли новий блок готовий, він додається до ланцюга і стає частиною загального реєстру. Інформація тепер стає доступною всім учасникам мережі, і вони можуть перевіряти та слідкувати за всіма транзакціями, які коли-небудь відбувалися в системі.

Ланцюг блоків є механізмом, що гарантує надійну і послідовну зберіганість даних, і це є однією з ключових переваг технології блокчейн. Він дозволяє створити

довіру до даних та транзакцій, забезпечуючи безпеку та прозорість у роботі системи.

Для захисту інформації в цій технології використовується криптографія. Кожен блок має унікальний хеш-код, який створюється за допомогою складних криптографічних алгоритмів. Це робить маніпулювання або фальсифікацію даних надзвичайно важким завданням.

У світі блокчейн та криптовалюти криптографічна безпека є настільки важливою, що можна сміливо сказати, що вона становить стовідсоткову гарантію захисту даних та транзакцій. Ця безпека ґрунтується на використанні сучасних криптографічних методів та алгоритмів для захисту конфіденційності, цілісності та автентичності інформації в блокчейні [8-10, 17] .

Усі дані, які зберігаються в блокчейні, можуть бути зашифровані. Це означає, що лише особи, які мають відповідний ключ, можуть отримувати доступ до інформації. Шифрування гарантує конфіденційність даних і захищає їх від несанкціонованого доступу.

Кожна транзакція в блокчейні підписується цифровим підписом, який відповідає ключу власника активу. Цей підпис гарантує, що транзакція була автентично створена власником та не була підмінена під час передачі.

Геш-коди є основною складовою частиною безпеки блокчейну. Кожен блок має унікальний геш-код, який залежить від його вмісту. Навіть найменша зміна в блоку призведе до зміни його геш-коду, що негайно виявиться при спробі зламу або фальсифікації.

Власники криптовалютних активів мають приватні ключі, які є їхнім особистим секретом. Вони використовують ці ключі для підпису та авторизації транзакцій. Публічні ключі використовуються для ідентифікації користувачів і генерації адрес, на які можна отримувати платежі.

Криптографічна безпека є фундаментальною складовою блокчейну та криптовалют, оскільки вона гарантує конфіденційність, непорушність та правдивість даних, які зберігаються в цих системах. Без цього рівня захисту, мережі блокчейн не змогли б надійно функціонувати та здобувати довіру користувачів.

Блокчейн також підтримує смарт-контракти - програми, які автоматизовано виконують угоди та умови без участі посередників. Це дозволяє сторонам автоматизовано виконувати угоди та контролювати їхнє виконання.

Смарт-контракти - це одна з найцікавіших та перспективних складових технології блокчейн. Вони представляють собою програми, які автоматизовано виконують угоди та умови, записані в блокчейні, без потреби в посередниках чи третіх сторін.

Смарт-контракти можна уявити як цифрові правила, які виконуються автоматично, якщо вони визначені умовами. Наприклад, угода про передачу власності на блокчейні може мати смарт-контракт, який автоматично передає власність однієї сторони іншій, якщо задовільняються певні умови, такі як внесення платежу або підтвердження доставки товару.

Однією з ключових переваг смарт-контрактів є їхній безпосередній характер. Вони виконують угоди точно відповідно до заданих правил, уникнувши потреби в посередниках або інших додаткових процедурах. Це робить їх ефективними та економічними засобами для автоматизації багатьох видів угод та операцій.

Крім того, смарт-контракти дозволяють зберігати дані про угоди та транзакції в блокчейні, що робить їх прозорими та піддається перевірці. Це зменшує ризик обману чи фальсифікації угод і сприяє відкритості та довірі в системі.

Завдяки смарт-контрактам, блокчейн стає не лише системою збереження даних, але й потужним інструментом для автоматизації бізнес-процесів та управління угодами. Вони знаходять застосування у фінансах, логістиці, охороні здоров'я та багатьох інших сферах, і мають великий потенціал для перетворення традиційних методів угод та транзакцій.

Однією з ключових переваг технології блокчейн є її здатність забезпечувати трасованість та прозорість в усіх операціях та транзакціях, які відбуваються в системі. Ця особливість має велике значення для різних галузей та сфер застосування.

У блокчейні кожна транзакція фіксується і трасується в реальному часі, і інформація про неї доступна для всіх учасників мережі. Це створює високий рівень прозорості та довіри.

Трасованість в блокчейні вказує на можливість відстежувати всі кроки та події в обраному процесі чи угоді. Кожна транзакція записується у блокчейні та отримує унікальний ідентифікатор. Завдяки цьому ідентифікатору будь-який учасник мережі може легко відстежити всю історію операцій та визначити, які зміни були внесені у дані.

Прозорість означає, що інформація у блокчейні доступна для перегляду всіма учасниками мережі, і вони можуть бачити всі транзакції та дії інших користувачів. Прозорість створює довіру серед учасників, оскільки ніхто не може приховувати чи змінювати дані без відображення цього у глобальному реєстрі.

Ці характеристики мають велике значення у багатьох сферах, включаючи фінанси, договірні відносини, логістику та охорону здоров'я. У фінансовому секторі, наприклад, блокчейн дозволяє трасувати всі грошові перекази та транзакції, що робить його більш стійким до шахрайства та відмивання грошей. У галузі логістики, прозорість дозволяє відстежувати рух товарів від виробника до споживача, запобігаючи підробці та втратам.

Завдяки трасованості та прозорості, блокчейн робить процеси більш надійними, ефективними та відкритими [10, 38-41].

Для прийняття рішень і підтвердження транзакцій використовується механізм консенсусу. Різні блокчейн мережі можуть використовувати різні алгоритми консенсусу, такі як Proof of Work (PoW) чи Proof of Stake (PoS), для визначення того, які транзакції будуть додані до блоку.

Механізм консенсусу є невід'ємною частиною функціонування технології блокчейн і відіграє важливу роль у забезпеченні вірогідності та надійності мережі. Цей механізм дозволяє визначити, яка версія подій чи транзакцій є правильною і буде включена до блокчейну.

У мережі блокчейн можуть бути сотні та тисячі вузлів, і кожен з них може вносити нові транзакції до системи. Проте виникає питання: як мережа визначає, які транзакції дійсно відбулися і мають бути включені до блокчейну?

Основними механізмами консенсусу є [9, 21]:

- Proof of Work (PoW). У цьому механізмі вузли розв'язують складну обчислювальну задачу, і перший, хто її вирішить, отримує право додати новий блок до ланцюга. PoW використовується, наприклад, у біткоїні. Він гарантує високий рівень безпеки мережі, але вимагає значної енергії;
- Proof of Stake (PoS). У цьому випадку, учасники мережі відбираються для створення нових блоків на основі кількості криптовалюти, яку вони вклали в систему. PoS більш енергоефективний, але вимагає значної інвестиції у криптовалюту;
- Delegated Proof of Stake (DPoS). Цей метод поєднує PoW та PoS, дозволяючи обрати обмежену кількість делегатів, які беруть на себе створення блоків. DPoS є швидшим і менш енерговитратним, але може бути менш децентралізованим.

Механізм консенсусу допомагає уникнути ситуацій, коли дві чи більше версій блокчейну конфліктують між собою. Він забезпечує єдність мережі і гарантує, що всі учасники приймають одну правильну версію подій [21].

Механізм консенсусу є ключовою складовою технології блокчейн і визначає її надійність та стійкість до атак. Він дозволяє мережі функціонувати децентралізовано та забезпечує довіру серед учасників, що робить блокчейн інноваційним і надійним інструментом для збереження та передачі даних.

Ці принципи роботи блокчейну роблять його інноваційною та надійною технологією для збереження, обміну та обробки даних у безпечних та децентралізованих системах.

1.2 Апаратні платформи для хешування

1.2.1 Огляд апаратних платформ

Для реалізації технології хешування в системі блокчейн необхідно вибрати відповідну апаратну платформу, яка забезпечить оптимальну продуктивність та ефективність. Завдяки росту інтересу до криптовалют та блокчейн-технологій, на ринку з'являються різні апаратні рішення, призначені для вирішення завдань хешування та обчислення криптографічних гешів. Ось загальний огляд основних видів апаратних платформ, які можуть бути використані для цієї мети [22-24]:

- графічні процесори, які в основному використовуються для обробки графіки в іграх та програмах для мультимедіа, можуть бути використані для обчислень хешування в блокчейні. Вони мають велику кількість паралельних обчислювальних одиниць і дозволяють розподіляти завдання на багато потоків, що робить їх ефективними для видобутку криптовалют;

- процесори загального призначення (CPU): Зазвичай використовуються для загальних обчислень, такі процесори також можуть використовуватися для хешування в блокчейні. Вони є більш універсальними, але менш продуктивними у порівнянні з GPU чи спеціалізованими ASIC-пристроями;

- спеціалізовані ASIC-пристрої (Application-Specific Integrated Circuit). Ці пристрої розроблені спеціально для виконання конкретних завдань хешування і криптографічних обчислень. Вони зазвичай мають велику продуктивність та низькі енерговитрати, але призначені виключно для одного типу завдань;

- FPGA (Field-Programmable Gate Array). Це апаратні пристрої, які можуть бути програмовані для виконання різних обчислювальних завдань, включаючи хешування. FPGA поєднують переваги GPU та ASIC, дозволяючи адаптувати їх до конкретних вимог блокчейн мережі.

1.2.2 Графічні процесори

Графічні процесори (GPU) та їхнє використання для хешування в технології блокчейну представляють собою цікаву область, яка стала дуже популярною

завдяки розвитку криптовалют та добуванню блоків. GPU є одним із ключових апаратних засобів, які використовуються для генерації хешів та підтримки операцій блокчейн мережі.

Основною перевагою GPU є їхня здатність обробляти багато потоків одночасно. Це створює ідеальні умови для обчислення хешів, оскільки багато операцій може бути обчислено паралельно. В порівнянні з центральними процесорами (CPU), які зазвичай обробляють лише декілька потоків одночасно, GPU набагато продуктивніші для завдань хешування.

GPU мають велику кількість обчислювальних ядер і велику швидкість тактової частоти, що дозволяє їм генерувати хеші дуже швидко. Це особливо важливо для добування криптовалют, де швидкість обчислень впливає на успішність добування нових блоків.

GPU відзначаються своєю високою ефективністю. Вони споживають менше енергії порівняно з центральними процесорами, що робить їх економічно вигідними для добування криптовалют та забезпечення безпеки блокчейн мережі.

GPU можуть бути використані для різних видів хешування та криптографічних операцій, що робить їх гнучкими для використання в різних блокчейн мережах.

Прикладом такої апаратної платформи може бути відеокарта NVIDIA GeForce RTX 3090 (рис. 1.1)



Рисунок 1.1 - NVIDIA GeForce RTX 3090

NVIDIA GeForce RTX 3090 є однією з найпотужніших графічних карт, яка призначена для високопродуктивних ігор та обчислень. Завдяки великій кількості CUDA-ядер і 24 ГБ високошвидкісної пам'яті GDDR6X, вона забезпечує вражаючу обчислювальну потужність і здатність генерувати реалістичні графічні ефекти та обробляти великі об'єми даних.

RTX 3090 також підходить для професійних завдань, таких як редагування відео, 3D-моделювання та обчислення. Завдяки великій кількості пам'яті і високій обчислювальній потужності, вона відмінно справляється з вимогливими завданнями.

Однак RTX 3090 відносно дорога і має високі вимоги до живлення та охолодження. Також, її розмір може вимагати великого корпусу для комп'ютера.

Незважаючи на плюси використання графічних процесорів для хешування блокчейну, важливо враховувати, що з часом розвиток технологій призводить до появи більш спеціалізованих апаратних рішень, таких як ASIC-мінери, які можуть бути набагато продуктивнішими в деяких випадках. Також, для коректного використання GPU для хешування, необхідне належне програмне забезпечення та налаштування, які враховують специфіку конкретного блокчейну.

У підсумку, GPU є важливим інструментом для реалізації хешування в блокчейні, завдяки своїй багатопотоковості, високій швидкості обчислень та ефективності. Вони використовуються як для добування криптовалют, так і для забезпечення безпеки та валідації транзакцій в блокчейн мережах.

1.2.3 Спеціалізовані ASIC-пристрої

Спеціалізовані ASIC-пристрої (Application-Specific Integrated Circuit) є ключовими апаратними рішеннями для хешування в технології блокчейну. Вони розробляються спеціально для виконання конкретних завдань хешування та криптографічних обчислень і є найбільш продуктивними пристроями для цих цілей.

Спеціалізовані ASIC-пристрої розробляються для виконання конкретного завдання, такого як хешування в алгоритмі SHA-256, який використовується у

біткоїні. Вони мають вбудовані апаратні блоки, спеціально адаптовані для швидкого та ефективного виконання цього завдання.

Основною перевагою ASIC-пристроїв є їхня надзвичайно висока продуктивність. Вони здатні генерувати хеші набагато швидше, ніж GPU або CPU, що робить їх ідеальними для добування криптовалют, які використовують алгоритми, що вимагають великих обчислень.

В порівнянні з GPU та CPU, спеціалізовані ASIC-пристрої споживають значно менше енергії при виконанні обчислень. Це робить їх більш економічно вигідними для довгострокового добування криптовалют, оскільки вони дозволяють знизити витрати на електроенергію.

Оскільки ASIC-пристрої призначені лише для виконання певного виду операцій, вони не мають загального призначення, що дозволяє їм досягати максимальної продуктивності в конкретній області використання. Наприклад, ASIC-мінери для біткоїна спеціалізовані саме на хешуванні SHA-256.

Як приклад ASIC-мінери можна розглядати Bitmain Antminer S19 Pro (рис. 1.2).



Рисунок 1.2 - Bitmain Antminer S19 Pro

Bitmain Antminer S19 Pro є потужним ASIC-майнером, спеціально розробленим для добування біткоїна та інших криптовалют, які використовують алгоритм SHA-256. Цей майнер має високу обчислювальну потужність в розмірі 110 TH/s, що дозволяє ефективно добувати криптовалюту.

Цей пристрій споживає приблизно 3250 Вт, що робить його ефективним з точки зору витрат на електроенергію на одиницю обчислювальної потужності. Він має високий рівень ефективності згідно з показником споживання енергії, який становить приблизно 30 Дж/ТН.

Майнер має відповідну систему охолодження, але він може видавати певний рівень шуму (приблизно 75 дБ), тому важливо розміщати його в відокремленому приміщенні або враховувати шумову ізоляцію.

Antminer S19 Pro є популярним вибором для майнерів, які спеціалізуються на добуванні біткоїна та інших криптовалют, які використовують алгоритм SHA-256. Важливо враховувати, що ціна майнера може значно варіюватися з часом і залежить від ринкової популярності та попиту.

Однак, слід враховувати, що спеціалізовані ASIC-пристрої не здатні змінювати свою функціональність і використовуються виключно для одного типу завдань. Тому, при зміні алгоритму хешування в мережі блокчейн, вони можуть стати застарілими і потребувати заміни на більш потужні пристрої.

Таким чином, спеціалізовані ASIC-пристрої є найпродуктивнішими і найефективнішими засобами для хешування в блокчейні, зокрема для добування криптовалют. Проте їх ефективність залежить від конкретного завдання та алгоритму хешування, який використовується в мережі, і вони вимагають певних інвестицій для придбання та підтримки.

1.2.4 FPGA

FPGA (Field-Programmable Gate Array) - це апаратний пристрій, який володіє унікальними можливостями в контексті реалізації хешування в технології блокчейну. FPGA представляє собою інтегральну схему, яка може бути програмована для виконання різних обчислювальних завдань, включаючи хешування та криптографічні операції. [24]

FPGA мають поля програмування, в які можна завантажувати програми, що визначають, які логічні елементи та з'єднання між ними створюються в пристрої. Це дозволяє FPGA виконувати різні завдання, в залежності від програми, яку завантажено. FPGA можуть бути повторно програмовані для виконання різних обчислювальних завдань.

Такі мікросхеми є дуже гнучкими і адаптованими пристроями. Вони можуть бути налаштовані для роботи з різними алгоритмами хешування та криптографічними операціями, що робить їх ідеальними для використання в різних блокчейн мережах та проектах.

FPGA володіють великою швидкістю обчислень, подібно до GPU, завдяки великій кількості логічних елементів і можливості паралельного обчислення. Вони здатні генерувати хеші та виконувати криптографічні операції дуже швидко.

FPGA споживають менше енергії порівняно з CPU та GPU, що робить їх ефективними з точки зору витрат на електроенергію. Це важливо для довгострокового використання та економічних розрахунків, особливо в ситуаціях, де велика кількість FPGA може бути використана для створення розподіленої обчислювальної мережі.

Використання програмованих мікросхем в блокчейн мережі може позитивно вплинути на її масштабованість. FPGA дозволяють розподілити обчислення між вузлами мережі, що поліпшує швидкість обробки транзакцій та доводить до покращення продуктивності.

Програмне забезпечення для FPGA також грає важливу роль у їхньому використанні в блокчейні. Програми повинні бути налаштовані для конкретних завдань, таких як добування криптовалют або підтримка безпеки мережі.

Як приклад такої платформи можна навести AMD Varium™ C1100 (рис. 1.3).



Рисунок 1.3 - AMD Varium™ C1100

AMD Varium™ C1100 - розробницька блокчейн-карта, призначена для прискорення обчислювальних задач, пов'язаних з технологією блокчейн, включаючи перевірку транзакцій та обчислення хеш-функцій, які використовуються в більшості алгоритмів Proof of Work (PoW) та Proof of Stake (PoS). Велика FPGA-структура на карті AMD Varium C1100 дозволяє користувачам реалізовувати паралельні обчислювальні потоки та виконувати одночасно кілька хеш-алгоритмів для досягнення найвищої продуктивності при мінімальному споживанні енергії.

Побудована на архітектурі AMD UltraScale+, акселераторна карта Varium™ C1100 має компактний форм-фактор у вигляді однієї слот-карти повної висоти та наполовину довгу, при максимальному споживанні енергії до 75 Вт (пасивне охолодження). Карта має подвійні QSFP28 мережеві входи/виходи, які підтримують 8 x 25 Гбіт/с, інтерфейс PCI Express® (PCIe®) Gen3x16 або gen4x8, а також 8 ГБ HBM (високопродуктивна пам'ять другого покоління).

Наявний інтерфейс USB надає підтримку відлагодженню та програмуванню. Він дозволяє користувачам отримувати доступ до BMC для моніторингу та керування системою, FPGA UART для підключення бітстріму та FPGA JTAG для програмування та відлагодження за допомогою Vivado.

Акселераторну карту Varium можна легко програмувати з використанням середовища розробки Vitis™ від AMD, що дозволяє криптогрошінам розробляти

різні алгоритми видобутку на високорівневих мовах програмування, таких як C та C++.

Як показано, FPGA представляють собою потужні та гнучкі апаратні пристрої, які можуть бути використані для реалізації хешування та криптографічних операцій в блокчейні. Вони забезпечують гнучкість та продуктивність, що робить їх привабливими для різних проектів і допомагає вдосконалювати технологію блокчейн.

Використання FPGA (Field-Programmable Gate Array) має безліч переваг порівняно з іншими технологіями. FPGA відзначаються гнучкістю і програмованістю, оскільки їхню апаратну структуру можна налаштовувати для виконання різних завдань. Це дозволяє адаптувати FPGA під конкретні потреби і завдання, що робить їх ідеальними для широкого спектра застосувань.

Іншою важливою перевагою є висока продуктивність FPGA. Вони можуть забезпечувати високу обчислювальну потужність і швидкодію, що робить їх ідеальними для завдань, де швидкість виконання має велике значення.

Крім того, FPGA зазвичай споживають менше енергії порівняно з іншими спеціалізованими апаратними рішеннями. Це важливо для застосувань, де важливе споживання енергії та теплова ефективність.

Як показано, FPGA представляють собою потужні та гнучкі апаратні пристрої, які можуть бути використані для реалізації хешування та криптографічних операцій в блокчейні. Вони забезпечують гнучкість та продуктивність, що робить їх привабливими для різних проектів і допомагає вдосконалювати технологію блокчейн.

FPGA також відзначаються високим рівнем паралельності і можуть виконувати одночасно багато завдань, що покращує їхню продуктивність.

Нарешті, FPGA є ідеальним вибором для розробників, оскільки їх можна перепрограмувати і вдосконалювати для різних завдань без необхідності масштабної зміни обладнання.

1.3 Алгоритми хешування функцій

1.3.1 SHA-256 (Secure Hash Algorithm 256-bit)

SHA-256 [25], який стоїть за Secure Hash Algorithm 256-bit, є одним з найбільш використовуваних алгоритмів хешування, особливо в області блокчейну та криптовалют. Цей алгоритм належить до сімейства криптографічних хеш-функцій SHA-2, розробленого Національним інститутом стандартів та технологій США. SHA-256 перетворює вхідні дані (незалежно від їх розміру) у унікальний 256-бітний (32-байтний) хеш-вихід. Ця хеш-функція широко відома своєю стійкістю до колізій, що означає, що знайти два різні вхідні набори даних, які приводять до однакового хешу, є надзвичайно складно.

Використання SHA-256 в блокчейн технологіях, зокрема в Bitcoin, забезпечує високий рівень безпеки. Воно використовується для створення цифрових підписів, які є ключовими для автентифікації транзакцій, а також в майнінгових процесах, де важлива швидкість та ефективність обчислень. Завдяки своїй надійності, SHA-256 забезпечує важливу основу для забезпечення цілісності та безпеки в сучасних криптовалютних системах.

Його використання в блокчейні не обмежується лише криптовалютами. Цей алгоритм застосовується і в ширшому спектрі кібербезпеки та криптографії, оскільки він дозволяє швидко перевіряти цілісність даних. Наприклад, SHA-256 використовується для перевірки цілісності файлів та програмного забезпечення, забезпечуючи користувачам впевненість у тому, що вони не були змінені або пошкоджені.

Ця надійність SHA-256 походить від його конструкції. Алгоритм розроблений таким чином, що навіть незначна зміна в вхідних даних призводить до значної та непередбачуваної зміни в хеші. Це властивість відома як "лавинний ефект". В результаті, важко здогадатися про вхідні дані, знаючи лише хеш, що робить SHA-256 ефективним в областях, де конфіденційність та безпека є критично важливими.

Також важливо відзначити, що, незважаючи на його широке використання та надійність, дослідники в області криптографії постійно аналізують і оцінюють SHA-256 на предмет потенційних вразливостей. Це дозволяє забезпечувати його стійкість до майбутніх загроз, таких як квантові обчислення, які теоретично можуть підірвати його безпеку. Однак, наразі SHA-256 залишається одним з найбільш безпечних і широко використовуваних алгоритмів хешування у світі.

1.3.2 Кессак

Кессак [26], відомий також як SHA-3 (Secure Hash Algorithm 3), є видатним алгоритмом хешування, який був обраний як наступник серії SHA-2 після відкритого конкурсу, проведеного Національним інститутом стандартів та технологій США. Цей алгоритм розрізняється від своїх попередників унікальною архітектурою та принципами роботи.

Особливість цього алгоритму полягає в його конструкції, заснованій на так званій "губці", що дозволяє алгоритму виконувати абсорбцію вхідних даних і потім "віджимати" хеш. Ця гнучкість означає, що Кессак може виробляти хеші різної довжини, на відміну від фіксованого розміру вихідних даних, як у випадку з SHA-256.

Кессак забезпечує високий рівень безпеки і вважається стійким до всіх відомих видів криптографічних атак. Це робить його відмінним вибором для застосування у сучасних криптографічних системах. Крім того, завдяки його високій швидкості та ефективності, Кессак широко використовується в різних областях, включаючи блокчейн технології, де він служить основою для хешування в Ethereum.

Завдяки своїй унікальній структурі та ефективності, він продовжує бути предметом досліджень та аналізу в криптографічній спільноті, демонструючи його важливість і релевантність в області інформаційної безпеки. Його здатність адаптуватися до різних потреб та вимог робить його важливим інструментом у світі цифрової безпеки і блокчейну.

Кессак, крім своєї універсальності та безпеки, вирізняється також своєю високою продуктивністю в різних обчислювальних середовищах. Ця особливість

робить його ідеальним для використання не тільки в потужних серверних системах, але й в обмежених за ресурсами пристроях, таких як мобільні телефони та вбудовані системи.

Однією з ключових переваг цього алгоритму є його гнучкість у термінах розміру хешу. Він може генерувати хеші різної довжини, що дає розробникам можливість налаштовувати рівень безпеки та продуктивності відповідно до їхніх конкретних потреб. Ця функція робить Кессак особливо привабливим для широкого спектру застосувань, від банківської безпеки до захисту персональних даних.

Ще один аспект, що заслуговує уваги, - це стійкість Кессак до квантових атак. У світлі зростаючої загрози з боку квантових комп'ютерів, здатність алгоритму протистояти таким атакам є важливим фактором для майбутньої безпеки криптографічних систем.

В кінцевому підсумку, цей алгоритм хешування стоїть як сильний кандидат для тривалого використання в криптографії завдяки своїм унікальним характеристикам та гнучкості. Його внесок у безпеку цифрових систем є значним і, ймовірно, буде залишатися актуальним на довгі роки.

1.3.3 RIPEMD-160

RIPEMD-160, що розшифровується як RACE Integrity Primitives Evaluation Message Digest, є одним із визнаних алгоритмів хешування, розроблений у Європі як частина проекту RACE (Research and Development in Advanced Communications Technologies in Europe). Алгоритм створює 160-бітний хеш, відомий своєю стійкістю до криптографічних атак, особливо щодо колізій, коли два різних вхідних повідомлення призводять до одного й того самого хешу [38-39].

Цей алгоритм був спеціально розроблений для заміни попередніх алгоритмів хешування, таких як MD4 та MD5, які з часом стали вразливими до криптографічних атак. Цей алгоритм знайшов широке застосування у сфері цифрової безпеки, зокрема в області криптовалют. Наприклад, він використовується в комбінації з іншими алгоритмами, як-от SHA-256, в блокчейні Bitcoin для створення криптографічно стійких адрес.

Надійність RIPEMD-160 випливає з його конструкції, яка забезпечує високий рівень безпеки та зменшує ризик потенційних вразливостей. Хоча він не так широко використовується, як деякі інші алгоритми хешування, такі як SHA-256, його важливість в контексті криптовалют і цифрової безпеки залишається значною. RIPEMD-160 продовжує відігравати ключову роль у забезпеченні безпеки та цілісності цифрових систем, де необхідна надійна криптографічна стійкість.

Завдяки своїй 160-бітній довжині хешу цей алгоритм, пропонує хороший баланс між безпекою та ефективністю. Цей розмір хешу робить його достатньо стійким до атак грубою силою та забезпечує прийнятний рівень безпеки для більшості застосувань, у той час як його обчислювальна ефективність забезпечує швидкість обробки даних.

Однією з особливостей RIPEMD-160 є його дизайн, який містить два паралельних шляхи обробки даних. Ця структура забезпечує додатковий рівень безпеки, оскільки зломисникам потрібно боротися з двома різними механізмами генерації хешу одночасно, що значно ускладнює атаки.

У сфері криптовалют він використовується для створення публічних адрес. Наприклад, у Bitcoin він використовується для скорочення довжини хешу, створеного за допомогою SHA-256, що робить публічні адреси більш компактними та керованими. Це є важливим, оскільки користувачі Bitcoin часто мають справу з довгими та складними рядками для транзакцій, і скорочення цих рядків робить їх більш зручними для використання.

Загалом, RIPEMD-160 залишається важливим інструментом у світі цифрової безпеки та криптографії. Його специфічна структура та перевірена часом стійкість до різних криптографічних атак роблять його цінним ресурсом для розробників та фахівців з безпеки.

1.3.4 Blake2

Blake2 є швидким та ефективним криптографічним алгоритмом хешування, який був розроблений як покращена альтернатива іншим відомим алгоритмам, таким як SHA-2 та SHA-3. Він виник як результат розвитку попереднього конкурента SHA-3, відомого як BLAKE. Основною метою Blake2 було

забезпечення високої швидкості обробки даних при збереженні високого рівня безпеки.

Цей алгоритм відрізняється своєю високою продуктивністю, особливо на сучасному апаратному забезпеченні, що робить його ідеальним для широкого спектру застосувань, від веб-серверів до мобільних пристроїв. Його швидкість перевершує багато традиційних алгоритмів хешування, при цьому він забезпечує порівнянний рівень безпеки [30, 40].

Однією з особливостей Blake2 є його гнучкість: він пропонує кілька варіантів, зокрема Blake2b і Blake2s, які оптимізовані для різних обчислювальних середовищ. Blake2b призначений для 64-бітних платформ, тоді як Blake2s більше підходить для 32-бітних систем.

Крім того, цей алгоритм має властивості, які роблять його корисним у застосуваннях, де потрібна одночасна швидкість та безпека, наприклад, при верифікації паролів або швидкому хешуванні великих обсягів даних. Його ефективність у використанні пам'яті та швидкість обробки роблять його відмінним вибором для систем, де ресурси обмежені.

Все це робить Blake2 дуже привабливим вибором для розробників та інженерів, які шукають надійний, швидкий та ефективний алгоритм хешування для сучасних криптографічних та безпекових застосувань. Він продовжує набувати популярності і визнання в галузі, що свідчить про його міцність та ефективність.

Цей алгоритм хешування, завдяки своїй конструкції, також забезпечує високий рівень гнучкості та налаштувань. Розробники можуть налаштувати довжину хешу відповідно до своїх потреб, що робить Blake2 придатним для різноманітних застосувань, від шифрування файлів до систем автентифікації. Ця здатність адаптуватися до конкретних вимог додатків є однією з ключових переваг Blake2.

Крім того, Blake2 пропонує вбудовані можливості для забезпечення ключової безпеки (keyed hashing), що робить його ефективним для використання в мак-функціях (message authentication codes). Ця особливість робить його ідеальним

для сценаріїв, де хешування повинно бути пов'язано з конкретним ключем, що підвищує безпеку даних.

У контексті блокчейну та криптовалют, Blake2 знайшов своє застосування завдяки своїй швидкості та ефективності. Він використовується в деяких криптовалютах як альтернатива більш традиційним алгоритмам хешування, забезпечуючи швидку обробку транзакцій та ефективний майнінг.

В цілому, цей алгоритм стоїть як алгоритм хешування, який вирішує багато викликів, пов'язаних із швидкістю та безпекою в сучасних цифрових системах. Його універсальність та ефективність роблять його цінним інструментом для широкого кола застосувань, від особистої безпеки до промислових систем кібербезпеки.

1.3.5 Scrypt

Scrypt є важливим криптографічним алгоритмом, розробленим з метою створення більш безпечних та витратних за часом алгоритмів хешування. Його особливість полягає в тому, що він спеціально розроблений, щоб бути високовитратним з точки зору пам'яті та часу, що робить його особливо стійким до атак за допомогою спеціалізованого апаратного забезпечення, такого як ASIC (application-specific integrated circuits) та FPGA (field-programmable gate arrays).

Такий алгоритм знайшов широке застосування в області криптовалют [27-28, 35], зокрема у Litecoin, де він використовується для майнінгу. Відмінність Scrypt від інших алгоритмів, таких як SHA-256, використовуваних у Bitcoin, полягає у його вищих вимогах до обчислювальної пам'яті. Це означає, що для ефективного майнінгу потрібні машини з великою кількістю оперативної пам'яті, що ускладнює створення спеціалізованих майнінг-пристроїв і робить майнінг більш доступним для звичайних користувачів зі стандартним ПК.

Крім того, Scrypt використовується в системах автентифікації та захисту паролів. Його висока вимогливість до ресурсів робить його корисним для обмеження можливостей швидкого підбору паролів, забезпечуючи вищий рівень безпеки у порівнянні з менш витратними алгоритмами.

В цілому, цей алгоритм відрізняється від інших алгоритмів хешування своїми властивостями, які вимагають значно більшої кількості пам'яті та часу для обробки. Ці особливості роблять його привабливим для застосувань, де безпека є важливішою, ніж швидкість обчислень.

Scrypt, у своїй ролі як алгоритм, що інтенсивно використовує пам'ять, дозволяє ефективно протистояти атакам, заснованим на використанні спеціалізованого обладнання. Це особливо важливо у контексті майнінгу криптовалют, де великі ферми з ASIC-пристроями можуть домінувати та централізувати процес. Завдяки Scrypt малі майнери, які використовують звичайні комп'ютери, можуть конкурувати більш ефективно, що сприяє децентралізації мережі.

Використання такого алгоритму для захисту паролів також є значущим, оскільки воно робить більш складними та витратними спроби зламу паролів за допомогою грубої сили або методів підбору. У середовищах, де безпека паролів є критичною, наприклад, у веб-сервісах та базах даних, Scrypt допомагає зменшити ризики, пов'язані з крадіжкою паролів та даних.

Однак, необхідно підкреслити, що його висока вимогливість до ресурсів може бути двояким мечем. З одного боку, це забезпечує більшу безпеку, але з іншого – може призводити до підвищеного енергоспоживання та потреби у більш потужному обладнанні, особливо у контексті майнінгу. Це може бути обмежувальним фактором для окремих користувачів, які не мають доступу до великої кількості обчислювальних ресурсів.

Загалом, Scrypt залишається цінним інструментом у світі криптографії та криптовалют, забезпечуючи унікальний баланс між безпекою та доступністю. Його роль у забезпеченні децентралізації та безпеки систем є важливою, і він продовжує бути використовуваним у різних криптовалютних проектах та застосуваннях, де важливий захист від атак.

1.3.6 Ethash

Ethash є ключовим алгоритмом хешування, використовуваним у Ethereum, одній з найбільших криптовалютних платформ. Цей алгоритм був спеціально

розроблений для оптимізації під GPU (графічні процесори), що робить його відмінним вибором для децентралізованого майнінгу. Ця особливість Ethash є важливою для забезпечення більшої рівності у майнінг-процесі, у порівнянні з алгоритмами, які краще підходять для ASIC-пристроїв, що часто домінують в інших криптовалютних мережах [30, 36].

Ефективність Ethash на GPU означає, що індивідуальні майнери можуть конкурувати з великими майнінговими фермами, що допомагає підтримувати децентралізацію мережі Ethereum. Це досягається за рахунок високих вимог до пам'яті, які ускладнюють створення ефективних ASIC-пристроїв для майнінгу Ethereum.

Цей алгоритм також включає механізми, що ускладнюють використання пулів майнінгу, великих груп майнерів, які об'єднують свої ресурси для збільшення шансів на успішний майнінг. Це робиться шляхом забезпечення високої вимогливості до оперативної пам'яті, що робить синхронізацію роботи великих груп майнерів складнішою.

Важливою його особливістю є його стійкість до зростання хешрейту, оскільки він був спроектований таким чином, щоб бути відносно стійким до швидкого розвитку апаратного забезпечення. Це допомагає забезпечувати більш стабільну та передбачувану мережу, що є важливим для забезпечення безпеки та стабільності платформи Ethereum.

Загалом, Ethash відіграє важливу роль у підтримці безпеки, децентралізації та стабільності мережі Ethereum, роблячи його ключовим елементом екосистеми цієї криптовалюти. Його здатність адаптуватися до різних обчислювальних середовищ та високі вимоги до пам'яті роблять його унікальним та ефективним рішенням для майнінгу.

1.3.7 X11

X11 є унікальним алгоритмом хешування, який знайшов своє застосування в криптовалюті Dash, а також у деяких інших криптовалютах. Основною відмінною рисою X11 є його конструкція, яка включає послідовне використання одинадцяти

різних криптографічних алгоритмів хешування. Ця каскадна структура забезпечує високий рівень безпеки та стійкість до атак.

Використання різноманітних алгоритмів у X11 не лише збільшує його безпеку, але й робить його менш сприйнятливим до вразливостей, які можуть бути виявлені в окремих алгоритмах. Навіть якщо один або кілька алгоритмів у ланцюжку стануть вразливими, інші продовжують забезпечувати безпеку хешу [31].

X11 також спроектований з метою бути більш доступним для майнінгу на звичайних комп'ютерах. У порівнянні з іншими алгоритмами, такими як SHA-256, використовуваним у Bitcoin, X11 менш сприйнятливий до майнінгу за допомогою спеціалізованих ASIC-пристроїв, що сприяє децентралізації майнінгу.

Ефективність X11 у використанні енергії є ще однією важливою перевагою, особливо в контексті зростаючих занепокоєнь щодо енерговитрат у майнінгу криптовалют. X11 був розроблений так, щоб бути ефективнішим з точки зору споживання енергії, порівняно з іншими алгоритмами, такими як SHA-256.

Однак, слід враховувати, що складність та багатоступінність X11 може зробити його менш прозорим та складнішим для розуміння і аналізу порівняно з більш простими алгоритмами. Тим не менш, X11 залишається важливим інструментом у світі криптовалют, забезпечуючи унікальне поєднання безпеки, ефективності та доступності майнінгу.

1.3.8 Poseidon

Poseidon є відносно новим алгоритмом хешування, який з'явився у сфері блокчейну та криптографії. Цей алгоритм був спеціально розроблений, щоб бути оптимальним для використання у системах, де важливі швидкість та ефективність, особливо у контексті zero-knowledge proofs (доказів з нульовим розголошенням), які є ключовими для приватності та анонімності у деяких блокчейн-системах.

Основна перевага цього алгоритму полягає в його здатності швидко обробляти великі обсяги даних з мінімальними вимогами до обчислювальних ресурсів. Це робить його ідеальним для блокчейн-платформ, які прагнуть до підвищення швидкості транзакцій та ефективності обробки даних.

Крім того, він забезпечує високий рівень безпеки та надійності. Його конструкція забезпечує стійкість до відомих методів криптографічного аналізу, що робить його безпечним вибором для застосувань, де це критично важливо.

Poseidon також вирізняється своєю універсальністю, оскільки він може бути ефективно інтегрований у різноманітні криптографічні системи та протоколи. Його здатність адаптуватися до різних обчислювальних середовищ та потреб користувачів робить його цінним інструментом у сучасній криптографії [36].

Цей алгоритм, крім зазначених переваг, вирізняється й іншими характеристиками, що роблять його особливо привабливим для широкого спектру застосувань. Однією з таких характеристик є його компактність: алгоритм здатен генерувати хеші, які займають відносно мало місця. Це особливо важливо для систем, де ефективність зберігання даних має велике значення, як у випадку з блокчейнами та іншими децентралізованими технологіями.

Крім того, Poseidon характеризується високою стійкістю до паралельних атак, що робить його більш безпечним у середовищах, де потенційні зловмисники можуть використовувати потужні обчислювальні ресурси для спроби розшифрування хешів. Ця стійкість до паралельної обробки також забезпечує, що Poseidon залишається ефективним у різних обчислювальних середовищах, від мобільних пристроїв до великих дата-центрів.

Він також має потенціал для інтеграції в системи, де потрібні швидкі та безпечні докази з нульовим розголошенням. Це робить його корисним в блокчейн-платформах, які прагнуть до забезпечення високого рівня приватності та анонімності транзакцій.

У підсумку, Poseidon представляє собою потужний інструмент в світі сучасної криптографії та блокчейну. Його унікальні характеристики, які поєднують швидкість, ефективність, безпеку та компактність, роблять його привабливим вибором для широкого спектру застосувань, від підтримки базових операцій блокчейну до забезпечення додаткових рівнів конфіденційності та безпеки.

1.4 Висновки до розділу 1

Технологія блокчейну, яка характеризується своєю інноваційністю та надійністю, пропонує новий підхід до зберігання, обміну та обробки даних у безпечних та децентралізованих системах. Ключовим аспектом блокчейну є його децентралізована природа, що відкидає ідею централізованого контролю і надає кожному учаснику мережі рівні права та можливості. Це сприяє створенню стійкої системи, захищеної від збоїв і кібератак. Використання блокчейну включає захист даних за допомогою криптографії, що забезпечує безпеку і прозорість у системі. Важливу роль у блокчейні відіграють смарт-контракти, які автоматизують угоди і зменшують потребу в посередниках, сприяючи ефективності та зниженню ризиків обману. Технологія також забезпечує трасованість та прозорість усіх операцій, підвищуючи довіру та відкритість у фінансових, логістичних, та інших сферах. Нарешті, механізми консенсусу, такі як Proof of Work та Proof of Stake, забезпечують надійність і єдність мережі, гарантуючи довіру серед учасників і підтримуючи децентралізовану структуру блокчейну.

У контексті хешування в блокчейні, різні апаратні платформи пропонують унікальні переваги та обмеження. Графічні процесори (GPU) ефективно обробляють багато потоків одночасно, що робить їх ідеальними для паралельних обчислень, характерних для добування криптовалют та обробки блокчейн-транзакцій. Однак, вони можуть бути відносно дорогими та мати високі вимоги до живлення та охолодження.

Спеціалізовані ASIC-пристрої розробляються конкретно для виконання завдань хешування і криптографічних обчислень, забезпечуючи надзвичайно високу продуктивність. Вони ефективні у використанні енергії, але мають обмежену функціональність і не можуть бути перепрограмовані для інших завдань.

З іншого боку, FPGA є гнучкими та програмованими пристроями, які можуть бути налаштовані для роботи з різними алгоритмами хешування та

криптографічними операціями. Вони поєднують переваги GPU та ASIC, дозволяючи адаптувати їх до конкретних вимог блокчейна.

Підрозділ про алгоритми хешування функцій охоплює детальний огляд різних криптографічних хеш-алгоритмів, кожен з яких має унікальні характеристики та застосування, особливо у контексті блокчейну та криптовалют. SHA-256 виділяється своєю надійністю та широким використанням у таких сферах, як автентифікація транзакцій та верифікація цілісності даних, зокрема у Bitcoin. Кессак, відомий як SHA-3, забезпечує високу безпеку та гнучкість, дозволяючи генерувати хеші різної довжини, що робить його ідеальним для різноманітних криптографічних систем. RIPEMD-160, зі своїм унікальним дизайном та стійкістю до криптографічних атак, є значущим в контексті криптовалют, особливо для створення криптографічно стійких адрес. Blake2 виділяється своєю швидкістю та ефективністю, що робить його підходящим для широкого спектру застосувань, включаючи верифікацію паролів та обробку великих обсягів даних. Scrypt, з його високими вимогами до пам'яті та обчислювальними витратами, сприяє децентралізації майнінгу криптовалют та забезпеченню безпеки систем автентифікації. Ethash, ключовий для Ethereum, оптимізований для GPU, відіграє важливу роль у підтримці децентралізації та стабільності мережі Ethereum. X11, із своєю унікальною каскадною структурою, забезпечує високий рівень безпеки та ефективності в енергоспоживанні, що робить його привабливим для децентралізації майнінгу та зменшення енерговитрат. Нарешті, Poseidon, зі своєю здатністю швидко обробляти великі обсяги даних та високим рівнем безпеки, є цінним доповненням до світу криптографії, особливо у системах з доказами з нульовим розголошенням. Кожен з цих алгоритмів вносить свій унікальний вклад у безпеку, ефективність та гнучкість у світі цифрових технологій і криптовалют.

2 РОЗРОБКА АРХІТЕКТУРИ СИСТЕМИ

2.1 Апаратне забезпечення системи

2.1.1 AMD Varium C1100 Blockchain Accelerator Card

AMD Varium™ C1100 - це розробницька картка для блокчейну, призначена для прискорення обчислювально інтенсивних додатків у блокчейні, включно з валідацією транзакцій та розрахунками хешу, які використовуються у більшості алгоритмів Proof of Work (PoW) та Proof of Stake (PoS) [42]. Велика FPGA Fabric на картці AMD Varium C1100 дозволяє користувачам реалізовувати паралельні обчислювальні потоки та виконувати декілька алгоритмів хешування паралельно, щоб забезпечити найвищу продуктивність при найменшому споживанні енергії.

Створена на архітектурі AMD UltraScale+, прискорювальна картка Varium™ C1100 забезпечує компактний однослотовий форм-фактор повної висоти та половинної довжини, працюючи при максимальній потужності до 75 Вт (пасивне охолодження). Картка оснащена подвійним мережевим вход/вихід QSFP28, що підтримує 8 x 25 Гб/с, PCI Express® (PCIe®) Gen3x16 або Gen4x8, та 8 ГБ HBM.

Інтерфейс USB забезпечує підтримку відлагодження та програмування. Він дозволяє користувачам отримати доступ до BMC для моніторингу та контролю системи, FPGA UART для з'єднання з бітстрімом, та FPGA JTAG для програмування та відлагодження у Vivado.

Прискорювальну картку Varium можна легко програмувати, використовуючи середовище розробки Vitis™ від AMD, що дозволяє криптомайнерам розробляти різні алгоритми майнінгу на високорівневих мовах програмування, таких як C та C++.

AMD UltraScale+ архітектура є втіленням передових інженерних ідей та інновацій, які змінюють уявлення про можливості програмованих логічних пристроїв (PLD), зокрема FPGA.

Перш за все, UltraScale+ відрізняється своєю масштабованістю. Архітектура забезпечує високу щільність інтеграції, дозволяючи розміщати більше логічних

елементів у меншому просторі. Це означає, що розробники можуть створювати більш складні та функціонально насичені системи на одному чипі, зменшуючи загальні вимоги до простору і споживання енергії.

Крім того, UltraScale+ вражає своєю енергоефективністю. Завдяки оптимізованому дизайну та передовим технологіям виробництва, чипи UltraScale+ споживають менше енергії, що є критично важливим для сучасних високопродуктивних обчислювальних систем. Це дозволяє не тільки знижувати операційні витрати, але й забезпечує менший тепловий вплив, що спрощує задачі з охолодження. Інновації цієї технології також торкаються аспектів інтеграції. Ця архітектура підтримує високошвидкісні мережеві інтерфейси та різноманітні опції підключення, включаючи передові стандарти PCI Express. Це забезпечує високу гнучкість при інтеграції FPGA в різні обчислювальні системи.

AMD UltraScale+ також розширені можливості для обробки даних. Із покращеним дизайном логічних блоків і підвищеною частотою роботи, розробники можуть досягати більшої продуктивності, оптимізуючи обчислювальні процеси та знижуючи затримки. Нарешті, ця технологія відрізняється своєю програмованістю та адаптивністю. Розробники можуть швидко та ефективно налаштовувати логіку FPGA відповідно до конкретних вимог проекту, що надає їм неперевершену гнучкість у реалізації складних алгоритмів та систем.

У світі блокчейну, де кожна секунда та кожен біт даних мають значення, AMD-Xilinx Varium C1100 відіграє ключову роль. Ця картка не просто компонент системи, але справжній каталізатор прогресу в області блокчейн-обчислень.

Традиційні виклики, з якими стикаються розробники блокчейн-додатків – це і величезні обсяги транзакцій, які потрібно перевірити та валідувати, і складності криптографічних обчислень, що забезпечують безпеку та надійність системи. Все це вимагає значної обчислювальної потужності. В цьому контексті Varium C1100 виступає як вирішення цих проблем, пропонуючи неперевершену продуктивність та ефективність.

На відміну від звичайних процесорів, Varium C1100 забезпечує високу паралелізацію завдяки своїм FPGA можливостям. Це означає, що вона може

одночасно обробляти величезні обсяги даних, значно прискорюючи процес валідації транзакцій та обчислення хешів. Для блокчейн-систем, де час - це не просто гроші, але й основа довіри та безпеки, така швидкість та ефективність є надзвичайно важливими.

Крім того, Varium C1100 має великий обсяг пам'яті та високошвидкісні I/O можливості. Це означає, що вона може ефективно зберігати та обробляти державу блокчейну, яка постійно розростається, без втрати продуктивності. У світі, де розмір блокчейнів вимірюється в гігабайтах, а навіть терабайтах, така здатність є неоціненною. Основні технічні характеристики подані в табл. 2.1.

Таблиця 2.1 - Ключові характеристики AMD Varium™ C1100

Характеристика	Значення
Модель	AMD Varium™ C1100
Архітектура	AMD UltraScale+
Форм-фактор	Однослотова карта повної висоти, наполовину довга
Обчислювальна потужність	Не вказано (спеціалізована для блокчейн-обчислень)
Максимальне споживання енергії	75 Вт (пасивне охолодження)
Пам'ять	8 ГБ HBM (високопродуктивна пам'ять другого покоління)
Мережеві можливості	Двійні QSFP28 порти, підтримка 8 x 25 Гбіт/с
Інтерфейс	PCI Express® (PCIe®) Gen3x16 або gen4x8
Програмування та відлагодження	USB інтерфейс для доступу до BMC, FPGA UART для підключення бітстріму, FPGA JTAG для програмування
Інструменти розробки	Підтримка середовища розробки Vitis™ від AMD
Особливості	Призначена для прискорення блокчейн-задач, включаючи перевірку транзакцій та обчислення хеш-функцій

Не менш важливим є той факт, що Varium C1100 розроблена з урахуванням енергоефективності. Вона виконує свої завдання, споживаючи значно менше енергії, ніж традиційні обчислювальні системи, що є великою перевагою у сфері, де енергоспоживання та екологічність стають все більш актуальними питаннями.

Розглянемо, як картка AMD-Xilinx Varium C1100 вносить свій вклад у два основні механізми консенсусу в блокчейні - Proof of Work (PoW) та Proof of Stake (PoS). Ці механізми, хоча й різні за своєю природою, є життєво важливими для безпеки та ефективності блокчейн-мереж.

PoW, відомий своїм використанням у Bitcoin, залежить від обчислювальної потужності для валідації транзакцій та створення нових блоків. Цей процес, званий "майнінгом", вимагає величезної кількості обчислювальних ресурсів. Varium C1100, з її високою обчислювальною потужністю і здатністю до паралельної обробки даних, ідеально підходить для цього завдання. Вона може швидко виконувати складні криптографічні обчислення, які є ключовими для PoW, тим самим підвищуючи ефективність та шанси на успішне майнінг.

З іншого боку, PoS, який використовується в таких мережах, як Ethereum 2.0, базується не на обчислювальній потужності, а на володінні певною кількістю валюти мережі. У PoS, валідатори вибираються на основі їхньої "частки" або участі в мережі. Тут Varium C1100 сприяє підвищенню ефективності валідації транзакцій та управління блокчейн-стейтом. Завдяки своїй високій продуктивності та мережевим можливостям, вона дозволяє валідаторам швидше обробляти транзакції та підтримувати синхронізацію з мережею, що є критично важливим для забезпечення безперервної роботи та надійності мережі PoS.

AMD Ryzen 3700x є частиною сімейства процесорів Ryzen 3-го покоління від AMD, відомого своєю високою продуктивністю та ефективністю. Цей процесор вирізняється використанням архітектури Zen 2, яка забезпечує значне поліпшення продуктивності порівняно з попередніми поколіннями.

2.1.2 AMD Xilinx AMD Ryzen 3700x

Ryzen 3700x є восьмиядерним процесором з шістнадцятьма потоками, що робить його ідеальним для багатозадачності та вимогливих обчислювальних завдань. Його базова тактова частота складає 3.6 ГГц, яка може бути підвищена до 4.4 ГГц у режимі Turbo, забезпечуючи додаткову потужність при необхідності [43].

Цей процесор відомий своєю високою енергоефективністю, завдяки чому він забезпечує відмінний баланс між продуктивністю та енергоспоживанням. Це

робить його привабливим вибором для користувачів, які прагнуть до високої продуктивності, не жертвуючи енергоефективністю. Ryzen 3700x також відомий своєю підтримкою швидкої оперативної пам'яті та великої кількості PCIe ліній, що робить його ідеальним для ігрових систем, віртуалізації, і високопродуктивних робочих станцій. Завдяки цьому, він здатен ефективно обробляти складні завдання, такі як 3D-моделювання, відеоредагування та інші обчислювально вимогливі програми.

Крім того, Ryzen 3700x має високий рівень сумісності з різними материнськими платами та системами охолодження, що дозволяє легко інтегрувати його у різні комп'ютерні конфігурації. Це робить його популярним вибором серед користувачів, які прагнуть до створення власних систем на замовлення.

Основні технічні характеристики подані у таблиці 2.2.

Таблиця 2.2 - Основні технічні характеристики AMD Ryzen 3700x

Характеристика	Значення
Модель	AMD Ryzen 7 3700x
Покоління	Ryzen 3-є покоління
Архітектура	Zen 2
Кількість ядер	8
Кількість потоків	16
Базова тактова частота	3.6 ГГц
Максимальна тактова частота	4.4 ГГц (режим Turbo)
Тип пам'яті	DDR4
Максимальна кількість пам'яті	Залежить від материнської плати
Тепловий пакет	65 Вт
Технології	AMD StoreMI, AMD SenseMI, AMD Ryzen Master
Сокет	AM4
Технологія виготовлення	7 нм
Підтримка PCIe	PCIe 4.0 (залежить від материнської плати)
Кеш L2	4 МБ
Кеш L3	32 МБ

На закінчення, AMD Ryzen 3700x є міцним процесором, який вирізняється своєю високою продуктивністю, енергоефективністю та гнучкістю використання.

Його потужність та ефективність роблять його відмінним вибором для широкого спектру застосувань, від ігрових систем до професійних робочих станцій.

2.2 Програмне забезпечення системи

2.2.1 Ubuntu

Ubuntu 20.04, також відома як Focal Fossa, є значним випуском в серії операційних систем Ubuntu, що базується на Linux. Ця версія, випущена у квітні 2020 року, є випуском з тривалою підтримкою (LTS), що означає, що вона отримуватиме оновлення та підтримку протягом п'яти років після свого виходу. Однією з ключових особливостей Ubuntu 20.04 є оновлений інтерфейс користувача з удосконаленням графічного середовища GNOME, що забезпечує більш зручний та інтуїтивно зрозумілий досвід користувача.

Ubuntu пропонує покращену підтримку апаратного забезпечення, включаючи оновлені драйвери, що забезпечують кращу сумісність з новішим обладнанням. Ця версія також включає останні версії численних популярних програм та пакетів, що робить її привабливою для широкого кола користувачів, від початківців до професіоналів. Безпека в Ubuntu 20.04 була значно покращена з введенням останніх оновлень безпеки та функцій, таких як підтримка WireGuard VPN за замовчуванням. Це робить систему більш надійною для використання в корпоративних та особистих середовищах, де безпека даних має велике значення.

Завдяки своїй стабільності, підтримці, гнучкості та безпеці Ubuntu 20.04 є відмінним вибором для різноманітних застосувань, включаючи сервери, настільні комп'ютери, розробку програмного забезпечення та використання в освітніх установах. Оновлені версії популярних пакетів і програм забезпечують користувачам доступ до найсучасніших інструментів та додатків. В цілому, Ubuntu 20.04 продовжує підтримувати репутацію Ubuntu як однієї з найбільш доступних та надійних Linux-дистрибутивів на ринку.

2.2.2 AMD Vitis Unified Software Platform

Vitis від AMD (раніше від Xilinx) - це комплексне інтегроване середовище розробки (IDE), призначене для полегшення розробки програмного забезпечення та апаратного забезпечення для широкого спектру платформ AMD [44], включаючи FPGA та адаптивні обчислювальні вузли. Робота з Vitis під операційною системою Ubuntu відкриває багато можливостей для розробників, оскільки Ubuntu є однією з найпопулярніших та найбільш підтримуваних Linux дистрибутивів.

Уніфікована програмна платформа Vitis надає фреймворк для розробки та доставки адаптивних SoC та FPGA прискорених додатків, використовуючи стандартні мови програмування, такі як C та C++. Робочий процес Vitis пропонує всі можливості стандартного середовища розробки програмного забезпечення, включаючи:

- компілятор або крос-компілятор для хост-додатків, що працюють на процесорах x86 або Arm®;
- крос-компілятори для створення fpga бінарних файлів;
- середовище відлагодження для виявлення та усунення проблем у коді;
- профайлери продуктивності для ідентифікації вузьких місць та оптимізації додатку;

Прискорений додаток Vitis складається з двох окремих компонентів: програмного забезпечення, що виконується на стандартному процесорі, такому як процесор X86 або вбудований процесор ARM, та бінарного файлу AMD (xclbin), що містить функції прискорення апаратних засобів, або ядра.

Програмне забезпечення, або хост-додаток, написаний на C/C++ і виконується на звичайному CPU. Програма використовує рідний API XRT, реалізований бібліотекою AMD Runtime (XRT), для взаємодії з ядром прискорення в пристрої AMD. Опис хост-додатка та необхідних API-викликів можна знайти в документації Vitis у розділі "Написання програмного забезпечення".

Апаратно прискорені ядра можуть бути написані на C/C++ або RTL (Verilog або VHDL) та виконуватися у програмованій логічній частині пристрою AMD. Для вимог до кодування дивіться розділи "Розробка PL ядер за допомогою C++" або

"Пакування RTL ядер" у документації Vitis. Ядра інтегруються з апаратною платформою Vitis за допомогою стандартних інтерфейсів AXI.

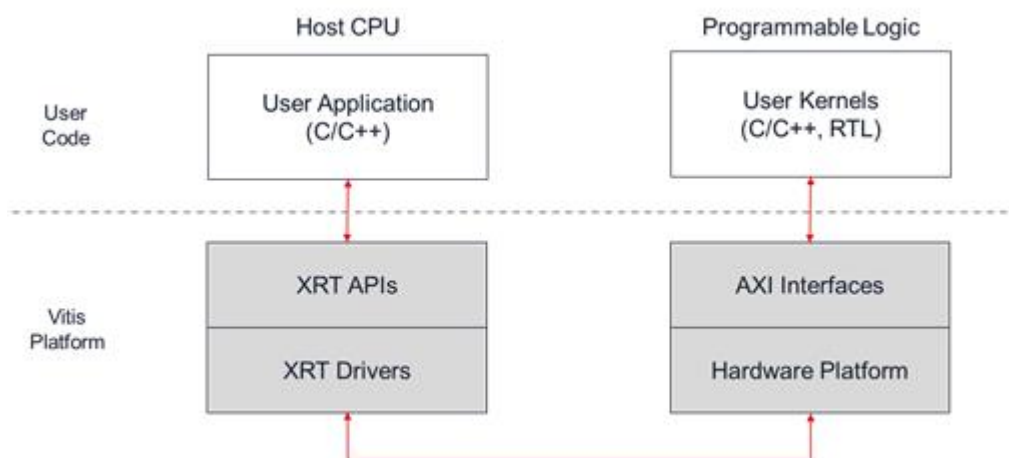


Рисунок 2.1 – Структура Vitis

Прискорені застосування Vitis можуть виконуватися на платформах прискорення обчислень у центрах даних або на вбудованих процесорних платформах:

На картках прискорення центрів даних програмне забезпечення виконується на сервері x86, а ядра працюють у FPGA на картках прискорення, підключених через PCIe®.

На вбудованих процесорних платформах програмне забезпечення виконується на процесорі Arm пристрою AMD MPSoC, а ядра виконуються всередині цього ж пристрою.

Оскільки програмні та апаратні компоненти застосування Vitis використовують стандартизовані інтерфейси (API XRT та протоколи AXI) для взаємодії один з одним, вихідний код користувача залишається майже незалежним від специфіки платформи і може бути легко перенесений між різними платформами прискорення.

Існує кілька способів, за допомогою яких програмне забезпечення може взаємодіяти з апаратними ядрами. Найпростіший метод можна розкласти на наступні кроки:

- хост-додаток записує дані, які потрібні ядру, у глобальну пам'ять пристрою FPGA;
- хост-програма налаштовує вхідні параметри ядра;
- хост-програма запускає виконання ядра;
- ядро виконує необхідні обчислення, доступуючись до глобальної пам'яті для читання або запису даних, як це необхідно. Ядра також можуть використовувати потокові з'єднання для спілкування з іншими ядрами, передаючи дані від одного ядра до наступного;
- ядро повідомляє хосту, що воно завершило своє завдання;
- хост-програма передає дані з глобальної пам'яті назад у пам'ять хоста, або може передати власність на дані іншому ядру.

Процес компіляції Vitis слідує стандартному процесу компіляції та зв'язування як для хост-програми, так і для коду ядра:

- хост-програма будується за допомогою компілятора GNU C++ (g++) для додатків центрів даних або крос-компілятора GNU C++ Arm для вбудованих процесорних пристроїв;
- бінарний файл FPGA будується за допомогою компілятора Vitis (v++). Спочатку ядра компілюються у файл об'єкта AMD (.xo). Потім .xo файли зв'язуються з апаратною платформою для генерації бінарного файлу пристрою AMD (.xclbin). Як описано у команді V++, компілятор та зв'язувач Vitis приймає широкий спектр параметрів для налаштування та оптимізації результатів.

Компілятор Vitis пропонує три різні цільові завдання побудови: два емуляційних завдання, які використовуються для відлагодження та валідації, та стандартне апаратне завдання, яке використовується для генерації фактичного FPGA бінарного файлу:

- програмна емуляція. Код ядра компілюється для виконання на хост-процесорі. Це дозволяє ітеративно вдосконалювати алгоритм через швидкі цикли побудови-запуску. Ця ціль корисна для ідентифікації синтаксичних помилок, виконання відлагодження на рівні вихідного коду ядра, що виконується разом з додатком, та перевірки поведінки системи;

- апаратна емуляція: Код ядра компілюється в апаратну модель (RTL), яка виконується в спеціалізованому симуляторі. Цикл побудови-запуску займає більше часу, але надає детальний, цикл-точний огляд активності ядра. Ця ціль корисна для тестування функціональності логіки, яка буде використана в FPGA, та отримання початкових оцінок продуктивності;
- апаратне завдання: Код ядра компілюється в апаратну мову опису (RTL), а потім синтезується та реалізується для цільового пристрою AMD, в результаті чого отримується бінарний файл (xclbin), який буде виконуватися на фактичному FPGA.

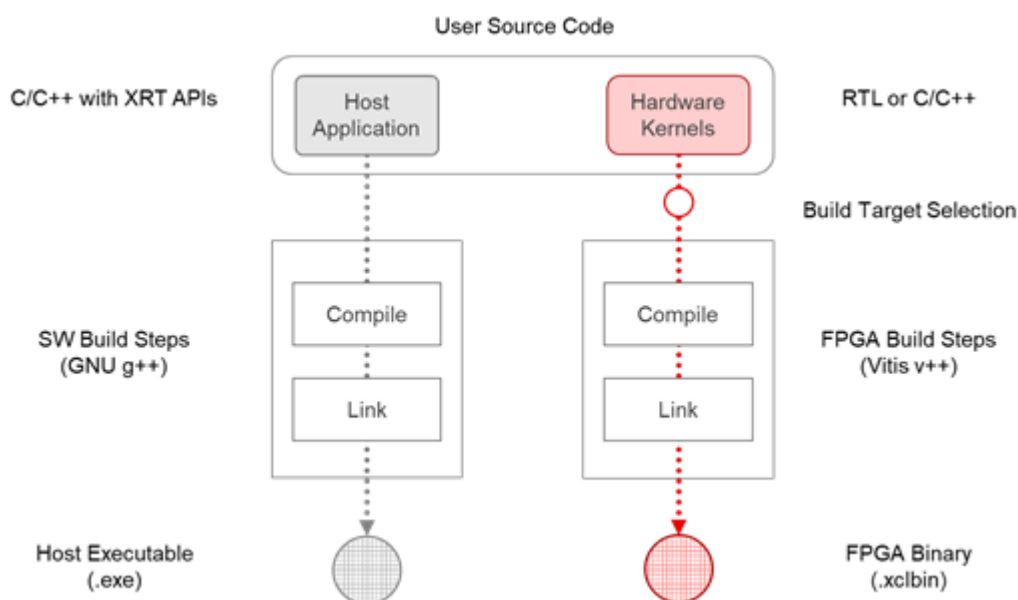


Рисунок 2.2 – Компіляція Vitis

Особливості роботи з Vitis під Ubuntu:

- Vitis можна встановити на Ubuntu шляхом завантаження відповідних пакетів з веб-сайту AMD. Процес встановлення включає налаштування залежностей і вимагає певних знань про Linux;
- ПЗ ефективно інтегрується з різними інструментами та утилітами Ubuntu, що полегшує процес розробки та відлагодження;

- Vitis дозволяє розробникам створювати як програмне, так і апаратне забезпечення, використовуючи високорівневі мови програмування. Це ідеально підходить для розробки FPGA програмування та керування ресурсами;
- ПЗ підтримує різні апаратні платформи AMD, включаючи FPGA, дозволяючи розробникам вибирати найбільш підходящу платформу для своїх проектів;
- Vitis оптимізований для забезпечення високої продуктивності та гнучкості при розробці, що є ключовим для складних обчислювальних задач;
- робота з Vitis під Ubuntu також означає доступ до великої спільноти користувачів та розробників. Існує багато ресурсів, форумів та документації для допомоги у вирішенні проблем та питань.

2.2.3 AMD Xilinx AMD-Xilinx Run Time (XRT) for Ubuntu 20.04

AMD-Xilinx Run Time (XRT) для Ubuntu 20.04 є спеціалізованим програмним забезпеченням, розробленим для підтримки прискорювачів FPGA (Field-Programmable Gate Array) та інших апаратних пристроїв від AMD Xilinx. XRT надає необхідні драйвери та бібліотеки для інтеграції цих пристроїв з операційною системою Ubuntu 20.04, забезпечуючи ефективне управління апаратними ресурсами та оптимізацію продуктивності.

Основні аспекти AMD-Xilinx XRT включають:

- інтеграція з FPGA та іншими прискорювачами. XRT дозволяє розробникам легко використовувати потужності FPGA та інших прискорювачів AMD Xilinx у своїх програмних продуктах;
- драйвери та API. Пакет включає драйвери для Ubuntu 20.04, які забезпечують стабільну роботу FPGA. Також надається API (Application Programming Interface), що дозволяє програмістам взаємодіяти з апаратними засобами на високому рівні;
- оптимізація продуктивності. XRT оптимізує використання апаратних ресурсів FPGA, підвищуючи продуктивність роботи програм та знижуючи витрати на обчислювальні ресурси;

- підтримка розробки. XRT надає інструменти для розробників, які полегшують інтеграцію FPGA в програмні рішення, включаючи підтримку різних мов програмування та фреймворків;
- сумісність з Ubuntu 20.04. XRT спеціально розроблений для сумісності з Ubuntu 20.04, що забезпечує легку інтеграцію з цією операційною системою і її великим екосистемним портфоліо.

AMD-Xilinx Run Time є ключовим компонентом для розробників, які використовують FPGA та інші апаратні прискорювачі AMD Xilinx у своїх рішеннях на базі Ubuntu 20.04, забезпечуючи необхідну інфраструктуру для максимального використання потенціалу цих пристроїв.

2.3 Висновки до розділу 2

AMD Varium™ C1100 є спеціалізованою розробницькою карткою для блокчейну, призначеною для прискорення обчислювально інтенсивних задач, таких як валідація транзакцій і розрахунки хешу. Заснована на архітектурі AMD UltraScale+, вона пропонує високу щільність інтеграції і енергоефективність, що дозволяє розміщати більше логічних елементів на меншій площі з меншим споживанням енергії. Картка забезпечує паралельне виконання обчислювальних задач і включає високошвидкісні мережеві інтерфейси. Її можна програмувати за допомогою середовища розробки Vitis™ від AMD, що робить її привабливою для криптомайнерів. Varium C1100 є ефективним рішенням для блокчейн-застосувань, надаючи високу продуктивність та енергоефективність, що є важливим у сфері, де енергоспоживання та екологічність стають все більш актуальними.

Також в цьому розділі було розглянуто три ключових програми: Ubuntu 20.04 (Focal Fossa), AMD Vitis Unified Software Platform, і AMD-Xilinx Run Time (XRT) для Ubuntu 20.04.

Ubuntu 20.04 - це важливий випуск серії операційних систем Ubuntu на базі Linux. Цей випуск має тривалу підтримку (LTS) та пропонує оновлений користувацький інтерфейс з удосконаленим середовищем GNOME. Ubuntu 20.04 покращує підтримку апаратного забезпечення, безпеку (включаючи підтримку WireGuard VPN), і включає оновлені версії численних програм та пакетів.

AMD Vitis - це інтегроване середовище розробки від AMD, призначене для полегшення розробки програмного та апаратного забезпечення, зокрема для FPGA та адаптивних обчислювальних вузлів. Vitis включає компілятори, середовище відлагодження, профайлери продуктивності і дозволяє створювати ядра на C/C++ або RTL. Це середовище інтегрується з хост-програмами через API XRT і підтримує стандартні інтерфейси для взаємодії програмного забезпечення та апаратних компонентів.

AMD-Xilinx Run Time для Ubuntu 20.04 - це програмне забезпечення, спеціально розроблене для підтримки FPGA та інших апаратних пристроїв AMD Xilinx. XRT включає драйвери та API для ефективного управління апаратними ресурсами, сприяючи оптимізації продуктивності FPGA. Це забезпечує сумісність із Ubuntu 20.04 і є важливим для розробників, що використовують FPGA та інші апаратні прискорювачі AMD Xilinx.

Взаємодія між цими програмами полягає у тому, що Ubuntu 20.04 забезпечує стабільну та безпечну операційну систему, на якій можна використовувати AMD Vitis для розробки програмного та апаратного забезпечення. AMD-Xilinx Run Time покращує інтеграцію і ефективне використання FPGA та інших апаратних прискорювачів AMD Xilinx у середовищі Ubuntu, забезпечуючи оптимізацію продуктивності та ефективне управління ресурсами.

3 РОЗРОБКА FPGA СИСТЕМИ ДЛЯ ХЕШУВАННЯ БЛОКЧЕЙНУ

3.1 Вимоги до системи

Для реалізації ефективної FPGA системи для хешування блокчейну, необхідно врахувати наступні вимоги:

- інтерфейсування коду. Система має включати первинні директиви для створення інтерфейсу в коді, що дозволить забезпечити правильне з'єднання компонентів;
- потокова обробка даних. Має бути забезпечена передача даних між функціями завантаження, обчислення і зберігання у формі потоку, замість обробки пакетами;
- паралелізм. Необхідно реалізувати використання конвеєра для максимізації пропускної здатності та забезпечення високої паралельності обчислень;
- оптимізація займаної площі. Логіка має бути адаптована до наявних ресурсів FPGA, з оптимальним використанням доступного простору;
- балансування. Потрібне точне налаштування системи для досягнення оптимальної продуктивності та ефективності.

Для забезпечення найкращої роботи системи необхідно врахувати:

- траєкторії переміщення даних, тобто необхідно зрозуміти, яким шляхом дані рухаються через мікросхему FPGA та як швидко це відбувається;
- взаємодію даних - важливо впевнитися, що зміни в даних не будуть негативно впливати на їх власний стан або стан інших даних у системі;
- використання ресурсів, тобто потрібно оцінити, скільки апаратних ресурсів необхідно для реалізації запланованого дизайну.

Важливим є те, що розробка на FPGA відбувається не тільки на рівні програмного забезпечення, але й включає фізичне створення схем на чіпі. Програмне забезпечення Vitis сприятиме ідентифікації можливих проблем в дизайні та допомагає в оптимізації ресурсів.

Для успішного створення визначення апаратних засобів HLS на мовах C/C++ необхідно забезпечити титаку структуру проєкту, де вся функціональність дизайну міститься в межах програми HLS, включаючи одну верхньорівневу функцію та відповідні підфункції чи бібліотеки. Також необхідно уникнути системних викликів до операційної системи, за винятком функцій введення/виведення, як то printf та cout.

Треба намагатися використовувати конструкції C/C++, які мають фіксований або обмежений розмір і є недвозначними, з чітко визначеними ресурсами без використання динамічної пам'яті.

3.2 Створення дизайну проєкту

На початку роботи створимо новий проєкт: File -> New -> Application Project.

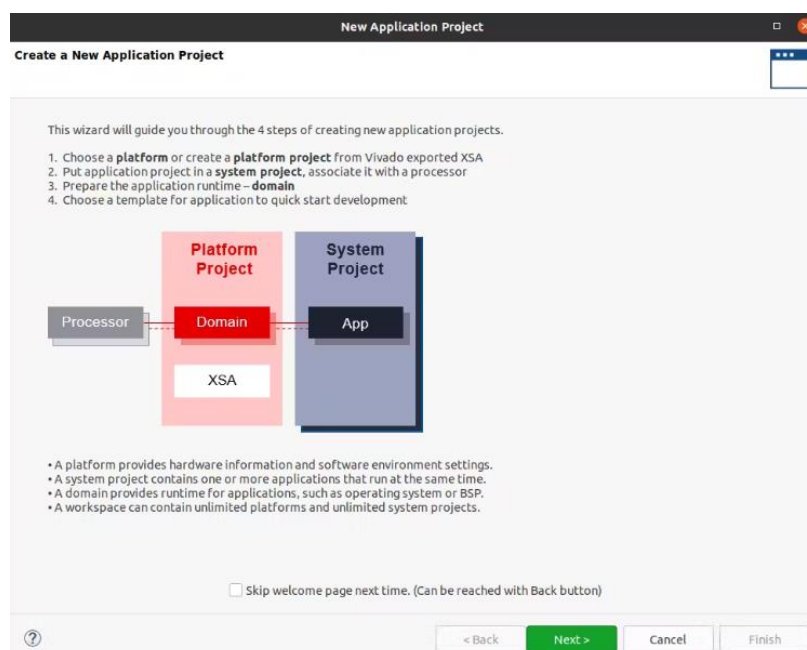


Рисунок 3.1 – Створення нового проєкту

Цей процес призведе до створення проекту з кількома рівнями. Хост-додаток, який запускається на процесорі, буде надсилати та отримувати дані від ядра, тоді як ядро-додаток, яке виконується на FPGA, забезпечуватиме прискорену обробку даних, а також здійснюватиме зв'язок між проектами. В цьому інтерфейсі налаштувань не передбачено.

Середовище розробки Vitis має визначати платформу, до якої спрямований проект. Для плати Varium C1100 вона позначається як u55n, із повною назвою: "xilinx_u55n_gen3x4_xdma_1_202110_1".

Ця платформа встановлюється в каталозі /opt/xilinx/platforms, якщо слідувати стандартним інструкціям встановлення.

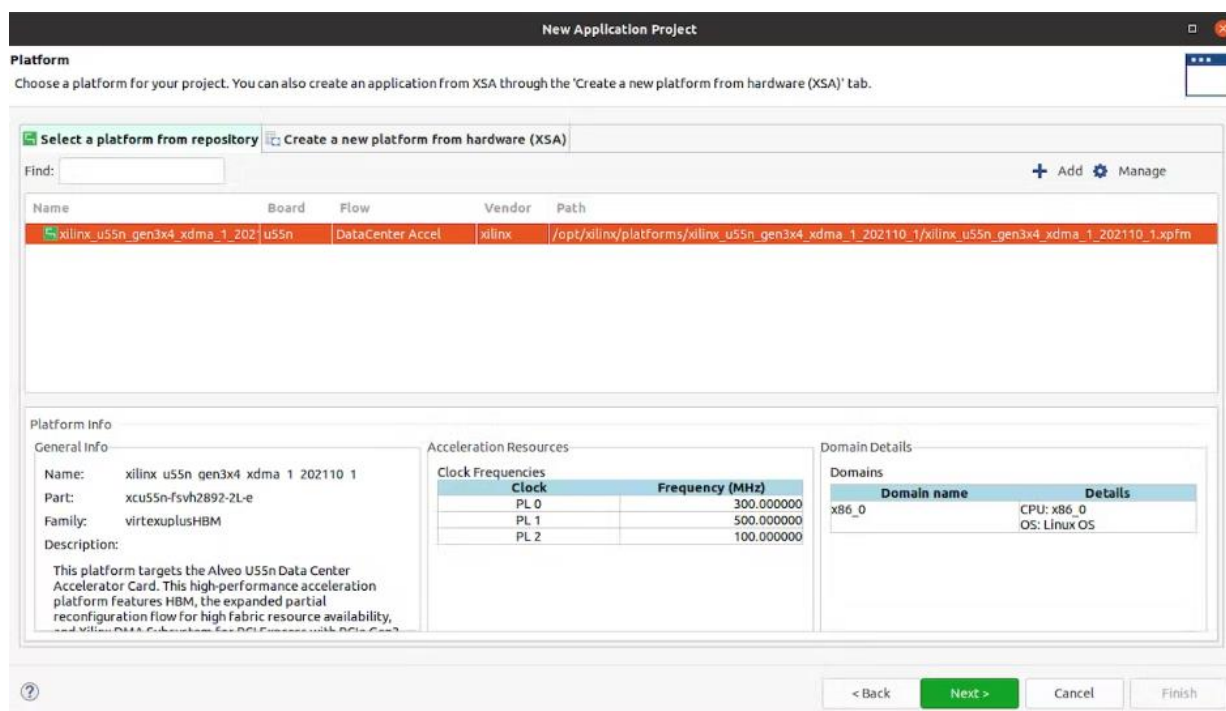


Рисунок 3.2 – Визначення платформи xilinx_u55n_gen3x4_xdma_1_202110_1

Натискаємо «Next».

Даємо проекту назву "Poseidon_Arity2" (рис. 3.3).

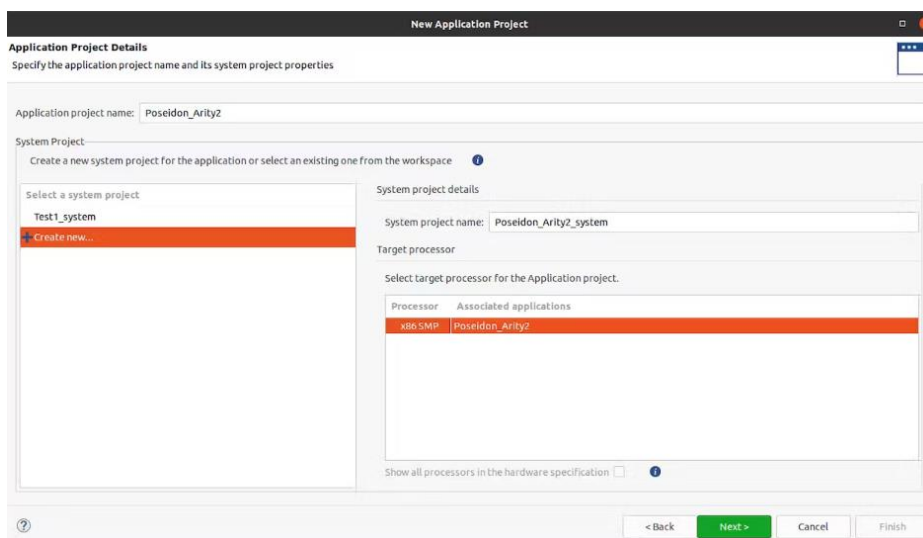


Рисунок 3.3 – Деталі проєкту

Натискаємо «Next».

Із шаблонів обираємо «Просте векторне додавання». За допомогою цього шаблону Vitis створить хост-додаток і невелике ядро (рис. 3.4).

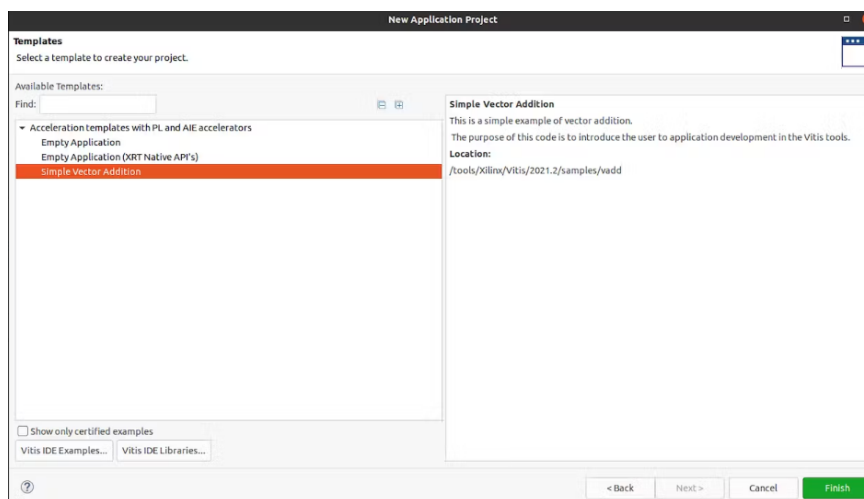


Рисунок 3.4 - Просте векторне додавання

Ми повинні отримати проєкт з трьома модулями, які представляють ядро, апаратний зв'язок та хост.

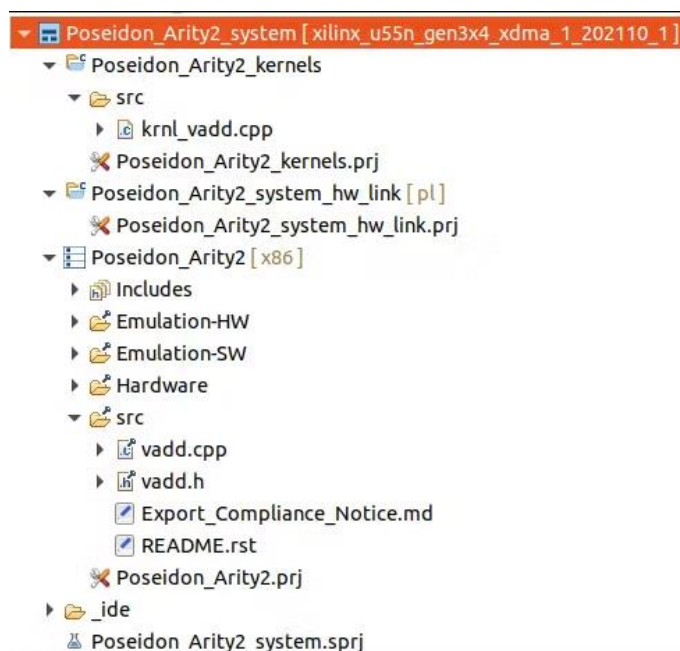


Рисунок 3.5 – Структура проєкту

3.3 Реалізація архітектури системи

Архітектура системи для хешування блокчейну на базі FPGA має бути розроблена з урахуванням вимог і особливостей Filecoin - інноваційної платформи для розподіленого зберігання даних. Filecoin.io використовує кооперативний підхід до зберігання та отримання даних, стимулюючи провайдерів використовувати невикористане сховище і винагороджуючи їх криптовалютою FIL. Система базується на IPFS (InterPlanetary File System) - протоколі, який дозволяє зберігати та отримувати доступ до файлів в межах розподіленої файлової системи.

Ключовим елементом, що визначає продуктивність системи Filecoin, є алгоритм хешування Poseidon, який застосовується для формування Меркле дерев. Спроможність цього алгоритму швидко обробляти нові файли є критичною для ефективності впровадження даних у блокчейн, а оптимізація його реалізації безпосередньо впливає на економічну вигоду, зменшуючи витрати на апаратне забезпечення та енергію.

В контексті проекту Neptune, Filecoin використовує додатковий підтримуючий код для прискорення оброблення за допомогою GPU. В рамках даного проекту ми зосередимося на перенесенні та адаптації коду Poseidon OpenCL для оптимізації його виконання на FPGA. Це вимагає детального розуміння математичного підґрунтя алгоритму, доступного на вебсайті специфікацій Filecoin, та адаптації його для роботи в умовах обмежених ресурсів FPGA.

Для ефективної обробки даних, блоки розділяються на сегменти по 32 байти та послідовно проходять через низку раундів обчислень (рис. 3.6). Кожен раунд включає виконання квінтичних S-боксів та множення на константи, визначені в матриці, над полем, використовуючи формулу Монтгомері для підвищення продуктивності за рахунок уникнення операцій ділення. Результатом є формування вузлів для фінального Меркле дерева, що є важливою частиною верифікації та безпеки в блокчейні.

Розробка архітектури системи повинна враховувати необхідність високої пропускної спроможності, мінімізацію латентності, оптимізацію використання ресурсів FPGA та забезпечення гнучкості системи для адаптації до змін у протоколі Filecoin. Основний фокус має бути на створенні модульної, масштабованої та ефективної системи, яка може бути легко оновлена або модифікована для відповідності майбутнім вимогам блокчейн технологій.

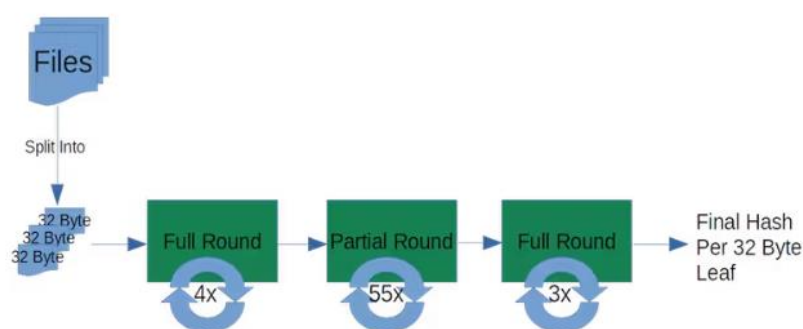


Рисунок 3.6 – Раунди обчислення

Кожен раунд складається з кількох квінтичних S-боксів (множення на x^5), якими слідує множення на кількість констант, знайдених у матриці. Все це

виконується над полем, використовуючи форму Монтгомері, щоб покращити продуктивність (уникаючи операцій ділення).

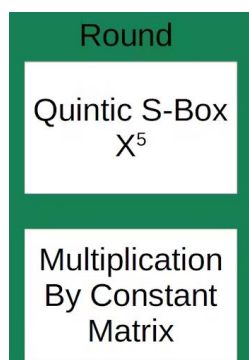


Рисунок 3.6 – Раунд

Створення проекту починається з аналізу та адаптації існуючого коду OpenCL, який був розроблений для проекту Filecoin/Neptune. Завдання полягає в тому, щоб перетворити оптимізовані для GPGPU операції в код, сумісний із архітектурою FPGA. Основна увага при цьому зосереджується на ефективному використанні ресурсів FPGA та адаптації алгоритмів для максимальної продуктивності.

Початковим кроком є ознайомлення з базовим кодом OpenCL з проекту Neptune, який доступний у розділі коду проекту "Baseline_OpenCL_from_Neptune.cl". Основне завдання полягає в спрощенні коду: видалення специфічного для виробника коду та зменшення непрямих звернень, щоб зрозуміти ключові елементи, які потрібно реплікувати та оптимізувати для FPGA.

Код поділяється на два основні розділи: перший, присвячений монтгомерійському модульному множенню, та другий, який займається реалізацією хешування Poseidon. Монтгомерійське модульне множення, зокрема, є важливою частиною алгоритму, яка вимагає точної адаптації для ефективної реалізації на FPGA.

На наступному етапі розробки ядра FPGA використовується мова програмування C++, яка дозволяє використовувати директиви High-Level Synthesis (HLS) для оптимізації та типи даних, специфічні для апаратних засобів. Це дозволяє більш гнучко управляти ресурсами FPGA та налаштовувати обробку даних з урахуванням їх особливостей.

Для початку створюється новий файл з назвою "krnl_poseidon_arity2.cpp" у директорії "Poseidon_Arity2_kernels/src". Цей файл стане основою для ядра, яке буде обробляти хешування за алгоритмом Poseidon. Крок за кроком у цьому файлі будуть розроблятися різні секції ядра, кожна з яких буде ретельно оптимізована для виконання специфічних завдань, передбачених алгоритмом.

Ключовим аспектом цієї роботи є використання HLS для контролю над конвеєрною обробкою та іншими аспектами синтезу, що дозволить максимально ефективно використовувати обмежені ресурси FPGA. Кожен крок розробки буде уважно перевірятися на предмет відповідності вимогам ефективності та продуктивності, встановленим для системи хешування блокчейну.



Рисунок 3.7 - Створення нового файлу з назвою "krnl_poseidon_arity2.cpp"

Створення проекту FPGA для алгоритму хешування Poseidon починається з визначення ключових компонентів та структури системи, яка відповідає API блокчейну Filecoin із використанням проекту Neptune.

Функція точки входу. Необхідно створити функцію `poseidon_hash(...)`, яка служитиме входом для обробки даних. Ця функція має відповідати API, заданому

в проєкті Neptune, але з додатковими оптимізаціями для потоку даних. Вона буде ядром вищого рівня, до якого Vitis IDE буде підключати хост-додаток.

Обгортка функції. Згідно з вимогами Vitis HLS, декларація функції верхнього рівня ядра має бути обгорнута зовнішнім "C", щоб забезпечити сумісність з різними мовами програмування та інструментами розробки.

Структура даних. Створюємо структуру Fr, яка буде використовуватися для передачі даних між хостом і FPGA. Ця структура забезпечує узгодженість із визначенням API, що використовується Neptune, та зберігає дані у відповідному форматі.

Внутрішня структура даних. Потрібно створити структуру `state_2_standard`, яка служитиме для внутрішнього зберігання стану даних під час виконання алгоритму.

Функції потоку даних. Налаштування функцій, які відповідатимуть за поточкову обробку даних, зокрема завантаження, обчислення та збереження результатів.

Мітки циклів. Для кращого аналізу та зрозумілості коду при дебагінгу та оптимізації через Vitis HLS необхідно додати мітки до циклів, наприклад `store_result::`.

Розробка хост-програми. Створення хост-програми, яка буде управляти взаємодією між системою та акселератором, використовуючи OpenCL. Структура хост-програми буде схожа на існуючий приклад `vadd.cpp`.

Створення основного файлу. Копіювання `vadd.cpp` з подальшим перейменуванням у `host_poseidon.cpp` та адаптацією файлу до специфіки проєкту.

Включення точних типів даних. В коді буде використано `#include "ap_int.h"` для підтримки точних типів даних, необхідних для обробки на FPGA.

Налаштування розміру даних: Визначення розміру даних для розробки, `static const int DATA_SIZE = 2`, для простоти тестування і перевірки функціональності.

Визначення ядра OpenCL. При створенні визначення ядра OpenCL необхідно зазначити назву функції точки входу, що була розроблена раніше, з використанням синтаксису OCL_CHECK для обробки помилок.

Змінюємо вхідні та вихідні змінні на ті, які потрібні для функції точки входу ядра.

В коді типового процесу використання OpenCL для програмування FPGA:

- перевіряється введення користувача та встановлюються шляхи до необхідних файлів;
- відбувається перебір доступних платформ та вибір цільової платформи;
- створюється контекст і командна черга;
- зчитується та завантажується програма (ядро) з файлу .xclbin;
- виділяється пам'ять на пристрої та передаються аргументи до ядра;
- виконується ядро та оцінюються результати.

Оскільки створюється власне ядро, то потрібно повідомити Vitis про нову функцію входу для ядра:

- відкриваємо файл "Poseidon_Arity2_kernels.prj";
- видаляємо апаратну функцію "krnl_vadd";
- натискаємо "Add Hardware Function" та виберіть "poseidon_hash".



Рисунок 3.8 - Add Hardware Function

3.4 Емуляція

3.4.1 Програмна емуляція

Основним (і найшвидшим) способом перевірки коректності логіки є за допомогою програмної емуляції. Це також найшвидший метод компіляції, тому рекомендується виявляти проблеми саме під час програмної емуляції.

Програмна емуляція буде виконувати код проти системного процесора CPU. Певні специфічні для апаратури оптимізації, як-от Dataflow, будуть аналізуватись під час виконання.

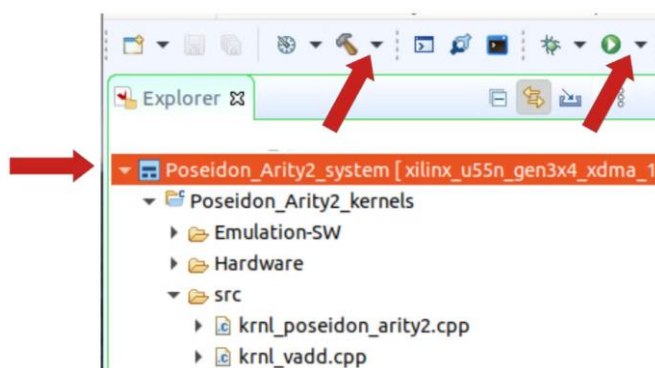


Рисунок 3.9 – Програмне емуляція

Вибір проєкту призведе до збірки всіх підмодулів. Якщо обрати лише один підмодуль, тоді буде зібрано тільки цей підмодуль.

3.4.2 Емуляція апаратного забезпечення

Після того, як наш код працює правильно, ми переходимо до оптимізації апаратних засобів. Тепер всі оптимізації набувають сили. Vitis будує RTL-дизайн, і коли запускається логіка, цей RTL-дизайн (на відміну від коду) є основою для виконання та виводу.

Журнали процесу збірки показують, як застосовуються оптимізації, помилки та деяку інформацію про ресурси, наприклад, скільки множників або м'ютексів буде створено.

3.4.3 Аналіз

Детальніший аналіз оптимізації апаратних засобів проводиться через додаток Vitis HLS, який запускається з IDE Vitis:

- будується, використовуючи "Emulate-HW";
- відкривається файл "Poseidon_Arity2_kernels.prj";
- вибирається апаратну функцію "poseidon_hash";
- "Launch Vitis HLS...";
- "OK" у вікні "Launch Vitis HLS";
- відкривається додаток Vitis HLS.

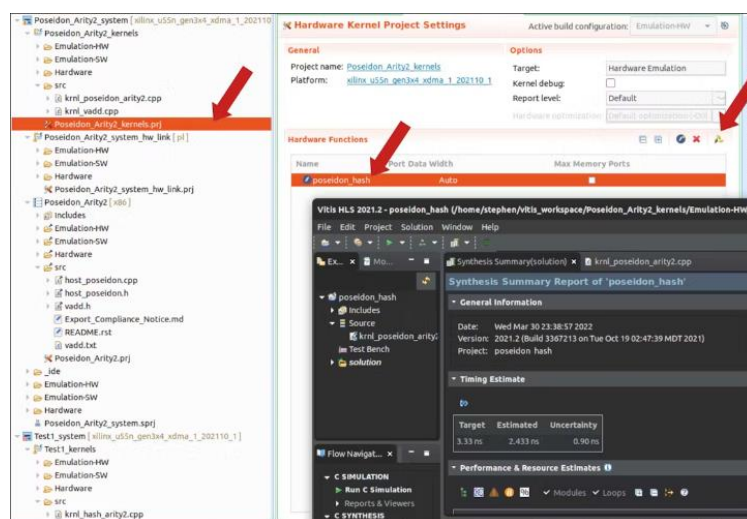


Рисунок 3.10 - Аналіз оптимізації апаратних засобів

3.4.4 Редагування та синтез кернела

Синтез кернела (kernel synthesis) - це процес перетворення високорівневого опису алгоритму (наприклад, написаного на мовах C або C++) в апаратний опис на рівні RTL (Register Transfer Level). У цьому контексті, "кернел" зазвичай відноситься до центральної частини програми або алгоритму, яка підлягає прискоренню за допомогою FPGA.

Запуск C синтезу (Емуляція-HW) дозволяє перетворити високорівневий код на RTL (Register Transfer Level), який можна буде емулювати на апаратному рівні.

Запуск C-синтезу (Емуляція-HW) дозволяє перетворити високорівневий код на RTL (Register Transfer Level), який можна буде емулювати на апаратному рівні.

Існує можливість переглянути кілька типів звітів, які надасть Vitis HLS після синтезу.

Можна перевірити, з якою швидкістю було оцінено таймінг годинника. Це допоможе зрозуміти, чи може ядро працювати з необхідною частотою.

Можна подивитися на показники продуктивності, такі як затримка, інтервал (II - iteration interval), чи були секції розроблені з конвеєром, та кількість використаних ресурсів.

Можна ознайомитися з виводом вкладки "Guidance" (журнал виводу від компілятора), який може дати рекомендації по подальшій оптимізації.

Потім необхідно повернутися до ядра, щоб провести оптимізацію. Можна внести зміни, щоб поліпшити таймінг, зменшити затримку, покращити конвеєрність або зменшити використання ресурсів.

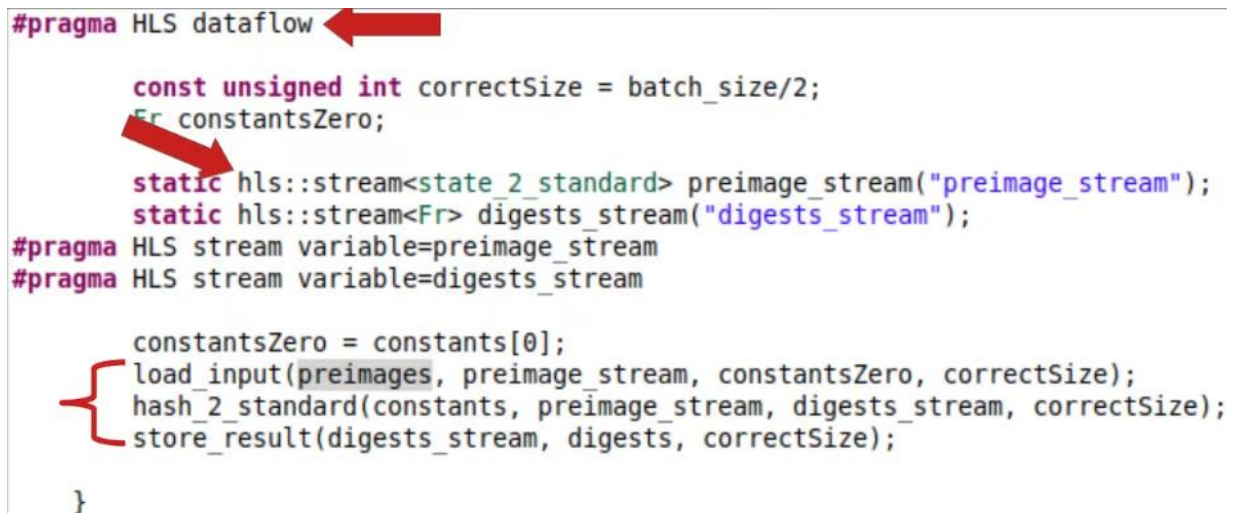
Будемо використовувати інформацію з отриманих звітів, щоб ідентифікувати, де можна здійснити оптимізацію, і які функції можуть бути застосовані для цього.

Після внесення змін, повторюється процес синтезу, щоб перевірити вплив оптимізацій.

Оптимізація ядра - це ітеративний процес, який може вимагати декількох циклів редагування, синтезу та аналізу, щоб досягти необхідних показників продуктивності.

Перша оптимізація, яку слід застосувати— це dataflow (потокowa обробка даних). Потокowa обробка дозволяє передавати дані між функціями у вигляді потоку (на відміну від партій). З потоковою обробкою не потрібно чекати завершення обробки всіх даних — кожен елемент даних передається наступному споживачу, як тільки його виробляє виробник. Функції, що беруть участь у потоковій обробці, використовують парадигму виробник-споживач, яка налаштовує канали між функціями, дозволяючи апаратному забезпеченню починати обробляти дані, як тільки вони стають доступними.

Vitis рекомендує налаштувати функції завантаження-обчислення-збереження та застосувати потокову обробку між ними. Саме тому було вказано іншу точку входу в ядро "knl_poseidon_arity2.cpp", щоб можна було налаштувати потокову обробку. Функція `poseidon_hash(...)` використовується для обробки потоку даних. `hls::stream<...>` визначає канал між функціями, а три функції внизу представляють завантаження-обчислення-збереження.



```
#pragma HLS dataflow

const unsigned int correctSize = batch_size/2;
Fr constantsZero;

static hls::stream<state_2_standard> preimage_stream("preimage_stream");
static hls::stream<Fr> digests_stream("digests_stream");
#pragma HLS stream variable=preimage_stream
#pragma HLS stream variable=digests_stream

constantsZero = constants[0];
{
    load_input(preimages, preimage_stream, constantsZero, correctSize);
    hash_2_standard(constants, preimage_stream, digests_stream, correctSize);
    store_result(digests_stream, digests, correctSize);
}
```

Рисунок 3.11 - Директива `#pragma HLS dataflow`

Директива `#pragma HLS dataflow` може бути визначена у функції або циклі.

Також необхідно враховувати певні обмеження потокової обробки:

- обмеження щодо читання з входів функції або запису в виходи функції всередині області потокової обробки;
- повинен бути підтриманий шаблон одного виробника-споживача;
- оминаючи завдання та розмірність каналів можуть знизити продуктивність;
- обмеження щодо зворотного зв'язку між завданнями;
- умовне виконання завдань не може бути використане;
- цикли з кількома умовами виходу не можуть бути використані.

З огляду на вузький скоуп (скоуп визначає область коду, де змінна або інша сутність програми є відомою і може бути доступною) завантаження-обчислення-збереження та те, що потокова обробка не є ієрархічною (не застосовується до підфункцій), не виникає проблеми з порушенням цих обмежень.

Змінна, що використовується в дизайні, може бути збережена в пам'яті або регістрах у дизайні RTL. Кожен з них має обмежену кількість портів читання і запису, і тому обмежену пропускну здатність до цього зберігання.

Масив не є винятком. Незважаючи на те, що він зберігає багато даних, уся структура може бути доступна лише як одне місце пам'яті.

Цей алгоритм покладається на масив `Fr constants[...]`, що використовується в кількох місцях логіки. Щоб уникнути затору на цьому масиві, потрібно зробити його таким, щоб він здавався для RTL набором невеличких місць пам'яті.

До цього часу було використано типи даних C++, такі як `unsigned long long` та `int`, але ми можемо бути більш точними щодо розміру цих змінних. У апаратному забезпеченні кожен зайвий біт означає додатковий дріт і місце пам'яті, яке має бути вбудоване в дизайн. Крім того, кожен додатковий біт може вимагати використання більшої кількості ресурсів, таких як DSP (цифрова обробка сигналів) або LUT (таблиця пошуку), для операцій над даними.

Описана структура використовує масив із 4 64-бітних значень через обмеження хоста, який може обробляти лише до `unsigned long long`. Таким чином, число, яке представляє ця структура, тоді є $\text{val}[0] + (\text{val}[1] \ll 64) + (\text{val}[2] \ll 128) + (\text{val}[3] \ll 192)$.

Вказана FPGA може обробляти повні 256-бітні числа. Це спрощує деякі арифметичні операції, які раніше потрібно було виконувати над частинами, а тепер можна виконувати над простими числами. Це також може підвищити продуктивність за рахунок меншої кількості циклів та ресурсів.

Для того, щоб враховувати обмеження хоста, створюємо `Fr_4Limb` і модифікуємо існуючий `Fr`, щоб використовувати `ap_uint<256>`.

Також переходимо від формату масиву `little endian` (найменш значущі дані зліва) до формату числа `big endian` (найменш значущі дані справа).

Коли зберігаємо дані для відправлення назад хосту, знову потрібно враховувати обмеження хоста. Цього разу ми використовуємо вибір діапазону щоб вибрати 64 біти з більшого числа для створення кожного сегмента.

Модульне множення Монтгомері є поліпшенням продуктивності, яке уникає операції ділення при обчисленні $a * b \bmod N$. Числа перетворюються у форму Монтгомері перед застосуванням арифметичних операцій, а потім повертаються до звичайної форми, коли логіка завершується.

Розглядаючи оригінальний (спрощений) код OpenCL, ми бачимо логіку навколо обробки сегментів (лімбів). У функціях для додавання та віднімання необхідно ітерувати через чотири сегменти і враховувати або переносити по ходу.

Оскільки FPGA може обробляти 256-бітне число, можемо виключити цю складність і просто додавати або віднімати числа.

Множення також стає трохи простішим, замість подвійно вкладеного циклу над лімбами.

Використання менших чисел у множенні безпосередньо впливає на зменшення кількості DSP, необхідних для завершення множення. Знаходження балансу між цією формою з додатковими множеннями та прямим 256-бітним множенням (результат 512 біт) є частиною процесу оптимізації.

Аналіз за допомогою Vitis HLS показує дуже низьке використання ресурсів (DSP, FF, LUT), але ми ще не реалізовано основну частину алгоритму.

Module	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
poseidon_hash			0.00	-	-	-	-	-	-	dataflow	62	0	8462	8300	0
entry_proc			-	-	0	0.0	-	0	-	no	0	0	3	37	0
Block_split7_proc			-	-	0	0.0	-	0	-	no	0	0	11	68	0
transform_constants7			0.00	-	448	1.493E3	-	448	-	no	0	0	663	816	0
transform_constants7_Pipeline_transform_constants			0.00	-	376	1.253E3	-	376	-	no	0	0	529	318	0
transform_constants			-	-	374	1.247E3	-	3	1	373	yes	-	-	-	-
load_input			0.00	-	586	1.963E3	-	586	-	no	0	0	662	637	0
load_input_Pipeline_load_input			0.00	-	514	1.713E3	-	514	-	no	0	0	527	95	0
load_input			-	-	512	1.706E3	-	3	1	511	yes	-	-	-	-
hash_2_standard			0.00	-	-	-	-	-	-	no	0	0	26	168	0
hash_2_standard_Pipeline_compute			-	-	-	-	-	-	-	no	0	0	12	93	0
compute			-	-	-	-	-	2	1	yes	-	-	-	-	-
store_result			0.00	-	-	-	-	-	-	no	0	0	414	653	0
store_result_Pipeline_store_result			0.00	-	-	-	-	-	-	no	0	0	271	97	0
store_result			-	-	-	-	-	3	1	yes	-	-	-	-	-

Рисунок 3.12 - Аналіз за допомогою Vitis HLS

Poseidon Hash Code. Найпростішим способом реалізації хешувального коду є переклад з OpenCL в C++ .

Виконання емуляції програмного забезпечення пройшло перевірку за допомогою зразка золотих даних, справжнім випробуванням є те, наскільки швидкою та ефективною є логіка, коли ми аналізуємо емуляцію апаратного забезпечення.

Аналіз. Стовпець "Iteration Latency" показує, скільки часу потрібно для завершення однієї ітерації логіки. Цей код виконується за 55 654 такти годинника на одну порцію даних.

"Latency(cycles)" показує, скільки часу потрібно, щоб завершити всі дані для тієї функції. Враховуючи, що `hash_2_standard` є нашою обчислювальною функцією, вона є основним драйвером продуктивності. Цей код виконуватиме 1000 хешів за 55 654 001 такт. Годинник з частотою 300 МГц (300 000 000 тактів за секунду) означатиме, що ми завершуємо 1000 хешів за 0.185 секунди. Це досить повільно.

Є деякі функції з конвеєрною обробкою, які Vitis змогла застосувати автоматично.

При реалізації програми на FPGA логіка стає мережею дротів, які з'єднують між собою таблиці пошуку (LUT), тригери (FF), цифрові сигнальні процесори (DSP) та інші компоненти усередині чіпа. Дані в кожен момент часу займають лише невелику частку цього апаратного забезпечення. Уявляючи наш FPGA як фабрику з внутрішньою конвеєрною лінією, ми прагнемо завантажувати нові дані на конвеєр якнайшвидше після того, як попередній набір даних переходить до наступної фази обробки. Така одночасна обробка численних елементів даних вздовж шляху схеми підвищує ефективність і забезпечує високу ступінь паралелізму.

Ідеальний сценарій полягає в досягненні інтервалу ітерації рівного 1, що означає обробку нового шматка даних на конвеєрі з кожним тактом годинника. На практиці потрібно знайти баланс між паралелізмом та доступністю ресурсів.

З поточним кодом просте застосування конвеєру на вході функції скоротить затримку лише наполовину, вимагаючи при цьому в 4.3 рази більше DSP, ніж є доступними у заданій FPGA.

Почнемо з переосмислення способу написання коду, щоб його конвеєрна обробка була ефективнішою.

Зменшення шляхів. Логіка перетворюється на схемні шляхи, чим більше варіацій логіки ми маємо, тим більше схемних шляхів потрібно створити. Ця конструкція `if/else if/else` призводить до створення трьох схемних шляхів крізь чіп, але дані завжди потрапляють лише на один з шляхів, залишаючи інші два невикористаними.

Якщо переписати це так, щоб об'єднати `apply_matrix_2_standard(...)`, то можемо отримати вищу утилізацію та, отже, вищу ефективність від схеми.

У `hash_2_standard` є декілька проблем:

Застосування `#pragma HLS pipeline` на початку функції вимагає надто великих ресурсів.

`#pragma HLS pipeline` є директивою в мові програмування C, яка використовується в контексті високорівневого синтезу (HLS) для вказівки компілятору на необхідність оптимізації певної ділянки коду за допомогою техніки, відомої як "пайплайнінг" (pipeline). Ця директива в основному застосовується при розробці програмного забезпечення для FPGA (полів програмованих вентилів) та інших апаратних пристроїв.

Пайплайнінг у контексті HLS дозволяє підвищити продуктивність шляхом паралельного виконання декількох етапів обчислення. У традиційному пайплайні кожен етап обробки даних виконується в окремому такті годинника, що дозволяє одночасно обробляти декілька різних наборів даних на різних стадіях.

Коли програміст додає `#pragma HLS pipeline` до ділянки коду, він або вона інструктує компілятор HLS організувати цю частину коду як пайплайн. Це може бути корисним для оптимізації циклів, наприклад, щоб кожна ітерація циклу виконувалася паралельно з іншими, значно підвищуючи продуктивність.

Застосування `#pragma HLS pipeline` на початку кожного циклу допомагає конвеєрно обробляти цикл, але для виходу з циклу потрібно, щоб конвеєр спорожнів. Враховуючи, що `dataflow_loop` не є конвеєрним, бачимо, що лише один шматок даних переміщується по всьому конвеєру в будь-який момент часу.

Бачимо неефективність дублювання шляхів з `full_round_2_standard`, який з'являється у двох місцях, а також `last_full_round_2_standard`, який дуже схожий на повний раунд.

Виправимо спочатку дублювання. Кожен повний, частковий та останній раунд дуже схожі один на одного. І вставимо функції, які чинять опір конвеєрній обробці в нашому поточному коді. Вказуємо `#pragma HLS inline` для:

```
scalar_product
apply_sparse_matrix_2_standard
apply_round_matrix_2_standard
```

Спостерігаємо значне покращення продуктивності з інтервалу ітерації $P=1$. Тепер ми відправляємо новий шматок даних через конвеєр кожного циклу. Якби FPGA працював з частотою 300 МГц, необхідно було б обчислювали 300 мільйонів раундів за секунду; з 65 раундами на кожну порцію даних, можна будл б обробляли приблизно 4,6 мільйона повних хешів за секунду.

Але є нюанс: порушено баланс між паралелізмом та займаною площею. При натисканні кнопки "Show Resource Utilization Percentage", бачимо, що використовується 182% доступних DSP в FPGA. DSP інтенсивно використовуються функціями `apply_matrix_2_standard` і `quintic_s_box`. Щоб зменшити використання ресурсів, потрібно збільшити інтервали ітерації і потенційно обмежити використання певних ресурсів у логіці. Застосуємо `#pragma HLS allocation` наступним чином: накажемо Vitis скоротити деякі з алокованих схем і замість цього повторно використовувати створені. Побічним ефектом є подовження інтервалу ітерації до 12.

З цією оптимізацією ми обробляємо новий раунд кожні 12 тактів годинника. З 65 раундами на кожну порцію даних, ми завершуємо хеш кожні 780 тактів. Якщо наш FPGA працює з частотою 300 МГц, то ми обробляємо 384 615 хешів за секунду.

3.5 Висновки до розділу 3

В розділі наголошується на важливості детальної розробки та оптимізації FPGA системи для хешування блокчейну. Система має включати ефективне інтерфейсування коду, забезпечувати потокову обробку даних, підтримувати високий рівень паралелізму, оптимізувати використання простору на FPGA та забезпечити балансування для оптимальної продуктивності. Важливо розуміти шляхи переміщення даних в мікросхемі, а також забезпечувати взаємодію даних і правильне використання ресурсів. Розробка на FPGA включає як програмування, так і фізичне створення схем на чіпі.

Також в цьому розділі подається детальний опис створення дизайну проекту, починаючи з створення нового проекту в середовищі розробки Vitis, вибору платформи та структури проекту. Описується процес реалізації архітектури системи, з орієнтацією на оптимізацію обробки даних для Filecoin, використовуючи алгоритм хешування Poseidon. Особлива увага приділяється трансформації існуючого коду OpenCL для оптимізованого виконання на FPGA. Розробка ядра FPGA включає створення точки входу, обгортки функції, структури даних, функцій потоку даних та належного використання HLS директив для оптимізації обробки даних.

Після успішного кодування та оптимізації коду, процес переходить до стадії створення RTL-дизайну. У цій фазі відбувається перевірка та застосування оптимізацій на апаратному рівні, що дозволяє отримати детальне уявлення про ресурси та можливі помилки. Журнали процесу збірки відіграють ключову роль у цьому аналізі, надаючи інформацію про використання ресурсів та ефективність оптимізацій.

На стадії аналізу відбувається аналіз оптимізації апаратних засобів проводиться через спеціалізоване програмне забезпечення Vitis HLS. Це дає можливість поглянути під капот процесу емуляції та оцінити, наскільки ефективно були застосовані зміни в коді. Використання додатку Vitis HLS ініціюється через

IDE Vitis, де можна аналізувати різні аспекти оптимізації, включаючи продуктивність, використання ресурсів та потенційні місця для подальшого поліпшення.

Синтез кернела включає перетворення високорівневих описів алгоритмів на апаратні описи на рівні RTL. Цей процес починається з виконання C-синтезу, який перетворює код на RTL, придатний для емуляції на апаратному рівні. Отримані звіти від Vitis HLS після синтезу надають цінну інформацію про продуктивність, використання ресурсів та потенційні напрямки для оптимізації. На основі цих даних вносяться зміни в код для поліпшення продуктивності, зниження затримки та оптимізації використання ресурсів. Весь процес є ітеративним та вимагає кількох циклів редагування, синтезу та аналізу для досягнення оптимальних результатів.

В цілому ця робота підкреслює необхідність ретельного планування та врахування багатьох технічних аспектів при створенні ефективної FPGA системи для хешування блокчейну, включаючи використання специфічних можливостей FPGA для оптимізації обробки даних та забезпечення високої продуктивності системи.

ВИСНОВКИ

Основна мета роботи полягала у створенні високоефективної та надійної FPGA-системи для хешування в контексті блокчейн технологій. Робота підкреслює важливість технології блокчейн у сучасному цифровому світі, особливо акцентуючи на необхідності ефективного хешування для забезпечення безпеки і надійності блокчейн-систем.

В роботі проведено глибокий аналіз апаратного забезпечення, зокрема AMD Varium C1100 і AMD Ryzen 3700x, а також програмного забезпечення, включаючи Ubuntu 20.04, AMD Vitis Unified Software Platform, і AMD Xilinx AMD-Xilinx Run Time (XRT) для Ubuntu 20.04. Магістерська робота детально описує процес розробки системи, включаючи створення дизайну, емуляцію, аналіз та оптимізацію. В кінцевому підсумку, розроблена FPGA-система для хешування демонструє значний потенціал для підвищення ефективності та безпеки в блокчейн мережах.

Інтеграція різних компонентів апаратного і програмного забезпечення, включаючи FPGA-картки, процесори, операційну систему Ubuntu і розробницькі платформи, як AMD Vitis, є ключовою для досягнення бажаного рівня продуктивності та ефективності. Процес розробки включав детальне планування, емуляцію апаратного забезпечення та оптимізацію, що дозволило створити надійну і ефективну систему.

Результати дослідження показали, що адаптивні можливості FPGA пропонують значні переваги для обробки блокчейн-завдань, особливо у сфері хешування, що є критично важливим для забезпечення безпеки та цілісності даних. Емуляція і тестування системи виявили її високу продуктивність та адаптивність, підкреслюючи потенціал FPGA у вирішенні складних задач хешування в блокчейн.

У підсумку, ця магістерська робота відображає важливість інноваційного підходу до розробки та застосування FPGA в блокчейн-технологіях. Показано, що збалансоване поєднання теорії та практики, а також інтеграція різних компонентів

та технологій може привести до створення високоефективних, надійних і гнучких систем, які зможуть забезпечити нові можливості для блокчейн-спільноти та інших галузей.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Nakamoto, Satoshi. "Bitcoin: A Peer-to-Peer Electronic Cash System" [Електронний ресурс]. bitcoin.org. 31 жовтня 2008. Архівовано з оригіналу (PDF) 20 березня 2014. URL: (дата звернення: 30.11.2023).
2. Dwork, Cynthia; Naor, Moni. Pricing via processing or combatting junk mail // CRYPTO 1992: Advances in Cryptology — CRYPTO' 92. 1992. С. 139-147.
3. Russell, Alexander et al. "Forkable Strings are Rare" [Електронний ресурс]. In: Cryptology ePrint Archive, Report 2017/241, (2017). DOI: <http://eprint.iacr.org/2017/241> (дата звернення: 30.11.2023).
4. Kovalchuk, Lyudmila et al. Analysis of Splitting Attacks on Bitcoin and GHOST Consensus Protocols // Proc. of the 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems. Romania, Bucharest, 2017.
5. Sompolinsky, Yonatan; Zohar, Aviv. "Accelerating bitcoin as transaction processing. fast money grows on trees, not chains" [Електронний ресурс]. In: IACR Cryptology ePrint Archive 881 (2013). DOI: <http://eprint.iacr.org/2013/881.pdf> (дата звернення: 30.11.2023).
6. Rosenfeld, Meni. "Analysis of hashrate-based double-spending" [Електронний ресурс]. In: arXiv preprint (2014). DOI: <https://arxiv.org/abs/1402.2009> (дата звернення: 30.11.2023).
7. Pinzon, Carlos; Rocha, Camilo. "Double-Spend Attack Models with Time Advantage for Bitcoin". In: Electronic Notes in Theoretical Computer Science. Том 329. 2016. С. 79–103.
8. Grunspan, Cyril; Pérez-Marco, Ricardo. "Double spend races" [Електронний ресурс]. CoRR abs/1702.02867 (2017). URL: <http://arxiv.org/abs/1702.02867> (дата звернення: 30.11.2023).
9. Feller, William. An Introduction to Probability Theory and Its Applications, Volume 2. 2nd Edition. 1991. 704 с.

10. "Bitcoin confirmed transactions per day stats" [Электронный ресурс]. 2018. URL: <https://www.blockchain.com/charts/n-transactions> (дата звернения: 30.11.2023).
11. Popov, Serguei. "The Tangle" [Электронный ресурс]. 2017. URL: https://iota.org/IOTA_Whitepaper.pdf (дата звернения: 30.11.2023).
12. Fitzi, Matthias; Gaʻzi, Peter; Kiayias, Aggelos; Russell, Alexander. "Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition" [Электронный ресурс]. Cryptology ePrint Archive, Report 2018/1119, 2018. URL: <https://eprint.iacr.org/2018/1119> (дата звернения: 30.11.2023).
13. Bagaria, Vivek; Kannan, Sreeram; Tse, David; Fanti, Giulia; Viswanath, Pramod. "De-constructing the blockchain to approach physical limits" [Электронный ресурс]. arXiv preprint arXiv:1810.08092, 2018.
14. Dilley, J.; Poelstra, A.; Wilkins, J. "Unfreezable blockchain" [Электронный ресурс]. bitcoin forum. 2016. URL: <https://bitcointalk.org/index.php?topic=57647.msg686497#msg68649> (дата звернения: 30.11.2023).
15. Lerner, Sergio Demian. "dagcoin: a cryptocurrency without blocks" [Электронный ресурс]. 2015.
16. Baird, Leemon. "Hashgraph consensus: fair, fast, Byzantine fault tolerance" [Электронный ресурс]. swirlds tech report. 2016. URL: <http://www.swirlds.com/wp-content/uploads/2016/06/2016-05-31-Swirlds-Consensus-Algorithm-TR-2016-01.pdf> (дата звернения: 30.11.2023).
17. Jia, Yaoqi. "Demystifying hashgraph: Benefits and challenges" [Электронный ресурс]. hackernoon.com. URL: <https://hackernoon.com/demystifying-hashgraph-benefits-and-challengesd605e5c0cee5> (дата звернения: 30.11.2023).
18. Butt, Kyle; Sorensen, Derek; Stay, Michael. "Casanova" [Электронный ресурс]. In: arXiv preprint (2018). URL: <http://arxiv.org/abs/1812.02232> (дата звернения: 30.11.2023).
19. Sompolinsky, Yonatan; Zohar, Aviv. "Phantom" [Электронный ресурс]. IACR Cryptology ePrint Archive, Report 2018/104 (2018).

20. Kovalchuk, Lyudmila; Kaidalov, Dmytro; Nastenka, Andrii; Rodinko, Mariia; Shevtsov, Oleksiy; Oliynykov, Roman. "Decreasing security threshold against double spend attack in networks with slow synchronization". Computer Communications. Том 154. 2020. С. 75–81.

21. Кудін, А.М.; Ковальчук, Л.В.; Коваленко, Б.А. "Теоретичні засади та застосування блокчейн технологій: імплементація нових протоколів консенсусу та краудсорсінг обчислень". Математичне та комп'ютерне моделювання. Вип. 19. 2019р. С. 62-68.

22. "ASIC" [Електронний ресурс]. CoinMarketCap. URL: <https://coinmarketcap.com/academy/uk/glossary/asic> (дата звернення: 30.11.2023).

23. "ASIC-майнери – ефективний та швидкий видобуток криптовалют" [Електронний ресурс]. Тернополяни. URL: <https://ternopoliany.te.ua/biznes-ta-finansy/82785-asicmainery-efektyvnyi-ta-shvydkyi-vydobutok-kryptovaliut> (дата звернення: 30.11.2023).

24. "ASIC" [Електронний ресурс]. PC Praha. URL: <https://pcpraha.cz/uk/asic/> (дата звернення: 30.11.2023).

25. "Різниця між алгоритмами хешування SHA-1, SHA-2, SHA-256" [Електронний ресурс]. E-Centrum. URL: <https://e-centrum.eu/uk/B8-sha-1-sha-2-sha-256/> (дата звернення: 30.11.2023).

26. "What is Keccak?" [Електронний ресурс]. BitDegree. URL: <https://www.bitdegree.org/crypto/learn/crypto-terms/what-is-keccak> (дата звернення: 30.11.2023).

27. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J. "The Skein Hash Function Family" [Електронний ресурс]. URL: <http://www.skein-hash.info/sites/default/files/skein1.3.pdf> (дата звернення: 30.11.2023).

28. Aumasson, J.-P., Çalik, Ç., Meier, W., Özen, O., Phan, R. C.-W., Varıcı, K. "Improved Cryptanalysis of Skein". Tokyo, Japan: International Conference on the Theory and Application of Cryptology and Information Security ASIACRYPT, 2009. С. 542-559.

29. Goldhamer, J. "SHA-3 Submission: Skein Hash Function Family" [Электронный ресурс]. Insight on skein hash function proposal. URL: <https://koclab.cs.ucsb.edu/teaching/cren/project/2010/goldhamer.pdf> (дата звернення: 30.11.2023).

30. Bellare, M., Kohno, T., Lucks, S., Ferguson, N., Schneier, B., Whiting, D., Callas, J., Walker, J. "Provable Security Support for the Skein Hash Family" [Электронный ресурс]. URL: <http://www.skein-hash.info/sites/default/files/skein-proofs.pdf> (дата звернення: 30.11.2023).

31. Mouha, N., Raunak, M.S., Kuhn, D.R., Kacker, R. "Finding Bugs in Cryptographic Hash Function Implementations". IEEE Transactions on Reliability, 2018. T. 99. С. 1-15.

32. Wang, D., Jiang, Y., Song, H., He, F., Gu, M., Sun, J. "Verification of Implementations of Cryptographic Hash Functions". IEEE Access, 2017. T. 5. С. 7816-7825.

33. "Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications". Special Publication 800-22 Revision 1a. NIST, 2010.

Glassnode Studio. "Bitcoin: Block Height" [Электронный ресурс]. 2022. URL: <https://studio.glassnode.com/metrics?a=BTC&category=&m=blockchain.BlockHeight&s=1230508800&u=1659484800&zoom=> (дата звернення: 30.11.2023).

34. Glassnode Studio. "Bitcoin: Market Cap" [Электронный ресурс]. 2022. URL: <https://studio.glassnode.com/metrics?a=BTC&category=&m=market.MarketcapUsd> (дата звернення: 30.11.2023).

35. Glassnode Studio. "Bitcoin: Number of Active Addresses" [Электронный ресурс]. 2022. URL: <https://studio.glassnode.com/metrics?a=BTC&category=Addresses&m=addresses.ActiveCount> (дата звернення: 30.11.2023).

36. The Cable. "Blockchain technology, non-fungible tokens, and the future of exchange" [Электронный ресурс]. 2022. URL: <https://www.thecable.ng/blockchain-technology-non-fungible-tokens-and-the-future-of-exchange> (дата звернення: 30.11.2023).

37. Briane, N. "Earth Day analysts say Bitcoin mining is naturally gravitating to green energy" [Електронний ресурс]. Cointelegraph. The future of money. April, 22, 2022. URL: <https://cointelegraph.com/news/this-earth-day-analysts-say-bitcoin-mining-is-naturally-gravitating-to-green-energy> (дата звернення: 30.11.2023).

38. Buterin, V. "The Search for a Stable Cryptocurrency" [Електронний ресурс]. Ethereum Foundation Blog. 2014. URL: <https://blog.ethereum.org/2014/11/11/search-stable-cryptocurrency/> (дата звернення: 30.11.2023).

39. Coin Market Cap. [Електронний ресурс]. 2022. URL: <https://coinmarketcap.com/> (дата звернення: 30.11.2023).

40. Cryptocurrency across the world [Електронний ресурс]. Triple A. 2022. URL: <https://triple-a.io/crypto-ownership-data/> (дата звернення: 30.11.2023).

41. Cryptocurrency Market Capitalizations [Електронний ресурс]. Investing.com. 2022. URL: <https://www.investing.com/crypto/charts?%20https://triple-a.io/crypto-ownership-old-290622/> (дата звернення: 30.11.2023).

42. "Mining CryptoNight-Haven on the Varium C1100 Blockchain Accelerator Card" [Електронний ресурс]. In: arXiv preprint. URL: <https://arxiv.org/abs/2212.05033> (дата звернення: 30.11.2023).

43. Moorhead, Patrick. "AMD Ryzen 3900X And 3700X Reviews Show The Company Delivered Big On Its Product Promises" [Електронний ресурс]. Moor Insights & Strategy. 25 липня 2019. URL: <https://moorinsightsstrategy.com/amd-ryzen-3900x-and-3700x-reviews-show-the-company-delivered-big-on-its-product-promises/> (дата звернення: 30.11.2023).

44. O'Neal, Terry; Cassells, William. "Vitis Unified Software Platform" [Електронний ресурс]. Xilinx Wiki. Останнє оновлення: 24 квітня 2023. URL: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/297009243/Vitis+Unified+Software+Platform> (дата звернення: 30.11.2023).

ДОДАТОК А

Цей код є частиною програми для обчислення хеш-функції Poseidon, яка оптимізована для виконання на FPGA за допомогою High-Level Synthesis (HLS).

Лістинг A.1 - Baseline_OpenCL_from_Neptune.cl

```
// Визначення, щоб код працював із CUDA та OpenCL
#ifdef __NVCC__
    // Для CUDA
    #define DEVICE __device__ // Визначення для функцій пристрою
    #define GLOBAL
    #define KERNEL extern "C" __global__ // Визначення для ядра
    #define LOCAL __shared__ // Визначення для спільної
локальної пам'яті
    #define CONSTANT __constant__ // Визначення для константної
пам'яті

    // Макроси для отримання ідентифікаторів у CUDA
    #define GET_GLOBAL_ID() blockIdx.x * blockDim.x +
threadIdx.x
    #define GET_GROUP_ID() blockIdx.x
    #define GET_LOCAL_ID() threadIdx.x
    #define GET_LOCAL_SIZE() blockDim.x
    #define BARRIER_LOCAL() __syncthreads() // Бар'єр для
синхронізації потоків

    typedef unsigned char uchar; // Визначення типу uchar

    #define CUDA
#else // Для OpenCL
    // Аналогічні визначення для OpenCL
    #define DEVICE
    #define GLOBAL __global
    #define KERNEL __kernel
    #define LOCAL __local
    #define CONSTANT __constant

    // Макроси для отримання ідентифікаторів у OpenCL
    #define GET_GLOBAL_ID() get_global_id(0)
    #define GET_GROUP_ID() get_group_id(0)
    #define GET_LOCAL_ID() get_local_id(0)
    #define GET_LOCAL_SIZE() get_local_size(0)
    #define BARRIER_LOCAL() barrier(CLK_LOCAL_MEM_FENCE)
#endif

// Визначення для NVIDIA OpenCL
#ifdef __NV_CL_C_VERSION
```

```

#define OPENCL_NVIDIA
#endif

// Визначення для різних GPU від AMD
#if defined(__WinterPark__) || defined(__BeaverCreek__) ||
defined(__Turks__) || \
    defined(__Caicos__) || defined(__Tahiti__) ||
defined(__Pitcairn__) || \
    defined(__Capeverde__) || defined(__Cayman__) ||
defined(__Barts__) || \
    defined(__Cypress__) || defined(__Juniper__) ||
defined(__Redwood__) || \
    defined(__Cedar__) || defined(__ATI_RV770__) ||
defined(__ATI_RV730__) || \
    defined(__ATI_RV710__) || defined(__Loveland__) ||
defined(__GPU__) || \
    defined(__Hawaii__)
#define AMD
#endif

// Функції для виконання арифметичних операцій з переносом у
64-бітному та 32-бітному форматах
// Ці функції використовують асемблерні вставки для
оптимізації на GPU

// ...

// Визначення структур та функцій для роботи з полями простих
чисел (prime-field)
// Це включає операції додавання, віднімання, множення та
піднесення до степеня

// ...

// Ядро для обчислення хешу з передзображень
// Використовує глобальні ідентифікатори для обробки батчів
даних
KERNEL void hash_preimages_2_standard(CONSTANT Fr
constants[373],
                                GLOBAL Fr *preimages,
                                GLOBAL Fr *digests,
                                int batch_size
                                ) {
    int global_id = GET_GLOBAL_ID();

    if (global_id < batch_size) {
        // Ініціалізація та обробка стану для кожного
передзображення
        // ...
        digests[global_id] = s.elements[1]; // Збереження
результату хешування
    }
}

```

Лістинг A.2 - krnl_poseidon_arity2_1.cpp

```

#include <stdint.h>
#include <hls_stream.h>
#include <cstdio>
#include "ap_int.h"
#include <iostream>
#include <math.h>
using namespace std;

// Структура для зберігання елементів поля
typedef struct Fr {
    unsigned long long val[4]; // Масив 4-х 64-бітних значень
} Fr;

// Структура для зберігання стану алгоритму
typedef struct state_2_standard {
    Fr elements[3]; // Елементи стану
    int current_round; // Поточний раунд обчислень
    int rk_offset; // Зсув ключа раунду
} state_2_standard;

// Функція для завантаження вхідних даних
static void load_input(Fr* preimages,
hls::stream<state_2_standard> &preimage_stream, const Fr
constantZero, int size) {
    #pragma HLS inline off
    load_input: for (int i = 0; i < size; i++) {
        state_2_standard s;
        s.elements[0] = constantZero;
        for (int j=0; j < 2; j++) {
            Fr preimage = preimages[i * 2 + j];
            s.elements[j+1] = Fr { (preimage.val[3],
preimage.val[2], preimage.val[1], preimage.val[0]) };
        }
        s.current_round = 0;
        s.rk_offset = 0;
        preimage_stream << s; // Передача стану в потік
    }
}

// Функція для виконання хешування
static void hash_2_standard(const Fr constants[],
hls::stream<state_2_standard> &in, hls::stream<Fr> &out, int size) {
    #pragma HLS inline off
    compute: for (int i = 0; i < size; i++) {
        state_2_standard s = in.read();

        int testAdd = size * 2;

        cout << "hello from the kernel " << testAdd << endl;
    }
}

```

```

        out.write(Fr { { 0, 0, 0, 0 } }); // Повернення
результату у вихідний потік
    }
}

// Функція для зберігання результату
static void store_result(hls::stream<Fr> &out_stream, Fr* out,
int size) {
    #pragma HLS inline off
    store_result: for (int i = 0; i < size; i++) {
        out[i] = out_stream.read(); // Читання результату з
потоків та збереження
    }
}

// Основна функція, доступна ззовні
extern "C" {
    void poseidon_hash(
        const Fr constants[373],
        Fr *preimages,
        Fr *digests,
        int batch_size
    ) {
        #pragma HLS interface m_axi port = preimages bundle = gmem0
        #pragma HLS interface m_axi port = digests bundle = gmem1

        #pragma HLS dataflow

        const unsigned int correctSize = batch_size/2; //
Обчислення коректного розміру
        Fr constantsZero;

        static hls::stream<state_2_standard>
preimage_stream("preimage_stream");
        static hls::stream<Fr>
digests_stream("digests_stream");
        #pragma HLS stream variable=preimage_stream
        #pragma HLS stream variable=digests_stream

        constantsZero = constants[0];
        load_input(preimages, preimage_stream, constantsZero,
correctSize);
        hash_2_standard(constants, preimage_stream,
digests_stream, correctSize);
        store_result(digests_stream, digests, correctSize);
    }
}

```

```

#include <stdint.h>
#include <hls_stream.h>
#include <cstdio>
#include "ap_int.h"
#include <iostream>
#include <math.h>
using namespace std;

typedef struct Fr_4limb {
    ap_uint<64> val[4];
} Fr_4limb;

typedef struct Fr {
    ap_uint<256> val;
} Fr;

typedef struct state_2_standard {
    Fr elements[3];
    ap_uint<7> current_round;
    ap_uint<9> rk_offset;
} state_2_standard;

const Fr Fr_ONE = { (ap_int<64>("1739710354780652911",10),
ap_int<64>("11064306276430008309",10),
ap_int<64>("6378425256633387010",10), ap_int<64>("8589934590",10))
};

const Fr Fr_P = { (ap_int<64>("8353516859464449352",10),
ap_int<64>("3691218898639771653",10),
ap_int<64>("6034159408538082302",10),
ap_int<64>("18446744069414584321",10)) };

const Fr Fr_R2 = { (ap_int<64>("524908885293268753",10),
ap_int<64>("419701826671360399",10),
ap_int<64>("3129137299524312099",10),
ap_int<64>("14526898881837571181",10)) };

const Fr Fr_ZERO = { 0 };
const ap_int<64> Fr_INV =
ap_int<64>("18446744069414584319",10);

// Greater than or equal
static bool Fr_gte(Fr a, Fr b) {
    return a.val >= b.val;
}

// Normal addition
static Fr Fr_add_(Fr a, Fr b) {
    return Fr { a.val + b.val };
}
static Fr Fr_sub_(Fr a, Fr b) {
    return Fr { a.val - b.val };
}

```

```

// Modular subtraction
static Fr Fr_sub(Fr a, Fr b) {
    Fr res = Fr_sub_(a, b);
    if(!Fr_gte(a, b))
        res = Fr_add_(res, Fr_P);
    return res;
}

// Modular addition
static Fr Fr_add(Fr a, Fr b) {
#pragma HLS INLINE
    Fr res = Fr_add_(a, b);
    if (Fr_gte(res, Fr_P))
        res = Fr_sub_(res, Fr_P);
    return res;
}

const ap_uint<64> Fr_P_first_limb = Fr_P.val(63, 0);
const ap_uint<192> Fr_P_remainder = Fr_P.val(255, 64);

static ap_uint<384> Fr_mul_part(Fr a, Fr b, ap_uint<384> t384,
ap_uint<3> i) {
#pragma HLS inline

    //ap_uint<8> range = i * ap_uint<7>(64);

    ap_uint<64> bSubRange = b.val.range((i * 64) + 63, i * 64);

    t384 += a.val * bSubRange;

    ap_uint<64> t_first_limb = t384.range(63, 0);
    ap_uint<64> m = Fr_INV * t_first_limb;

    ap_uint<129> res = m * Fr_P_first_limb + t384(63, 0);
    ap_uint<64> carry = res.range(128, 64);

    return m * Fr_P_remainder + t384(383, 64) + carry;
}

static Fr Fr_mul(Fr a, Fr b) {
#pragma HLS inline
    ap_uint<384> t384 = ap_uint<384>(0);

    Fr_mul_loop: for (ap_uint<3> i = 0; i < 4; i++) {
        t384 = Fr_mul_part(a, b, t384, i);
    }

    Fr result = Fr { (t384(255, 192), t384(191, 128), t384(127,
64), t384(63, 0)) };

    if (Fr_gte(result, Fr_P)) {
        result = Fr_sub_(result, Fr_P);
    }
}

```

```

    }

    return result;
}

static Fr Fr_mont(Fr a) {
    return Fr_mul(a, Fr_R2);
}

static Fr Fr_unmont(Fr a) {
    Fr one = Fr { 1 };
    return Fr_mul(a, one);
}

static ap_uint<256> Fr_unmont_to_decimal(Fr a) {
    return a.val;
}

static void transform_constants(const Fr_4limb constants[], Fr
constantsFr[], ap_uint<9> size) {
    #pragma HLS inline off
    transform_constants: for (ap_uint<9> i=0; i<size; i++) {
        constantsFr[i] = Fr { (constants[i].val[3],
constants[i].val[2], constants[i].val[1], constants[i].val[0]) };
    }
}

static void load_input(Fr_4limb* preimages,
hls::stream<state_2_standard> &preimage_stream, const Fr
constantZero, ap_uint<9> size) {
    #pragma HLS inline off
    load_input: for (ap_uint<32> i = 0; i < size; i++) {
        state_2_standard s;
        s.elements[0] = constantZero;
        for (ap_uint<2> j=0; j < 2; j++) {
            Fr_4limb preimage = preimages[i * 2 + j];
            s.elements[j+1] = Fr { (preimage.val[3],
preimage.val[2], preimage.val[1], preimage.val[0]) };
        }
        s.current_round = 0;
        s.rk_offset = 0;
        preimage_stream << s;
    }
}

static Fr quintic_s_box(Fr l, Fr pre_add, Fr post_add) {
    #pragma HLS allocation operation instances=mul limit=15
    Fr tmp = Fr_add(l, pre_add);
    tmp = Fr_mul(l, l);
    tmp = Fr_mul(tmp, tmp);

```

```

        tmp = Fr_mul(tmp, 1);
        tmp = Fr_add(tmp, post_add);

        return tmp;
    }

    static Fr scalar_product(const Fr* a, Fr* b, ap_uint<2> size)
{
    #pragma HLS inline
        Fr res = Fr_ZERO;

        multiply_add_loop: for (ap_uint<2> i = 0; i < size; ++i) {
            Fr tmp = Fr_mul(a[i], b[i]);
            res = Fr_add(res, tmp);
        }

        return res;
    }

    static state_2_standard apply_matrix_2_standard(const Fr
matrix[9], state_2_standard s) {
    #pragma HLS allocation operation instances=mul limit=30
        Fr tmp[3];
        initialize: for (ap_uint<2> i = 0; i < 3; ++i) {
            tmp[i] = s.elements[i];
            s.elements[i] = Fr_ZERO;
        }

        multiply_add_loop: for (ap_uint<2> j = 0; j < 3; ++j) {
            for (ap_uint<2> i = 0; i < 3; ++i) {
                s.elements[j] = Fr_add(s.elements[j],
Fr_mul(matrix[i * 3 + j], tmp[i]));
            }
        }
        return s;
    }

    static state_2_standard apply_sparse_matrix_2_standard(const
Fr sm[5], state_2_standard s) {
    #pragma HLS inline
        Fr first_elt = s.elements[0];

        s.elements[0] = scalar_product(sm + 0, s.elements, 3);

        multiply_add_loop: for (ap_uint<2> i = 1; i < 3; ++i) {
            Fr val = Fr_mul((sm + 3)[i-1], first_elt);
            s.elements[i] = Fr_add(s.elements[i], val);
        }

        return s;
    }

```

```

static state_2_standard apply_round_matrix_2_standard(const Fr
constants[373], state_2_standard s) {
#pragma HLS inline
    if ((s.current_round > 3) && (s.current_round < 4 + 55)) {
        ap_uint<7> index = s.current_round - 3 - 1;
        ap_uint<9> index2 = index * 5;
        s = apply_sparse_matrix_2_standard(constants + 98 +
(index2), s);
    } else {
        Fr *matrix;
        if (s.current_round == 3) {
            matrix = (Fr (*)(constants + 89));
        } else {
            matrix = (Fr (*)(constants + 80));
        }

        s = apply_matrix_2_standard(matrix, s);
    }

    return s;
}

static state_2_standard add_full_round_keys_2_standard(const
Fr constants[373], state_2_standard s) {
    add_loop: for (ap_uint<2> i = 0; i < 3; ++i) {
        s.elements[i] = Fr_add(s.elements[i], (constants +
1)[s.rk_offset + i]);
    }
    s.rk_offset += 3;
    return s;
}

static state_2_standard all_rounds_2_standard(const Fr
constants[], state_2_standard s, ap_uint<1> full, ap_uint<1>
partial, ap_uint<1> last) {
    // #pragma HLS pipeline
    #pragma HLS allocation operation instances=mul limit=30
    // 95 is the fastest at 300MHz with all DSP

    state_2_standard s2;
    s2.current_round = s.current_round;

    loop_quintic: for (ap_uint<2> i = 0; i < 3; ++i) {
        Fr post_add;
        if (full || partial) {
            post_add = (constants + 1)[s.rk_offset + i];
        } else {
            post_add = Fr_ZERO;
        }

        if (i == 0 || full || last) {
            s2.elements[i] = quintic_s_box(s.elements[i],
Fr_ZERO, post_add);

```

```

        } else {
            s2.elements[i] = s.elements[i];
        }
    }

    if (full) {
        s2.rk_offset = s.rk_offset + 3;
    } else if (partial) {
        s2.rk_offset = s.rk_offset + 1;
    } else {
        s2.rk_offset = s.rk_offset;
    }

    state_2_standard s3 =
    apply_round_matrix_2_standard(constants, s2);

    s3.current_round = s.current_round + 1;

    return s3;
}

static void hash_2_standard(const Fr constants[],
hls::stream<state_2_standard> &in, hls::stream<Fr> &out, ap_uint<32>
size) {

    state_2_standard s[512];
    ap_uint<7> round[512] = {0};
    ap_uint<9> index = 0;
    ap_uint<32> totalReads = 0;

    ap_uint<32> fitToProcessingSize = (size >> 9) << 9;

    if (fitToProcessingSize < size) {
        fitToProcessingSize += 512;
    }

    ap_uint<39> processSize = fitToProcessingSize * 65;

    main_loop: for (ap_uint<32> i=0; i < processSize; i++) {
#pragma HLS loop_tripcount min=65000 max=65000
#pragma HLS PIPELINE

        if (round[index] == 0) {
            if (totalReads < size) {
                s[index] =
add_full_round_keys_2_standard(constants, in.read());
                round[index] = 1;
                totalReads = totalReads + 1;
            }
        } else if (round[index] >= 1 && round[index] <= 63) {
            ap_uint<7> currentP = round[index];

```

```

        ap_uint<1> full = ((currentP >= 1 && currentP <=
4) || (currentP >= 60 && currentP <= 62));
        ap_uint<1> partial = (currentP >= 5 && currentP
<= 59);
        ap_uint<1> last = (currentP == 63);

        s[index] = all_rounds_2_standard(constants,
s[index], full, partial, last);

        round[index] = round[index] + 1;
    } else {
        out.write(s[index].elements[1]);
        round[index] = 0;
    }

    index++;
    if (index >= 512) {
        index = 0;
    }
}

static void store_result(hls::stream<Fr> &out_stream,
Fr_4limb* out, ap_uint<9> size) {
    #pragma HLS inline off
    store_result: for (ap_uint<9> i = 0; i < size; i++) {
        Fr data = out_stream.read();
        out[i] = Fr_4limb { { data.val(63, 0), data.val(127,
64), data.val(191, 128), data.val(255, 192) } };
    }
}

extern "C" {
    void poseidon_hash(
        const Fr_4limb constants[373],
        Fr_4limb *preimages,
        Fr_4limb *digests,
        int batch_size
    ) {
        #pragma HLS interface m_axi port = preimages bundle = gmem0
        #pragma HLS interface m_axi port = digests bundle = gmem1

        #pragma HLS dataflow

        Fr constantsFr[373];
        #pragma HLS array_partition variable=constantsFr complete

        const unsigned int correctSize = batch_size/2;
        Fr constantsZero;

        static hls::stream<state_2_standard>
preimage_stream("preimage_stream");

```

```

        static hls::stream<Fr>
digests_stream("digests_stream");
        #pragma HLS stream variable=preimage_stream
        #pragma HLS stream variable=digests_stream

        transform_constants(constants, constantsFr, 373);
        constantsZero = constantsFr[0];
        load_input(preimages, preimage_stream, constantsZero,
correctSize);
        hash_2_standard(constantsFr, preimage_stream,
digests_stream, correctSize);
        store_result(digests_stream, digests, correctSize);

    }
}

```

Лістинг A.4 - host_poseidon_1.cpp

```

#define OCL_CHECK(error, call)
\
    call;
\
    if (error != CL_SUCCESS) {
\
        printf("%s:%d Error calling " #call ", error code is:
%d\n", __FILE__, __LINE__, error); \
        exit(EXIT_FAILURE);
\
    }

#include <stdlib.h>
#include <fstream>
#include <iostream>
#include "vadd.h"
#include "ap_int.h"
using namespace std;

static const int DATA_SIZE = 1;

static const std::string error_message =
    "Error: Result mismatch:\n"
    "i = %d CPU result = %d Device result = %d\n";

typedef struct Fr {

```

```

        unsigned long long val[4];
    } Fr;

    int main(int argc, char* argv[]) {

        //TARGET_DEVICE macro needs to be passed from gcc command
line
        if(argc != 2) {
            std::cout << "Usage: " << argv[0] <<" <xclbin>" <<
std::endl;
            return EXIT_FAILURE;
        }

        std::string xclbinFilename = argv[1];

        // Compute the size of array in bytes
        size_t size_in_bytes = DATA_SIZE * sizeof(int);
        size_t constants_size_in_bytes = 373 * sizeof(Fr);

        // Creates a vector of DATA_SIZE elements with an initial
value of 10 and 32
        // using customized allocator for getting buffer alignment
to 4k boundary

        std::vector<cl::Device> devices;
        cl::Device device;
        cl_int err;
        cl::Context context;
        cl::CommandQueue q;
        cl::Kernel krnl_poseidon_hash;
        cl::Program program;
        std::vector<cl::Platform> platforms;
        bool found_device = false;

        //traversing all Platforms To find Xilinx Platform and
targeted
        //Device in Xilinx Platform
        cl::Platform::get(&platforms);
        for(size_t i = 0; (i < platforms.size() ) & (found_device
== false) ;i++){
            cl::Platform platform = platforms[i];
            std::string platformName =
platform.getInfo<CL_PLATFORM_NAME>();
            if ( platformName == "Xilinx"){
                devices.clear();
                platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR,
&devices);

                if (devices.size()){
                    device = devices[0];
                    found_device = true;
                    break;
                }
            }
        }
    }

```

```

    }
    if (found_device == false){
        std::cout << "Error: Unable to find Target Device "
            << device.getInfo<CL_DEVICE_NAME>() << std::endl;
        return EXIT_FAILURE;
    }

    // Creating Context and Command Queue for selected device
    OCL_CHECK(err, context = cl::Context(device, NULL, NULL,
    NULL, &err));
    OCL_CHECK(err, q = cl::CommandQueue(context, device,
    CL_QUEUE_PROFILING_ENABLE, &err));

    std::cout << "INFO: Reading " << xclbinFilename <<
std::endl;
    FILE* fp;
    if ((fp = fopen(xclbinFilename.c_str(), "r")) == nullptr)
    {
        printf("ERROR: %s xclbin not available please
build\n", xclbinFilename.c_str());
        exit(EXIT_FAILURE);
    }
    // Load xclbin
    std::cout << "Loading: '" << xclbinFilename << "'\n";
    std::ifstream bin_file(xclbinFilename,
std::ifstream::binary);
    bin_file.seekg (0, bin_file.end);
    unsigned nb = bin_file.tellg();
    bin_file.seekg (0, bin_file.beg);
    char *buf = new char [nb];
    bin_file.read(buf, nb);

    // Creating Program from Binary File
    cl::Program::Binaries bins;
    bins.push_back({buf,nb});
    devices.resize(1);
    OCL_CHECK(err, program = cl::Program(context, devices,
bins, NULL, &err));

    // This call will get the kernel object from program. A
kernel is an
    // OpenCL function that is executed on the FPGA.
    OCL_CHECK(err, krnl_poseidon_hash =
cl::Kernel(program,"poseidon_hash", &err));

    // These commands will allocate memory on the Device. The
cl::Buffer objects can
    // be used to reference the memory locations on the
device.
    OCL_CHECK(err, cl::Buffer buffer_constants(context,
CL_MEM_READ_ONLY, constants_size_in_bytes, NULL, &err));
    OCL_CHECK(err, cl::Buffer buffer_preimages(context,
CL_MEM_READ_ONLY, size_in_bytes, NULL, &err));

```

```

        OCL_CHECK(err, cl::Buffer buffer_digests(context,
CL_MEM_WRITE_ONLY, size_in_bytes, NULL, &err));

        //set the kernel Arguments
        int narg=0;
        OCL_CHECK(err, err =
krnl_poseidon_hash.setArg(narg++,buffer_constants));
        OCL_CHECK(err, err =
krnl_poseidon_hash.setArg(narg++,buffer_preimages));
        OCL_CHECK(err, err =
krnl_poseidon_hash.setArg(narg++,buffer_digests));
        OCL_CHECK(err, err =
krnl_poseidon_hash.setArg(narg++,DATA_SIZE));

        //We then need to map our OpenCL buffers to get the
pointers
        Fr *ptr_constants;
        Fr *ptr_preimages;
        Fr *ptr_digests;
        OCL_CHECK(err, ptr_constants = (Fr*)q.enqueueMapBuffer
(buffer_constants , CL_TRUE , CL_MAP_WRITE , 0,
constants_size_in_bytes, NULL, NULL, &err));
        OCL_CHECK(err, ptr_preimages = (Fr*)q.enqueueMapBuffer
(buffer_preimages , CL_TRUE , CL_MAP_WRITE , 0, size_in_bytes, NULL,
NULL, &err));
        OCL_CHECK(err, ptr_digests = (Fr*)q.enqueueMapBuffer
(buffer_digests , CL_TRUE , CL_MAP_READ , 0, size_in_bytes, NULL,
NULL, &err));

        cout << "setting up data" << endl;
        for (unsigned long long i = 0; i < 373; i++) {
            ptr_constants[i] = Fr { { i, 0, 0, 0 } };
        }

        for (unsigned long long i = 0; i < DATA_SIZE; i++) {
            ptr_preimages[i] = Fr { { i, 0, 0, 0 } };
        }

        // Data will be migrated to kernel space
        OCL_CHECK(err, err =
q.enqueueMigrateMemObjects({buffer_preimages}, 0/* 0 means from
host*/));

        //Launch the Kernel
        OCL_CHECK(err, err = q.enqueueTask(krnl_poseidon_hash));

        // The result of the previous kernel execution will need
to be retrieved in
        // order to view the results. This call will transfer the
data from FPGA to
        // source_results vector

```

```

        OCL_CHECK(err,
q.enqueueMigrateMemObjects({buffer_digests},CL_MIGRATE_MEM_OBJECT_HOST));

        cout << "waiting" << endl;
        OCL_CHECK(err, q.finish());
        cout << "finish" << endl;

        //Verify the result
        int match = 0;
        for (int i=0; i< (DATA_SIZE >> 1); i++) {
            cout << " output " << ptr_digests[i].val[0] << " " <<
ptr_digests[i].val[1] << " " << ptr_digests[i].val[2] << " " <<
ptr_digests[i].val[3] << endl;
        }

        OCL_CHECK(err, err =
q.enqueueUnmapMemObject(buffer_constants, ptr_constants));
        OCL_CHECK(err, err =
q.enqueueUnmapMemObject(buffer_preimages, ptr_preimages));
        OCL_CHECK(err, err =
q.enqueueUnmapMemObject(buffer_digests, ptr_digests));
        OCL_CHECK(err, err = q.finish());

        cout << "TEST " << (match ? "FAILED" : "PASSED") <<
std::endl;
        return (match ? EXIT_FAILURE : EXIT_SUCCESS);

    }

```