БАКАЛАВРСЬКА РОБОТА

БР. ІП - 07.00.00.000 ПЗ

Група ІП-21-1

Гетюк Андрій

2025

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Гетюк Андрій Дмитрович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Оптимізаційні техніки компіляції

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня ____________Гетюк А.Д.______________
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Гобир Ліда Мирославівна, асистент_________
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц.______________Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“____” ________________202 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Гетюк Андрі. Дмитровичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Оптимізаційні техніки компіляції"

керівник проекту (роботи) асист. Гобир Лідія Мирославівна

затверджені наказом вищого навчального закладу від “ 28 ” квітня 2025 р. № 264/7

2. Строк подання студентом проекту (роботи) 10 червня 2025 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1 Вступ до проблеми
2 Автоматична ідентифікація та вибір прискорювачів
3 Автоматична оптимізація для спільного проектування HW/SW
4 Ідентифікація та вибір системно-залежних прискорювачів

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1.Приклад графа потоку керування функції, позначений кольором частоти виконання (рис. 1.1 ст.13).

2. Час виконання алгоритму точного кадрування (рис. 1.10, ст.30).

3. Автоматично згенеровані прискорювачі включають кілька шляхів даних, які виконують тіло циклу двовимірних SCoP 40 (рис 2,3, ст.40)

4. Розподіл помилок прогнозування фактору розгортання циклу для 18 000 прогнозів поза вибіркою(рис. 2.8, ст.51)

5. Приклад графа викликів (рис 3.3, ст.61)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання 2025 р.

Керівник ______________________________
(підпис)

Завдання прийняв до виконання______ _______________
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обґрунтування теми роботи	15.02.2025	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	25.02.2025	виконано
3	Побудова моделі або алгоритму власного рішення	15.03.2025	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	25.04.2025	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	10.06.2025	виконано

Студент ____________________
(підпис)

Керівник роботи ____________________
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 80 сторінок, 34 рисунки, 6 таблиць, список використаних джерел із 20 найменування,

Метою роботи є розробка та дослідження ефективних методів та програмних інструментів для автоматизованої ідентифікації, вибору та оптимізації апаратних прискорювачів з урахуванням системних обмежень та енергоефективності, з метою підвищення продуктивності та ефективності сучасних обчислювальних систем.

Об'єкт дослідження: Процеси автоматичної ідентифікації, вибору та оптимізації апаратних прискорювачів у гетерогенних обчислювальних системах.

Предмет дослідження: Методи та алгоритми вибору, оптимізації та впровадження системно-залежних апаратних прискорювачів з використанням машинного навчання, аналізу даних та спеціалізованих фреймворків.

Результати дослідження: Досліджено питання гетерогенних обчислень та їх інтеграції в процес автоматизованого спільного проектування апаратного та програмного забезпечення.

В першому розділі описана нова методологія RegionSeeker, яка представляє ідеї, що стосуються дослідницького питання.

В другому розділі описується автоматизована методологію, яка визначає потенціал повторного використання даних певного типу програм за допомогою поліедрального аналізу.

В третьому розділі представлено AccelSeeker, який пропонує розширений діапазон кандидатів на прискорення – для всього графа викликів функцій програми.

Висновок: у роботі було досліджено проблему автоматичної ідентифікації, вибору та оптимізації апаратних прискорювачів у гетерогенних обчислювальних системах.

КЛЮЧОВІ СЛОВА: ГЕТЕРОГЕННИХ ОБЧИСЛЕНЬ, FRAMEWORK REGIONSEEKER, ПРИСКОРЮВАЧ, ГРАФ, ІДЕНТИФІКАЦІЯ

ANNOTATION

The bachelor's thesis contains 53 pages, 34 figures, 6 tables, a list of used sources with 20 names,

The purpose of the work is to develop and study effective methods and software tools for automated identification, selection and optimization of hardware accelerators taking into account system constraints and energy efficiency, in order to increase the productivity and efficiency of modern computing systems.

Object of research: Processes of automatic identification, selection and optimization of hardware accelerators in heterogeneous computing systems.

Subject of research: Methods and algorithms for selection, optimization and implementation of system-dependent hardware accelerators using machine learning, data analysis and specialized frameworks.

Research results: The issue of heterogeneous computing and its integration into the process of automated joint design of hardware and software is investigated.

The first section describes the new RegionSeeker methodology, which presents ideas related to the research question.

The second section describes an automated methodology that determines the data reuse potential of a certain type of program using polyhedral analysis.

The third section presents AccelSeeker, which offers an extended range of acceleration candidates - for the entire graph of program function calls.

Conclusion: the paper investigated the problem of automatic identification, selection and optimization of hardware accelerators in heterogeneous computing systems.

KEYWORDS: HETEROGENEOUS COMPUTING, FRAMEWORK REGIONSEEKER, ACCELERATER, GRAPH, IDENTIFICATION

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CFG - граф потоку керування

SESE - Single Entry Single Exit

SE - Scalar Evolution

CFG - граф потоку керування

SVM - методи опорних векторних машин

SHOC - Scalable Heterogeneous Computing

SVM - векторні машини ()

NN - найближчі сусіди

CNN - Згорткові нейронні мережі ()

DNN - низькопотужних глибоких нейронних мереж ().

AES - Advanced Encryption Standard

XPE - Xilinx Power Estimator

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ……………………………………………6

ВСТУП ....… 8

РОЗДІЛ 1. АВТОМАТИЧНА ІДЕНТИФІКАЦІЯ ТА ВИБІР ПРИСКОРЮВАЧІВ ……………………………………………………. 11

1.1. Мотивація 13

1.2. Формування проблеми 15

1.3. Алгоритми вибору 16

1.4. Framework RegionSeeker 20

1.5. Експериментальні результати 24

1.6 Висновки по розділу ………………………………………………..33

РОЗДІЛ 2. АВТОМАТИЧНА ОПТИМІЗАЦІЯ ДЛЯ СПІЛЬНОГО ПРОЕКТУВАННЯ HW/SW 34

2.1. Аналіз повторного використання даних 35

2.2. Підхід машинного навчання для прогнозування фактора розгортання циклу ……………………………………………………………………45

2.3 Висновки по розділу ………………………………………………..57

РОЗДІЛ 3. ІДЕНТИФІКАЦІЯ ТА ВИБІР СИСТЕМНО-ЗАЛЕЖНИХ ПРИСКОРЮВАЧІВ 58

3.1. AccelSeeker: прискорювачі 59

3.2. EnergySeeker: прискорювачі для енергоефективності 78

3.3. Гетерогенні обчислення .... …84

3.4 Висновки по розділу **……………………………………………..85**

ВИСНОВКИ 86

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ……………………………………87

Змн.	Арк.	№ докум.	Підпис	Дата	ДРБ.ІП – 07.00.00.000 ПЗ		
Розроб.		Гетюк А.Д.			Оптимізаційні техніки компіляції Пояснююча записка	Літ. / Арк. 9 / Акрушів	
Перевір.		Гобир Л.М.					
Реценз.		Саманів Л.В.				ІФНТУНГ ІП-21-1	
Н. Контр.		Піх М.М.					
Затверд.		Бандура В. В.					

ВСТУП

Актуальність теми дослідження. Підвищення продуктивності з точки зору швидшого виконання та високої енергоефективності є метою нескінченних дослідницьких зусиль і не надається безкоштовно. Живучи в епоху, коли існує величезна кількість даних, попит на продуктивність сучасних обчислювальних систем зростає ще більше. Такі технологічні гіганти, як Google і Facebook, збирають і обчислюють масу даних, наприклад, під час ¬програм, пов'язаних з машинним навчанням, і тривалих симуляцій. Обробка цього великого обсягу даних вимагає величезної обчислювальної потужності та призводить до все більш тривалого часу виконання.

Закон Мура [8], спостереження, зроблене співзасновником Intel Гордоном Муром, передбачає, що кількість транзисторів, які можна використовувати в тій самій частині інтегральної схеми, подвоюватиметься приблизно кожні 18 місяців. Додатково до цього масштабування Деннарда [22], також відоме як масштабування MOSFET, стверджує, що напруга та струм пропорційні розміру транзистора. Тому, доки зберігається та сама площа мікросхеми, потужність залишається постійною, і в той же час на ній може поміститися більше транзисторів меншого розміру. На жаль, це вже не так. Розмір транзистора з роками зменшився, але кількість потужності на транзистор нещодавно перестала зменшуватися відповідно, явище, також відоме як порушення масштабування Деннарда [24].

Порушення масштабування Деннарда разом із, здавалося б, неминучим кінцем економічного аспекту закону Мура [3], представляють новий виклик для комп'ютерних ¬архітекторів, які прагнуть досягти кращої продуктивності в сучасних комп'ютерних системах. Гетерогенні обчислення з'являються як одне з рішень для підтримки тенденції зростання продуктивності. Це досягається шляхом зосередження уваги на спеціальному апаратному забезпеченні (HW), яке може прискорити виконання програми (SW) або частини цієї програми. Спеціалізовані апаратні прискорювачі реалізуються ¬на платформах, де вони можуть бути перепрограмованими, що забезпечує високу гнучкість, оскільки

різні реалізації можуть мати місце з використанням апаратних ресурсів платформи (наприклад, плата FPGA), або жорстко підключеними, такими як інтегральна схема, призначена для конкретного застосування (ASIC). Перший тип реалізації HW приносить в жертву частину потенційної продуктивності, досягнутої завдяки дозволу гнучкі конструкції, оскільки ті самі апаратні ресурси можна перепрограмувати. Останній не пропонує гнучкості, але може забезпечити кращу продуктивність у порівнянні з FPGA. У рамках цього дослідження розглядалися обидві реалізації HW.

Оскільки продуктивність центрального процесора загального призначення стає обмеженою через фізичні та технологічні обмеження, потрібні альтернативні комп’ютерні архітектури. Однорідні паралельні процесори використовуються для виявлення паралелізму обчислень у програмах програмного забезпечення, але продуктивність все ще обмежена частинами обчислень, які не можна розпаралелювати, факт, також відомий як закон Амдала. Замість ЦП загального призначення або однорідних паралельних ЦП, які керують виконанням програм ПЗ, спеціалізовані компоненти апаратного забезпечення, а саме прискорювачі, можна використовувати разом із ЦП загального призначення та виконувати найвимогливіші частини програми з точки зору обчислень. Отже, потреба в одному потужному центральному процесорі вже не є такою критичною, оскільки виконання також можна перекласти на інші частини HW. У результаті ми досягаємо як більш збалансованого виконання з використанням різних апаратних ресурсів, так і перекладаємо виконання конкретних, більш вимогливих частин обчислень на спеціалізовані апаратні прискорювачі.

Одним із прикладів широко розповсюдженої гетерогенної архітектури є додавання графічного процесора до центрального процесора на тому самому чіпі, щоб використовувати паралелізм і обчислювальну потужність, яку може запропонувати графічний процесор, коли йдеться про обробку зображень і візуалізацію 3D-графіки. Іншими прикладами є центральні процесори загального призначення в поєднанні зі спеціальним апаратним забезпеченням, які

виконують певні ядра або навіть повні програми. Остання архітектура може мати ряд варіацій, з одним або декількома HW прискорювачами та різними типами зв'язку, жорсткого чи слабкого [20]. Розробка першого варіанту, тісно або співпроцесорної моделі, виконується за допомогою прискорювача як розширення набору інструкцій у конвеєрі ЦП за замовчуванням. Останній реалізує зв'язок між процесором і прискорювачем вільно, без будь-яких знань про базову мікроархітектуру процесора.

Метою роботи є розробка ефективних гетерогенних комп'ютерних архітектур, щоб час затримки та вимоги до енергії постійно зменшувалися.

Для досягнення поставленої мети в роботі необхідно вирішити **наступні завдання:** Провести аналіз сучасних підходів до автоматичної ідентифікації та вибору апаратних прискорювачів у гетерогенних системах. Сформулювати проблеми та вимоги до автоматизованого проектування апаратно-програмних рішень. Розробити ефективні алгоритми та програмні засоби для вибору та оптимізації прискорювачів. Впровадити фреймворки для виявлення та оцінки обчислювально важливих регіонів програм Провести експериментальну перевірку ефективності запропонованих рішень у практичних завданнях.

Об'єктом є процеси автоматичної ідентифікації, вибору та оптимізації апаратних прискорювачів у гетерогенних обчислювальних системах.

Предмет дослідження: методи та алгоритми вибору, оптимізації та впровадження системно-залежних апаратних прискорювачів з використанням машинного навчання, аналізу даних та спеціалізованих фреймворків.

Методи дослідження – були використанні аналіз та узагальнення літературних джерел. формалізації та моделювання, алгоритмічні методи — для розробки ефективних процедур аналізу програмного коду, Методи машинного навчання, Експериментальні методи, Порівняльний аналіз

Бакалаврська робота містить 80 сторінки, 34 рисунків, 6 таблиці, 3 розділи, список використаних джерел із 20 найменуванням.

РОЗДІЛ 1. АВТОМАТИЧНА ІДЕНТИФІКАЦІЯ ТА ВИБІР ПРИСКОРЮВАЧІВ

Рухаючись до гетерогенної ери, апаратні прискорювачі, призначені для конкретного завдання, можуть покращити як прискорення виконання, так і енергоефективність у порівнянні з ЦП загального призначення або набором однорідних ЦП. Тим не менш, ідентифікація та вибір частин обчислень, які будуть реалізовані в HW, є складним і вимогливим завданням. Необхідно глибоке розуміння програми, яку потрібно прискорити, бюджет апаратних ресурсів (області) часто обмежений, а деталізація кандидатів на прискорення може істотно вплинути на загальний час виконання. Крім того, оптимізація може бути застосована до заданого ідентифікованого апаратного прискорювача, і це може створити кілька версій еквівалентних екземплярів обчислення, що, у свою чергу, може призвести до різноманітних гетерогенних архітектур з різними характеристиками та різним приростом продуктивності. Щоб вирішити ці проблеми, я представляю автоматизовану методологію, яка отримує вихідний код певної програми та виводить кілька апаратних прискорювачів, які слід розглянути для прискорення. Серед цих кандидатів відбувається відбір, який максимізує колективне прискорення, враховуючи постійний дощ на території. Нарешті, на етапі відбору можна розглянути кілька версій одного і того ж кандидата.

1.1 Мотивація

Яке обґрунтування вибору дизайнера, коли вручну вибираються частини додатка, які потрібно прискорити в HW, і як натомість цей вибір можна відтворити автоматизованим інструментом? Хоча можливо , що *все* обґрунтування дизайнера не може бути відтворено автоматично - потенційно тому, що це вимагає глибоке знання програми під рукою - безумовно, все ще бажано визначити принаймні підмножину дій, які можна автоматизувати

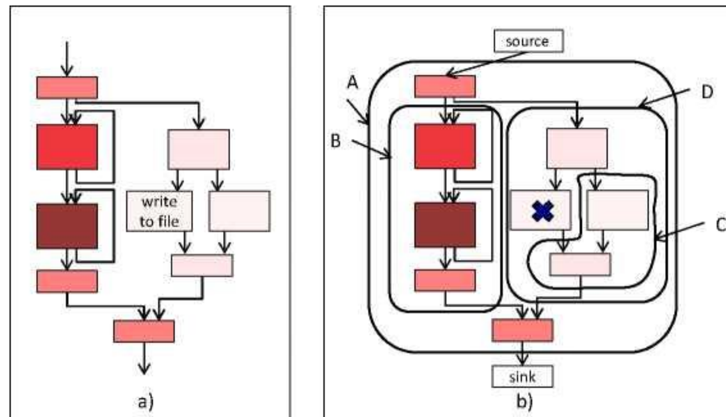


Рисунок 1.1 - а) Приклад графа потоку керування функції, позначений кольором частоти виконання (чим темніший базовий блок, тим частіше), б) B і C є дійсними підграфами; A і D не є дійсними підграфами, оскільки вони містять заборонений вузол. B також є областю CFG, оскільки вона має єдиний вхід і вихід потоку керування.

Зазвичай метою розробника буде: враховуючи доступну область прискорювача, вилучити якомога більше обчислень, за обмеженням вимагати не більше цієї області, щоб максимізувати результуюче прискорення. У рамках цього дослідження я ідентифікую підграфи графа потоку керування, які мають одну вхідну контрольну точку та одну вихідну контрольну точку, які тут будуть називатися областями, як хороші кандидати на прискорення. Обґрунтування полягає в тому, що ці підграфи мають єдину точку входу, яка відповідає моменту виконання, коли викликається прискорювач, і єдину точку виходу, отже належним чином повертаючись до єдиного місця в програмному забезпеченні, коли прискорювач завершує роботу. Зверніть увагу, що цей тип підграфа потоку керування був раніше запропонований і досліджений у дослідженні компілятора - під назвою SESE (Single Entry Single Exit) у [2], [38] і під назвою Simple Region у реалізації LLVM [42] - з метою покращення якості генерації програмного коду та як область для застосування оптимізації компілятора та розпаралелювання. Ідея визначення того самого типу підграфа запозичена та застосована тут новим способом і до іншого сценарію та мети: автоматичного вибору HW прискорювачів. Приклад мотивації наведено на рисунку |1.1| а, який зображує CFG прикладу функції, позначений кольором частоти виконання (темніший

основний блок, тим частіше). Можливим вибором під час ідентифікації прискорювачів вручну є робота над деталізацією функцій: реалізація в HW функції, яка найчастіше виконується. Однак цей вибір може бути не ідеальним, оскільки недоліки можуть бути подвійними: 1) частина функції може виконуватися рідше, ¬ніж інші частини (права сторона CFG, у прикладі на Рисунку 1.1 а), таким чином ефективно витрачаючи нерухомість прискорювача. 2) частина функції може містити несинтезовані конструкції, такі як системний виклик «запис у файл» на Рисунку 1.1a або виклик функції, який не можна вбудовано. З іншого боку спектру, вибір просто в межах одного базового блоку - отже, тіла часто виконуваного циклу на зображенні - також може бути не ідеальним, оскільки прискорювач буде викликатися один раз на кожній ітерації циклу, що може призвести до великих накладних витрат. Крім того, деякий потенціал прискорення може бути втрачений, оскільки більші регіони CFG можуть піддавати кращу оптимізацію синтезу. Тому регіони CFG пропонуються як кандидати на прискорювачі, враховуючи деталізацію, яка може переходити від одного циклу до цілої функції та будь-чого між ними. Основною частиною мого дослідження для цієї роботи є розгляд регіонів CFG як кандидатів і метод автоматичної ідентифікації та вибору цих регіонів.

1.2 Формування проблеми

Запропонована методологія визначає та досліджує продуктивність регіонів -шляхом аналізу на рівні проміжного представлення (IR) графів потоку керування функцій, що входять до цільової програми. CFG представляє потік керування через програму. Визначення: CFG. CFG — це орієнтований циклічний граф G, де У (G) — множина вузлів, а E (G) — множина ребер. Кожен вузол у CFG відповідає базовому блоці у функції, а кожне ребро — потоку керування в цій функції. Додається вихідний вузол, пов’язаний лише з базовим блоком входу функції, і вузол-приймач, пов’язаний лише з виходом функції. На Рисунку 1.1b показано приклад CFG. Вузол у CFG позначається як заборонений, якщо він

відповідає базовому блоку, що містить інструкції, які не можуть бути синтезовані в HW - наприклад, виклики операційної системи. Визначення: дійсний підграф. Дійсний підграф — це будь-який підграф CFG, який не містить забороненого вузла. На Рисунку 1.1b: B і C є дійсними підграфами ¬; A і D не є дійсними підграфами, оскільки вони містять заборонений вузол. Визначення: регіон. Область R CFG G є дійсним підграфом, так що існує одне ребро, що йде від вузла в У (G) \ У (R) до вузла в У (R) і а єдине ребро, що йде від вузла в У (R) до вузла в У (G) \ У (R). На Рисунку 1.1b: B є регіоном, а C ні. У рамках цього розділу всі і тільки регіони розглядаються як ¬кандидати на ідентифікацію прискорювачів. Враховуючи функцію переваг M() і вартість C () для кожного регіону, ми можемо сформулювати задачу вибору регіонів наступним чином: Проблема: вибір регіону Нехай R = {R 1 ,R 2 ,...,R n } — набір регіонів із пов'язаними функціями вартості та переваг C і M . Для будь-якої підмножини X ✓ {1,2,..., n} регіонів позначимо M (X) = ^ ieX M (R i) суму заслуг її регіонів, і позначимо C (X) = ^ тобто X C (R {) сума витрат її регіонів. Ми хочемо вибрати підмножину X регіонів таким чином, що 1. Жодні дві області, що належать до однієї CFG, не перекриваються, тобто У (R t) П У (Rj) = 0, для всіх 1 < i, j < n 2. Вартість C(X) знаходиться в межах наданого користувачем бюджету витрат C max 3. ПеревагаM(X) максимізована Це визначення проблеми відповідає тому, що ми визначили в попередньому розділі як мету розробника: враховуючи доступну область прискорювача, вилучити якомога більше обчислень, з обмеженням вимагати не більше, ніж ця область, щоб максимізувати результуюче прискорення

1.3 Алгоритми вибору

Проблема *вибору регіону* потребує попередньо ідентифікованого набору регіонів як вхідних даних. Для ідентифікації регіонів і, отже, збору такого набору повторно використовується існуючий перехід LLVM, який, у свою чергу, базується на алгоритмі лінійної часової складності, опублікованому в [38]. Тоді,

враховуючи доступний набір регіонів, потрібно вирішити більш дорогу з точки зору обчислень проблему *вибору регіону*, і алгоритми її вирішення пояснюються нижче.

Надається експоненціальний, точний метод розгалужень і меж, заснований на пошуку за двійковим деревом; по-друге, швидкий (поліноміальний) неточний, **жадібний** метод; по-третє, підхід зустрічі посередині, все ще експоненціальний, але масштабується швидше, ніж **точний** , який ми називаємо точною обрізкою **.**

Перш ніж заглибитися в пояснення алгоритму, на Рисунку 1.2 наведено приклад роботи. На Рисунку зображено CFG двох функцій і виділено п’ять областей, визначених у них (позначених A, B, C, D, E на Рисунку)

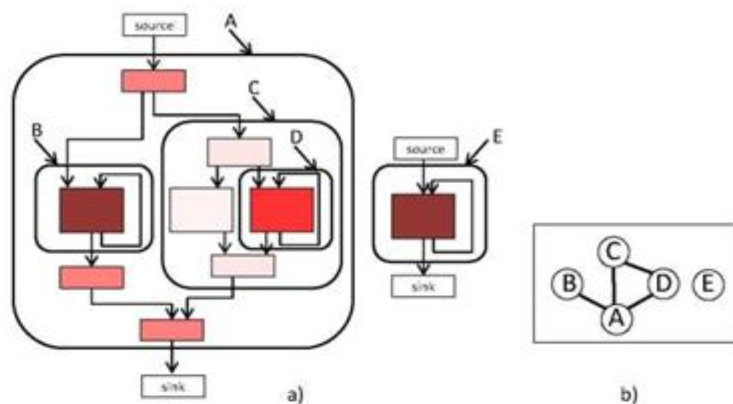


Рисунок 1.2 - а) Робочий приклад, що використовується для пояснення алгоритмів вибору: зображено ДФГ двох функцій, на яких виділено п'ять областей, позначених від A до E. б) Граф перекриття для п'яти областей.

Далі ми позначаємо O множину перекривань областей, тобто множину {(f, j) I V(R t) П V(Ry) Ф 0} . Ребра графіка, представленого на Рисунку 1.2 ç, відповідають набору O перекривань областей для поточного прикладу. 1.3.1

Точний метод У точному методі Вибір Проблемної Області зводиться до самостійної заданої задачі. Зокрема, ми будуємо неорієнтований граф G , де V(G) = {1,2, ...,n}, тобто існує один вузол для кожної області, і £(G) = O, тобто два вузли з’єднані, якщо відповідні області перекриваються. Легко бачити, що

множина {i 1г i 2 ,...,i r } задовольняє умову 1 задачі вибору області, якщо це незалежний набір G. Рисунок |1.2| b показує граф перекриття G , що відповідає поточному прикладу: область A перекривається з усіма областями, крім E, тому додаються ребра, що з'єднують A з B, C і D тощо. Прикладами незалежних наборів на цьому графіку є {A}, {B,D}, {B,C,E}. Алгоритм рекурсивно досліджує незалежні набори G, подібно до алгоритму Брона-Кербоша [11], і його кроки будуть прослідковані за допомогою поточного прикладу та відповідного дослідження дерева, показаного на рис. Алгоритм підтримує набір X, який є активним незалежним набором (ініціалізовано 0) і набір P доступних вузлів (ініціалізовано V(G)). На кожній ітерації алгоритм вибирає вузол u у P так, що C(lu {u}) < C max , тобто він задовольняє умову 2 проблеми *вибору регіону* та рекурсивно досліджує конфігурації

1. $X' = X \cup \{u\}, P' = P \setminus (\{u\} \cup N(u))$
2. $X' = X, P' = P \setminus \{u\}$

де $N(u) = (v \mid (u, v) \in E(G)\}$ — множина сусідів *и* у графі перекриття. Конфігурація 1 проходить усі незалежні множини, які містять *X* і u. Вибір *P'* зберігає цей інваріант, оскільки всі сусіди *и* видаляються з *P*. Натомість конфігурація 2 проходить усі незалежні множини, які містять *X* , але не u. Зауважте, що будь-який незалежний набір відвідується *лише один раз.*

Цей процес можна проілюструвати на Рисунку 1.3: корінь дерева представляє порожню множину, а множина *P* у цій точці містить усі регіони. Потім спочатку досліджується включення регіону A, а набір P оновлюється шляхом видалення всіх регіонів, що перекриваються з A: $P = (E\}$. Відповідно до переваг і витрат усіх регіонів у цьому прикладі, показаному в таблиці на Рисунку, також оновлюються переваги (60) і вартість (35) досліджуваного рішення.

У кожній точці дослідження новий вузол *и* розглядається для додавання в поточний незалежний набір. Якщо немає вузла *и,* який задовольняє умову 2 проблеми *вибору регіону* , алгоритм записує набір *X* і повертається назад,

оскільки *X* є максимальним щодо умови 2. Для поточного прикладу на рисунку 1.3 бюджет витрат C_{max} дорівнює 35. Отже, дослідження зупиняється на *X* = (A}, оскільки бюджет витрат було досягнуто, і повертається назад. Наступним вибраним регіоном є *B,* набори *X* і *P* знову оновлюються відповідно до *X = (B}* і *P* = (C, D, *E* }), і дослідження триває.

Оптимізація 1: щоб пришвидшити пошук, алгоритм підтримує максимальну перевагу M_{max} досліджених на даний момент незалежних наборів. Таким чином, якщо *M(X* UP *P*) < M_{max} , алгоритм може повернутися назад, оскільки жодна надмножина *X* не має переваги, більшої за максимальну, знайдену на даний момент. Цю оптимізацію можна побачити в роботі, серед іншого, у вузлі дерева, де *X = (B}* і *P* = (D,E}). Фактично, M_{max} становить 75 на той момент дослідження (його було досягнуто набором *X = (B, C* , E}), тоді як поточна перевага *M* ((B}) дорівнює 30, а залишковий потенційний приріст *P = (D, E* }) становить 40. M_{max} не може бути досягнуто, і алгоритм може назад.

Оптимізація 2: Щоб зробити наведену вище дію точного відсікання ефективною, алгоритм приймає стратегію вибору вузла *и* з максимальною перевагою серед тих, які задовольняють умову 2 проблеми *вибору регіону* . На практиці це означає, що регіони-кандидати розглядаються в порядку зменшення заслуг.

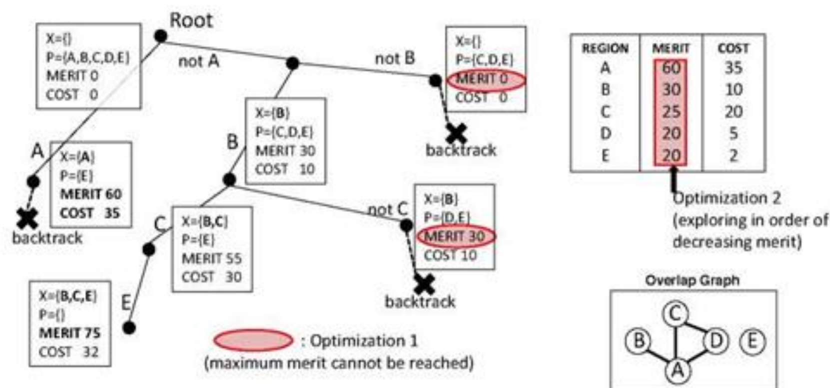


Рисунок 1.3 - Дослідження дерев, виконане за допомогою точного, Рисунок II.21, і для бюджету 35.

Наприкінці дослідження алгоритм повідомляє про набори, записані з перевагою, яка дорівнює M_{max}, тобто задовольняє умову 3 проблеми *вибору регіону*. У поточному прикладі це відповідає набору A = $\{B, C, E\}$.

Алгоритм, що реалізує жадібний вибір, підтримує набір A (ініціалізований 0), який є поточним частковим рішенням, і набір P доступних областей (ініціалізований {1,2,..., n}). На кожній ітерації алгоритм вибирає область u в P з найбільшою перевагою так, що C(AU {u}) < C_{max}, і продовжує до наступної ітерації з $X' = X$ U {u} і $P' = P \setminus$ ({u} U {v | (u, v) є O}). Вибір P' гарантує, що множина X задовольняє умову 1 на кожній ітерації. Якщо немає області u, що задовольняє умову 2, алгоритм завершує роботу та повідомляє X. У поточному прикладі на Рисунку |1.3j це відповідає простому припиненню дослідження на наборі A = {A}.

Оскільки жадібний метод ніколи не повертається назад, він часто потрапляє в пастку локальних мінімумів і тому не може гарантувати оптимальність. З іншого боку, він дуже швидко сходиться до рішення і використовується як наївна стратегія для отримання порівняльних результатів.

Попередні два алгоритми представляють два кінці спектра: точний і експоненціальний з одного боку; неточним, швидким і наївним з іншого. Третє рішення, який встановлює баланс між ними, походить із спостереження, що хоча список регіонів, визначених у заявці, є довгим — потенційно надто довгим для точної обробки — список значущих регіонів короткий, де під значенням -мається на увазі відчутний внесок у загальне прискорення. Іншими словами, розподіл регіонів щодо наданого прискорення дуже нерівний. Таким чином, третя альтернатива алгоритму полягає у застосуванні точного алгоритму, але лише до вирізаного списку регіонів у вхідних даних. На практиці це ¬відповідає ігноруванню ряду областей з низькою швидкістю в задачі відбору.

1.4 Framework RegionSeeker

Framework RegionSeeker — це автоматизована методологія, яка визначає

та вибирає кандидатів для апаратного прискорення з вихідного коду програми. Широкий аналіз програмного забезпечення, заснований на інфраструктурі компілятора LLVM, виконує, окрім ідентифікації, оцінку приросту продуктивності (заслуг), а також ресурсів апаратного забезпечення (вартість) кожного кандидата. Згодом, враховуючи обмеження апаратних ресурсів, відбувається вибір визначених апаратних прискорювачів, які максимально - підвищують сукупний приріст продуктивності.

Проаналізовано створений для цієї мети інструментальний ланцюжок LLVM; потім детально описано використану модель платформи та контрольні показники, використані для порівняльної оцінки.

Проходи аналізу RegionSeeker були побудовані в версії 3.8 *компілятора LLVM і інструментального ланцюжка* [12]. Інфраструктура LLVM стала основою компілятора для розробки моїх власних проходів аналізу, а також інструментів, які використовуються для профілювання. Пропуск *ідентифікації*

регіону , як показано в Алгоритмі 1, було розроблено для ідентифікації та надання початкової оцінки вартості та переваг визначених регіонів. Пропуск отримує як вхідні дані програми, розроблені на C або C++, і виконує аналіз на рівні проміжного представлення (IR).

ідентифікації регіону повторює кожну функцію наданої програми введення та, використовуючи наявний прохід *RegionInfo* LLVM [19], визначає регіони в кожній функції. Згодом заборонені вузли в регіонах ідентифікуються та позначаються, наприклад, системні виклики або виклики функцій, які не знаходяться в лінії. Області, що містять ці вузли, позначені як недійсні. І навпаки дійсні регіони оцінюються шляхом профілювання за допомогою інструментальної процедури. Профілювання за допомогою інструментарію вимагає генерації інструменталізованої версії коду, яка дає більш детальні результати, ніж профайлер вибірки. Використовуючи це формування, основні блоки анотуються в кожній функції з їхньою відповідною частотою виконання за допомогою проходу LLVM *ClrFreqCFGPrinter*.

Перепустка *ідентифікації регіону* також виконує ранню оцінку переваг і

вартості регіону, реалізовану безпосередньо в ланцюжку інструментів LLVM. Така оцінка базується на LLVM IR і не потребує будь-яких модифікацій вручну для виконання функцій, окреслених у вихідному коді тесту. Він оцінює вартість регіону як площу, необхідну для реалізації своїх вузлів DFG, і його переваги як цикли, збережені між виконанням ПЗ і УЗ, де останнє є затримкою вузлів на критичних шляхах DFG. Остаточним результатом нашого проходження аналізу є список дійсних регіонів або кандидатів на акселератор, кожен з яких містить приблизні переваги та вартість.

Вихідний список регіонів зберігається у файлі, який, у свою чергу, обробляється алгоритмами вибору **exact** , **greedy** і **exact-on-cropped** , реалізованими як окремі програми на C++.

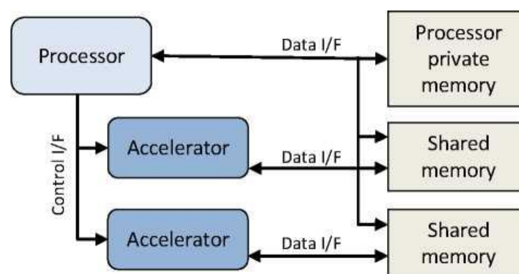


Рисунок 1.4 - Цільова модель ASIP

Дана модель включає в себе головний процесор, що поєднує спеціальні функціональні прискорювачі через інтерфейси керування та спільну пам’ять даних.

Перевага продуктивності, досягнута прискоренням для конкретної програми, залежить від багатьох цільових параметрів, включаючи прийняту ієрархію пам’яті, використовуваний протокол шини, стратегію з’єднання та кількість розглянутих процесорів, прискорювачів і пам’яті.

Щоб оцінити продуктивність RegionSeeker, було припущено, що система, що складається з одного процесора та кількох прискорювачів, обмінюється спільними даними з блокнотною пам’яттю (Рисунок 1.4). Процесор активує прискорювачі через інтерфейс із відображенням пам’яті, що вимагає транзакції

на системній шині. Після активації прискорювачі зчитують і записують дані в блокноти та з них, обчислюючи їхні виходи, до яких потім може отримати доступ процесор. Таким чином, переміщення даних навколо виконання прискорень не потрібне. Прискорювачі підключаються до блокнотної пам'яті з портами, які мають затримку в один такт. Інтерфейс керування між процесором і прискорювачами має затримку 10 тактів. Ці параметри відповідають тим, які використовуються інтерфейсами **ap_memory** та **s_axilite** відповідно, наданими Xilinx Vivado.

Час роботи неприскореної частини розглянутих контрольних тестів вимірюється за допомогою симулятора gem5 [8], що моделює процесор ARMv8-A з шириною випуску 1. Модель процесора є атомарною, з порядковим виконанням. Він пов'язаний з окремою пам'яттю інструкцій і даних із
затримкою доступу в один такт.

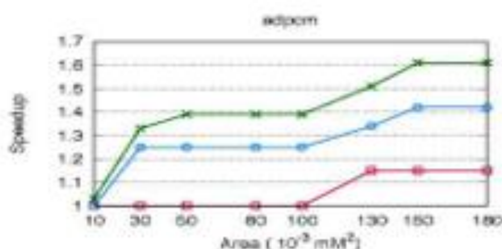
Інструменти високорівневого синтезу, як згадувалося у Вступі, вдосконалювалися протягом багатьох років, і після вибору цілі для прискорення вони можуть створювати відповідний екземпляр HW, який буде реалізовано в гетерогенній системі. У рамках нашої оцінки час виконання обладнання отримується за допомогою 1.4 дві різні структури HLS: симулятор Aladdin і комерційний набір інструментів Xilinx Vivado HLS. Aladdin націлений на впровадження ASIC. Це дозволяє швидко оцінити, але не створює синтезований список з'єднань як вихід; незважаючи на це, стверджується, що оцінки, запропоновані цим інструментом, знаходяться в межах 1% від тих, які отримані ¬з реалізації RTL [21]. Екземпляри апаратного забезпечення, створені за допомогою Vivado HLS, натомість призначені для проектів FPGA. Синтез-запуски в цьому рамках займають більше часу, але дають точні показники вартості (апаратних ресурсів) і переваг (прискорення) кожного прискорювача, а також прямий шлях до його реалізації. Вартість C () регіонів (і, для порівняння, базових блоків і функцій) обчислювалася як кількість необхідних ресурсів. Ми виразили їх у термінах площі IC у випадку Aladdin (мМ 2) і максимуму між їхніми необхідними тригерами та пошуковими таблицями на Virtex7 FPGA у випадку Vivado. Достоїнства M ()

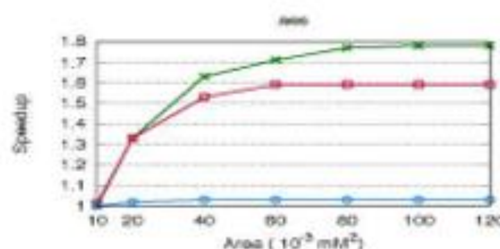
регіону було встановлено як різницю між апаратним і програмним часом виконання для всіх його викликів у програмі.

Були розглянуті реальні додатки різного розміру з набору тестів вбудованих програм CHStone [17]. adpcm виконує процедуру кодування, тоді як sha є безпечним хеш-алгоритмом шифрування, який широко використовується для створення цифрових підписів та обміну криптографічними ключами. aes — це алгоритм шифрування з симетричним ключем. gsm виконує лінійний прогнозний аналіз кодування, який використовується для мобільного зв'язку. dfmul і dfsin є меншими ядрами, які виконують множення подвійної точності з плаваючою комою та функції синуса, використовуючи тегерну арифметику. jpeg і mpeg2 є більшими програмами, які реалізують стиснення JPEG і MPEG-2
відповідно.

1.5 Експериментальні результати

У цьому розділі досліджуються та кількісно оцінюються результати та внески RegionSeeker з різних точок зору. По-перше, оцінюється прискорення, що випливає з розгляду регіонів як цілей для прискорення, з огляду на найсучасніші рішення, засновані на функціях і базових блоках. Потім аналізується продуктивність запропонованих алгоритмів для вирішення проблеми *вибору регіону* . Нарешті, досліджується надійність прискорення на основі регіону при зміні специфічних архітектурних параметрів.





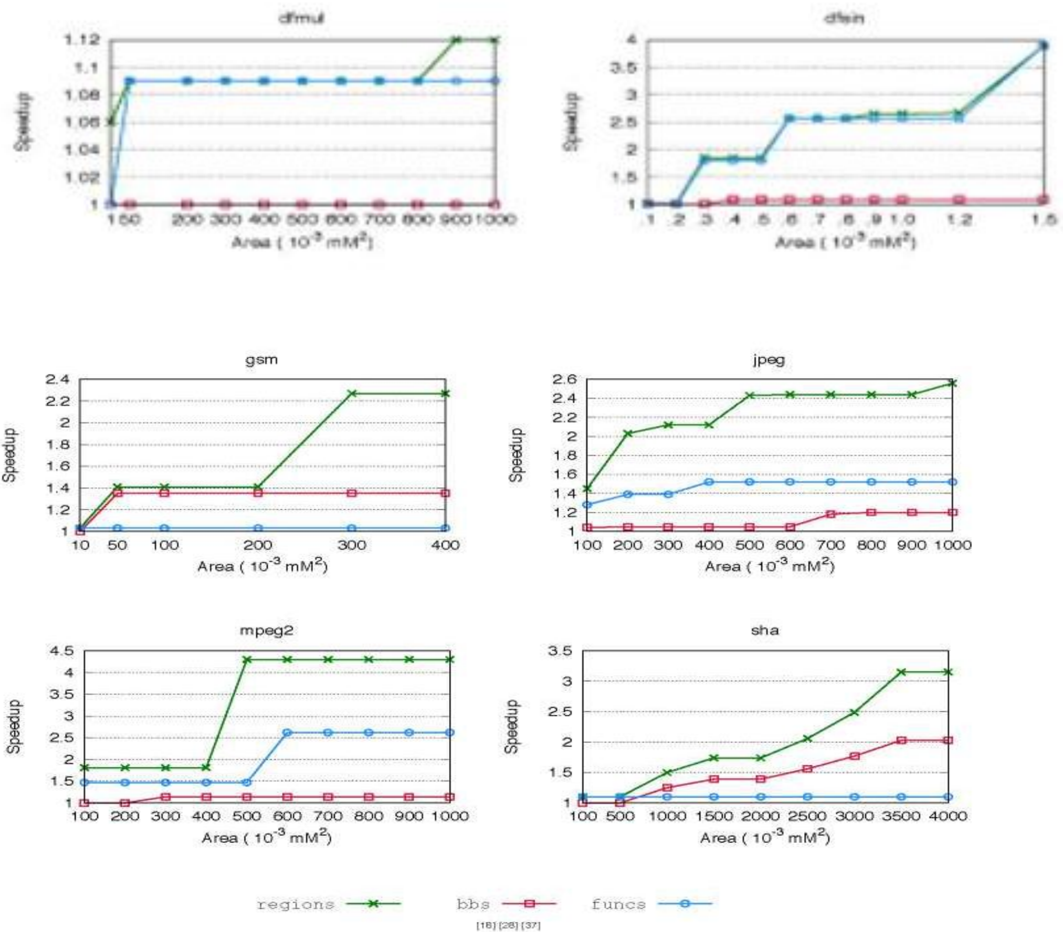


Рисунок 1.5 - Порівняння прискорень

Які були отримані на восьми контрольних тестах CHStone шляхом вибору регіонів, лише основних блоків і лише функцій, зміни обмеження області, використання Aladdin і gem5 для оцінки переваг і вартості

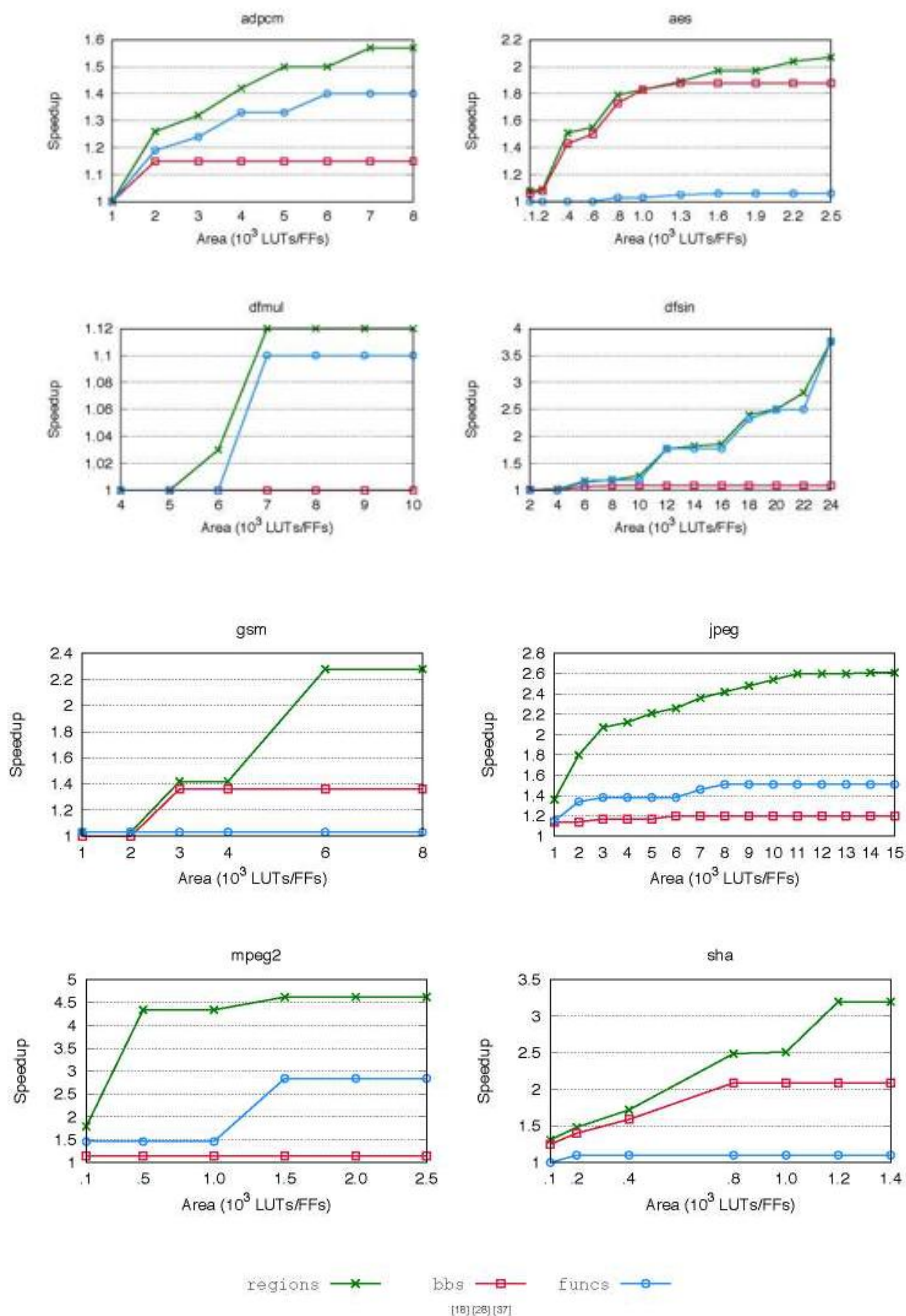


Рисунок 1.6 - Порівняння прискорень,

Які отримані у восьми контрольних тестах CHStone шляхом вибору регіонів, лише основних блоків і лише функцій, зміни обмеження області, використання Vivado_HLS і gem5 для оцінки переваг і вартості

Щоб оцінити переваги RegionSeeker, ми порівнюємо його з двома найсучаснішими альтернативами. Перший полягає в тому, щоб ідентифікувати прискорювачі *автоматично* , але лише в межах потоку даних — тобто в межах

окремих базових блоків — як це робиться за допомогою найсучасніших підходів, таких як [17], [28], [12], і [11], щоб назвати лише деякі. Зокрема, реалізовано найсучасніший алгоритм, запропонований у [12] і використаний у [28] і в [10], який ідентифікує максимальні опуклі підграфи в базових блоках. Ці методи - ідентифікують найбільшу частину, яку можна синтезувати та прискорити в базовому блоці, і, отже, являють собою верхню межу прискорення, якого можна досягти методами ідентифікації, які працюють на рівні потоку даних (базового блоку). По-друге, це імітація *ручного* підходу вибору цілих функцій, що також є сферою, що підтримується інструментами синтезу високого рівня [15] [23] [25].

Було проведено дві серії експериментів: спочатку Aladdin, а потім Vivado HLS використовувалися для оцінки переваг і вартості, підкреслюючи, що методологію RegionSeeker можна використовувати в різних інструментах синтезу високого рівня, і, що більш важливо, перевіряючи, що вибрані регіони в основному однакові, незалежно від використовуваної моделі оцінки вартості та переваг. В експериментах використовувався RegionSeeker із методом **точного вибору обрізки**, відкидаючи регіони, які забезпечують менше 10% максимальної ефективності.

На рисунку 1.5 показано прискорення, досягнуте за допомогою Aladdin за допомогою прискорювачів, вибраних RegionSeeker (на рисунку позначено - **регіонами),** щодо всього часу виконання програм і для різних обмежень області. Для додатків малого та середнього розміру, таких як **adpcm** , **aes** , **gsm** і **sha** , коефіцієнт прискорення RegionSeeker коливається від 1,6x до 3,2x. Для менших ядер можна спостерігати більші варіації, оскільки для **dfmul** і **dfsin** прискорення досягає 1,12x і 3,9x відповідно. Нарешті, для більших тестів, таких як **jpeg** і **mpeg2,** прискорення є досить значним: за допомогою RegionSeeker можна досягти 2,5x для першого та до 4,3x для останнього.

Подібні тенденції спостерігаються, коли замість цього для синтезу прискорювача використовується Vivado HLS, як показано на Рисунку 1.6 RegionSeeker стабільно перевершує найсучасніші підходи, які націлені або на окремі базові блоки, або на цілі функції, у всіх тестах. Ці результати

підкреслюють, що на досяжні прискорення значною мірою впливає те, які сегменти додатків вибрано для прискорення, і що на такий вибір лише незначно впливає прийнятий інструмент оцінки переваг і вартості. Насправді було підтверджено, що в двох наборах експериментів вибрані регіони були *однаковими* у 80% випадків. Наприклад , із 10 регіонів, вибраних для досягнення 2,2-кратного прискорення для порівняльного тесту jpeg 1.58 є однаковими при використанні Aladdin або Vivado HLS для вимірювання та оцінки витрат, а ті, що відрізняються, сприяють менш ніж 14% наданого приросту. Прискоренню, яке можна отримати за рахунок прискорення базових блоків, перешкоджає їх невелика деталізація і, як наслідок, велика кількість перемикачів між програмним і апаратним виконанням. Більше того, у цьому налаштуванні багато можливостей оптимізації під час апаратної реалізації прискорювачів втрачено, оскільки вони виникають лише тоді, коли розглядається потік керування, як натомість у випадку з регіонами. З іншого боку, прискорення, отримане шляхом вибору цілих функцій, відстає від тієї, що відповідає регіонам, з двох причин. По-перше, вибір функцій обмежено тими, які не містять заборонених вузлів, і це може виключити перспективні регіони в них. По-друге, що більш важливо, він також негнучкий з точки зору області, що особливо помітно, коли для прискорення доступна лише невелика кількість апаратних ресурсів. У таких випадках вибір функцій часто виявляє лише кілька можливих кандидатів із невеликою перевагою (наприклад, у jpeg та mpeg2 для площі менше 0,5 мМ 2). Це обмеження не існує для регіонів, оскільки можна вибрати лише частину, що стосується окремих гарячих точок у функції. Дійсно, продуктивність RegionSeeker випливає з високої гнучкості підходу до вибору, оскільки він дозволяє розглядати весь спектр деталізації, починаючи від цілих функцій і закінчуючи окремими циклами, що зрештою дозволяє краще використовувати прискорення для певного бюджету області.

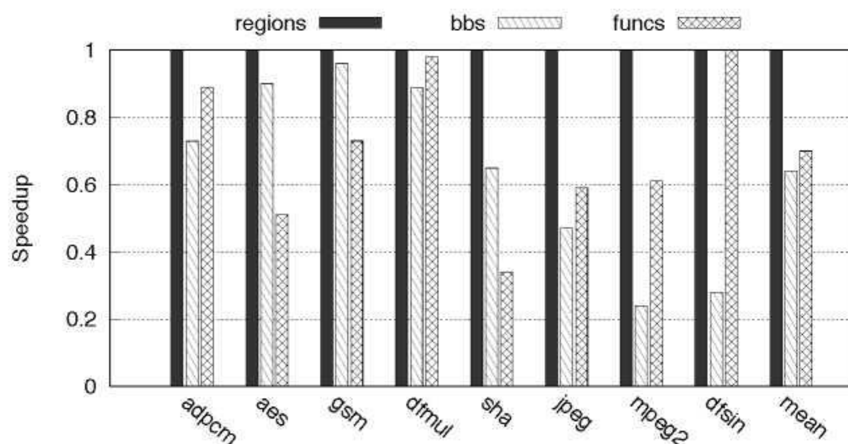


Рисунок 1.7 - Нормована швидкість роботи RegionSeeker

залежно від вибору функції та базового блоку, враховуючи для кожного бенчмарку фіксоване обмеження на площу. Синтез за допомогою Vivado HLS

Підсумок проведених експериментальних досліджень представлено на рисунку 1.7. Він повідомляє про нормалізоване прискорення, отримане RegionSeeker, порівняно з ідентифікацією основного блоку та функції, коли використовується максимальний бюджет розглянутої області та Vivado_HLS. ¬Крайній правий набір стовпців ілюструє, що в середньому RegionSeeker використовує приблизно на 30% вищі скачки швидкості щодо двох базових методів. Крім того, хоча в деяких випадках базові показники збігаються з продуктивністю RegionSeeker (наприклад: gsm для базових блоків, dfsin для функцій), жоден з них не може зробити це послідовно в різних ¬обмеженнях області та в різних програмах, демонструючи придатність областей потоку керування як кандидатів на прискорювач.

У Розділі 1.3 було представлено три алгоритми вибору: точний алгоритм, який може не масштабуватися для великих тестів, наївний жадібний підхід і підхід «зустріч посередині», де точний алгоритм застосовується лише до вирізаного списку регіонів, на відміну від усіх регіонів тесту. У цьому розділі оцінюється продуктивність цих алгоритмів з точки зору масштабованості та якості знайденого рішення.

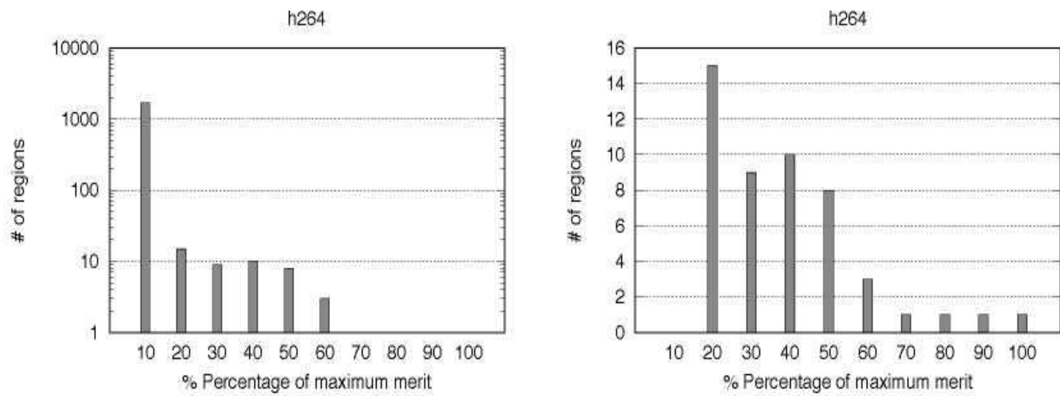


Рисунок 1.8 - Ліворуч: 1745 регіонів H.264 розділені тут на десять бункерів відповідно до їх якості ***M*** ().

Регіони, які забезпечують до 10% максимальної заслуги, потрапляють у перший бак, від 10% до 20% – у другий бак тощо. Зауважте, що розподіл дуже спотворений. Праворуч: дев'ять найприбутковіших бункерів тут показані в лінійному масштабі для ясності. Нерівність цього розподілу враховується алгоритмом **точного** відбору.

З цією метою націлений складний тест (а саме H.264 [25]), який має розмір коду понад десять тисяч рядків коду та містить тисячі регіонів. Зауважте, що хоча Aladdin надає оцінку ефективності та вартості для регіону дуже швидко (лічені мілісекунди на регіон) і, отже, міг би бути використаний для цього експерименту масштабованості алгоритму, він, однак, вимагає окреслення вибраного прискорювача саме *як виклик функції* у вихідному коді програми - і це наразі потрібно робити вручну. Це технічне обмеження означає, що оцінки Aladdin не можна використовувати для експериментів зі здатністю масштабування в цьому розділі. Таким чином, була застосована більш абстрактна модель, яку можна було реалізувати безпосередньо в ланцюжку інструментів LLVM, спираючись на проміжне представлення LLVM і без необхідності ручного втручання в вихідний код тесту. Вартість регіону оцінювалася як площа, необхідна для впровадження його вузлів DFG, а його достоїнства як цикли, збережені між виконанням програмного та апаратного забезпечення, де останнє є затримкою вузлів на критичних шляхах DFG, кожного з яких помножено на їх відповідну частоту.

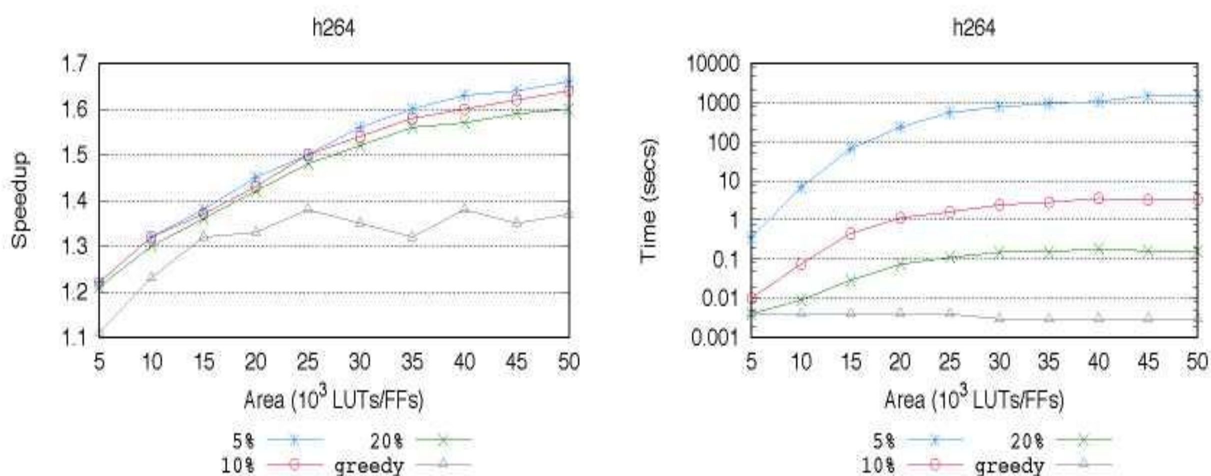


Рисунок 1.9 - Ліворуч: прискорення, досягнуте в еталонному тесті H.264 за допомогою **точного обрізання** , обрізання з урахуванням лише регіонів, які забезпечують щонайменше 5%, 10% і 20% максимальної ефективності, а також **жадібним** . Праворуч: час виконання відповідних алгоритмів

По-перше, на рисунку 1.8 показано розподіл усіх регіонів для еталонного тесту H.264 щодо їх якості *M* (). На цьому Рисунку стає очевидним, що розподіл надзвичайно спотворений: дуже небагато регіонів мають високі якості, а більшість – незначні. Це слід було очікувати, оскільки добре відомо, що великий відсоток часу під час роботи програмного додатку зазвичай витрачається на невеликий відсоток коду. І, звичайно, перевага регіону пропорційна частоті виконання відповідного сегмента коду.

Для великого тесту, такого як H.264, **точний** алгоритм не масштабується, якщо він подається на всі 1745 регіонів. Однак розумно очікувати, що, враховуючи розподіл , що спостерігається, добротність знайденого рішення лише трохи зменшиться при відкиданні великої кількості низькопотенційних регіонів, тоді як масштабованість може відчутно зрости. Це припущення підтверджується даними на рисунку 1.9, де виконується порівняння продуктивності та масштабованості алгоритму вибору **точного обрізання** на трьох різних рівнях обрізання, тобто враховуючи будь-які регіони, що забезпечують щонайменше 5%, 10% і 20% максимальної якості відповідно.

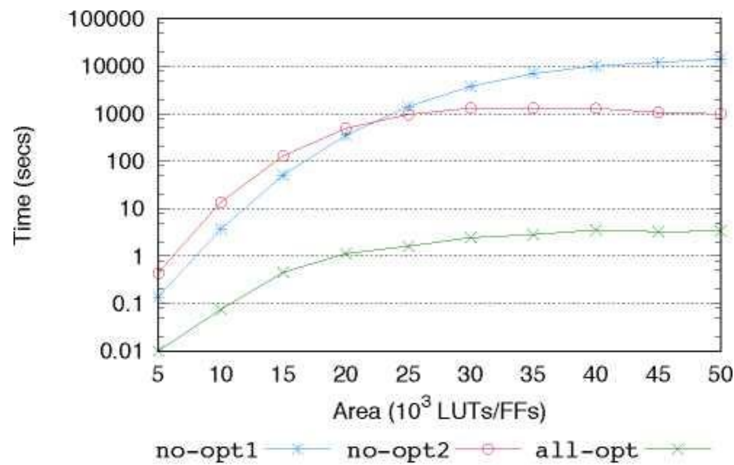


Рисунок 1.10 - Час виконання алгоритму **точного кадрування** (рівень кадрування 10%), коли дві оптимізації, описані в підрозділі 1.3.1, видалено

На рисунку 1.9 (ліворуч) показано очікуване прискорення, досягнуте впровадженням набору регіонів в апаратному забезпеченні, вибраних трьома рівнями **точного обрізання** та **жадібним** для різних обмежень області. Можна побачити, що при зміні рівня обрізання ефективність рішень, знайдених за допомогою **точного обрізання,** лише незначно відрізняється (як і очікувалося, мало що втрачається, коли ігноруються деякі області з низьким потенціалом), і в цілому воно значно перевершує швидке, але наївне **жадібне** .

Однак, як друге питання, на яке варто звернути увагу, простір дослідження для трьох різних рівнів кадрування значно відрізняється, а отже, і час, витрачений кожним із цих алгоритмів на завершення. Це можна побачити на Рисунку 1.9 (праворуч): порядки величини розділяють час, витрачений кожним, з **точним обрізанням** на 5% займають години, а на 10% – секунди.

Нарешті, показано ефект оптимізації, розробленої для покращення дослідження дерева пошуку. Ці оптимізації були описані в розділі 1.3.1 і показано на рисунку 1.3. Вони 1: обрізання дерева дослідження, коли певний найкращий результат, знайдений на даний момент, не може бути досягнутий, і 2: обробка регіонів у порядку зменшення результату. На Рисунку 1.10 можна побачити, як дві оптимізації впливають на час роботи алгоритму. При вимкненій обрізці більше трьох порядки величини втрачаються в часі. Коли список регіонів обробляється в порядку, відмінному від порядку зменшення ефективності (у

цьому експерименті розглядається порядок зменшення щільності g , тобто перевага поділена на вартість), втрачається більше ніж два порядки величини.

Ініціація прискореної процедури на виділеному апаратному блоці завжди тягне за собою тимчасове покарання T Overhead . Такі накладні витрати сильно залежать від ¬протоколу інтерфейсу між процесором і прискорювачами для конкретної програми. Хоча визначення такого протоколу виходить за рамки цієї роботи, варто дослідити вплив прийняття різних значень для цього параметра, коли використовуються різні методи відбору. повторно портовано для тесту sha. По-перше, зміна значення T Overhead між одним, десятьма та двадцятьма циклами не впливає на прискорення, отримане RegionSeeker, у той час як на прискорення для базових блоків це справді впливає. Це цілком очікувано, оскільки прискорювачі базового рівня блоку потребують більшої кількості викликів (наприклад, для кожної ітерації інтенсивного циклу), ніж прискорювачі регіонального рівня. По-друге, хоча зі зменшенням значення T Overhead прискорення основного блоку збільшується, воно не збільшується суттєво і все ще менш відчутне, ніж прискорення, досягнуте RegionSeeker

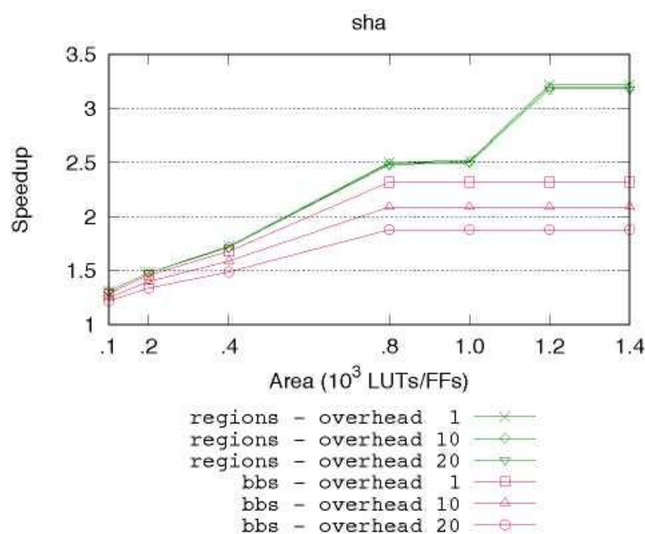


Рисунок 1.11 - Вплив накладних витрат на ініціалізацію для стратегій вибору регіонів та базових блоків, враховуючи значення TOverhead в 1, 10 та 20 тактів.

1.6 Висновки по розділу

Була описана нова методологія RegionSeeker, яка представляє ідеї, що стосуються дослідницького питання про те, яку частину або які частини програми слід синтезувати в HW за обмеженого бюджету області. RegionSeeker виконує автоматичну ідентифікацію та вибір апаратних прискорювачів, деталізація яких знаходиться в межах графа потоку керування функції. Включення потоку керування до відбору кандидатів для прискорення HW на виставці забезпечило відчутне підвищення продуктивності до 2 разів у порівнянні з найсучаснішими запропонованими методами та досягнуто прискорення до 4,5 разів порівняно з виконанням лише програмного забезпечення.

РОЗДІЛ 2 . АВТОМАТИЧНА ОПТИМІЗАЦІЯ ДЛЯ СПІЛЬНОГО ПРОЕКТУВАННЯ HW/SW

Виявлення хороших кандидатів для апаратного прискорення є першим кроком до реалізації гетерогенних дизайнів обчислювальних систем, які пропонують підвищену продуктивність порівняно з однорідною системою, обмеженою програмним процесором(ами) загального призначення. Однак, як ми бачили, набір оптимізацій, застосованих до апаратних прискорювачів, може ще більше скоротити час обчислень, що призведе до покращення продуктивності порівняно з неоптимізованими реалізаціями апаратних прискорювачів за замовчуванням. Сучасні інструменти високорівневого синтезу можуть застосовувати такі оптимізації до апаратних прискорювачів і підвищувати продуктивність їх реалізацій, а також загальну продуктивність усієї гетерогенної системи. Інструменти HLS, такі як Vivado HLS [85], якими б ефективними вони не були, вимагають від програміста багато ручних рішень, коли справа доходить до вибору *способу* синтезу цих прискорювачів. Крім того, визначення того, які оптимізації можна застосувати до апаратних прискорювачів, може бути складною проблемою, оскільки це сильно залежить від характеристик кожного апаратного прискорювача.

Щоб зробити автоматизацію ще одним кроком вперед у спільному проектуванні HW / SW, у рамках цієї частини мого дослідження я розглянув проблему автоматизації процесу прийняття рішень щодо того, які оптимізації слід застосовувати до кандидатів на прискорення HW у певному контексті. потенційні оптимізації, як уже згадувалося, включають керування пам'яттю даних, що споживаються та створюються апаратними прискорювачами, набір оптимізацій, націлених на цикли (наприклад, конвеєрування циклу, розгортання циклу, зведення циклу тощо), конвеєрне розміщення послідовних фрагментів обчислень, таких як наступні виклики функцій або тіла циклу послідовних ітерацій, оптимізація масиву, наприклад розбиття масиву в блоки однакового розміру. Серед різноманітних оптимізацій я зосередився на двох категоріях: а) Аналіз

повторного використання даних і b) Прогнозування фактора розгортання циклу. Ці випадки пояснюються більш детально в наступних двох розділах.

2.1 Аналіз повторного використання даних

Петлі є ідеальними кандидатами для прискорення. Майже в кожній програмі існує кілька циклів, які містять велику кількість ітерацій, і під час їх виконання витрачається велика кількість обчислювального часу. Крім того, існують вкладені цикли, які зазвичай демонструють високий рівень повторного використання даних. Ефективне використання повторного використання даних у послідовних ітераціях циклів може значно знизити необхідний обсяг обміну даними між апаратними прискорювачами та основною пам'яттю, таким чином зменшуючи пропускну здатність до та від прискорювачів і підвищуючи їх продуктивність.

Програми з ковзними вікнами, поширені в області обробки зображень, є звичайною ціллю для прискорення. Приклад багаторазового використання даних можна спостерігати в цих програмах, де зазвичай існує вікно доступу, яке сканує більш широкий домен, наприклад двовимірний масив. З огляду на те, що рівень і модель повторного використання даних відомі апріорі, можливо розробити спеціальні структури пам'яті, також відомі як буфери пам'яті, приєднані до апаратних прискорювачів. Ці буфери пам'яті можуть використовувати повторне використання даних, зберігаючи дані локально та, отже, мінімізуючи затримку пам'яті через зв'язок із основною пам'яттю.

Повторне використання даних у високорівневому синтезі все ще знаходиться на передчасній стадії. Сучасні методи [20] покладаються або на ручне переписування вихідного коду, що передує HLS, або на переклад з джерела на джерело [21] [20], і тому погано інтегровані в ланцюги інструментів HLS.

Методологія, детально описана в підрозділі 2.1.3, намагається подолати цю прогалину. Він представляє керовану компілятором структуру, засновану на бібліотеці LLVM Polly [29], здатну автоматично ідентифікувати потенціал

повторного використання даних в обчислювальних ядрах, щоб керувати синтезом складних HW-прискорювачів.

У сфері ідентифікації автоматичних прискорювачів дослідження поки що зосереджувалися переважно на прискоренні потоку даних [28] [23], не беручи до уваги потенціал оптимізації доступу до пам’яті. Виняток становлять роботи [9] [34], де автори підтверджують твердження, що прискорювачі з 2.1 Tom Storage може забезпечити кращу швидкість порівняно з тими, які прискорюють лише потік даних. Однак ці документи зосереджені на ідентифікації прискорювачів і не представляють методологію автоматичної ідентифікації оптимізаційного потенціалу, а також синтезу їх відповідно. У додатках із ковзними вікнами наукові та промислові кола досліджують можливість повторного використання даних. Інтелектуальні буфери [22], створені компілятором ROCCC [23], дозволяють автоматично виявляти можливості повторного використання даних, але не можуть бути пов’язані з інтерконнектами різної ширини. Методологія , описана в [28], використовує буфери повторного використання, що охоплюють кілька стовпців кадру, що створює значні накладні витрати на область. І [22], і [28] не можуть поєднати -апаратне розгортання та конвеєрне забезпечення, які натомість спільно підтримуються методологією, детально описаною тут. Альтернативний підхід, описаний в [23], вимагає великої кількості апаратних ресурсів, оскільки вимагає зберігання великих частин кадру, що обробляється всередині спеціального обладнання. У [20] автори пропонують аналітичний метод збору параметрів мікроархітектури для додатків із ковзним вікном на FPGA. Однак їх дизайн зрештою потрібно реалізовувати вручну, і, отже, робота нехтує аспектами високорівневого синтезу. Комерційний інструмент Vivado HLS вимагає значного ручного переписування вихідного коду, щоб створити екземпляр повторного використання буфера пам’яті. Натомість представлений тут підхід -покладається на автоматичний аналіз коду для визначення характеристик -цільової програми.

Щоб створити спеціальні прискорювачі зберігання, ми використовуємо двоетапну методологію, де спочатку виконуємо аналіз програмного забезпечення, а потім синтезуємо частину обчислень, яка буде реалізована в HW.

Етапи аналізу та синтезу програмного забезпечення зображені на рисунку 2.1 разом із процедурою оцінювання, якої ми дотримуємося. Перший етап включає аналіз повторного використання статичних даних, тоді як на етапі синтезу надаються деталі щодо реалізації апаратного забезпечення. Ці етапи призводять до розробки та впровадження спеціальних прискорювачів зберігання, яким вдається мінімізувати затримку через передачу даних між основною пам'яттю та апаратними прискорювачами.

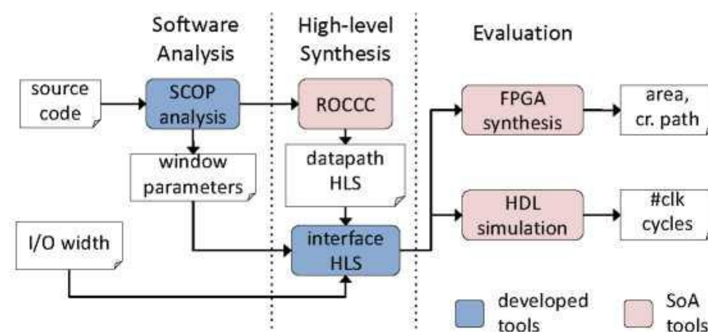


Рисунок 2.1 - Блок-схема нашого каркаса.

Можливості повторного використання даних. Наша структура використовує апаратне розгортання та конвеєрне забезпечення, щоб досягти високого рівня повторного використання даних між ітераціями. рисунок 2.2 ілюструє можливість повторного використання cept з точки зору високого рівня, припускаючи, без втрати загальності, що ковзне вікно рухається спочатку у вертикальному напрямку, а потім, у кінці кожного стовпця кадру, у горизонтальному.

Аналіз коду отримує параметри ядер SCoP, які використовуються разом із обмеженнями вводу/виводу для керування автоматизованим проектуванням високоефективних апаратних прискорювачів.

Віконні програми обчислюють вихідні значення з підмножини цілого кадру, локалізованого в невеликому двовимірному блоці. Таким чином, можна обмежити внутрішнє зберігання локальної пам'яті прискорювача даними, що використовуються в одному вікні, що називається *керованим набором,* що призводить до компактної апаратної реалізації (рис. 2.2а).

Тим не менш, завдяки використанню більшої локальної пам’яті дані, необхідні для кількох вікон, можна зберігати одночасно. Ми спостерігаємо (рисунок 2.2b) , що керовані набори горизонтально суміжних вікон сильно перекриваються, відрізняючись лише кількістю стовпців елементів, що дорівнює вертикальному кроку програми. Таким чином, можна підтримувати кілька (перекриваються) керованих наборів з невеликими витратами на розмір локального буфера. Потім кожне вікно, укладене в буфер, може оброблятися паралельно іншим шляхом обробки даних, реалізуючи розгортання з повторним використанням даних в апаратному забезпеченні.

Повторне використання даних також присутнє у вертикальному вимірі. Насправді керований набір наступних ітерацій відрізняється кількістю рядків, що дорівнює горизонтальному кроку (рис. 2.2c). Це джерело повторного використання даних можна ефективно використовувати за допомогою апаратної конвеєрної обробки, реалізуючи локальне сховище як регістр зсуву по рядках.

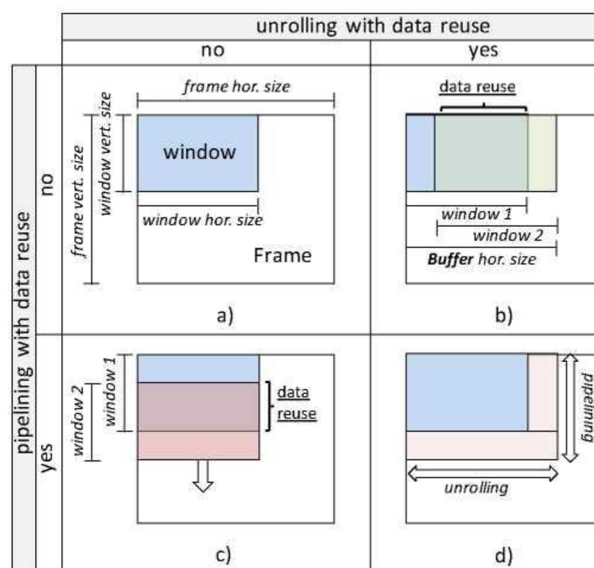


Рисунок 2.2 - На кожній ітерації програми ковзного вікна обробляють підмножину вхідних даних (a). Керований набір наступних ітерацій має високий ступінь перекриття як у горизонтальному (b), так і у вертикальному (c) вимірах. Наша структура автоматично використовує обидва, максимізуючи повторне використання даних (d).

Наша структура підтримує обидва типи повторного використання даних одночасно (Рисунок 2.2d). У цьому параметрі оновлення керованого набору передбачає передачу з основної пам’яті даних, що відповідають рядку буфера (на відміну від рядка вікна на рисунку 2.2c). Ми використовуємо цей додатковий ступінь свободи, щоб адаптувати як локальну структуру зберігання, так і кількість реалізованих шляхів даних відповідно до введення/виведення ширина комунікаційного інтерфейсу (тобто: кількість елементів даних, які можна одночасно читати або записувати).

Аналіз повторного використання даних

Щоб ідентифікувати розділи коду, що відповідають ядрам ковзних вікон, граф потоку керування (CFG) програми аналізується, шукаючи гнізда циклів. Потім бібліотека LLVM Polly [19] використовується для перевірки того, чи структура гнізд CFG є SCoP (Static Control Part), яка є підграфом CFG, де потік керування відомий статично. Якщо це так, його поліедральну модель, отриману Поллі, аналізують, щоб забезпечити параметри SCoP, необхідні для її апаратної реалізації .

Щоб автоматизувати використання повторного використання даних у SCoP, інформацію про розмір вікна, крок і розмір кадру потрібно зібрати з вихідного коду програми. Розмір вікна визначає шаблон доступу всередині самого внутрішнього тіла циклу. Крок у самій внутрішній і крайній петлі — це значення збільшення змінної індукції для внутрішньої та крайньої петлі відповідно. Нарешті, розмір кадру визначається як простір ітерації, в якому рухається ковзне вікно. Щоб отримати ці значення, ми розробили проходження аналізу компілятора **Алгоритм 2** LLVM Analysis Pass - ідентифікація SCoP та аналіз повторного використання даних **Введення:** програма, написана на C/C++

Вихід: список ідентифікованих SCoP та їхні відповідні розміри/значення рами, вікна, кроку.

```
1:  function RunOnRegion()
2:      getAnalysis(ScopInfo)
3:      scop = getScop()
4:      RunOnScop(scop)
5:
6:  function RunOnScop(scop)
7:      LI = getLoopInfo()
8:      SE = getSE()
9:      if L == OutermostLoop then
10:         getTripCountFortLoop()
11:         getStrideForLoop()
12:     else if L == InnerMostLoop then
13:         getReadMemoryAccesses()
14:         ComputeDistancesForReadAccesses()
15:         ComputeWindowSize()
```

на можливості, які пропонує структура LLVM Polly [79]. Спеціальні параметри програми розглядаються в поєднанні з обмеженнями архітектури - (ширина входу/виводу) для автоматичного синтезу ефективних прискорювачів SCoP.

Пропуск аналізу, який ми розробили, повторює області функцій додатків, визначені Polly як статичні частини керування. Як повідомляється в алгоритмі 2, для кожного циклу SCoP і Scalar Evolution (SE) інформація витягується з поточного тіла циклу за допомогою проходів аналізу, наданих LLVM. Інформація про цикл надає глибину циклу, і, отже, про те, чи є цикл внутрішнім чи зовнішнім, інформація SCoP SE включає метод підрахунку циклу, який обчислює простір ітерації для кожного циклу. Ця інформація дозволяє обчислити розмір кадру.

Горизонтальний і вертикальний крок SCoP обчислюється функцією *getStrideForLoop* , яка приймає як аргумент базовий блок, що відповідає тілу циклу. Ми розробили його, використовуючи метод *getStride*, включений у проходження LLVM ScopInfo Analysis, і функції, включені в Integer Set Library (isl) [11].

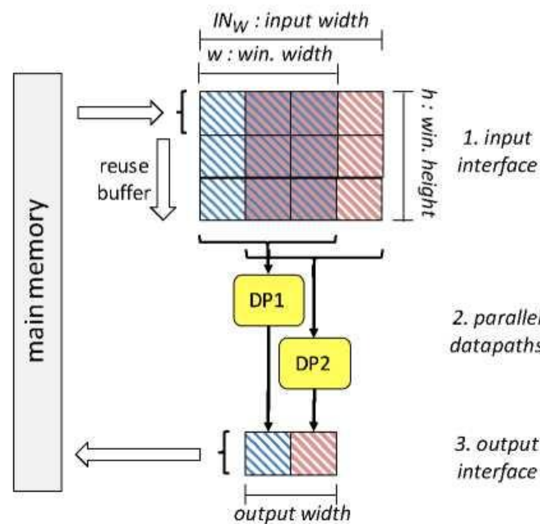


Рисунок 2.3 - Автоматично згенеровані прискорювачі включають кілька шляхів даних, які виконують тіло циклу двовимірних SCoP

Нарешті, доступи до пам’яті для читання, що знаходяться у самому внутрішньому тілі циклу, ідентифікуються за допомогою функцій isl у нашій власній функції ComputeDistancesForReadAccesses. Ми обчислюємо відстань (або дельту) кожного з цих доступів відносно першого ідентифікованого. З шаблону доступу розмір вікна обчислюється як його мінімальний охоплюючий прямокутник. Апаратна реалізація Параметри, отримані під час проходу аналізу (розмір кадру, крок, горизонтальний і вертикальний розмір вікна) і характеристики з’єднання (вхідна і вихідна ширина) використовуються для отримання ефективної реалізації прискорювачів HW з локальним зберіганням і повторним використанням даних. Запропонована реалізація показана на Рисунку 2.3. Прискорювачі вбудовують кілька комбінаторних шляхів даних, кожен з яких виконує одну ітерацію тіла циклу цільової програми. Інтерфейс введення містить локальне сховище, горизонтальний розмір якого відповідає доступній ширині вхідних даних IN W елементів даних, тоді як його вертикальний розмір дорівнює вертикальному розміру вікна програми h. Він реалізований як регістр зсуву IN W * h , що працює у вертикальному (зверху вниз) напрямку. Під час виконання перший рядок регістра зсуву заповнюється вхідними даними в кожному такті. Підмножина Елементів, що зберігаються в регістрі зсуву, підключається до

кожного з різних шляхів даних відповідно до їх керованих наборів, наприклад: перший має входи, що відповідають ¬стовпцям буфера в діапазоні від 0 до w — 1 (горизонтальний розмір вікна), а другий відповідає стовпцям буфера в діапазоні від 1 до w. Рисунок 2.3 ілюструє таку схему для простого випадку IN W = w + 1. На початку виконання h рядків зберігаються в регістрі зсуву перед активацією логіки шляхів даних. Після цього ця активація виконується для кожного нового рядка, відкидаючи останній (верхній) рядок і зберігаючи новий у першій (нижній) позиції регістра зсуву. Після завершення вертикального ковзання вікна через раму починається нове, збільшуючи горизонтальне переміщення буфера на IN W — w + 1 елемент. Нарешті, оскільки для виходів немає можливостей повторного використання, вихідний ¬інтерфейс просто об’єднує значення, згенеровані шляхами даних, і передає їх як один і широкий доступ до пам’яті.

Порівняння запропонованого підходу проведено з двома найсучаснішими інструментами HLS: ROCCC і Vivado HLS. Vivado HLS порівнюється у двох режимах, один — за замовчуванням (Vivado_norew), а інший — після значного ручного переписування вихідного коду (Vivado_rew) для отримання додаткових даних.

Оцінка нашої методології виконується за трьома тестами різного розміру вікна. Sobel — це алгоритм виявлення країв із шаблоном доступу вікна 3x3. BlockSAD є ядром у H.264 і використовується для виявлення подібності між блоками 4x4 . Нарешті, максимальний фільтр обчислює найяскравіший піксель серед сусідів у блоках 8x8. Було розглянуто три різні конфігурації, які охоплювали один шлях даних і мінімальну ширину вхідного сигналу (Conf.l) до декількох шляхів даних і збільшену ширину вхідного сигналу (Conf.2 і Conf.3), як підсумовано в таблиці 2.2. Кілька шляхів даних перетворюються на виконання більшої кількості паралельних вікон і, отже, збільшення попиту на ресурси області.

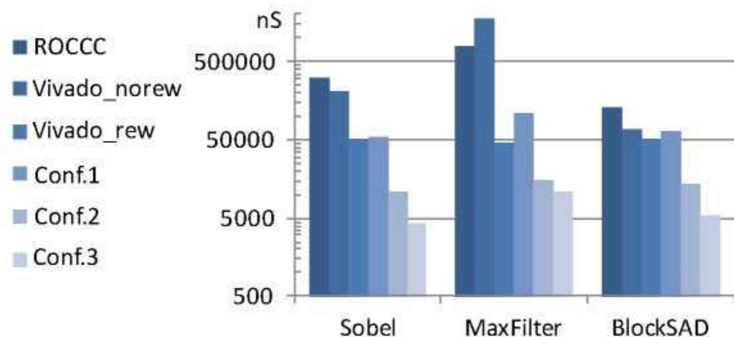


Рисунок 2.4 - Час виконання для обробки кадру 100x100

Час виконання. Час виконання, показано на рисунку 2.4 отримано з цільової платформи Xilinx Virtex7 FPGA. Можна помітити, що системи ROCCC мають подібну продуктивність до систем Vivado_norew. Прискорювачі Conf.l – навіть хоча вони не вимагають модифікації коду - такі ж ефективні, як і Vivado_rew. Конф.2 та Конф. 3, що представляє структуру, представлену в 2.1.3, різко ¬зменшує час виконання, із прискоренням на порядок величини в середньому між Conf.1 і Conf.3. Інші найсучасніші інструменти не можуть забезпечити еквівалентне рішення з таким малим часом виконання.

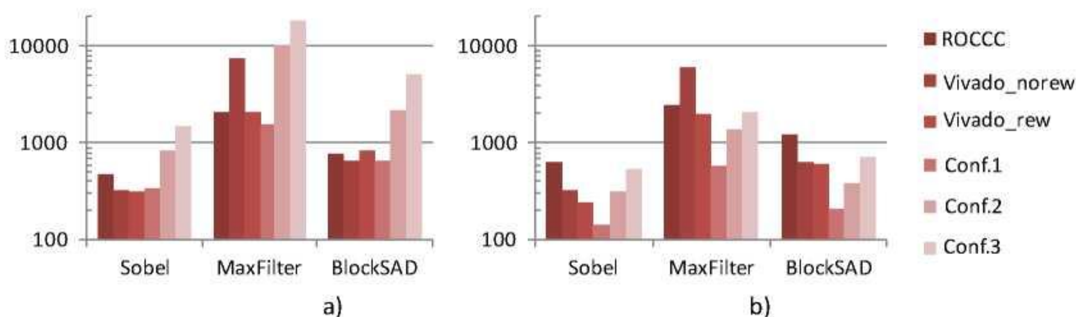


Рисунок 2.5 - Порівняння необхідних ресурсів для створених нами систем і базових підходів: LUT (a) і тригери (b).

Необхідні ресурси . На рисунку 2.5 показано кількість ресурсів площі, - необхідних для RO CCC, Vivado HLS і наших власних прискорювачів. Не -дивно, що прискорювачі з великою кількістю шляхів даних (Conf.3) потребують більше ресурсів, ніж підходи з одним шляхом (Conf.1, Vivado). Збільшення площі з точки зору тригерів можна порівняти з іншими найсучаснішими інструментами, оскільки розмір буфера лише трохи збільшено для підтримки високого ступеня

паралелізму. З іншого боку, результати підкреслюють, що складні прискорювачі вимагають більшої кількості комбінаторної логіки (LUT) відносно ROCCC і Vivado HLS.

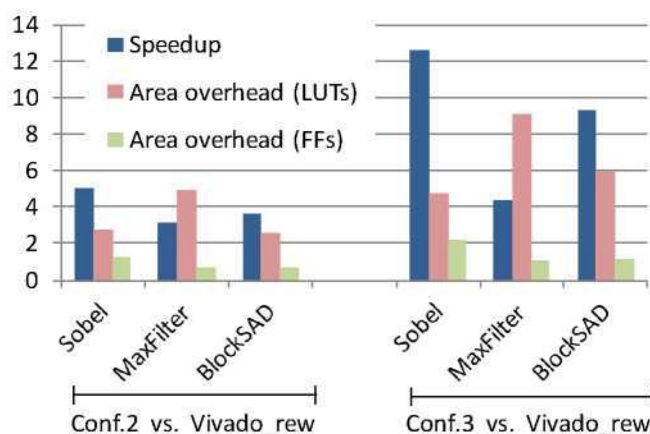


Рисунок 2.6 - Порівняння реалізацій Multi-Datapath і оптимізованої Vivado.

Як показано на рисунку 2.6, покращення прискорення за рахунок паралельних шляхів даних порівняно з накладними витратами на область: у випадку тесту BlockSAD, наприклад, Conf.3, який вбудовує 13 паралельних шляхів даних, вимагає в 6 разів більше LUT порівняно з реалізацією Vivado_rew, але в той же час призводить до 9,2-кратного прискорення. Знову ж таки, важливо зазначити, що найсучасніші інструменти не підтримують розгортання з повторним використанням даних, але зупиняються на рівні прискорення, якого можна досягти за допомогою рішень з єдиним шляхом до даних.

Структура, представлена у 2.1.3, на крок наближає синтез високого рівня до імітації рішень дизайнера обладнання, прийнятих вручну. Прискорювач Conf.3 для BlockSAD, представлений вище, по суті не відрізняється від розробленого вручну від Hameed et. al [25], у статті, спрямованій на ручне проектування прискорювачів для програми H.264. Автори справді вирішили інвестувати площу для 16 шляхів даних BlockSAD паралельно, щоб 1) максимізувати прискорення та повторне використання та 2) використовувати високу пропускну здатність між процесором і прискорювачем у своєму процесорі Cadence Tensilica Xtensa [28]. Ця робота імітує обґрунтування, яке

дотримується там, але може робити це автоматично. Сучасні інструменти HLS можуть автоматизувати деякі рішення, прийняті фреймворком, представленим у цьому розділі, але не всі - зокрема, вони не можуть автоматично та спільно використовувати розгортання та конвеєрне під час розгляду повторного використання, а отже, забезпечити рівні прискорення, надані тут. Статичний аналіз вихідного коду може мати вирішальне значення під час автоматичної оптимізації синтезу прискорювачів, призначених для додатків із ковзним вікном. Мій аналіз програмного забезпечення визначає повторне використання даних, а також локальність даних, а згодом дозволяє використовувати ці характеристики, використовуючи відповідні буфери пам'яті. Експериментальні результати демонструють підвищення продуктивності на порядок у порівнянні з найсучаснішими методологіями. Ця робота була опублікована на сьомому міжнародному семінарі HiPEAC IMPACT 2017 з техніки багатогранної компіляції [93].

2.2 Підхід машинного навчання для прогнозування фактора розгортання циклу

Інструменти високорівневого синтезу, як згадувалося на початку розділу, вимагають ручного прийняття рішень для створення ефективних прискорювачів. Ці рішення включають вибір високорівневих оптимізацій і трансформацій, які будуть застосовані, тому добре розуміння частин ПЗ, які необхідно прискорити, є важливим. Крім того, вибір оптимізацій є складним завданням через дві основні причини. По-перше, апаратний синтез є трудомістким процесом, який на практиці обмежує кількість можливих реалізацій, які можна оцінити. По-друге, ефект призначення різних значень директивам важко передбачити через низькорівневі характеристики програми.

Інструменти моделювання, такі як Aladdin [21], згаданий у Розділі 1, були розроблені для швидкої оцінки продуктивності та вартості (області) проектів, визначених HLS. Тим не менш, навіть при використанні інструментів оцінки,

вичерпна оцінка всіх налаштувань директив для кожного потенційного прискорювача в гетерогенній системі все ще неможлива, окрім простих випадків . Для вирішення цієї проблеми запропоновано структуру машинного навчання, яка здатна зробити висновок про належну реалізацію дизайну HLS на основі його характеристик, автоматично отриманих із проходу аналізу вихідного коду на основі структури компілятора LLVM [42].

У цьому розділі основна увага приділяється *розгортанню циклу,* вже добре відомій оптимізації з домену компілятора, а також домену HW. Оптимізація розгортання циклу повторює тіло циклу задану кількість разів, щоб виявити паралелізм, який, особливо в апаратній реалізації, може призвести до значного прискорення [41]. Тим не менш, цю директиву слід застосовувати розумно, оскільки вона тягне за собою високу вартість площі для дубльованої логіки; крім того, його наступні переваги можуть бути перешкоджені залежностями, що передаються циклом, і частими зверненнями до пам'яті.

Таким чином, зрозуміло, що існує компроміс між часом виконання та бюджетом області, який має під рукою комп’ютерний архітектор, а також рівнем складності та більшою кількістю потенційних побічних ефектів, які потрібно брати до уваги. Оскільки реалізації HW цільові, мета цієї роботи полягає в тому, щоб досягти *найкращої точки* між продуктивністю та ресурсами HW.

У рамках цієї дослідницької роботи зроблено наступні внески. По-перше, est класифікація, замість моделей на основі оцінки, щоб точно передбачити оптимальний коефіцієнт розгортання циклів у програмах, які будуть синтезовані в HW. Використання цієї методології може забезпечити результати з кращою оцінкою прогнозу та за набагато менший час порівняно з сучасними. По-друге, весь процес повністю автоматизований, від аналізу вхідних додатків, використовуючи інфраструктуру компілятора LLVM [22], до навчання Класифікатора випадкового лісу. Нарешті, навчений класифікатор Random Forest може бути використаний для генерації точних прогнозів розгортання циклу для будь-якої конкретної програми – фрагмент інформації, який може безпосередньо використовуватися інструментом HLS, таким як Vivado HLS, для синтезу частин

цих програм у HW.

Дослідницькі статті досліджували застосовність машинного навчання для застосування оптимізації компілятора. У компіляторах програмного забезпечення його використовували Агаков та ін. [1] для прискорення ітераційної компіляції, Monsifrot et al. [21] для створення евристики компілятора та Кулкарні та ін. [20], щоб вибрати порядок, у якому повинні виконуватися проходи оптимізації. Стефенсон та ін. [24] використовували контрольовану класифікацію, таку як класифікація найближчих сусідів (NN) і методи опорних векторних машин (SVM), щоб отримати точні прогнози в оптимальних факторах розгортання.

Лю та ін. [24] використовував класифікаційну модель випадкового лісу в контексті HLS, розширюючи структуру ітеративного уточнення, запропоновану в [26] [25] [29]. Вони вирішують проблему, відмінну від тієї, що розглядається в цьому розділі: проблему отримання набору оптимальних за Парето реалізацій даного проекту шляхом навігації в просторі його конфігурації. Подібну позицію щодо проектування системного рівня проілюстровано Ozisikyilmaz et al. [24]. На відміну від цих робіт, моєю метою є виконання прогнозного призначення директив синтезу на основі навчання, виконаного на незалежному вхідному наборі. Цю проблему також досліджували Kurra et al. [21]. Всупереч їхній стратегії, моя методологія не залежить від детальної моделі затримки оцінки тіла циклу, щоб передбачити фактори розгортання циклу в екземплярах HLS.

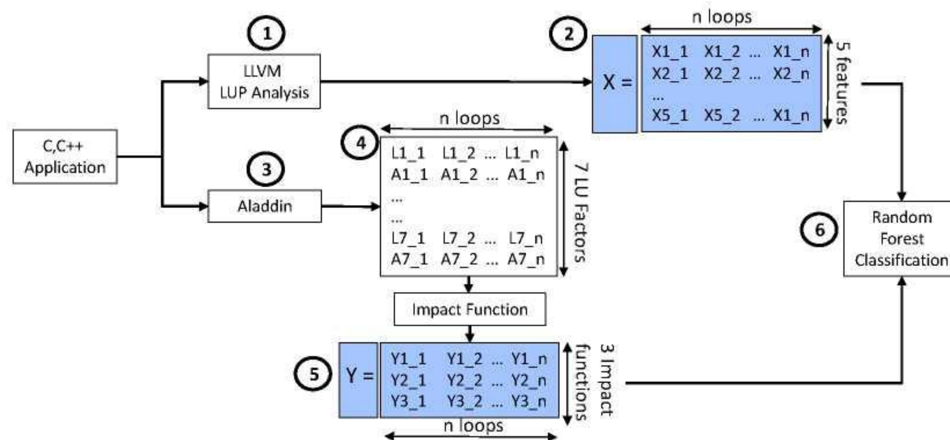


Рисунок 2.7 - Огляд методології прогнозування розгортання циклу

У цьому розділі спочатку представлена використовувана цільова функція, яка визначає оптимальний коефіцієнт розгортання циклу. Потім детально описано проходження аналізу LLVM, який було розроблено для автоматичного вилучення релевантних функцій циклу, і підхід, використаний для отримання області та продуктивності під час виконання проектів HLS. Нарешті, демонструється метод класифікатора навчання під наглядом, який під час фази навчання збирає дані з попередніх кроків для створення предиктора розгортання циклу, а під час фаз тестування призначає фактори розгортання циклу на основі характеристик циклу.

Фактор оптимального розгортання циклу - цільова функція

Оптимальний *коефіцієнт розгортання петлі* визначається наступним чином. Враховуючи визначений набір S факторів розгортання, наприклад S: <1, 2, 4, 8, 16, 32, 64>, існує один для кожного циклу, який максимізує **Вплив (I),** визначений такою формулою:

$$I(L,A) = \alpha \cdot \frac{(L_1 - L)}{L_1} + \beta \cdot \frac{(A_1 - A)}{A_1},\ \alpha + \beta = 1 \qquad (2.1)$$

де – затримка функції, що містить цикл і синтезується як апаратний прискорювач, для коефіцієнта розгортання циклу (LUF), який дорівнює одиниці, тобто повністю згорнутий цикл. L — затримка апаратного прискорювача для будь-якого можливого LUF із визначеного набору. Відповідно, A г є вимогою до площі HW прискорювача з LUF, що дорівнює одиниці, і A – площа для будь-якого можливого LUF від визначений набір. Згодом оптимальна LUF визначається як така, яка максимізує ¬вищезазначену функцію Impact. Зауважте, що коли LUF = 1, тоді I (L,A) = 0, що ¬відповідає базовій реалізації . I (L , A) також може бути від’ємним для неоптимального вибору LUF (де розгортання може збільшити площу без зменшення затримки), але завжди буде > 0 для оптимальних коефіцієнтів розгортання. Для оцінки розглядалися три різні стратегії: а) Оптимізація як для затримки, так і для площі (а = в = 0,5). У цій конфігурації зберігається баланс між зменшенням кількості ¬циклів виконання

та низьким використанням ресурсів HW області в даній реалізації. б) Оптимізація для затримки (а = 0,7, в = 0,3). Цей підхід сприяє мінімізації затримки, таким чином зосереджуючись на збільшенні швидкості програми, і, нарешті, с) Оптимізація для області (а = 0,3, в = 0,7). Цей підхід спрямований на зменшення територіального бюджету впровадження, отже, досягнення середнього прискорення, але збереження низького рівня використання ресурсів HW області. Усі три конфігурації задовольняють різні архітектурні потреби та досліджують реалістичні альтернативні сценарії.

LLVM Analysis Pass - Loop Features Extraction

Функції циклу автоматично ідентифікуються проходом аналізу (зображеним як точка 1 на Рисунку 2.7), який був розроблений в інфраструктурі компілятора LLVM [22]. Функції витягуються, починаючи з додатків, написаних на C або C++, що працюють на їхньому проміжному представленні, наданому передовими проходами LLVM.

Мій *LLVM Loop Unrolling Prediction Analysis Pass* повторює функції додатків і визначає петлі. Для кожного з них він виконує цикл, скалярну еволюцію та аналіз залежностей, щоб виділити їх характеристики, зведені в таблиці 2.3 критичний шлях, кількість спрацьовувань, наявність залежностей, що передаються циклом, і необхідні звернення до пам’яті (завантаження та збереження).

Вибір функцій базується на факторах, які впливають на вартість і досяжне прискорення апаратних розгорнутих циклів: цикл із довгим критичним шляхом може бути дорогим для дублювання, тоді як перенесені циклом залежності та доступи до пам’яті можуть призвести до серіалізації виконання незалежно від ступеня розгортання. Ці спостереження спонукають нас розглянути суттєво відмінний список функцій щодо робіт, що зосереджуються на цілях програмного забезпечення, наприклад, у Stephenson et al. (Таблиця 2.4).

Щоб зібрати необхідні значення функцій, ми створили існуючі методи (наприклад, метод getTripCount, що належить до посилання на клас ScalarEvolution), і реалізували аналіз LLVM, псевдокод якого представлено в

алгоритмі 3. Результатом алгоритму є, для кожного циклу, вектор ознак X із зазначенням.

Таблиця 2.3.

Функції, отримані за допомогою LLVM LU Analysis Pass.

Особливості - X Vector
Критичний шлях
Підрахунок відключення циклу
Має залежності, що передаються циклом
 # Завантажити інструкції
 # Інструкції магазину

його характеристики, представлені як точка 2 на рисунку 2.7. Вектори ознак використовуються під час навчання для налаштування класифікатора, а також під час фази тестування для прогнозування факторів розгортання циклу з великим впливом.

Таблиця 2.4.

Вектори ознак, вибрані Stephenson et al. [24].

Особливості - X Vector 1	**Особливості - X Vector 2**
# Операнди	*# Операції з плаваючою комою*
Розмір діапазону	*Рівень петлі гнізда*
Критичний шлях	*# Операнди*
# Операції	*# Відділення*
Підрахунок відключення циклу	*# Операції з пам'яттю*

Затримка та оцінка площі

Щоб встановити зв’язок між LUF і продуктивністю/вартістю реалізацій, значення затримки та площі повинні бути витягнуті як для циклів у навчальному наборі (для оптимізації класифікатора), так і для циклів у тестовому наборі (для вимірювання його точності). Aladdin [21] (точка 3 на Рисунку 2.7), симулятор потужності перед RTL для апаратних прискорювачів був використаний для отримання затримки. та інформацію про місцевість. Усі функції в розглянутих контрольних тестах були змодельовані шляхом використання кожного можливого фактора розгортання в наборі S , визначеному вище, для кожного з

їхніх циклів. Aladdin повідомляє про затримку в тактах, тоді як площа виражається в *і ат* ^{2} за 45-нм технологією. Результат показано як точка 4 на Рисунку 2.7.

Вплив (I) було обчислено пізніше для різних a і $в$ значення, щоб отримати оптимальний коефіцієнт розгортання циклу для кожного циклу функції, який є індексом LUF, який максимізує I . Результатом є три вектори $\{Y1, Y2, Y3\}$ (точка 5 на рисунку 2.7), які містять цільові значення для алгоритму класифікації.

Випадкова класифікація лісу

Ця інформація (вектори X і Y), отримана, як описано вище, використовується як вхідна інформація для класифікатора випадкового лісу (RF) (точка 6 на рисунку 2.7). Контрольоване навчання виконується шляхом виявлення кореляції між вхідними даними — отриманою компілятором - інформацією, яка використовується як вектор ознак X — і виходом, який є оптимальним фактором розгортання циклу для кожного циклу.

Випадковий ліс використовувався як контрольована модель навчання, що було продемонстровано Лю та ін. [24], щоб перевершити такі альтернативи, як багатошарові нейронні мережі та класифікація опорних векторних машин у контексті дослідження простору дизайну HLS. Алгоритми випадкового лісу дотримуються методології дерева рішень, поєднуючи багато слабких класифікаторів для отримання сильного, що дозволяє генерувати надійні класифікатори низької складності. Застосований алгоритм, представлений в Алгоритмі 4, дотримується підходу, подібного до стратегії k-кратної перехресної перевірки. Весь набір даних розподіляється випадковим чином між навчальним набором і тестовим набором, де навчальний набір дорівнює 80% усього набору даних, а решта 20% є тестовим набором. Потім для процесу навчання на навчальному наборі використовується модель Random Forest, а для кожного елемента тестового набору виконуються прогнози поза вибіркою. Після того, як усі прогнози для тестового набору були обчислені, прогнозний бал і середня помилка обчислюються для поточного навчального сеансу.

Для оцінки ефективності класифікації навченого класифікатора було

прийнято дві різні метрики. Прогнозна *оцінка* вказує на відсоток оптимальних (відповідно до $I(L, A)$) LUF, які були правильно ідентифіковані на тестовому наборі поза вибіркою. Натомість середня *помилка* вимірює середню відстань між індексами в S правильного та прогнозованого LUF.

Для порівняльної оцінки запропонованої методології, яка поєднує класифікацію Random Forest і виділення ознак циклу на основі LLVM, були розглянуті тести різної складності. До малих і середніх належать adpcm, ядро кодування аудіо, трафарет, реалізація ітераційного алгоритму, який оновлює елементи масиву відповідно до заданого шаблону, і sha, безпечний метод хеш-шифрування, який використовується в області інформаційної безпеки. Натомість jpeg і mpeg2 є більшими тестами, які стискають зображення та відео відповідно. Додатки були взяті з наборів тестів CHStone [27] і Scalable Heterogeneous Computing (SHOC) [21]. Загалом вони складаються з 87 різних петель. Класифікація Random Forest була реалізована за допомогою набору Scikit-learn [56], який включає найсучасніші реалізації моделей машинного навчання на Python. Scikit-learn також використовувався для повторної реалізації двох методів, запропонованих Stephenson et al. [24], які вважаються вихідними. повторно портує різницю між індексами прогнозованих оптимальних факторів розгортання циклу та тих, що були отримані за допомогою вичерпного дослідження, враховуючи 18 000 прогнозів поза вибіркою на всіх контрольних циклах. Результати сильно зосереджені на нулі, що вказує на високий рівень правильних прогнозів. Класифікаційні моделі та особливості

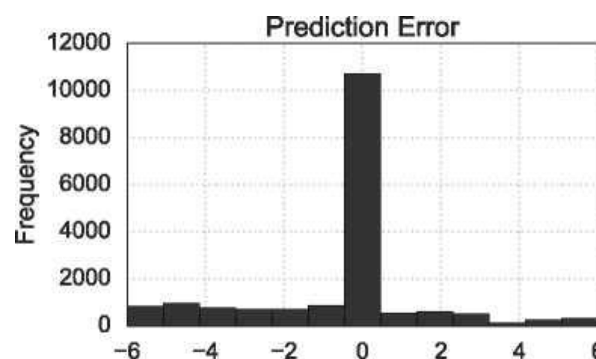


Рисунок 2.8 - Розподіл помилок прогнозування фактору розгортання циклу для 18 000 прогнозів поза вибіркою.

Оцінка вибору ознак (вектор X у таблиці 2.3) і моделі навчання Випадковий ліс (RF) відбувається за двома найсучаснішими методологіями, - запропонованими Stephenson et al. [24]. Останні представляють різні стратегії класифікації: опорні векторні машини (SVM) і найближчі сусіди (NN) і інший вибір досліджуваних характеристик циклу, наведених у таблиці 2.4. На Рисунку 2.9 показано, для вибору a = 0,5, оцінка прогнозу та середня помилка дев'яти стратегій, що є результатом різних векторів ознак і стратегій класифікації. Експериментальні результати підкреслюють, що представлена структура (вектор ознак X і радіочастотна класифікація) перевершує інші варіанти, досягаючи показника прогнозу вище 60% і середньої помилки менше 1,4. Подібні результати були отримані для функцій впливу, що сприяють площі або затримці (вектори Y2 і *Y 3).*

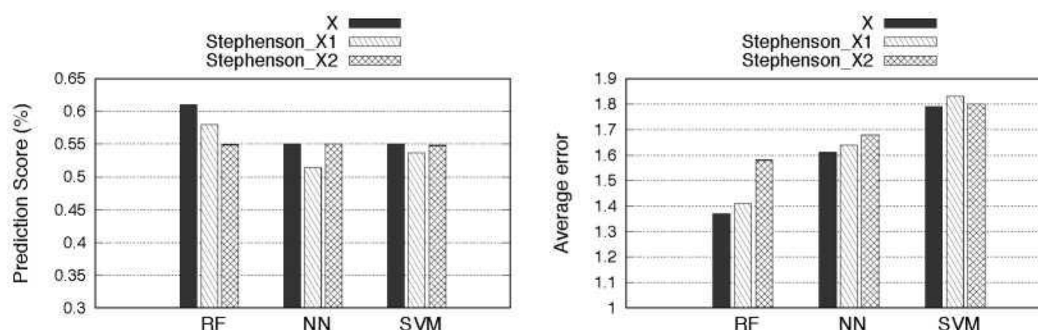


Рисунок 2.9 - Ліворуч: Порівняння оцінки прогнозу в моделях Random Forest, Nearest Neighbor, Support Vector Machines і відповідного вибору функції: вектор X, Stephenson et al. вектори X1 і X2 [24]. Праворуч: функції, отримані моїм пропуском аналізу розгортання циклу LLVM

Ітеративне уточнення

У другому раунді експериментів було проведено порівняння запропонованого методу з підходом ітеративного уточнення, використаним у [26] [25] [29]. Ітеративне уточнення використовує частину навчального набору даних для отримання першої версії класифікатора, продуктивність якого потім покращується за допомогою другого, неперетинного набору вхідних і вихідних даних. Було розглянуто три різні параметри цільових векторів Y *{Y* 1, *Y* 2, *Y 3}*,.

Використані дані, характеристики (вектор X) і модель навчання (випадковий ліс) були однаковими як для алгоритму 4, так і для алгоритму, що використовує ітеративне уточнення. Для ітеративного уточнення 75% навчального набору виділяється для початкової фази навчання, а решта 25% – для фази уточнення.

Оцінка прогнозу, як показано на Рисунку 2.10, коливається від 53% до 63% для трьох векторів *Y*. Незважаючи на це, наша методологія незмінно перевершує підхід ітеративного уточнення, досягаючи при цьому найвищої оцінки прогнозу (63%) для параметрів, які віддають перевагу ресурсам з низькими площами (Y3). Подібне спостереження можна зробити для значень середньої помилки, де запропонований підхід зберігає нижчу середню помилку для всіх прогнозів, по всіх векторах, причому той, що стосується вектора *Y* 3, є найнижчим (1,32).

На Рисунку 2.11 представлені показники прискорення, площі та впливу конструкцій HLS, оптимізованих за допомогою нашого прогнозного методу, порівнюючи їх із підходом ітеративного вдосконалення та результатами, отриманими в результаті вичерпного дослідження. Графіки відповідають функції впливу з a = 0,9 (подібні результати були отримані для інших значень a). З представлених даних можна зробити два висновки:

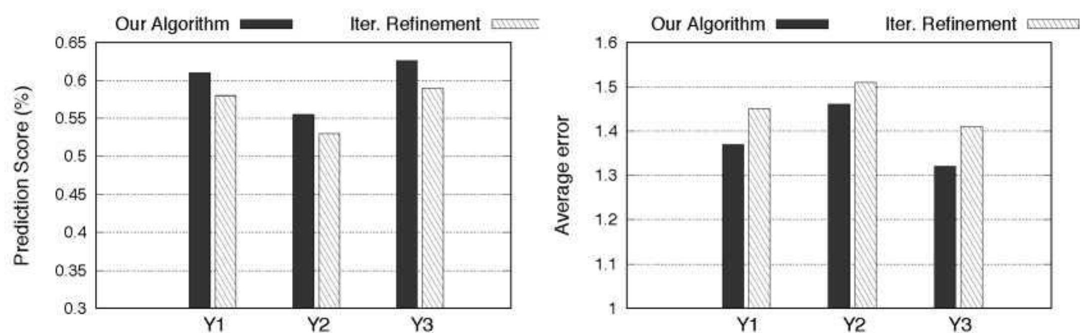


Рисунок 2.10 - Ліворуч: Порівняння оцінки прогнозу в моделях Random Forest, Nearest Neighbor, Support Vector Machines і відповідного вибору функції: вектор X, Stephenson et al. вектори X1 і X2 [74]. Праворуч: функції, отримані моїм пропуском аналізу розгортання циклу LLVM

По-перше, у більшості випадків методологія, представлена в розділі, точно відстежує визначений користувачем компроміс між площею та затримкою. У цьому відношенні jpeg-huff і stencil-stencil є винятками, оскільки їх структура циклу досить складна, що ускладнює автоматизацію їх оптимізації. По-друге, вплив, досягнутий нашим підходом, дорівнює або вищий (на 66% у випадку mpeg2), ніж вплив, досягнутий ітеративним уточненням. У середньому наша методологія забезпечує 86% прискорення, досягнутого завдяки оптимальному фактору, отриманому за допомогою дорогого вичерпного дослідження (90% для $a = 0{,}5$ і 92% для $a = 0{,}1$).

Порівняння часу конвергенції

Окрім отримання високоякісних прогнозів LUF, представлена тут структура також вимагає менших обчислювальних зусиль порівняно з іншими методами. У цьому відношенні експериментальні докази показані на рисунку 2.12, де показано час, необхідний для навчання та тестування, порівнюючи різні стратегії класифікації, вектори ознак і функції впливу.

Як і очікувалося, вибір використовуваних функцій, а також відносна релевантність площі та затримки істотно не впливають на час обчислення. З іншого боку, тип використовуваного класифікатора має помітний вплив, причому Random Forest повільніше сходиться, ніж NN і SVM. Тим не менш, наш підхід вимагав лише 14 секунд. Однак різниця між підходом ітеративного уточнення та нашою методологією навіть більш суттєва, оскільки перший вимагає майже в чотири рази більше, ніж наша методологія, щоб зблизитися.

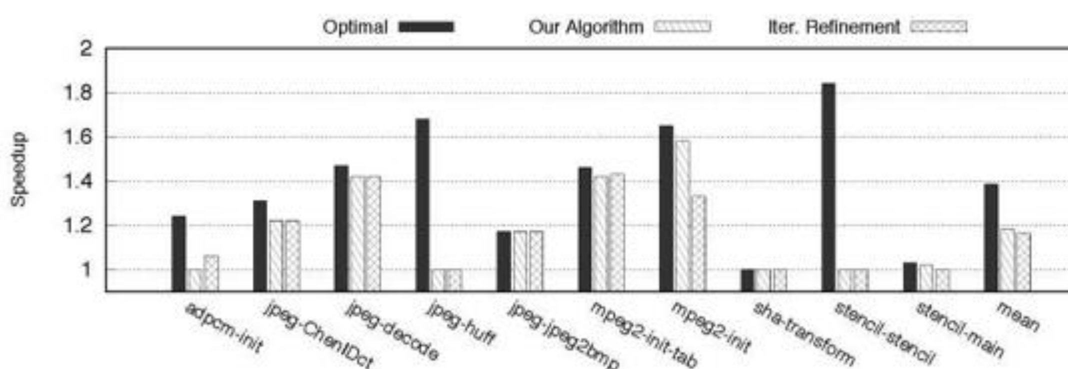


Рисунок 2.11. Порівняння прискорення, площі та впливу, досягнутих для кожної функції за допомогою нашого алгоритму та підходу ітеративного вдосконалення [26] [25] [29],

порівняно з оптимальними рішеннями, отриманими в результаті вичерпних досліджень. Номери прискорення та зон нормалізуються щодо повністю згорнутих конфігурацій

Рисунок 2.12. Ліворуч: час, необхідний для конвергенції для найсучасніших моделей машинного навчання з нашим вектором ознак r X і Stephenson et al. вектори X1 і X2 [24].

Справа: час збіжності нашого алгоритму та ітераційного уточнення по всіх трьох векторах Y.

2.3 Висновки по розділу

Вданому розділі описується автоматизована методологію, яка визначає потенціал повторного використання даних певного типу програм (програм із ковзним вікном) за допомогою поліедрального аналізу. Ця інформація використовується для розробки буферів пам’яті відповідно до вимог кожної програми. Експериментальні результати показали невелике або незначне підвищення продуктивності порівняно з найсучаснішими методологіями. Було представлено новий метод прогнозування фактора розгортання циклу за допомогою класифікації машинного навчання. Автоматичний аналіз компілятора використовувався для вилучення характеристик циклу, які використовуються для навчання алгоритму класифікації випадкового лісу. Цей підхід досяг кращої оцінки прогнозування порівняно з найсучаснішими методами машинного навчання. Експериментальні дані показали, що точні прогнози факторів розгортання циклу призвели до реалізації високопродуктивного прискорювача , водночас уникаючи трудомістких вичерпних досліджень.

РОЗДІЛ 3. ІДЕНТИФІКАЦІЯ ТА ВИБІР СИСТЕМНО-ЗАЛЕЖНИХ ПРИСКОРЮВАЧІВ

У цьому розділі поняття автоматичного вибору HW прискорювачів - розширено за рахунок збільшення обсягу кандидатів на прискорення, щоб націлити на платформи зі складною ієрархією пам'яті, такі як плата Xilinx Zynq Ultrascale+ PSoC. Знання характеристик цільової системи може мати вирішальне значення для вибору апаратних прискорювачів, які будуть реалізовані. Наприклад, система пам'яті платформи може значно впливати на затримку через обмін даними між основною пам'яттю системи та апаратними прискорювачами.

Ідентифікація та вибір прискорювачів для великих і складних програм, таких як декодер H.264 [25], на платформі, такій як згадана вище, отже, потребуватиме врахування накладних витрат на зв'язок, які в деяких випадках перевищують час обчислення. Щоб вирішити цю проблему, деталізація - кандидатів на прискорення розширюється до підграфа всього графа програми, щоб розширити сферу прискорення та, отже, визначити кандидатів із вищим співвідношенням обчислення до зв'язку.

AccelSeeker, ланцюжок інструментів на основі LLVM, представлений як структура, яка виконує автоматичну ідентифікацію та вибір апаратних прискорювачів, націлених на конкретну систему на кристалі (SoC). Він виконує ретельний аналіз вихідного коду програм і оцінює затримку пам'яті разом із затримкою обчислення кандидатів на прискорення. Потім AccelSeeker вибирає прискорювачі, які можуть мінімізувати загальну затримку програми та можуть запропонувати підвищене прискорення порівняно з підходами, заснованими лише на інформації профілювання, які імітують ручні рішення дизайнера.

На додаток до прискорювачів, спрямованих на прискорення, розширення цієї методології зосереджено на енергозберігаючих гетерогенних конструкціях. EnergySeeker розширює оцінку затримки попередньої структури та враховує оцінку потужності апаратного та програмного забезпечення для націлювання на енергоефективні апаратні прискорювачі; отже, EnergySeeker виконує ідентифікацію та вибір прискорювачів, які можуть запропонувати значні

енергетичні переваги гетерогенній обчислювальній системі порівняно з потужним, але енергоємним програмним процесором і проти методологій, заснованих виключно на інформації профілювання.

3.1 AccelSeeker: прискорювачі

Проектування системного рівня та гетерогенні обчислення в цілому стають проривом. Нові найкращі практики, засновані на високорівневому синтезі, забезпечують безпрецедентний рівень продуктивності. Інструменти HLS, як згадувалося в попередніх розділах, скорочують час, необхідний для синтезу HW прискорювачів, приймаючи вихідний код програми як вхідні дані. Проте розробка гетерогенних систем, що включають програмні процесори та апаратні прискорювачі, все ще є вимогливою роботою, під час якої ключові рішення залишаються виключно ручними зусиллями та експертним досвідом дизайнера [12]. Крім того, як уже обговорювалося, синтез HW потребує значної кількості часу, і в поєднанні зі значно великим простором альтернативних реалізацій, відкритих реальними додатками, кількість варіантів прискорювача, які розробник може розглянути вручну, перш ніж буде вирішено розділення апаратного та програмного забезпечення, на практиці обмежена.

Для вирішення цих проблем були запропоновані оцінювачі продуктивності, які, хоча і не забезпечують робочі апаратні реалізації, можуть вимірювати характеристики різних альтернатив реалізації прискорювача [22] [29]. Тим не менш, ці інструменти можуть оцінити лише один вибір прискореної функції одночасно. Отже, під час їх використання оцінка кожної потенційно життєздатної альтернативи розподілу апаратного/програмного забезпечення вимагає окремих експериментальних прогонів, що займає багато часу для цільових програм великого розміру.

Тому для того, щоб обмежити зусилля дизайнера, дуже важливо визначити набір життєздатних варіантів прискорення швидко, а також на ранніх стадіях процесу проектування, перш ніж виконувати більш пізні та більш детальні оцінки. Цей ключовий крок зараз погано підтримується інструментами

автоматизації проектування. Дійсно, найсучасніші стратегії раннього розподілу базуються виключно на інформації профілювання [22], яка, як також було показано авторами [27], часто може вводити в оману. Щоб заповнити цю прогалину та запропонувати ефективне вирішення проблеми, зазначеної вище, представлено AccelSeeker. AccelSeeker пропонує методологію для ідентифікації та вибору відповідних системних кандидатів прискорення в програмі з вихідного коду програмного забезпечення. Будучи реалізованим у компіляторі LLVM [42] у структурі, він спочатку виконує оцінку вартості (необхідних ресурсів) і переваг (потенційного прискорення) усіх потенційних прискорювачів за один швидкий прохід, а потім вибирає набір, який максимізує оцінене прискорення, не перевищуючи при цьому цільовий ресурс. Таким чином, використання AccelSeeker може скеровувати архітекторів інтегральних схем на ранніх етапах проектування, підкреслюючи, на які сегменти обчислювального потоку слід націлити HLS, а отже, на чому зосередити процес застосування оптимізації. Крім того, це вказує на те, які частини, ймовірно, не принесуть відчутних переваг, якщо їх реалізувати в HW - або через те, що вони мають низький обчислювальний слід, або через те, що їхні характеристики перешкоджають їх потенціалу для прискорення HW, наприклад, вони вимагають надмірної кількості передачі даних під час виконання обмежених обчислень. Підхід AccelSeeker помітно відрізняється від підходу оцінювачів продуктивності, оскільки найбільш перспективні кандидати визначаються в одному високорівневому дослідженні, що зменшує обсяг подальших і більш детальних оцінок. Він також відрізняється від підходів, заснованих виключно на інформації про профілювання, оскільки профайлери не пропонують вимірювання вартості та часу роботи реалізацій HW. Крім того, вони не враховують накладні витрати на виклик, що потенційно може призвести до вибору часто викликаних, але невеликих кандидатів. Нарешті, передача даних є важливим фактором, який не враховують найсучасніші профайлери, отже потенційно пропонуючи кандидатів, які потребують надмірної кількості зв’язку, що, у свою чергу, може значно послабити будь-яку потенційну отриману продуктивність. Натомість

AccelSeeker враховує витрати на зв'язок і характеристики основної платформи, щоб створити модель оцінки, яка веде до вибору високопродуктивних апаратних прискорювачів.

За останні роки інструменти високорівневого синтезу значно вдосконалилися [19]. Зараз ¬доступні комерційні інструменти (наприклад: XilinxVivado HLS [15], Cadence Stratus HLS [13]), а також академічні альтернативи (наприклад: Legup [15], Bambu [28]) підтримують проектування дуже великих прискорювачів з коду C/C++ . Вони досягають рівня продуктивності, порівнянного з ручними реалізаціями, визначеними мовами опису обладнання (HDL) низького рівня, такими як VHDL або Verilog [25]. Тим не менш, автоматизований вибір частин програми найбільш піддатливий до апаратного прискорення все ще є відкритою темою для досліджень. Підходи вибору, засновані на результатах синтезу [16], погано масштабуються для складних програм, оскільки вони доступні лише на пізній стадії процесу проектування. Платформи оцінювання пропонують детальний аналіз - продуктивності та вимог до ресурсів апаратно-прискореної системи, уникаючи повного синтезу, або шляхом поєднання програмного забезпечення та симуляторів апаратного забезпечення (наприклад, gem 5 [8] і Aladdin [21] у [22]), або шляхом прийняття гібридної позиції, у якій оцінюється час виконання апаратного забезпечення, тоді як час програмного забезпечення вимірюється на цільовій платформі (як у Xilinx SdSoC [29]). Однак в обох випадках оцінки - виконуються після розділення обладнання та програмного забезпечення, що залишається на основі проб і помилок. Натомість у цьому розділі пропонується методичне рішення для розділення. Зворотна сторона неправильного вибору розділів і, як наслідок, важливість автоматизованих інструментів, таких як AccelSeeker, які керують вибором високоякісних прискорювачів, стає ще помітнішою, якщо врахувати великі зусилля, необхідні для оптимізації реалізації HLS-defmed прискорювачів. Оптимізація дизайну передбачає специфікацію кількох директив для спрямування дизайну до бажаного компромісу щодо продуктивності. Зв'язок між директивами та продуктивністю реалізацій не є

простим, тому для досягнення запланованих результатів потрібна оцінка кількох альтернатив, як показано в [25] [26] [24]. Тому важливо заздалегідь зосередити ці зусилля з оптимізації лише на тих потенційних прискорювачах, які можуть призвести, з точки зору додатків, до відчутного прискорення. З цією метою представлений тут підхід ґрунтується на попередніх роботах з au- томатична ідентифікація розширень набору інструкцій. Більшість методів у цій галузі націлені на настроювані процесори, доповнені спеціальними функціональними блоками в межах конвеєрів процесорів. Отже, ці методи зазвичай обмежують свій пошук обсягом окремих базових блоків [22] [28], як показано на рисунку 3.1| а. У Розділі 1, у якому було представлено структуру RegionSeeker, і в [23] натомість було зосереджено увагу на ідентифікації більших сегментів коду, включаючи структури потоку керування, що належать до окремих функцій (зображено на Рисунку 3.1 b). Однак такий обсяг все ще недостатній, якщо орієнтуватися на платформи, де прискорювачі підключаються до системної шини [20]. У цьому налаштуванні вартість переміщення даних стає настільки значною, що навіть структури потоку керування всередині функцій не забезпечують продуктивність. Відповідні кандидати на прискорювачі повинні потім охоплювати цілі функції, включаючи, у свою чергу, усі функції, викликані в їхньому дереві викликів. AccelSeeker враховує таку саму деталізацію (рисунок 3.1c), удосконалюючи сучасний рівень автоматичної ідентифікації прискорювачів.

Методологія AccelSeeker, як показано на Рисунку 3.2. Спочатку ми визначаємо, що таке кандидат на прискорення, а потім докладно описуємо, як ми вибираємо серед пулу таких кандидатів, отриманих із вихідного коду програми, підмножину, яку буде реалізовано в апаратному забезпеченні (A і C на Рисунку). Нарешті, представлено підхід, використаний для оцінки ефективності кандидатів (Merit) і вимог до ресурсів (Cost) (B на Рисунку)

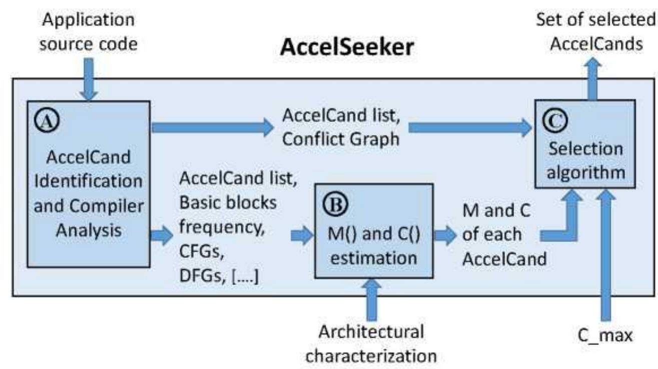


Рисунок 3.2. Фази підходу AccelSeeker.

А) Кандидати для прискорення ідентифікуються за допомогою аналізу вихідного коду. В) Враховуючи оцінку переваг і вартості для кожного кандидата, і С) враховуючи максимальну доступну вартість, AccelSeeker виконує вибір підмножини таких кандидатів

Щоб виявити, які частини програми можна найбільш вигідно прискорити за допомогою апаратного забезпечення, ми досліджуємо її граф викликів функцій, тобто спрямований ациклічний граф G(N,E), де кожен вузол n є N відповідає функції, а кожне ребро e = (u, v) є E відповідає виклику функції u до функції v. Корінь – це вузол, який досягає всіх інших вузлів графа, тобто для кожного іншого вузла n є N, від кореня до нього існує шлях. Граф викликів функцій G має корінь, який представляє функцію верхнього рівня програми. Рисунок 3.3 а показує приклад графа викликів (зауважте, що напрямки країв тут не показані для ясності зображення; однак вони спрямовані зверху вниз).

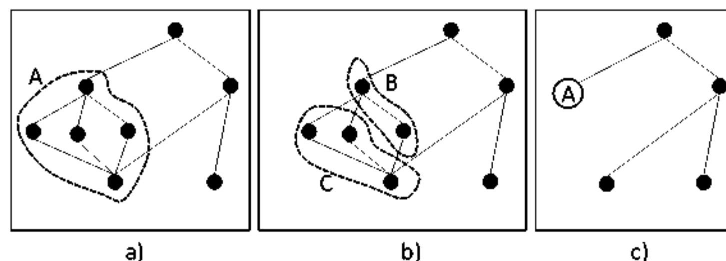


Рисунок 3.3. а) Приклад графа викликів. Чорні вузли — це функції, присутні в програмному забезпеченні, а краї — виклики функцій.

Підграф A є AccelCand. b) Підграфи B і C не є AccelCand (B має вихідні ребра, C не має кореня), c) Граф викликів, що є результатом вибору A як прискорювача

Прискорювач-кандидат визначається та називається **AccelCand** як підграф $S(N_s, E_s)$ графа G, що демонструє такі дві характеристики: підграф має корінь; підграф має нуль вихідних ребер. Перше означає, що підграф має вузол, який досягає всіх інших вузлів у підграфі; останнє означає, що для кожного вузла n_s є N_s, у G не існує ребра (n_s, m_s) такого, що $m_s \pounds N_s$.

Рисунок ЗТЗІа та 3.3 b показують три приклади підграфів, позначених A, B, C. Хоча підграф A є AccelCand, підграф B – ні, оскільки він не має нульових вихідних ребер, а підграф C також не є AccelCand, оскільки він не має кореня. Граф викликів, отриманий у результаті вибору AccelCand A як прискорювача, показано на Рисунку ЗТЗІс: увесь підграф підводиться до одного виклику (прискорювача).

Методологія обмежена розглядом графів викликів, які є ациклічними, і, отже, з такими конструкціями, як рекурсія, неможливо працювати. Це відповідає обмеженням інструментів HLS.

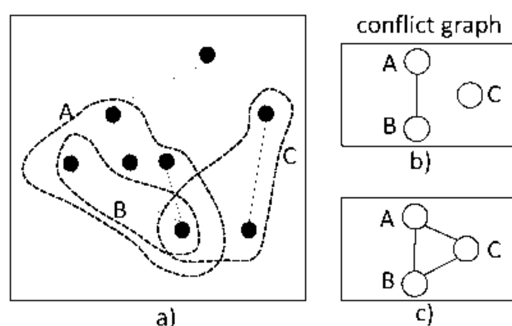


Рисунок 3.4. a) Три перекриваються AccelCand: A, B, C.

b) Конфліктний граф, який розглядається у формулюванні задачі: повне перекриття є конфліктом, тоді як часткове перекриття допускається с)

Конфліктний граф, прийнятий у [96] замість нього, де будь-який вид перекриття вважається конфліктом.

Дано граф викликів *G(N,E),* існує AT AccelCands у ньому; кожен вузол *G* є, по суті, коренем одного і єдиного AccelCand. Проблема *відбору* полягає в тому, щоб вибрати серед усіх |1V| AccelCands, підмножина, яка буде реалізована як прискорювачі.

Кожен AccelCand пов'язаний із заслугою M() – оцінкою кількості збережених циклів, коли він реалізований у HW на відміну від SW – і вартістю C() – оцінкою площі, необхідної йому, коли він реалізований у HW. Зауважте, що запропонована тут методологія не залежить від способу визначення вартості та переваг. Звичайно, їхнє визначення повинно правильно відображати архітектури програмного та апаратного забезпечення, на які орієнтована методологія.

Враховуючи набір AccelCands, визначений і ідентифікований, як описано в попередньому підрозділі, і враховуючи вартість і переваги, пов'язані з кожним із них, проблема, яка розглядається, полягає в наступному.

Проблема: вибір прискорення

Нехай A = $\{A_1, A_2, \ldots, A_n\}$ — набір AccelCand із пов'язаними функціями вартості та переваг C і M. Для будь-якої підмножини *S* с A вона позначається як M(S) = ^ $_{!<£S}$ M(A,) як сума переваг кандидата, а як C(S) = XLes C(A,-) як сума їхніх витрат.

Підмножина *S* AccelCands вибирається так, що:

1. Перевага M(S) максимізована
2. Вартість C(S) знаходиться в межах заданого користувачем бюджету витрат C_{ma}
3. Жодні два кандидати в наборі *S не* конфліктують

Концепція *конфлікту* визначається таким чином: два кандидати A_i та *Aj* у *S* конфліктують тоді і тільки тоді, коли A_t *с Aj_Aj с* A_t, тобто якщо один повністю включає іншого. Це показано на рисунку 3.4. Показано граф викликів, де виділено 3 AccelCands (з 8 можливих, оскільки в графі викликів 8 вузлів), і

вони позначені A, B і C. *Граф конфлікту* також представлено на Рисунку 3.4b, який є неорієнтованим графом, де вузли представляють AccelCands, а ребро додається між двома вузлами, якщо два кандидати конфліктують. A і B конфліктують, оскільки B повністю міститься в A.

Причина концепції конфлікту полягає в тому, що в формулюванні проблеми (і в реалізації для її вирішення) переваги набору кандидатів визначаються як *додаткові:* це сума індивідуальних переваг обраних кандидатів. Оскільки заслуги B вже повністю зараховуються до заслуг A, два кандидати не повинні бути обрані одночасно (і їхні заслуги не слід зараховувати двічі). Зауважте, що якщо натомість часткове збігання – як у випадку з кандидатами A та C, які в цьому прикладі спільним викликом однієї функції – дві переваги правильно моделюються окремо, оскільки наш інструмент може ідентифікувати «спільні» функції з кандидата A, а також кількість з кандидата B. Отже, часткове збіг не є конфліктом.

Формулювання проблеми *Accel Selection* запозичено з того, що знайдено в Розділі 1. Однак воно має важливу відмінність. У розділі 1.2 кандидати на прискорення знаходяться в межах графа потоку керування однієї функції, тому жодне збігання в межах одного вибору не допускається. І навпаки, у поточному розділі розглядаються підграфи графа викликів функцій, отже, дозволяючи рішення, включаючи частково перекриваючі AccelCands.

Розв'язування задачі Accel Selection на графі викликів функцій програми відповідає розв'язанню задачі незалежного набору на результуючому графі конфлікту. Граф конфліктів показує, які пари AccelCand знаходяться в конфлікті; таким чином, незалежний набір графа конфлікту задовольняє умову 3 проблеми вибору прискорення. Тому реалізовано алгоритм, який рекурсивно досліджує незалежні набори графа конфлікту, подібно до алгоритму Брона-Кербоша [11], Реалізований алгоритм повертає набір S з найвищою перевагою M (S) (отже задовольняє умову 1 постановки задачі) і вартість якого C(S) не перевищує а задана користувачем максимальна вартість C max (отже задовольняє умову 2 задачі для формулювання). Цей повернутий набір представляє оптимальне

рішення проблеми вибору прискорення .

Алгоритм, що розв'язує проблему незалежної множини, звичайно, має неполіноміальну складність. Точна реалізація все ще може знайти оптимальне рішення для експериментів у Підрозділі 3.1.9 за лічені мілісекунди, навіть для значного розміру графа викликів функцій розглянутої програми (граф з 63 вузлів, «Результати експерименту»). Приклад вибору можна побачити на Рисунку 3.5. Спочатку зображено приклад графа викликів, який має 8 вузлів і, отже, 8 AccelCand, кожен з яких вкорінений в одному з 8 вузлів. Вісім відповідних AccelCand не зображені на цьому Рисунку, але деякі з них можна побачити на Рисунку 3І4 а: наприклад, AccelCand, укорінений у вузлі 2, зображений там і позначений A, AccelCand, укорінений у вузлі 3, також зображений на тому ж Рисунку та позначений B тощо. На Рисунку 3.5f > зображено повний граф конфлікту, що відповідає цьому прикладу. Як видно, кандидат 1 (відповідає всьому графіку) конфліктує з усіма іншими кандидатами, кандидати 2 і 3 не конфліктують (вони лише збігаються у функції 7) і т. д. Тепер, враховуючи приклади значень вартості та заслуг для кожного кандидата (на Рисунку 3.4c), максимальний незалежний набір знайдено на графі конфлікту, який максимізує суму переваг обраних кандидатів, але не долає максимальну суму вартість і не включає конфлікти. Два приклади (для C max = 25 і для C max = 40) показані на рисунку 3.4d.

Тут докладно описано абстрактну вартість і переваги, які автоматично обчислюються з вихідного коду (рис. 3.2, B). Оскільки метою представленої структури є вибір найефективніших кандидатів перед їхньою оптимізацією, AccelSeeker розглядає їх реалізації за замовчуванням, наприклад, такі, де жодна функція не вбудована та жоден цикл не розгортається. Високопродуктивні реалізації, ймовірно, матимуть більші вимоги до ресурсів, у свою чергу потенційно вимагаючи відкинути деякі з вибраних AccelCand. Тим не менш , ці додаткові дизайнерські рішення виконуватимуться в обмеженому діапазоні набору кандидатів, отриманого Ac celSeeker (на відміну від усього дизайну), таким чином полегшуючи наступні зусилля. Архітектурна характеристика.

AccelSeeker базує свої оцінки на кількох параметрах, що характеризують цільову платформу. Оскільки вони пов’язані лише з змодельованою архітектурою, але не залежать від програми, характеристика являє собою одноразове зусилля для даної апаратної цілі. Для експериментів у розділі 3.1.9 це завдання було виконано за допомогою серії мікротестів, синтезованих на програмованій системі на кристалі (PSoC) Zynq. Однак методологія не обмежується цією метою. Навпаки, його можна адаптувати до різних обчислювальних архітектур (наприклад, реалізацій ASIC) шляхом вимірювання 1) області та критичного шляху окремих операторів (суматорів, помножувачів тощо), 2) накладних витрат, пов’язаних із ініціюванням виклику прискорення, 3) часу, необхідного для передачі входів і виходів, і 4) ресурсів, що використовуються для реалізації зв’язків прискорювач-пам’ять. (за замовчуванням реалізовано як головні порти осі в системах Zynq). Оцінка вартості. Вартість C () AccelCand обчислюється як сума його оціночної логіки та пам’яті. Щодо логіки, обчислюється сума необхідних ресурсів (незалежно ¬для таблиць пошуку та блоків DSP) арифметичних операцій, присутніх у його верхній функції. Якщо присутні виклики функцій, то рекурсивно область викликаних функцій також береться до уваги. Крім того, імітуючи реалізацію за замовчуванням прискорювачів XilinxPSoC, додавання вартості логіки, необхідної головному порту axi для кожного масиву, присутнього в AccelCand додається список параметрів. Потім область пам’яті виходить із розміру масивів, що зберігають вхідні/вихідні та проміжні значення, необхідні для прискорювача. Розмір введення/виведення визначається шляхом аналізу елементів у списку параметрів верхньої функції-кандидата, тоді як пам’ять, необхідна для проміжних значень, виводиться з оголошень змінних у кожному кандидаті, остаточно визначаючи кількість необхідних блоків BRAM. Відповідно до обмежень інструментів HLS, динамічний розподіл пам’яті не підтримується. Оцінка заслуг. Переваги M() AccelCand виражаються в термінах кількості тактових циклів, які зберігаються завдяки застосуванню його як апаратного прискорювача замість виконання в програмному забезпеченні. У свою чергу, оцінка часу роботи апаратного забезпечення повинна враховувати як

межі обчислення, так і накладні витрати на зв'язок хост-прискорювач. Останні витягуються шляхом врахування кількості необхідних звернень до пам'яті, масштабованих за фактором, що залежить від архітектури. було застосовано висхідний підхід, як також показано на рисунку 3.6. По-перше, максимум обчислюється затримка поширення кожного з основних блоків (BB), присутніх в AccelCand (як у верхній функції, так і в її викликах). Це досягається шляхом обходу їхніх DFG та обліку затримок операцій, отримуючи таким чином найдовший шлях вводу-виводу (рис. 3.6 a). Потім критичні шляхи BB виражаються в тактах, ділячи затримки поширення на період системного годинника. Помноживши критичні шляхи на кількість виконань кожного BB обчислюється відповідне робоче навантаження. Нарешті, оцінка часу обчислення AccelCand є сумою робочих навантажень його складових BB (рис. 3.6bc).

Час роботи програмного забезпечення оцінюється подібним чином, але замість обчислення критичних шляхів на рівні BB обчислюється сума затримки (у тактах) усіх його складових операцій, таким чином моделюючи, що вони послідовно обробляються програмним забезпеченням.

На основі зібраних даних перевага AccelCand *i* обчислюється таким чином:

$$M(i) = [T_{Sw}(i) - (T_{oh} + max(T_{Hw}^{comp}(i), T_{Hw}^{comm}(i)))] \times n_{exec}$$

де $T_{Sw}(i)$ — час виконання AccelCand у програмному забезпеченні, T_{oh} — фіксовані накладні витрати, необхідні для налаштування та запуску апаратного прискорення, *T^^ii)* і *TH ₁ < WJ !m (i)* — час виконання, коли *i* апаратно прискорюється, припускаючи, що його продуктивність залежить від обчислень або зв'язку. Нарешті, n_{exec} — це кількість разів, коли AccelCand виконується в програмі.

AccelSeeker реалізовано як перехід компілятора в інфрас-структурі LLVM 3.8 [22], як видно з алгоритму 5. Результуюча реалізація містить методи для

ідентифікації та аналізу AccelCand (рис. 3.2, A), для оцінки їх якості та вартості (рис. 3.2, B), а також для їх вибору (рис. 3.2, C). Додаткові відомості надано про те, як дані, необхідні на цих етапах, витягуються за допомогою аналізу ІЧ-рівня LLVM.

Ідентифікація та аналіз AccelCands. Для генерації графа викликів кожна функція анотована зв'язками абонент/виклик; граф викликів послідовно - обходиться для ідентифікації всіх дійсних AccelCand, як визначено в розділі 3.1.3. На цьому рівні виявляється інформація щодо перекриття таких AccelCand, необхідна для створення графа конфлікту та для подальшого вибору. Графік потоку керування кожного кандидата та графік потоку даних кожного базового блоку витягуються, щоб їх можна було використовувати як вхідні дані для розрахунку вартості та переваг відповідно до методу, який уже описано вище.

Періодичність виконання. Кількість викликів кожного кандидата, а також частота виконання кожного основного блоку в кожному кандидаті витягується через LLVM з динамічним профілюванням. Використовується процедура профілювання за допомогою приладів, яка вимагає генерації інструментальної версії коду, а потім дозволяє анотувати отримані частоти назад до рівня ІЧ.

Інструкції IR, включені в даний кандидат, а також кількість IR в інструкціях, включених у виклики функцій, якщо такі є. Кількість викликів кожного кандидата виходить із частоти виконання, отриманої за допомогою профілювання часу виконання. Обчислення HWLatency. І навпаки, ми оцінюємо затримку апаратного забезпечення, обчислюючи затримку інструкцій, відповідно охарактеризованих для цільової реалізації апаратного забезпечення. Реалізована функція обчислює затримку апаратного забезпечення кожного базового блоку як критичний шлях базового блоку, помножений на його відповідну частоту виконання. Загальна затримка HW кандидата отримується шляхом підсумовування всіх затримок HW усіх базових блоків, включених до can didate. Оцінка як SW, так і HW відбувається за принципом «знизу вгору», спочатку виконуючи оцінки кінцевих кандидатів і переходячи вгору до тих, що містять виклики іншим. Оцінка затримки апаратного зв'язку. Щоб врахувати

накладні витрати на затримку пам’яті через обмін даними між реалізованими апаратними прискорювачами та основною пам’яттю (THWm m (i)), вимоги до вводу/виводу для кожного AccelCand оцінюються в рамках LLVM шляхом отримання списку параметрів кожного кандидата та отримання вимог до даних для кожного типу кандидата (наприклад, розмір масиву цілих чисел, розмір структури тощо). Область оцінювання логіки HW. Вартість кандидата оцінюється як загальна необхідна площа ресурсів. Площа логічних одиниць обчислюється шляхом обліку таблиць пошуку (LUT) і DSP характеризованих операцій в межах одного базового блоку, а потім підсумовування всіх ресурсів усіх базових блокувань, включених до кандидата.

Область оцінки портів Master AXI. Щоб врахувати апаратні ресурси, необхідні для головного порту AXI, аналізується список параметрів кожного кандидата. Кожен ідентифікований масив враховує додаткові логічні одиниці (LUT), які вносять свій внесок у загальну площу.

Результати відбору кандидатів оцінювали шляхом впровадження відповідних апаратно-прискорених систем на Xilinx Zynq Ultrascale+ PSoC під керуванням операційних систем Linux. Система працює на частоті 100 МГц, а один із процесорів Cortex A53 призначений для виконання програмних (неприскорених) частин розглянутого тесту.

Вихідні положення для порівняльної оцінки

Якість вибору прискорювачів, запропонованих AccelSeeker, порівнювали з тими, які дизайнер отримав би, керуючись лише інструментом профілювання програмного забезпечення. Для таких базових рішень використовувався інструмент *gprof [29].* Gprof отримує дані про час виконання програмного забезпечення для всіх функцій, але не надає підтримки для оцінки часу виконання апаратного забезпечення, області апаратного забезпечення, а також накладних витрат на введення/виведення та виклик. Імітуючи можливі стратегії, яких слідував би дизайнер на основі даних профілювання, було розглянуто три можливі альтернативи:

- У підході «спершу в ширину» (називається *gprofl)* спочатку

вибирається кінцева функція з найвищим часом обчислення . Подальші функції розглядаються для апаратного виконання рекурсивно, як ті, що a) мають найвищий час обчислення в програмному забезпеченні, і b) які є або листами в графі викликів, або, якщо вони мають виклики, усі вони вже були вибрані на попередніх кроках. Після синтезу кандидат реалізується апаратно, якщо його включення в набір прискорювачів не порушує обмеження області.

• І навпаки, *gprof2* приймає позицію глибини. Він також починається з найбільш інтенсивного обчислювального листа в графі викликів програми. Потім він проходить через нього, ітеративно розглядаючи батьків поточного кандидата в порядку зменшення робочого навантаження, вибираючи найвище робоче навантаження, яке не перевищує місцевий бюджет.

• Нарешті, *gprof3 спочатку* вибирає функції, які потребують найбільшого обчислення (без урахування їх викликів), незалежно від ієрархії графа викликів.

У всіх випадках ці базові лінії не враховують функції, які становлять менше 0,5% загального часу виконання, оскільки вони не будуть цікавими для дизайнера. У наступному підрозділі показано, що AccelSeeker перевершує наведені вище стратегії, підкреслюючи, що більш повна інформація, яку він пропонує, є вирішальною для точного визначення кандидатів, які призведуть до вищого прискорення, і d визначає високопродуктивне апаратне/програмне розділення за певних ресурсних обмежень.

Порівняльний додаток

Експерименти проводилися на тесті декодування відео H.264, випущеному Університетом Іллінойсу [25], обробляючи три відеосегменти , надані авторами тесту (у форматах QCIF (176x144), CIF (352x288) і VGA (640x480) відповідно). Цільова реалізація містить 63 функції та понад 6000 рядків коду. Він походить від довідкового коду H.264, описаного в [25], який був адаптований для уникнення несинтезованих конструкцій. Його граф викликів представлений на Рисунку 3.7 разом із назвами деяких функцій.

Рейтинг кандидатів на акселерацію

У цьому підрозділі демонструється ефективність AccelSeeker у

визначенні AccelCand, які найбільше піддаються апаратному прискоренню. Для цього раунду експериментів ми застосували найкращі кандидати, запропоновані gprof або by

Таблиця 3.1.

Ранжування AccelCands на основі прискорення додатків, реалізованих як прискорювачів у реалізації Zynq PSoC (стовпець 2), а також відповідно до ранніх стратегій оцінки (AccelSeeker, стовпець 3, і gprof, стовпець 4).

Кандидат	Перевірка Рейтинг Zynq Ultrascale+	Оцінка Рейтинг (AccelSeeker)	Оцінка Рейтинг (gprof)
residual-block-cavlc-16	1	1	4
TrailingOnes-TotalCoeff	2	2	2
inter-prediction-chroma-double	3	3	5
шкала-залишок4x4	4	7	6
загальна кількість нулів	5	5	9
передбачення-Chroma	6	10	13
ІнтраІнфо	7	9	18
перед виконанням	8	4	15
шоубіти	17	17	1
Кліп3	18	18	3

AccelSeeker, не враховуючи ті, які є занадто великими для відображення в програмованій логіці використовуваної тестової системи (Xilinx Zynq XCZU9EG). У таблиці 3.1 AccelCands упорядковано за прискоренням, яке вони забезпечують на Zynq PSoC, коли реалізовано як прискорювачі (стовпець 2), порівняно з повним програмним виконанням. AccelSeeker оцінює дуже подібний рейтинг (зазначений у стовпці 3), лише з незначними відмінностями. Натомість рейтинг, заснований лише на інформації профілювання, як-от gprof (стовпець 4), погано корелює з фактично досягнутим прискоренням. Дійсно, деякі запропоновані кандидати (наприклад: *Clip3()* і *showbits())* фактично мають більший час роботи в апаратному забезпеченні, ніж у програмному забезпеченні, і мають поганий рейтинг як за AccelSeeker , так і за перевіркою. Результати посилаються на тестове відео QCIF. Дуже схожі результати були отримані за допомогою входів CIF і VGA, як показано в таблиці 3.2. Дев’ять із десяти кандидатів з найвищими заслугами однакові, з майже ідентичним рейтингом,

оскільки в середньому рейтинг змінюється на 0,7 у рейтинговій послідовності, порівнюючи QCIF із CIF, і на 0,3, порівнюючи QCIF із рейтингом VGA.

Продуктивність вибору прискорювача з обмеженими ресурсами

Щоб оцінити продуктивність запропонованого методу, прискорення додатків систем з апаратним прискоренням, вибраних AccelSeeker, за різних обмежень C_{max}, порівнюються з прискореннями, вибраними базовим методом.

Таблиця 3.2.

Ранжування AccelCands на основі оцінки переваг AccelSeeker для різних розмірів вхідних даних (QCIF, стовпець 2, CIF, стовпець 3 і VGA, стовпець 4).

Кандидат	**AccelSeeker Рейтинг QCIF**	**AccelSeeker Рейтинг CIF**	**AccelSeeker Рейтинг VGA**
residual-block-cavlc-16	1	1	1
TrailingOnes-TotalCoeff	2	2	2
inter-prediction-chroma-	3	3	3
перед виконанням	4	4	4
загальна кількість нулів	5	6	5
inter-prediction-chroma-	6	5	-
residual-block-cavlc-4	7	9	6
шкала-залишок4x4	8	8	8
TrailingOnes-ChromaDc	9	10	10
GetAnnexbNALU	10	-	9

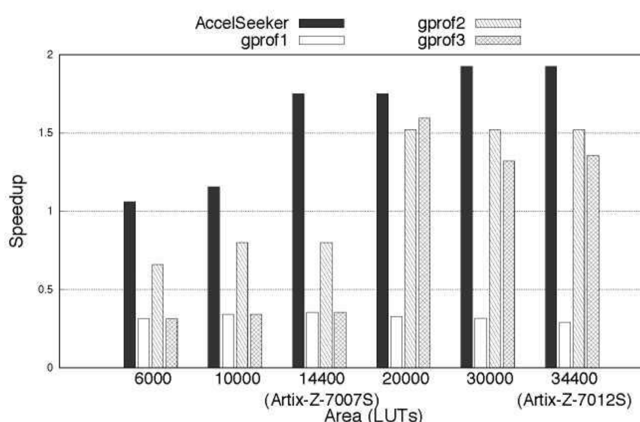


Рисунок 3.8 - Прискорення, досягнуте протягом усього часу виконання декодера H.264

Шляхом реалізації, як апаратних прискорювачів, наборів кандидатів, отриманих за допомогою AccelSeeker, і наборів, отриманих за допомогою стратегій профілювання gprof1, gprof2 і gprof3, змінюючи обмеження області

Таке обмеження виражається як максимальна кількість LUT, призначених для впровадження прискорювачів (включаючи два реальних PSoC, а саме Xilinx Artix Z-7007S і Z-7012S [86]); Подібні міркування можуть бути виведені шляхом обмеження BRAM або DSP, або їх комбінації. На Рисунку 3.8 показано ці результати для тестового введення QCIF. Прискорення досягається шляхом порівняння часу виконання тестової програми на прискорених системах (де вибрані AccelCand виконуються апаратно) з неприскореною (де всі частини працюють на процесорі PSoC). На Рисунку порівняльно також показано прискорення, отримані при використанні трьох стратегій на основі профілювання. Результати показують, що запропонований тут підхід забезпечує підвищення продуктивності навіть за дуже низьких обмежень області та прискорення в 1,9 раза для бюджету області в 34 400 таблиць пошуку (кількість, доступна для Artix Z-7012S середнього класу).

З іншого боку, кандидати, визначені всіма стратегіями профілювання, не можуть заощадити час виконання (натомість призводять до уповільнення) для вузьких областей, оскільки переваги апаратного прискорення применшуються накладними витратами на виклик і передачу даних, які не оцінюються інструментами, заснованими лише на даних профілювання. Навіть якщо досягнуто деякого підвищення продуктивності, як у випадку з *gprof2* і *gprof3* для більш м'яких обмежень, отримані вибрані елементи мають нижчу якість порівняно з AccelSeeker. Крім того, у базових стратегіях збільшення ресурсів, виділених на апаратне прискорення, може навіть погіршити фактичну продуктивність системи, оскільки все більше і більше неефективних кандидатів призначені для апаратного виконання. І навпаки, набори AccelCand, вибрані AccelSeeker, монотонно збільшують продуктивність у міру послаблення обмеження C_{max}. детально описано результати цієї методології та розглянуті базові лінії на основі профілювання, у таблиці 3.3 наведено кореневу функцію

AccelCands, вибраних для апаратного прискорення за різних обмежень області, а на Рисунку 3.9 показано їх на графіку викликів H.264 для бюджету 30 тис. LUT. Ці експериментальні дані підкреслюють, що підхід *gprofl* , орієнтований на ширину, має тенденцію до вибору великої кількості малих кінцевих функцій, які через високі накладні витрати не досягають високої продуктивності. Натомість позиція глибини (втілена в *gprof2)* може вибрати занадто мало кандидатів, оскільки вона обмежена лише зосередженням лише на одній гілці графіка викликів функцій. Можливості прискорення також втрачаються через повне ігнорування ієрархії графа викликів, як це зроблено в *gprof3.* Зрештою, вищу продуктивність можна досягти завдяки неочевидному вибору наборів прискорювачів, визначених AccelSeeker.

Причини цієї переваги подвійні. По-перше, AccelSeeker керується не лише частотою виконання, як це робить профілювання: натомість він може врахувати потенційне прискорення, яке можна використати за допомогою виконання апаратного забезпечення, а також компроміс із накладними *витратами* через передачу даних між процесорами та прискорювачами. Потім він може оцінити це в світлі *вартості ресурсів* , які вимагає виділений апаратний блок. По-друге, AccelSeeker уповноважений алгоритмом вибору який вирішує проблему *вибору прискорення* , максимізуючи переваги за обмеження вартості. Враховуючи такий екземпляр, як H.264, з графом викликів із 63 функціями, що призводить до графа конфлікту з 63 вузлів і 361 ребра, стає очевидним, що проблему не слід залишати на розв’язання розробниками вручну. На відміну від ручних підходів, заснованих лише на профілюванні, запропонована стратегія на основі компілятора добре підходить для вирішення цієї складної задачі.

Аналіз зусиль дизайнера

Один виклик AccelSeeker отримує повний набір даних про прискорення , зосереджуючись на тих, які можуть найкраще використовувати апаратне прискорення. І навпаки, усі базові лінії на основі профілювання вимагають методу проб і помилок, оскільки оцінки ресурсів недоступні, і на них не можна покладатися, щоб відхилити попередні AccelCand, які перевищують наявні

бюджети. Таким чином, ці стратегії або передбачають велику кількість прогонів синтезу для багатьох можливих варіантів *(gprofl , gprof3)*, або надмірно обмежують набір можливих кандидатів на прискорення, тим самим - перешкоджаючи результуючому прискоренню (як у випадку *gprof2).* Дійсно, ця спроба представлена на Рисунку 3.10: більшість кандидатів, визначених шляхом профілювання, зрештою порушують обмеження ресурсів через різні стратегії та обсяги доступних ресурсів. Синтезу таких кандидатів уникають, замість цього використовуючи AccelSeeker, що значно зменшує зусилля розробника щодо вибору високоефективного апаратно-програмного забезпечення. Дійсно, збір - усіх фаз AccelSeeker зайняв час порядку мілісекунд для експериментів.

3.2 EnergySeeker: прискорювачі для енергоефективності

Оскільки пристрої System-on-Chip з живленням від батареї стають все більш помітними та користуються високим попитом, потреба в спеціалізованому обладнанні, яке може служити для збереження значної кількості енергії, на додаток до прискорення додатків, як показано в попередньому розділі, стає більшою. Інструменти HLS і HW Description Language (HDL), як згадувалося в попередніх розділах, вимагають ручного прийняття рішень з боку програміста. Коли бюджет обмежений, рішення про те, які частини обчислень є більш вимогливими з точки зору споживання енергії, щоб матеріалізувати їх у HW прискорювачах малої потужності, не є тривіальним. Це вимагає глибокого розуміння програмного застосування та його характеристик, оскільки аспекти обчислювальної інтенсивності та керування пам’яттю є складними та їх важко визначити. Крім того, як обговорювалося в попередньому розділі, апаратний синтез -вимагає значної кількості часу і, враховуючи величезну кількість можливих альтернативних реалізацій, також обмежує кількість малопотужних прискорювачів, які інженер може розглянути вручну. З іншого боку, інструменти моделювання, такі як Aladdin, використаний у розділах 1 і 2, пропонують оцінку енергетичних і апаратних ресурсів вибраної цілі, а отже, більш швидку оцінку

порівняно з інструментами HLS. Однак ці інструменти не створюють функціональну апаратну реалізацію, яку можна використовувати на фізичній платі PSoC, як-от Xilinx Zynq Ultrascale+, яку використовували на етапі експериментів AccelSeeker .

Крім того, інструменти моделювання все ще потребують значних ручних зусиль для налаштування експериментальних середовищ, завдання, яке потрібно повторити для кожного розглянутого кандидата. У цьому розділі представлено EnergySeeker, щоб запропонувати вирішення проблеми визначення та вибору частин програми, які запропонують найбільші переваги енергозбереження за певного територіального бюджету. Це методологія, яка автоматично оцінює придатність кандидатів на енергозберігаюче обладнання з вихідного коду програми, що згодом дозволяє їх автоматичну ідентифікацію та вибір. EnergySeeker, реалізований в інфраструктурі компілятора LLVM [22] і заснований на AccelSeeker, спочатку забезпечує вимірювання вартості (необхідних ресурсів) і переваг (потенційно збереженої енергії) потенційних прискорювачів, а потім вибирає набір, який максимізує очікуваний приріст енергоефективності в межах визначеного бюджету HW ресурсів. Використання EnergySeeker може допомогти ¬інженерам на ранніх етапах проектування, вказуючи, які частини обчислень є енергоємними та їх слід використовувати для синтезу малопотужних прискорювачів, а які частини, натомість, навряд чи призведуть до значної економії енергії. Причиною останнього може бути те, що час обчислення в HW може бути різко збільшений порівняно з тим, що в SW (наприклад, вони мають високі накладні витрати на зв’язок з пам’яттю) або те, що необхідні ресурси HW є настільки енергоємними, що вони відповідають або перевищують відповідні енергетичні потреби сторони, яка використовує лише програмне забезпечення.

Велика частина дослідницької роботи була присвячена енергоефективності в -гетерогенних обчислювальних системах. У [19] [20] автори представляють кластерну багатоядерну обчислювальну систему з тісно пов'язаними 32-розрядними елементами обробки OpenRISC.

Енергоефективність досягається за допомогою паралелізму, але стверджується, що більша увага до ієрархії пам'яті та більша спеціалізація в реалізаціях HW підвищить як продуктивність, так і енергоефективність у запропонованій системі.

Застосовувані на малих ядрах, ядра RISC-V із наднизьким енергоспоживанням призводять до уповільнення, але забезпечують покращення енергозбереження, коли в додатках Інтернету речей (IoT) є тривалі періоди простою [29]. також присутні в пристроях IoT. У [25] пропонується реалізація спеціального апаратного забезпечення для виконання CNN на 65-нм SoC і, таким чином, для досягнення зниженого споживання енергії. Проте в жодному з цих випадків ідентифікація, оцінка або вибір частин виконання, які будуть синтезовані в апаратному забезпеченні, не розглядаються. Також були досліджені методи, включаючи автоматичне вставлення апаратної транзакційної пам'яті на етапі попередньої вибірки додатків для енергоефективності [22]. У [26] представлена автоматизована методологія, яка оцінює продуктивність і застосовує оптимізацію апаратних прискорювачів для низькопотужних глибоких нейронних мереж (DNN). Симулятор Aladdin [21] використовується для виконання оцінки вимог до потужності, а остаточна реалізація перевіряється на 40-нм технології CMOS. EnergySeeker, однак, пропонує автоматизовану фазу вибору, яка максимізує потенційний приріст енергії за даного бюджету території, окрім оцінки потужності. Крім того, дослідники націлилися на прискорювачі машинного навчання. Прискорювач Support Vector Machine і Active Learning Data Selection об'єднані з процесором з низьким енергоспоживанням [23] для запуску медичних програм і мінімізації вимог до електроенергії. Для досягнення кращої енергоефективності в алгоритмі Advanced Encryption Standard (AES) було досліджено розділення HW/SW [24] за допомогою бібліотеки OpenSSL [24]. Відповідно до розміру блоків даних шифрування автоматизований метод розподіляє вхідні завдання шифрування на апаратні реалізації або реалізації розширеного набору інструкцій існуючого програмного процесора. Ці методології зосереджені на впровадженні енергоефективних технічних засобів

для а) конкретних програм, а також б) автоматизації деяких процесів ідентифікації, оцінки та вибору частин виконання, які будуть матеріалізовані в апаратному забезпеченні. Натомість EnergySeeker достатньо загальний, щоб приймати будь-яку програму як вхідну інформацію, написану на C або C++, при цьому повністю автоматизуючи всі процеси, необхідні для визначення та вибору найбільш підходящих апаратних прискорювачів, що призводить до підвищення енергоефективності. пропонується автоматизована методологія, яка виконує оцінку продуктивності та потужності для розділення програмного/апаратного програмного забезпечення. Враховуючи функціональний опис C/C++ і визначене користувачем відображення HW/SW, надається оцінка, а потім фаза дослідження та оптимізації простору проектування для покращення даної конфігурації. Всупереч запропонованій методології, процес відбору кандидатів на апаратне забезпечення не автоматизований і не враховується бюджет місцевих ресурсів.

Подібний підхід до методології AccelSeeker був використаний для Energy -Seeker. Визначення кандидата для енергозбереження разом із оцінкою вартості та переваг та алгоритмом остаточного вибору підмножини кандидатів, які будуть реалізовані в апаратному забезпеченні, є такими самими, як представлено в підрозділах 3.1.3, 3.1.4, 3.1.6, 3.1.5. Основна відмінність полягає в тому, що під час оцінки продуктивності ¬апаратних прискорювачів Merit M () виражається не в збережених циклах, а в збереженій енергії. Зауважте, що, як зазначено в 3.1.6, методологія не обмежується певною цільовою метою і може бути адаптована до різних змінних, пов’язаних із заслугами та/або вартістю (наприклад, збережені цикли, збережена енергія для заслуг). Вартість C () оцінюється як сума оцінених логіки та пам’яті. Перший — це сума таблиць пошуку (LUT) і блоків DSP. Остання (пам'ять) залежить від вимог прискорювача до введення/виведення, які визначають кількість необхідних блоків BRAM. Оцінка збереженої енергії або Merit M () для апаратного прискорювача і натомість вимірюється в нано Джоулях (нДж) і виводиться за такою формулою:

$$M(i) = E_{SW}(i) - E_{HW}(i) \quad (3.1)$$

Енергоспоживання програмного ЦП (E SW) і відповідне енергоспоживання апаратних прискорювачів (E HW) визначаються за

$$E_{SW}(i) = P_{SW}(i) \times T_{SW}(i) \times n_{exec}$$
$$E_{HW}(i) = P_{HW}(i) \times T_{HW}(i) \times n_{exec}$$

формулою:

де Енергія виражається як добуток потужності (P), часу (T) і кількості викликів (n exec) даного прискорювача. P SW і P HW вимірюються у Ватах (Вт), і хоча реалізація програмного процесора, очевидно, фіксована, для апаратного прискорювача необхідна потужність залежить від його апаратної реалізації - (наприклад, кількості LUT, DSP, BRAM тощо). Час роботи T SW програмного процесора вимірюється в циклах (0,83 наносекунд на цикл для частоти процесора 1,2 ГГц), а для відповідного часу роботи апаратного прискорювача T HW тактується на частоті 100 МГц, що означає 10 наносекунд на цикл. Нарешті початкове рівняння заслуг стає таким:

$$M(i) = [(P_{SW}(i) \times T_{SW}(i)) - (P_{HW}(i) \times T_{HW}(i))] \times n_{exec} \quad (3.2)$$

У рамках визначення та вибору прискорювачів енергозбереження розширено аналіз компілятора на основі LLVM для EnergySeeker. Затримка апаратного забезпечення через обчислення, затримка програмного забезпечення та затримка зв’язку між основною пам’яттю та прискорювачами виконуються, як описано в підрозділі 3.1.7. Крім того, оцінка площі для апаратної логіки та портів Master AXI виконується, як описано в 3.1.7 Крім того, виконується оцінка потужності для апаратних прискорювачів, яка вимагає: a) площі логічних одиниць, b) кількості BRAM, c) кількості DSP і d) кількості масивів, підключених

до кожного прискорювача, оскільки останні враховуватимуть оцінку енергоспоживання через їх взаємозв'язки.

Вибір кандидатів EnergySeeker оцінювався шляхом впровадження апаратних прискорювачів на платі Xilinx Zynq Ultrascale+ PSoC з урахуванням зменшення динамічної потужності. Програмний процесор Cortex A53 мав тактову частоту 1,2 ГГц, а апаратні прискорювачі — 100 МГц. Основною пам'яттю системи є DDR4 SDRAM. Для вимог до потужності як програмного процесора, так і прискорювачів використовувався [27]. XPE від Xilinx пропонує інструмент аналізу потужності в найгіршому випадку, який оцінює енергоспоживання даної конструкції на будь-якій фазі циклу проектування. Основою для порівняльної оцінки були стратегії профілювання на основі інструменту gprof (gprof1, gprof2 і gprof3). Еталонним тестом програми, який використовувався для проведення експерименту, був декодер H.264 [25], той самий, що використовувався обробляючи як вхід відео у форматі QCIF (176x144).

Щоб виконати оцінку продуктивності EnergySeeker, порівнюється енергоефективність прикладної програми частин із апаратним прискоренням, вибраних EnergySeeker, за різних обмежень області (максимальна кількість LUT), з тими, що вибрані базовими методами. Енергоефективність отримується шляхом порівняння енергоспоживання за весь час роботи еталонної програми на програмному процесорі з енергоспоживанням для гібридної конструкції, де вибрані апаратні прискорювачі використовуються разом із неприскореними частинами, які залишаються в програмному процесорі. На рисунку 3.11 порівняльно показана енергоефективність, отримана при використанні трьох стратегій на основі профілювання, проти EnergySeeker. Було проведено три різні раунди експериментів. Один із одним активним ядром програмного процесора, другий із двома активними ядрами й один із чотирма. Можна помітити, що EnergySeeker стабільно перевершує всі три стратегії профілювання на основі gprof для різних обмежень області та в усіх трьох налаштуваннях (одне, два та чотири активних ядра ЦП). Апаратно-програмний підхід, керований

EnergySeeker, є в 2,2 рази енергоефективнішим порівняно з підходом лише програмного забезпечення завдяки тому факту, що апаратне забезпечення споживає значно менше енергії порівняно з енергоємним процесором ЦП, який працює на вищій частоті щодо програмованої логіки. Порівнюючи EnergySeeker із найсучаснішими методологіями, заснованими лише на профілюванні, наш підхід використовує модель оцінки затримки, як показано в попередньому розділі, яка поєднується з точною оцінкою потужності необхідної HW ресурси. На додаток до цього EnergySeeker підтримується алгоритмом вибору, описаним вище, який максимізує підвищення енергоефективності за обмеження площі. EnergySeeker, таким чином, веде до більш ефективного вибору з точки зору енергоефективності, оскільки для бюджетів рівних площ робиться кращий вибір, який потребує як менше часу для роботи прискорювачів, так і меншої потужності, порівняно з вибором, заснованим виключно на профільній інформації.

Фреймворки AccelSeeker і EnergySeeker були представлені для допомоги системним архітекторам на ранній стадії проектування систем з апаратним прискоренням. Вони націлені на краще прискорення та покращену енергоефективність, відповідно, у дизайні апаратного та програмного забезпечення. Автоматично оцінюючи потенційне прискорення різних варіантів апаратного прискорення в їх реалізації за замовчуванням, а також потрібні апаратні ресурси, AccelSeeker дозволяє архітекторам точно визначити розділи коду, які є гідними цілями для подальшого, більш детального аналізу та оптимізації. AccelSeeker виконує ідентифікацію потенційних прискорювачів, а також оцінки їх площі та прискорення за допомогою проходів аналізу компілятора, реалізованих у компіляторі LLVM, не вимагаючи тривалих і детальних оцінок кожного кандидата на прискорення окремо. Потім він автоматично вибирає набір кандидатів, які максимізують оцінене прискорення за заданого обмеження ресурсу. Експериментальні дані підкреслюють, що апаратне/програмне забезпечення, вибране AccelSeeker, значно перевершує варіанти, які ґрунтуються виключно на інформації профілю. Ця дослідницька

робота була опублікована на Міжнародній конференції з комп'ютерного дизайну (ICCD) 2019 року [25]. EnergySeeker пропонує розширення попереднього ланцюжка інструментів, який зосереджується на енергоефективному виборі, виконуючи оцінку потужності прискорювачів разом із оцінкою їхньої затримки, запропонованою фреймворком AccelSeeker. Експерименти виявили значну економію енергії до 55% менше необхідної енергії порівняно з підходом лише програмного забезпечення та незмінно кращу продуктивність порівняно з методологіями, заснованими на інформації про профілювання.

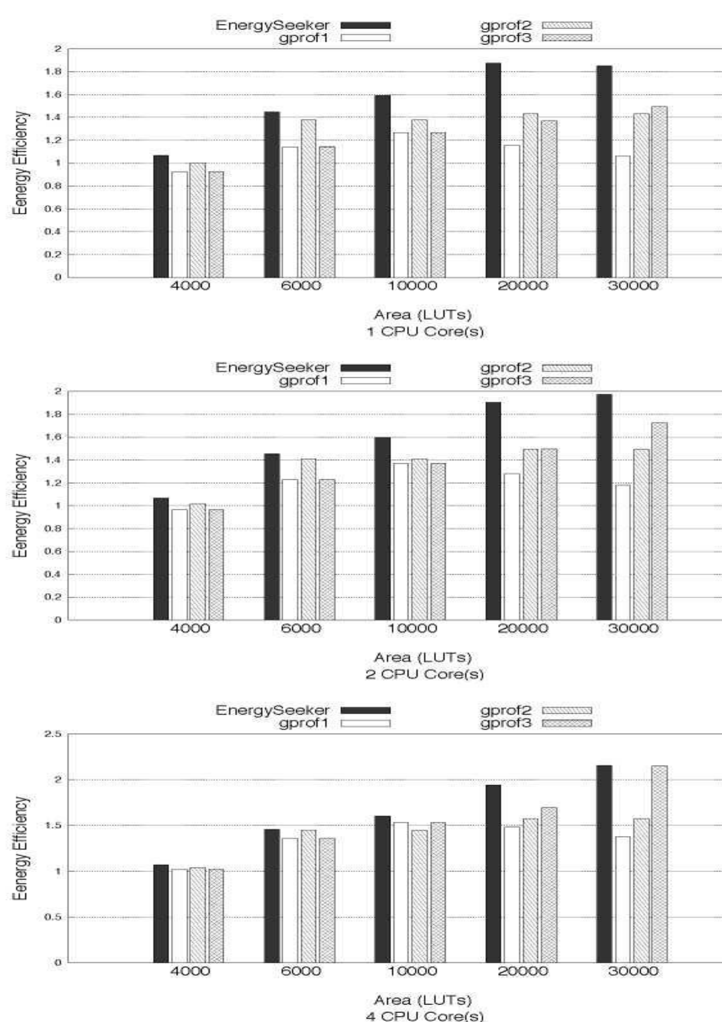


Рисунок 3.11 - Енергоефективність, отримана за весь час роботи декодера H.264

при використанні в якості апаратних прискорювачів наборів-кандидатів, отриманих за допомогою EnergySeeker, та наборів, отриманих за допомогою стратегій профілювання даних gprof1, gprof2 та gprof3, з різним обмеженням на

область. Кількість активних ядер центрального процесора програмного забезпечення: Одне ядро, два ядра та чотири ядра.

3.3 Гетерогенні обчислення

Гетерогенні обчислення є одним з найбільш перспективних підходів до підвищення продуктивності майбутніх обчислювальних систем. Тим не менш, повторний пошук розробки ефективних і ефективних платформ виконання HW/SW вимагає великої уваги та викликів. Вибір частин програми, які будуть прискорені в HW в автоматичному режимі, вибір оптимізації, яка буде застосована до прискорених частин HW, а також розгляд характеристик платформи, на якій реалізовано апаратні прискорювачі, — все це складні проблеми та питання дослідження.

Щоб вирішити їх, я розробив нову керовану компілятором методологію, яка виконує аналіз вихідного коду програмних додатків. Основна вигода від такого підходу двояка. По-перше, робота та рішення, які раніше виконувалися дизайнерами та інженерами вручну, виконуються систематично й автоматично, що призводить до швидших і менш схильних до помилок процесів. По-друге, досягнута продуктивність, або перекладена як прискорення, або як енергоефективність, у реалізованих гетерогенних конструкціях значно покращена порівняно з найсучаснішими методами. Нижче наведено короткий виклад моїх внесків, детально викладених у попередніх розділах дисертації.

3.4 Висновки по розділу

В даному розділі враховано цільову платформу , на якій синтезовано HW-реалізації. Коли націлені вимогливі обчислювальні системи зі складною ієрархією пам’яті, затримка через зв’язок між основною пам’яттю та апаратними прискорювачами може бути значною. Представлено AccelSeeker , який пропонує розширений діапазон кандидатів на прискорення – для всього графа викликів функцій програми. AccelSeeker виконує ідентифікацію прискорювачів-

кандидатів, а також оцінки їх площі та прискорення за допомогою аналізу компілятора. Згодом відбувається автоматичний вибір набору кандидатів, який максимізує оцінене прискорення за заданого обмеження ресурсу. Експериментальна оцінка складного тесту, декодера H.264, продемонструвала, що AccelSeeker значно перевершує варіанти, які ґрунтуються виключно на інформації профілювання. Як розширення попереднього ланцюжка інструментів, EnergySeeker зосереджується на енергоефективному виборі. Експерименти виявили значну економію до 55% менше енергії порівняно з підходом лише програмного забезпечення та незмінно кращу продуктивність порівняно з методологіями, заснованими лише на профілюванні.

ВИСНОВКИ

Дослідження оптимізаційних технік компіляції, представлене в даній роботі, підкреслює важливість комплексного підходу до аналізу та вдосконалення конструкцій програмування, оптимізації коду та системно-орієнтованих аспектів. Проведений аналіз конструкцій програмування виявив ефективність різних алгоритмів вибору, таких як точний, жадібний та комбінований методи, а також запропонував фреймворк RegionSeeker для оцінки продуктивності парадигм. Експериментальні результати підтвердили, що правильний вибір конструкцій значно впливає на продуктивність програм, особливо в задачах з високими накладними витратами.

Оптимізація коду в парадигмах показала, що аналіз повторного використання даних та передбачення оптимізацій за допомогою методів машинного навчання, зокрема для прогнозування факторів розгортання циклів, дозволяють досягти суттєвого підвищення ефективності. Розроблені методології та випущене програмне забезпечення створюють міцну основу для подальшого вдосконалення компіляторів.

Системно-орієнтовані аспекти підкреслили важливість паралелізму та енергоефективності в сучасних обчислювальних системах. Алгоритми вибору акселераторів (AccelSeeker та EnergySeeker) продемонстрували здатність оптимізувати продуктивність та зменшувати енергоспоживання, що є критично важливим для масштабних застосувань.

Загалом, результати роботи вказують на необхідність інтеграції різноманітних технік — від аналізу конструкцій до системної оптимізації — для створення ефективних компіляторів. Подальші дослідження можуть бути спрямовані на автоматизацію вибору оптимізаційних стратегій та адаптацію цих технік до нових архітектур і парадигм програмування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Muchnick S. S. Advanced Compiler Design and Implementation / Steven S. Muchnick. — Morgan Kaufmann, 1997. — 856 с. — Режим доступу: https://www.elsevier.com/books/advanced-compiler-design-and-implementation/muchnick/978-1-55860-320-2
2. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, Techniques, and Tools / Alfred V. Aho et al. — 2nd ed. — Addison-Wesley, 2006. — 1009 с. — Режим доступу: https://www.pearson.com/us/higher-education/product/Aho-Compilers-Principles-Techniques-and-Tools-2nd-Edition/9780321486813.html
3. Cooper K. D., Torczon L. Engineering a Compiler / Keith D. Cooper, Linda Torczon. — 2nd ed. — Morgan Kaufmann, 2011. — 824 с. — Режим доступу: https://www.elsevier.com/books/engineering-a-compiler/cooper/978-0-12-088478-0
4. Allen R., Kennedy K. Optimizing Compilers for Modern Architectures: A Dependence-based Approach / Randy Allen, Ken Kennedy. — Morgan Kaufmann, 2001. — 790 с. — Режим доступу: https://www.elsevier.com/books/optimizing-compilers-for-modern-architectures/allen/978-1-55860-286-1
5. Wolfe M. High-Performance Compilers for Parallel Computing / Michael Wolfe. — Addison-Wesley, 1995. — 592 с. — Режим доступу: https://www.pearson.com/us/higher-education/product/Wolfe-High-Performance-Compilers-for-Parallel-Computing/9780805327304.html
6. Bacon D. F., Graham S. L., Sharp O. J. Compiler Transformations for High-Performance Computing / David F. Bacon et al. — ACM Computing Surveys, 1994. — Vol. 26, No. 4. — С. 345–420. — Режим доступу: https://dl.acm.org/doi/10.1145/197405.197406
7. Kuck D. J., Kuhn R. H., Padua D. A., Leasure B., Wolfe M. Dependence Graphs and Compiler Optimizations / David J. Kuck et al. — ACM SIGPLAN Notices, 1981. — Vol. 16, No. 6. — С. 207–218. — Режим доступу: https://dl.acm.org/doi/10.1145/800206.806366

8. Lattner C., Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation / Chris Lattner, Vikram Adve. — Proceedings of the International Symposium on Code Generation and Optimization (CGO), 2004. — С. 75–86. — Режим доступу: https://dl.acm.org/doi/10.1109/CGO.2004.1281665

9. Kennedy K., Allen J. R. Automatic Translation of Fortran Programs to Vector Form / Ken Kennedy, J. R. Allen. — ACM Transactions on Programming Languages and Systems, 1987. — Vol. 9, No. 4. — С. 491–542. — Режим доступу: https://dl.acm.org/doi/10.1145/29873.29875

10. Padua D., Wolfe M. Advanced Compiler Optimizations for Supercomputers / David Padua, Michael Wolfe. — Communications of the ACM, 1986. — Vol. 29, No. 12. — С. 1184–1201. — Режим доступу: https://dl.acm.org/doi/10.1145/7902.7904

11. Sarkar V. Automatic Selection of High-Order Transformations in the IBM XL Fortran Compilers / Vivek Sarkar. — IBM Journal of Research and Development, 1997. — Vol. 41, No. 3. — С. 233–264. — Режим доступу: https://ieeexplore.ieee.org/document/6276737

12. Appel A. W. Modern Compiler Implementation in ML / Andrew W. Appel. — Cambridge University Press, 1998. — 552 с. — Режим доступу: https://www.cambridge.org/core/books/modern-compiler-implementation-in-ml/9780521582742

13. Gupta R., Epstein D., Schwon P. Machine Learning for Compiler Optimization / Rajeev Gupta et al. — Proceedings of the IEEE, 2003. — Vol. 91, No. 7. — С. 1049–1066. — Режим доступу: https://ieeexplore. ieee.org/ document/1213648

14. Compiler Optimization Techniques. — Режим доступу: https://www.cs.rice.edu/~keith/Research/compiler-opt.html. — Дата доступу: 2025-04-29.

15. Loop Optimization in Modern Compilers. — Режим доступу: https://www.cs.tufts.edu/~nr/cs257/archive/loop-optimization.pdf. — Дата доступу: 2025-04-29.

16. Parallelizing Compilers: Techniques and Challenges. — Режим доступу: https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/parallel-compilers/index.html. — Дата доступу: 2025-04-29.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи бакалавра: " **Оптимізаційні техніки компіляції** "

Обсяг пояснювальної записки: 83 аркушів

Дата закінчення дипломної роботи 10 червня 2025р.

Підпис студента _______________