НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

«На правах рукопису»
УДК 004.891.2

До захисту допущено:
Завідувач кафедри
_____ Олександр РОЛІК
«06» червня 2022 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Інтегровані інформаційні системи»

зі спеціальності 126 «Інформаційні системи та технології»

на тему: «Програмна система автоматизації створення оптимізованих додатків користувача для паралельних вбудованих систем»

Виконав:
студент ІІ курсу, групи ІА-01мн
Душабаєв Рустам Толкинбайович _____

Керівник:
Професор, д.т.н, с.н.с,
Чемерис Олександр Анатолійович _____

Консультант з _____:
_____
_____ _____

Рецензент:
молодший науковий співробітник
Інституту проблем моделювання в енергетиці
ім. Г.Є. Пухова НАН України, к.т.н.,
Сушко Сергій Володимирович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2022 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Рівень вищої освіти – другий (магістерський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-наукова програма «Інтегровані інформаційні системи»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олександр РОЛІК
«06» червня 2022 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Душабаєву Рустаму Толкинбайовичу

1. Тема дисертації «Програмна система автоматизації створення оптимізованих додатків користувача для паралельних вбудованих систем», науковий керівник дисертації Чемерис Олександр Анатолійович, Професор, д.т.н, с.н.с, затверджені наказом по університету від «26» квітня 2022 р. № НС/88/2022

2. Термін подання студентом дисертації 08.06.2022

3. Об’єкт дослідження: паралельні вбудовані системи

4. Предмет дослідження: автоматизована система створення оптимізованих додатків користувача для паралельних вбудованих систем

5. Перелік завдань, які потрібно розробити: огляд та аналіз проблеми оптимізації додатків, аналіз систем для оптимізації додатків, опис математичної моделі для оптимізації багаторівневих вкладених циклів методом тайліенгу, опис методу оптимізації розміру тайлів з використанням генетичного алгоритму, розробка системи автоматичної оптимізації розміру тайлу, експериментальні дослідження отриманої системи

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: блок-схема алгоритму розробленої системи, графіки залежності номеру покоління та кращого рішення, порівняльні графіки замірів часу виконання додатків, порівняльні

графіки замірів часу виконання додатків з використанням OpenMP, графік середенього часу виконання програми в кожному поколінні, графік відхилення в кожному поколінні, графік порівняння замірів часу виконання базової реалізації алгоритму, графіки залежності часу виконання програм від розміру тайлу

7. Орієнтовний перелік публікацій: стаття «Оптимізація розміру тайлу при розпаралелюванні вкладених циклів за рахунок використання генетичного алгоритму», конференція «Кібербезпека енергетики-2022»

8. Консультанти розділів дисертації*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання 31.01.2022

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Огляд предметної області	01.02.22 – 04.02.22	
2	Аналіз існуючих рішень	07.02.22 – 18.02.22	
3	Опис математичної моделі для оптимізації багаторівневих вкладених циклів методом тайліенгу	21.02.22 – 11.03.22	
4	Опис методу оптимізації розміру тайлів з використанням генетичного алгоритму	14.03.22 – 18.03.22	
5	Аналіз та вибір технічних засобів для реалізації системи	21.03.22 – 01.04.22	
6	Розроблення системи	04.04.22 – 22.04.22	
7	Оформлення текстової документації	28.04.22 – 24.05.22	
8	Подання дисертації до попереднього захисту	25.05.2022	

Студент Рустам ДУШАБАЄВ

Науковий керівник Олександр ЧЕМЕРИС

* Якщо визначені консультанти. Консультантом не може бути зазначено наукового керівника магістерської дисертації.

РЕФЕРАТ

Магістерська дисертація на здобуття ступеня «магістр» за освітньо-науковою програмою підготовки «Інтегровані інформаційні системи» на тему «Моделювання динамічних процесів крокуючого робота». Дисертація містить 102 сторінки, 75 рисунків, 14 таблиць, 1 додаток, 30 джерел.

Актуальність. Компілятори, які використовують для підготовки програмного забезпечення, розвиваються та постійно покращують техніки оптимізації програмного коду. Однак, існуючі методи оптимізації не дають ідеального результату, а також існує множина програм, що потребує додаткової оптимізації. Таким чином задача автоматизації оптимізації потребує творчого підходу, тому вона залишається актуальною.

Метою магістерської дисертації є підвищення ефективності оптимізації додатків методом тайліенгу.

Об`єкт дослідження: паралельні вбудовані системи.

Предмет дослідження: автоматизована система створення оптимізованих додатків користувача для паралельних вбудованих систем.

Наукова новизна одержаних у магістерській дисертації результатів полягає у вдосконаленні ефективності оптимізації додатків методом тайліенгу, а саме – у реалізації пошуку оптимальних розмірів тайлів методом генетичного алгоритму.

Публікація результатів дисертації. За результатами роботи було опубліковано статтю:

Chemerys OA Optimization of the tile size for parallelizing nested loops using the genetic algorithm // OA Chemerys, RT Dushabaiev International scientific magazine “Internauka”. – 2022 - №5 https://doi.org/10.25313/2520-2057-2022-5-8002

Ключові слова: ВКЛАДЕНІ ЦИКЛИ, ТАЙЛІЕНГ, ГЕНЕТИЧНИЙ АЛГОРИТМ, PLUTO, POLYBENCH, DEAP.

ABSTRACT

Master's dissertation on the educational-scientific level training program “Master” on the theme "Software system for automating the creation of optimized user applications for parallel embedded systems". The dissertation contains 102 pages, 75 figures, 14 tables, 1 application, 30 sources.

Relevance. Compilers used to prepare software are constantly evolving and improving software optimization techniques. However, the existing optimization methods do not give a perfect result, and there are many programs that need additional optimization. Thus, the task of optimization optimization requires a creative approach, so it remains relevant.

The purpose of the master's dissertation is to increase the efficiency of optimization of applications by tiling.

Object of research: parallel embedded systems.

Subject of research: automated system for creating optimized user applications for parallel embedded systems.

The scientific novelty of the results obtained in the master's dissertation is improvement efficiency of application optimization by the method of tailing, namely - in the implementation of the search for optimal tile sizes by the method of genetic algorithm.

Publication of dissertation`s results. According to the results of the work, an article was published:

Чемерис О. А., Душабаєв Р. Т. Оптимізація розміру тайлу при розпаралелюванні вкладених циклів за рахунок використання генетичного алгоритму // Міжнародний науковий журнал "Інтернаука". — 2022. — №5. https://doi.org/10.25313/2520-2057-2022-5-8002

Keywords: NESTED LOOPS, TILING, GENETIC ALGORITHMS, PLUTO, PLOYBENCH, DEAP.

ЗМІСТ

ВСТУП

У зв’язку зі стрімким розвитком багатоядерних процесорів, постає питання як їх ефективно використовувати. Така потреба прослідковується і в сфері програмного забезпечення для наукової діяльності. Так, в загальному випадку, в наукових обчислювальних програмах присутня велика кількість добре структурованих циклів, що займають найбільшу частину часу виконання. Одним зі способів пришвидшити роботу таких програм є використання багатопоточності. Запустивши цикли паралельно в окремих потоках можна було б поліпшити час виконання усієї програми. Однак, через наявність складних залежностей цикли не можливо запустити паралельно без додаткових втручань, наприклад, зміна циклів місцями або розчеплення циклів. Цикли, що не мають вкладених циклів або мають їх незначну кількість (<= 3), зазвичай можна легко розкласти на паралельні шляхом їх перетворень. Випадки з великою кількістю вкладених циклів слід розглядати окремо. В деяких випадках вони все ще можуть бути безпосередньо запущені паралельно. Нажаль такі ситуації виникають рідко. Найчастіше ні один зі вкладених циклів не може бути запущений паралельно без додаткових втручань. Зі збільшенням кількості вкладених циклів залежності між ітераціями циклів стають дедалі заплутаними, що ускладнює процес аналізу їх аналізу.

Мета: підвищення ефективності оптимізації додатків методом тайліенгу.

Для досягнення поставленої мети були вирішені наступні **задачі**:

- огляд і аналіз предметної області
- огляд і аналіз існуючих рішень для оптимізації вкладених циклів
- дослідження методу оптимізації циклів шляхом розбиття їх на тайли певного розміру
- реалізація системи оптимізації на базі програмного пакету Pluto
- виконання модифікації розробленої системи за допомогою генетичного алгоритму
- дослідження розробленої системи на тестових програма з пакету PolyBench

Для вирішення задач магістерської дисертації були використані наступні методи:

- метод розбиття вкладених циклів на тайли фіксованого розміру
- генетичний алгоритм
- інструменти зі стандартної бібліотеки Python

Об’єктом дослідження паралельні вбудовані системи, **предмет дослідження** – автоматизована система створення оптимізованих додатків користувача для паралельних вбудованих систем.

Наукова новизна полягає у вдосконаленні ефективності оптимізації додатків методом тайліенгу, а саме – у реалізації пошуку оптимальних розмірів тайлів методом генетичного алгоритму.

Публікація результатів дисертації. Відповідно до результатів роботи було опубліковано статтю:

Чемерис О. А., Душабаєв Р. Т. Оптимізація розміру тайлу при розпаралелюванні вкладених циклів за рахунок використання генетичного алгоритму // Міжнародний науковий журнал "Інтернаука". — 2022. — №5. https://doi.org/10.25313/2520-2057-2022-5-8002

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

Різні дослідження показують, що цикли є найбільш критичними частинами багатьох програм. Як видно в Таблиці 1.1 кілька популярних алгоритмів обробки сигналів витрачають в середньому 95% часу виконання в кількох циклах із інтенсивними обчисленнями.

Таблиця 1.1 – кількість циклів в популярних алгоритмах [1]

Назва алгоритму	Кількість циклів	Кількість циклів, що потребують >1% часу від загального часу виконання	Загальний відсоток часу витрачений в циклах, %
Wavelet image compression	25	13	99%
EPIC encoding	132	13	92%
UNEPIC decoding	62	15	99%
Media Bench ADPCM	3	3	98%
MPEG-2 encoder	165	14	85%
Skipjack encryption	6	2	99%

Як наслідок створюються методи і техніки трансформації циклів задля прискорення їх роботи. Розглянемо теорію трансформації циклів.

1.1 Граф простору ітерацій

Вкладені цикли глибиною n можуть буди представлені у вигляді графу простору ітерацій з осями, що відповідають різним циклам (рисунок 1.1). Осі помічені відповідними змінними, що являють собою індекс і обмежені верхніми межами циклів. Кожна ітерація представлена у вигляді вузла в графі та визначається його індекс-вектором $p = (p1, p2, \dots, p_n)$, де p_i – значення індексу i-го циклу. Рахуються вони від зовнішнього до внутрішнього циклу. Припускаючи, що крок циклу більше нуля, ми можемо зазначити, що індекс-вектор p ітерації є лексикографічно більшим ніж q і позначити як $p \succ q$ тоді і тільки тоді, коли $p_1 > q_1$ або $p_1 = q_1$ і $(p_2, \dots, p_n) \succ (q_2, \dots, q_n)$. До того ж, $p \succcurlyeq q$ тоді і тільки тоді, коли $p \succ q$, або $p = q$. В загальному випадку, ітерація p буде виконана після ітерації q тоді і тільки тоді, коли $p \succ q$. Порядок виконання може бути представлений у вигляді ребер між вузлами на графі простору ітерацій, тим самим позначаючи напрямок проходу простору ітерацій.

Залежність даних, тобто обмеження порядку переходів між вузлами ітерацій, визначає допустимий порядок виконання вузлів, що семантично еквівалентний до лексикографічного виконання вузлів.

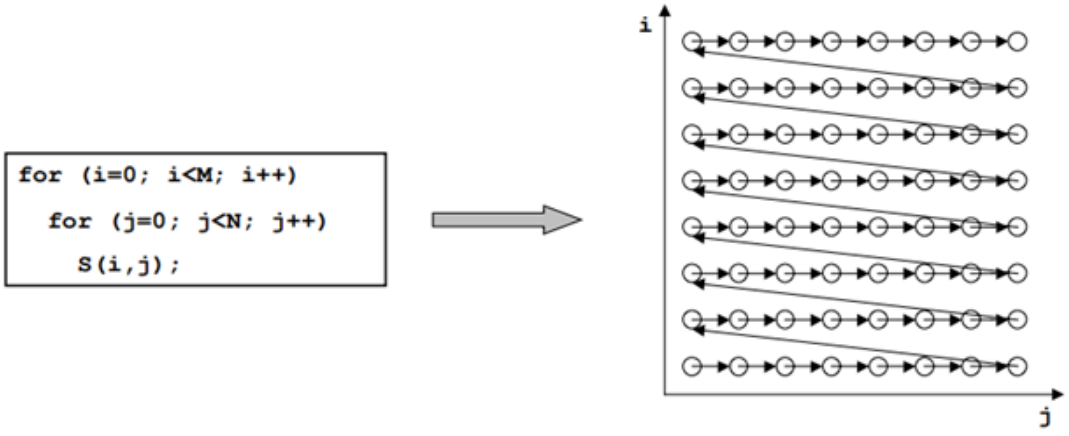


Рисунок 1.1 – граф простору ітерацій

Порядок виконання ітерацій може бути розширений операцією над циклом, що позначається «⊳». Нехай маємо дві операції $S_1[\vec{p}]$ і $S_2[\vec{q}\,]$, де $\vec{p}$ і $\vec{q}$ – це ітерації, що мають S_1 і S_2 відповідно. $S_1[\vec{p}] \rhd S[\vec{q}]$ означає, що $S[\vec{q}]$ буде виконана після того як завершиться $S_1[\vec{p}]$. В загальному випадку, $S_1[\vec{p}] \rhd S[\vec{q}]$ тоді і тільки тоді, коли в послідовності операцій S_1 йде після S_2 і $\vec{p} \succcurlyeq \vec{q}$ або S_1 є такою самою операцією, що і S_2 або передує її і $\vec{p} \succ \vec{q}$. Порядок виконання операції може бути зображено графічно, так само як і порядку обходу простору ітерацій,

1.2 Залежність даних

Нехай маємо дві операції доступу до однієї і тієї ж частини памяті $S_1[\vec{p}]$ і $S_2[\vec{q}]$, такі що $S_1[\vec{p}] \rhd S[\vec{q}]$. Тобто між двома операціями існує залежність в даних, а значить і між двома ітераціями. Для опису таких залежностей даних між циклами використовуються так звані вектори відстаней і залежностей

Вектор відстаней для вкладених циклів з глибиною n є n-мірний вектор $\vec{\delta} = (\delta_1, \dots, \delta_n)$, такий що ітерація з індекс-вектором $\vec{p} + \vec{\delta} = (p_1 + \delta_1, \dots, p_n + \delta_n)$, залежить від індекс-вектору $\vec{p}$. Якщо існує залежність даних між $S_1[\vec{p}]$ і $S_2[\vec{q}]$, тоді вектор відстаней буде мати вигляд: $\vec{\delta} = \vec{p} - \vec{q}$.

Вектор залежностей для циклів з глибиною n є n-мірний вектор $\vec{d} = ([d_1^-, d_1^+], \dots, [d_n^-, d_n^+])$, що узагальнює множину векторів відстаней, яка має вигляд:

$$DV(\vec{d}) = \{(\delta_1, \dots, \delta_n) | d_i^- \leq \delta_i \leq d_i^+\} \tag{1.1}$$

Слід заначити, що відстань залежності $(0, \dots, 0)$ немає жодного ефекту на трансформацію циклів і залишає порядок окремих операцій і інструкцій незмінним.

Варто відмітити, що залежність може не стосуватись внутрішніх циклів так само як і мати відношення до зовнішніх.

1.3 Методи трансформації

На сьогоднішній день існує низка методів трансформації циклів [2]:

Поділ – перетворення обчислювального циклу в кілька інших обчислювальних циклів з тим же діапазоном індексів, але різними частинами початкового циклу

Злиття - об'єднання сусідніх обчислювальних циклів в один цикл з однаковим діапазоном індексів

Переміщення коду, інваріантно до циклів – переміщення тих частин коду за межі тіла циклу, які знаходяться всередині циклу, але які не залежать від обчислень всередині циклу;

Розпаралелювання – одночасне виконання одних і тих самих операцій обчислювального циклу на різних обчислювальних пристроях

Конвеєрування – модифікація тіла циклу для найбільш ефективного конвеєра команд або обчислювальних блоків;

Планування – поділ тіла циклу на кілька частин, які можуть працювати одночасно на кількох процесорах;

Відхилення – переміщення ітерацій багатовимірного обчислювального циклу з метою переміщення залежностей ітерацій у крайній обчислювальний цикл;

Розділення – спрощення обчислювального циклу або усунення залежностей шляхом розбиття тіла циклу на кілька обчислювальних циклів, які мають однакові команди, але з меншим діапазоном ітерацій;

Тайліенг – реорганізація обчислювального циклу для зменшення використання кількості даних циклу з метою покращення локальності даних;

Векторизація – групування кількох ітерацій циклу для використання векторних операцій;

Розгортання – розгортання тіла циклу на еквівалентну кількість нециклових операцій

Розглянемо деякі методи більш детально

1.3.1 Паралелізм

Зараз існує два підходи до проблеми паралельності багаторівневих вкладених циклів. Одним з підходів є використання технології багатопотокової реалізації базового апаратного забезпечення, в якій відповідний рівень циклу вибирається для багатопотокового розпаралелювання за допомогою базового апаратного забезпечення, наприклад, техніки спекуляцій на рівні потоків (TLS) і транзакційної пам'яті (TM). Однак можливості аналізу залежностей базового апаратного забезпечення обмежені та не мають гнучкості, що ускладнює метод TLS для аналізу складних залежностей багаторівневих вкладених циклів і, таким чином, неможливо забезпечити ефективний паралелізм. Наприклад, апаратна техніка TM має обмеження на розмір транзакції (буфер транзакцій) і тривалість (події та переривання операційної системи можуть перервати транзакцію); тому при застосуванні до паралельності вкладених циклів на рівні потоків це часто призводить до переповнення базових апаратних ресурсів через надмірну деталізацію самої транзакції, що в кінцевому підсумку проявляється низькою ефективністю паралельного виконання [3].

Іншим поширеним підходом є використання традиційних програмних методів статичної компіляції, в яких модуль аналізу залежностей у компіляторі використовується для перевірки, чи є залежності між ітераціями циклу покроковим способом. Однак зі збільшенням кількості рівнів вкладених циклів, які підлягають перевірці, кількість залежностей, які необхідно перевірити, збільшується експоненціально або навіть факторно, що робить час компіляції нестерпно довгим. Зіткнувшись з цією ситуацією, основні компілятори (наприклад, gcc) зазвичай використовують консервативний підхід; тобто вони припускають, що залежності присутні між усіма ітераціями циклів, і, таким чином, паралелізм не може бути реалізований. Як правило, наукові обчислювальні програми містять велику кількість циклів, а також велику кількість рівнів вкладення.

Дослідники з Університету Міннесоти та Intel провели опитування щодо максимальної кількості вкладених рівнів циклів і циклів у деяких наукових обчислювальних програмах у SPEC2006, і результат наведено в Таблиці 1.2 [4]. Очевидно, що кількість вкладених рівнів циклу в стандартному тестовому наборі SPEC2006 дуже велика (наприклад, глибина вкладення 445.gobmk досягає навіть 22 рівнів), з великою кількістю таких циклів. Тому вирішення проблеми паралелізму у вкладених циклах може не тільки підвищити загальну продуктивність загальнонаукових обчислювальних програм, але й мати велике практичне значення для розширення сфери застосування паралельності на рівні потоків та ефективного використання багатоядерних процесорів.

Таблиця 1.2 – кількість циклів

Назва алгоритму	Найбільша глибина вкладених циклів	Загальна кількість циклів
401.bzip2	11	232
429.mcf	5	52
445.gobmk	22	1265
456.hmmer	5	851
458.sjeng	10	254
462.libquantum	4	94
464.h264ref	15	1870
473.astar	6	116
433.milc	11	421
444.namd	4	619
453.povray	15	1311
470.lbm	3	23

Багато дослідників неофіційно зазначали, що вибір найменшого набору шарів циклу з багаторівневих вкладених циклів для отримання максимального паралелізму є недетермінованою поліноміальною повною (NPC) проблемою. Зі збільшенням кількості вкладених рівнів циклу масштаб проблеми зростає дуже швидко (у факторному порядку)

Іншим методом трансформації циклів є метод тайліенгу.

1.3.2 Метод тайліенгу

Метод тайліенгу це добре відомий метод, що нерідко запроваджується компіляторами для оптимізації як послідовних, так і паралельних програм. Для того шоб продукувати високопродуктвний код, що працює на сучасних архітектурах, шляхом збільшення кількості повторного використання даних, а також локальності даних. В основі методу лежить зміна порядку ітерацій циклів, тому доступ до близьких даних відбувається за відносно короткий проміжок часу.

Зміна порядку досягається шляхом додаткового розбиття простору ітерацій на менші блоки, що носять назву тайли. В коді це відображається у вигляді циклів, що додаються до вже існуючих (рисунок 1.2)

Нехай маємо вкладені цикли глибиною n. Метод тайліенгу може перетворити їх на деякі більш глибокі від $(n+1)$ до $2n$ вкладені цикли. Тайліенг одного циклу – це заміна одного циклу парою циклів. Внутрішній має оригінальний крок циклу, а зовнішній має крок рівний

$$step = ub - lb + 1$$

де ub – верхня межа внутрішнього циклу, lb – нижня межа внутрішнього циклу.

Кількість ітерацій внутрішнього циклу, тобто, тайлу, називається розміром тайлу.

В загальному випадку розбиття циклів на тайли можливе тоді і тільки тоді, коли всі вкладені цикли є повністю змінними [5]

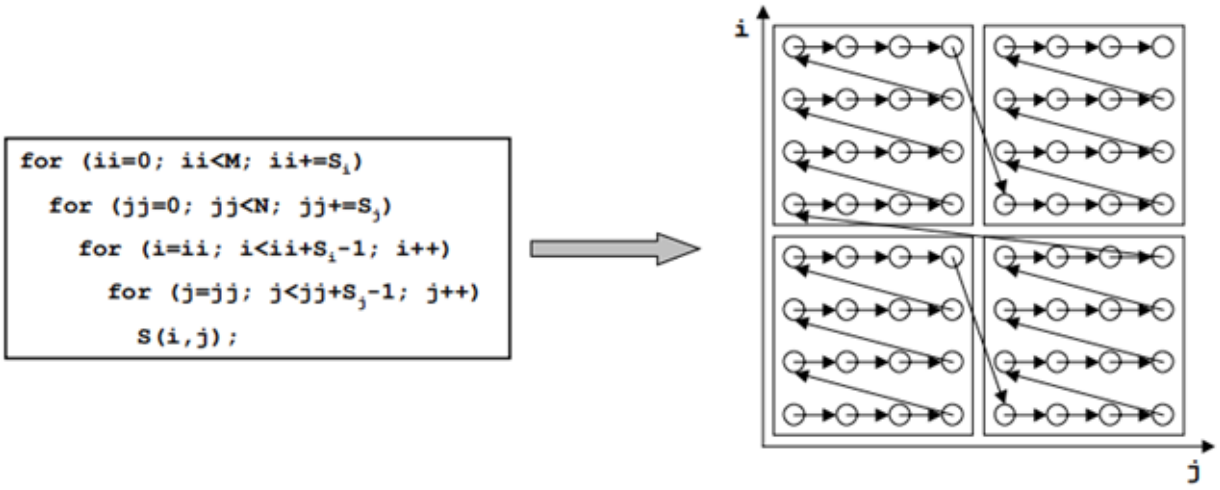


Рисунок 1.2 – Приклад тайліенгу

Дві вище згадані трансформації простору ітерацій, а саме метод паралелізації та метод тайліенгу підходять до вирішення проблеми оптимізації часу виконання програм абсолютно різними способами. Слід зазначити, що ці два способи можна використовувати разом. Оскільки метод тайліенгу передбачає використання більшої кількості потоків, то якщо між тайлами немає залежностей даних, можна очікувати додаткового покращення при сумісному використанні ще методу розпаралелювання.

Метод тайлінгу може бути застосований за допомогою окремих утилітами, що не є частиною компіляторів. Так, в результаті своєї роботи вони генерують перетворений вихідний код з вже трансформованими циклами. На даний момент існує кілька програмних пакетів, які реалізують метод тайліенгу шляхом перетворення вихідного коду.

Розглянемо метод тайліенгу та його параметри більш детально

2 ОПТИМІЗАЦІЇ ЦИКЛІВ ДЛЯ ПАРАЛЕЛЬНИХ ВБУДОВАНИХ СИСТЕМ МЕТОДОМ ТАЙЛІЕНГУ

Тайліенг являє собою трансформацію циклів, що розбиває всі обчислення, тобто простір ітерацій на підмножини менших розмірів, щоб ефективно використовувати ієрархію пам’яті цільової машини. Процедура поділу полягає в наступних етапах.

- Розділенні даних, що підлягають обробці, на блоки
- Виконання для кожного блоку його відповідних обчислень, що ведуть до часткових результатів
- Об'єднання останніх для отримання остаточних глобальних результатів.

В загальному випадку тайліенг використовується для векторизації, паралелізму, а також у багатьох оптимізаціях програм високого рівня, щоб зменшити затримку доступу до пам’яті та збільшити локальність даних.

Тайліенг базується на трьох основних компонентах:

- Фігура, яка визначається обмежуючими гіперплощинами паралелепіпеда або напрямками його охоплюючих векторів
- Форма, яка визначається співвідношенням довжин паралелепіпеда в різноманітних вимірах
- Розмір, який може бути фіксований для всіх розмірів і визначається коефіцієнтом масштабування.

Слід зазначити, що дві характеристики фігура і розмір мають найбільше значення. Насправді, більшість робіт, що стосуються тайлеінгу нічого не згадують про форму. Дослідники часто плутають фігуру з формою [7].

Таким чином тайліенг можна класифікувати за двома основними характеристиками фігура та розмір. Розглянемо їх біль детально.

2.1 Фігури тайлів

Як вже було зазначено вище даний метод розділяє простір ітерацій обчислень циклу на тайли певного розміру і фігури так, що прохід по цим тайлам призводить до обходу всього простору ітерацій. Фігура тайлу випливає з напрямків, що були обрані для розділення простору ітерацій. Такі напрями представлені шляхом розділення гіперплощин на тайли. Огляд статей з приводу оптимізації тайлів показує, що основний акцент зосереджений на теселяційному тайліенгу, атомарним тайлам та фігурам тайлів, такими як, трапеція, прямокутник, шестукутнк, тощо.

Слід зазначити, що визначення фігури тайлу також передбачає врахування обмежень щодо валідності такого розділення, наприклад, відсутність циклічних залежностей. Тож показники швидкодії для різних фігур відрізняються між тестовими програмами. Розглянемо деякі найбільш розповсюджені фігури тайлів.

2.1.1 Прямокутні тайли

Прямокутна фігура означає, що всі тайли мають форму боксів з ребрами паралельними до осей (рисунок 2.1). Дана фігура може розглядатись як різновид методу перестановки циклів.

Така фігура підтримується такими програмними пакетами як LooPo, PPCG, Pluto.

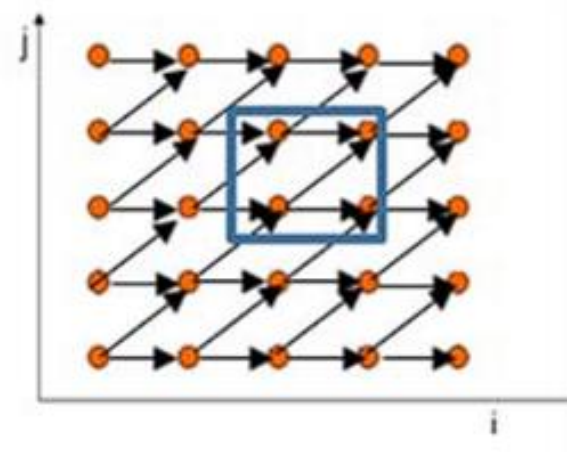


Рисунок 2.1 – Прямокутний тайл

2.1.2 Трапецеєвидні тайли

Серед найефективніших схем розбиття можна виділити алгоритм трапецієвидного розбиття, виконаний компілятором Pochoir.

Фріго і Стрампен [7] запропонували алгоритм, який обмежує асимптотичну частоту промахів кешу, забезпечуючи масштабовану паралельність. В свої роботі вони спирались на трапецієподібні тайли. Однак завдання реалізувати його в компіляторі загального призначення не було досягнуто. З цих причин розробили компілятор Pochoir, заснований на запропонованому методі Фріго і Струмпена з вищим ступенем асимптотичного паралельності.

Пізніше, спираючись на трапецієподібний алгоритм Pochoir, Був запропонуваний підхід до тайліенгу, який зменшує кількість синхронізацій та залежностей. Цей підхід спочатку розкладає простір ітерацій на тайли (рисунок 2.2), щоб на першому етапі побудувати граф залежностей на основі трапецієвидного алгоритму Pochoir. Потім застосовуються оптимізації до отриманого графу, щоб покращити динамічне планування, зберігаючи при цьому всі попередні залежності між тайлами. Зазначимо, що Pochoir запропонував спеціалізовану мову для обчислень.

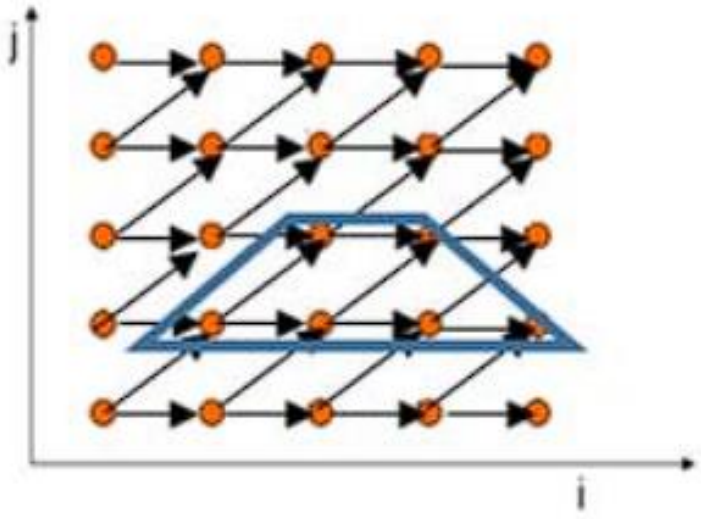


Рисунок 2.2 – трапцієвидний тайл

2.1.3 Hexagonal Tiling

Метод розбиття на шестикутні тайли тісно пов’язаний з розбиттям на тайли у вигляді ромбів, який буде описана нижче, за винятком деяких відмінностей. Так, верхня частина ромбовидного тайлу завжди вузька, тоді як верхня частина шестикутника варіюється по ширині [8].

Шестикутник забезпечує однакову кількість обчисленнь в кожному тайлі. Однак ромбовидна форма може мати варіції. Шестикутний тайл демонструє багаторівневу паралельність так само як і повторне використання даних (рисунок 2.3).

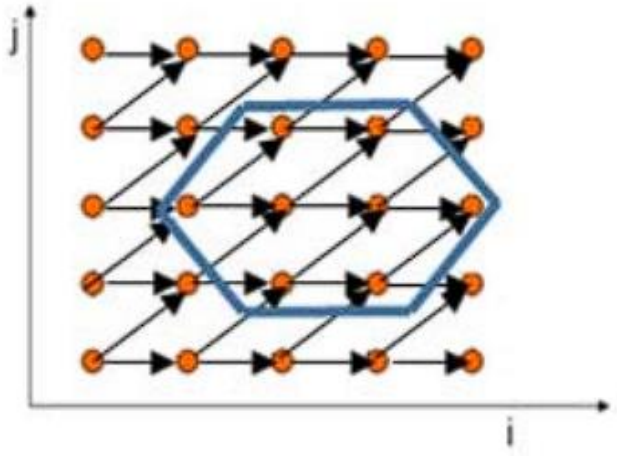


Рисунок 2.3 – шестикутний тайл

2.1.4 Паралелограмовидна фігура

Ця фігура не була практично перевірена через проблеми з реалізацією. Жодна реалізація ніколи не була випущена. У окремих випадках, тайли у вигляді паралелограма можуть бути реалізовані не прив’язуючись до розбиття простору ітерацій. Однак з точки зору генерації коду вони були корисні для бібліотек Omega [9] для створення необхідних примітивів комунікації.

Воннакот [10] працював з паралелевидними тайлами як з паралелепіпедами (рисунок 2.4). Спочатку кожен тайл виконує свій отриманий фрагмент, далі набір ітерацій генерує необхідні значення для іншого тайлу та передає їх далі. Потім обчислюється набір ітерацій, що залишився. В кінці, після отримання вхідних значень, кожен тайл виконує свій отриманий фрагмент. Крім того, Воннакот розглядав можливість розширення паралелограмів на ромби, наприклад, у випадку більших розмірів.

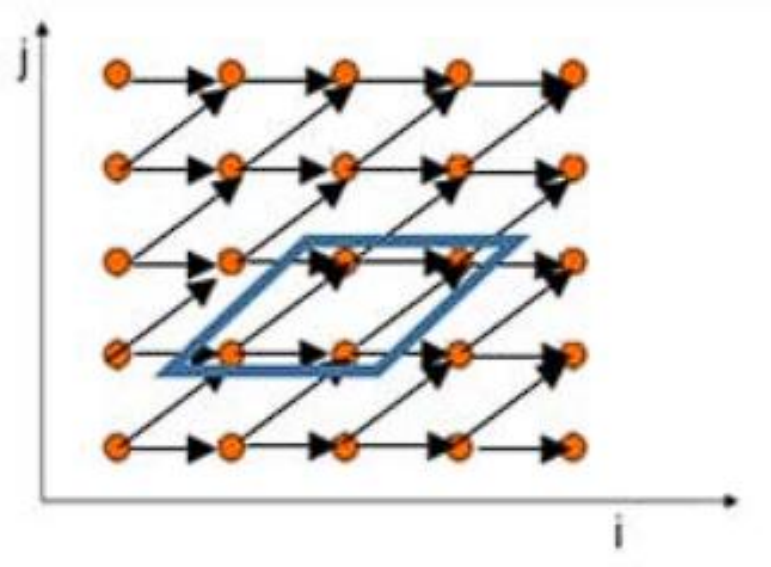


Рисунок 2.4 – паралелограмовидний тайл

2.1.5 Суміш фігур

Огляд робіт по оптимізації фігури тайлу в основному зосереджені на правильних і неподільних фігурах. Одним із методів розбиття, що покращує локальність даних, не перешкоджаючи паралельності між тайлами, є розбиття на змішані фігури, де тайли розкладаються на послідовність різних трапецієподібних обчилювальних тайлів (рисунок 2.5). Іншими словами, такий спосіб можна розглядати як метод з більш ніж однією формою тайлу, тобто він може мати сусідні тайли різної форми, такі як паралелепіпеди, трикутні та трапецієподібні.

Підхід до компіляції в даному випадку виконує спочатку розподіл обчислень, а потім розподіл даних. Зокрема, тайліенг неатомарними тайлами призводить до досягнення збалансованого розкладу при паралелізації методом планування. Щоб досягти збалансованого розкладу, в роботі [11] була запропонована технологія розщеплення тайлів, яка подібна до спряжено-трапецієподібного тайлу. Так, кожен тайл поділений на підтайли. Таким чином, спочатку обробляється кожен підтайл, який не містить залежностей, а потім підрегіон, який містить їх.

Отже даний спосіб розбиття –це схема з подільними фрагментами, такими, що комунікація між тайлами відбувається одночасними з обчисленням підтайлів усередині тайлу.

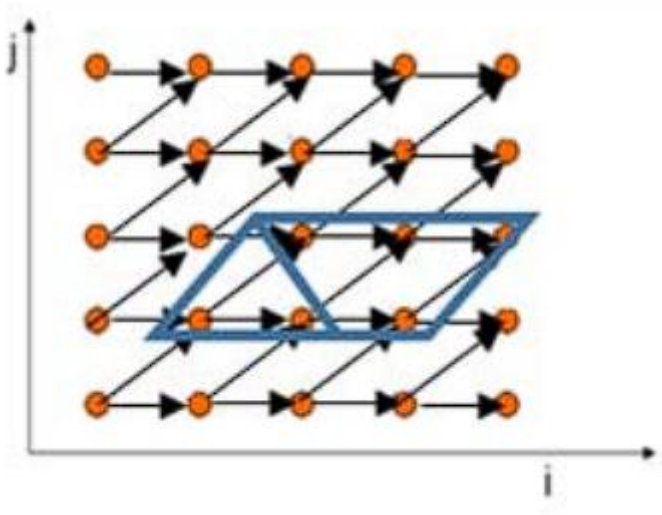


Рисунок 2.5 – розбиття на підтайли

2.1.6 Розбиття з пересіченнями

Орієнтуючись на трафаретні обчислення, кожне ядро в рамках даного методу може виконати більше однієї операції, перш ніж прокомунікувати з іншими ядрами. На рисунку 2.6 зображено два окремі тайли: трапецієвидну (сірим кольором) і паралелограмну (синім кольором). Зазначимо, що невелика сіра та синя область є спільною для обох тайлів і, таким чином, являє собою надлишкове обчислення. Це накладання означає, що всі плитки вздовж кожної спільної області можуть виконуватися паралельно, при цьому покращуючи тимчасову локальність даних. Даний метод гарантує, що всі тайли можуть виконуватися паралельно. Єдина проблема, пов’язана з перекриттям, полягає у великій кількості створених надлишкових обчислень.

Таким чином, стає необхідним компроміс між паралельною масштабованістю та часом виконання [12]

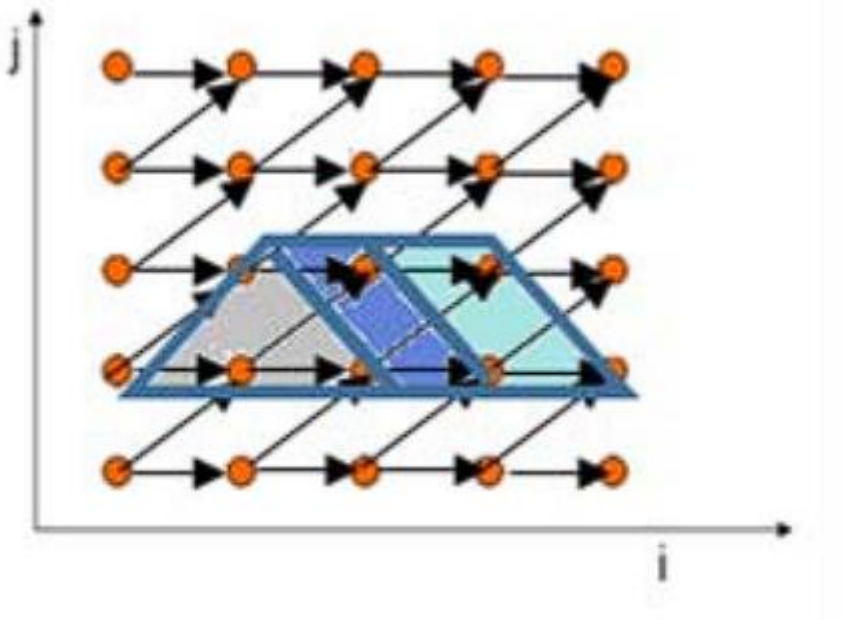


Рисунок 2.6 – розбиття на тайли з перекриттями

Метод ієрархічного перекриття тайлів, який є нещодавнім розширенням оригінальної ідеї, що лежить в основі розбиття з перекриттям, спрямований на зменшення накладних витрат на комунікацію шляхом введення надлишкових обчислень [12]. Він використовує переваги ієрархії обладнання та намагається збалансувати витрати на зв'язок і зайві обчислення, усуваючи міжтайлові комунікації. Фактично, оскільки кількість надлишкових обчислень має бути меншою, ніж накладні витрати на синхронізацію, даний метод покладається на ефективність каналів зв'язку відповідно до цільової платформи, наприклад, глобальної пам'яті, шини та спільного кешу. Таким чином, ефективність нового методу стає чутливою до апаратного забезпечення, на якому виконується програма.

2.1.7 Розбиття на ромби

Серед останніх методів розбиття на тайли, можна відмітити ромб (рисунок 2.7). Справді, він забезпечує як одночасний запуск, так і ідеальний баланс навантаження за достатніх умов для конкретних програм (трафаретних) з афінним доступом до даних.

Зауважте, що тайли у вигляді ромбів можуть перевершити налаштований генератор трафаретних кодів для певної області [13]. Такі методи важко включити до технік оптимізації компіляторів загального призначення через потребу міжпроцедурного аналізу покажчиків і масивів, на додаток до необхідності налаштування стратегії планування та параметрів розміру тайлу для кожної пари обчислення трафарету та машини. Розбиття на ромби дозволяє одночасний запуск без зайвих обчислень, використовуючи багатовимірний паралелепіпед як форму тайлу, на відміну від попередніх підходів, де обрана форма тайлу не випливає безпосередньо з аналізу залежностей даних.

Розбиття на ромби дозволяє вибрати розміри тайлу вздовж інших вимів, щоб отримати бажану ширину вздовж найбільш швидко змінного виміру

простору. Тому даний метод намагається отримати вигоду від ефективності попередньої апаратної вибірки та векторізації.

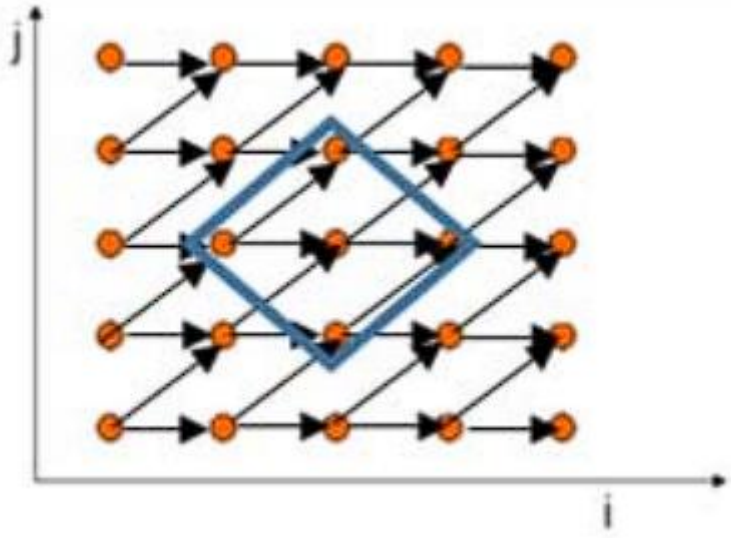


Рисунок 2.7 – ромбовидний тайл

2.2 . Розміри тайлів

Важливою характеристикою тайліенгу є розмір тайлу. На нього може безпосередньо впливати необхідний розмір локальної пам’яті, наприклад, розмір кеш-пам'яті. Насправді, оптимальна розбивка для ієрархічної пам’яті вимагає таких тайлів, щоб усі дані для однієї плитки вписувалися в найвищий рівень пам’яті та демонстрували багаторазове повторне використання даних, таким чином зменшуючи загальний трафік пам’яті. Розрізняємо два основних типи тайліенгу, які відрізняються з точки зору розміру тайлу

2.2.1 Статичний тайліенг

Кілька робіт присвячено генерації послідовного та паралельного коду з фіксованими розмірами тайлів. Коли розміри тайлів фіксовані, простір ітерацій тайлами можна розглядати як багатогранник, визначений набором лінійних обмежень. Таким чином, будуть згенеровані розбиті на тайли вкладені цикли.

У цьому контексті автор роботи [14] запропонував перший класичний підхід для генерування точного розбиття на тайли, коли розміри плитки постійні.

Ефективність, що обумовлена точністю розбиття циклів, залежить від вартості фіксування розмірів тайлу під час генерації. Зазначимо, що використовуючи фіксовані розміри тайлів можна отримати певну деградацію генерації циклів, коли існує залежність на всьому ітераційного простору циклів. Було доведено, що час генерації циклу може зростати експоненціально. Тому вони запропонували розкласти задачу генерації циклу на дві підзадачі, тобто генерувати цикли розбиті на тайли та звичайні. Такий підхід веде до масштабованого рішення для тайліенгу з фіксованим розміром при чому зменшивши кількості обмежень [14].

Слід зазначити, що якщо розміри тайлів не фіксовані, вони можуть створювати нелінійні обмеження. Таким чином, необхідна параметрична стратегія, особливо при використанні багатогранних компіляторів. Кілька досліджень, освітили параметризований тайліенг описані нижче.

2.2.2 Параметризовний тайліенг

Основна відмінність методу параметризованого тайліенгу від статичного полягає в тому, що розміри тайлів не фіксуються під час компіляції, а залишаються символічними параметрами. Такий підхід є досить корисними для ітеративних компіляторів коли вони оптимізують наукові програми, а саме математичні обчислення. Так, параметризований тайліенг дозволяє адаптувати програму під час виконання, особливо при ітераційній компіляції. Генерація коду методом параметризованого тайліенгу для загального випадку опуклих ітераційних просторів із прямокутними тайлами була вирішена за допомогою підходів Фур’є-Моцкіна [14]

З одного боку, існують різноманітні роботи щодо генерації тайлів, які зосереджені на параметризованому розбитті послідовних програм, але в у випадку

паралельних програм в більшості роботах в якості розмірів тайлів використовувались константи, що були відомі ще на етапі компіляції.

Також був запропонований систематичний метод, який керує декількома трансформаціями циклів за допомогою вирішення лінійних нерівностей. Також представив вирішення основної проблеми попередніх робіт - генерацію паралельного варіанту параметризованого тайліенгу.

З іншого боку, в літературі використовується багато добре відомих інструментів тайліенгу, що включають трансформацію з вихідного коду у вихідний код. Деякі з них підтримують параметризований тайліенг, наприклад, PPCG, PrimeTile, HITLOG та TLO. Слід зазначити, що TLOG був першим інструментом, який генерує параметризовані тайли для ідеально вкладених циклів з афінними межами циклу, а потім покращений HITLOG TLOG для обробки багаторівневих циклid.

Метод параметричного багатогранного аналізу використовується для повторного використання даних, що використовуються суміжними тайлами і математичну модель, щоб обґрунтувати деякі наближення доступу та передачі даних для читання та запису. Наприклад, аналіз працездатності який може оцінити розмір локальних буферів зберігання та метод їх відображення. Крім того, таке повторне використання даних між тайлами може бути виражено в наближеному параметричному вигляді.

Цей метод дозволяє досягти ефективності генерації коду як для статичних, так і для параметризованих розмірів тайлу. Він заснований на трьох компонентах, які добре показують різницю повних і часткових тайлів. Це метод для створення тайлів з відповідного абстрактного синтаксичного дерева та є характеристикою параметризованого розбиття циклів.

Підсумовуючи, усі ці роботи можна побачити як прогрес від генерації коду для багатогранних просторів ітерацій до генерації розбитих на постійного розміру тайли цикли, а потім до ефективного генерування параметризованих тайлів [15].

Так утворилась низка програмних пакетів, що повністю або частково реалізують вище згадані методи тайліенгу.

2.3 Програмні пакети для генерації коду

Розглянемо деякі добре відомі фреймворки для генерації коду, що використовують тайліенг.

2.3.1 Pluto

Pluto [16] є поліедральним компілятором, який перетворює програми на мові програмування C по типу код-код для крупнозернистої паралельності та локалізації даних. Разом з тим може генерувати паралельні програми на мові C з використанням пакету OpenMP. В даному програмному пакеті використовується алгоритм планування, який намагається знайти афінні перетворення, що дозволяють найефективніше розбивати цикли на тайли.

Іншими словами, Pluto намагається створити множину вкладених циклів, де кілька послідовних циклів можуть бути поміняні місцями. Основна мета – мінімізувати кількість комунікацій між окремими тайлами і забезпечити паралельність між ними.

Планування в Pluto добре оптимізовано. Так, він зменшує кількість залежностей, що виходять за межі конкретного тайлу.

Нещодавно Pluto почав підтримувати ромбовидне розбиття з одночасним запуском на виконання тайлів лише в одному вимірі, тобто це частковий паралельний старт. Використовуючи гіперплощини ромбів тільки для формування тайлів і використовуючи гіперплощини планування існуючого алгоритму Pluto для сканування точок всередині тайлів, можна отримати багато від тайліенгу.

2.3.2 LooPo

LooPo [6] є поліедральним напівавтоматичним компілятором типу код-код, який інтегрує різні методи та інструменти для аналізу заданих багаторівневих вкладених циклів. Паралелізація спирається в основному на цілочислене лінійне програмуванні для того щоб створити граф задач, у якому групи незалежних задач розташовані послідовно.

2.3.3 Par4All

Par4All [16] є компілятором типу код-код для послідовних програм на мовах програмування C і Fortran, які фінансуються за рахунок HPC (High Performance Computing). Дане рішення фактично інтегрує різноманітні фреймворки в автоматичний компілятор що здатен паралелизувати, оптимізувати та перетворювати існуючі програми для таргетування їх на різні архітектури. Він генерує ефективний цільовий код з точки зору ефективності та енергоспоживання для багатоядерних процесорів і високопродуктивних платформ.

В дійсності, Par4All являє собою обєднанням таких бібліотек як C-to-OpenMP і OpenMP-to-CUDA. C-to-OpenMP – програмний пакет, що дозволяє автоматично трансформувати код на мові програмування C так, щоб він використовував бібліотеку для паралельних обрахунків OpenMP. OpenMP-to-CUDA – дозволяє трансформувати код з використанням OpenMP в код, близький до мови C, але він буде виконуватись паралельно на відеокарті, в той час як OpenMP використовує лише можливості процесора.

2.3.4 PPCG

PPCG [17] є поліедричним компілятором типу код-код, який генерує код OpenCL або Cuda GPU з послідовних програм. Він включає в себе основні функції генерації коду Cuda, а саме вилучення послідовностей вкладених циклів, які

піддаються розпаралелюванню, алгоритм планування, багато евристик розбиття циклів і безпосередньо сам генератор коду Cuda.

2.3.5 PTile

PTile – це фреймворк часу компіляції для тайліенгу афінних вкладених циклів. В той же час, розміри тайлів обробляються вже під час виконання [18].

Даний фреймворк став першим ефективним інструментом, який генерує код з паралельних вкладених циклів для пааметризованих паралельних тайлів.

PTile дозволяє не тільки створювати паралельний код із параметризованими розмірами тайлів для ідеально вкладених циклів, але й для неідеально також.

2.3.6 PrimeTile

PrimeTile [19] — це фреймворк, який генерує багаторівневий тайліенг довільних неідеально вкладених циклів.

Даний фреймворк було реалізовано з можливістю керувати глибиною рекурсії тайліенгу для граничних тайлів. Однак PrimeTile має обмеження у вигляді лише послідовного тайліенгу. PrimeTile реалізовано в Python, і, отже, працює повільніше, ніж написані на мові програмування C у Pluto та HiTLOG.

2.3.7 DynTile

DynTile [20] була першою реалізованою системою для паралельного виконання параметризованого афінного коду розбитого на тайли. Вона обробляє паралелізм хвильового фронту в просторі ітерацій вхідної програми. Потім вона генерує код інспектора для сканування кількох тайлів, які будуть динамічно заплановані для збільшення кількості хвильових фронтів для паралельного виконання.

2.4 Класифікація генераторів коду

На основі раніше вивчених характеристик тайлів таких як вигляд та розмір, проведемо класифікацію найбільш відомих засобів генерації коду, що були згадані в попередньому розділі.

Для цього була створена порівняльна таблиця, з описом цих компіляторів відповідно до цікавих характеристик. З таблиці 2.1 можна зробити наступні висновки.

З одного боку, PPCG є найкращим генератором поліедрального коду для графічних процесорів завдяки ефективності коду Cuda. Слід зазначити, що існує кілька компіляторів, здатних генерувати ефективний код тайліенгу із параметризованими тайлами, наприклад. PrimeTile , PTile і DynTile.

З іншого боку, Pluto є найкращим генератором поліедрального коду для архітектур зі спільною пам’яттю, оскільки він забезпечує паралельність і оптимізацію кешу.

Таблиця 2.1 – порівняння програмних пакетів

Генератор / Тайли	luto	PPCG	LooPo	Par4All	tile	PrimeTile	DynTile
Прямокутник		+	+	+			+
Трапеція		+	+	+		+	+
Перекриті		+					
Шестикутник		+					
Змішана		+					
Ромбовидна		+					

Зафіксований			+	+			
Параметр		+				+	+
Паралелізм		+	+	+			+
Кеш				+		+	

Крім того, Pluto заснований на дуже ефективних техніках тайліенгу, наприклад. ромбовидний тип. Однак обмеження Pluto полягає в тому, що розміри тайлів можуть бути лише постійними і вказаними як вхідні дані для системи трансформації. Оскільки Pluto генерує код лише для фіксованих розмірів тайлів, такі розміри призначаються вручну перед компіляцією, що підкреслює основне обмеження Pluto.

3 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ВПЛИВУ РОЗМІРУ ТАЙЛУ НА ЧАС ВИКОНАННЯ КОДУ

Розглянемо приклад з трьома вкладеними циклами (рисунок 3.1), що ітеруються по трьом масивам. Даний фрагмент коду написаний без будь яких оптимізацій.

```
for (i=1; i<NMAX; i++) {
  for (j=0; j<NMAX; j++) {
    for (k=0; k<i; k++) {
      b[j][k] += a[i][k] * b[j][i];
    }
  }
}
```

Рисунок 3.1 – Вкладені цикли

Метод тайліенгу (рисунок 3.2) покращує час виконання фрагменту коду, а також покращує роботу з пам’яттю.

```
for (t1=0;t1<=floord(NMAX-1,32);t1++) {
  for (t2=0;t2<=floord(NMAX-1,32);t2++) {
    for (t3=0;t3<=min(floord(NMAX-2,32),t2);t3++) {
      for (t4=32*t1;t4<=min(NMAX-1,32*t1+31);t4++) {
        for (t5=max(32*t2,32*t3+1);t5<=min(NMAX-1,32*t2+31);t5++) {
          for (t6=32*t3;t6<=min(32*t3+31,t5-1);t6++) {
            b[t4][t6] += a[t5][t6] * b[t4][t5];;
          }
        }
      }
    }
  }
}
```

Рисунок 3.2 – Результат роботи методу тайліенгу

Були зроблені відповідні заміри часу виконання фрагментів коду та порівнянні між собою. Для того щоб оцінити відносні зміни в швидкодії, час за

який виконується фрагмент коду без використання оптимізацій був прийнятий за 100%. Тоді відносний час виконання оптимізованого коду обраховується за наступною формулою:

$$T = \frac{T_{tiled}}{T_{orignal}} * 100\% \quad (3.1)$$

де T – відносний час виконання оптимізованого коду, T_{tiled} – час виконання оптимізованого коду, $T_{original}$ – час виконання оригінального фрагменту коду.

Як результат (рисунок 3.3) можна бачити значний приріст у швидкодії.

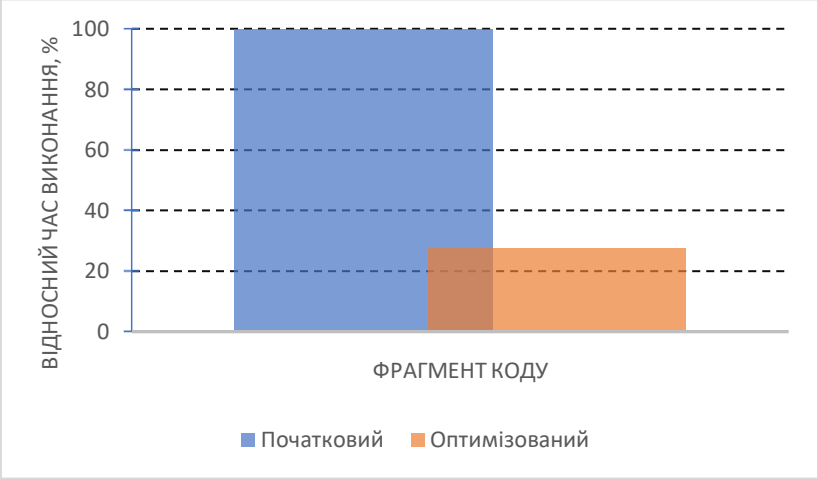


Рисунок 3.3 – Приріст швидкодії оптимізованого коду

3.1 Вплив розміру тайла на час виконання програми

Для дослідження того як впливає розмір тайлу на час виконання програми були взяті три алгоритми різних типів з набору PolyBench, а саме jacobi-1d, trisolve, symm.

3.2 Jacobi-1D

На рисунку 3.4 зображено алгоритм jacobi-1d, з пакету PolyBench без оптимізацій.

```
for (t = 0; t < _PB_TSTEPS; t++)
  {
    for (i = 1; i < _PB_N - 1; i++)
      B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
    for (i = 1; i < _PB_N - 1; i++)
      A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
  }
```

Рисунок 3.4 – фрагмент коду з пакету Pluto, jacobi-1d

На рисунку 3.5 зображено згенерований код після застосування методу тайліенгу.

Для отримання залежності часу виконання від розміру тайла була розроблена програма, що перебирає усі розміри тайлів від 1 до 100 по обом осям. Для кожного такого варіанту знову генерується код, компілюється, запускається та замірюється час виконання. Час кожного запуску зберігається та заноситься до таблиці.

На основі зібраних даних був побудований графік залежності часу від розміру тайлів, що зображений на рисунку 3.6

```
/* Start of CLooG code */
if ((N >= 4) && (T >= 1)) {
  for (t1=0;t1<=floord(T-1,32);t1++) {
    for (t2=2*t1;t2<=min(floord(2*T+N-3,32),floord(64*t1+N+61,32));t2++) {
      if (t1 <= floord(32*t2-N+1,64)) {
        if ((N+1)%2 == 0) {
          a[(N-2)] = b[(N-2)];;
        }
      }
      if (N == 4) {
        for (t3=max(32*t1,16*t2-1);t3<=min(min(T-1,32*t1+31),16*t2+14);t3++) {
          b[2] = 0.33333 * (a[2 -1] + a[2] + a[2 + 1]);;
          a[2] = b[2];;
        }
      }
      for (t3=max(ceild(32*t2-N+2,2),32*t1);t3<=min(min(min(floord(32*t2-N+32,2),T-1),32*t1+31),16*t2-2);t3++) {
        for (t4=32*t2;t4<=2*t3+N-2;t4++) {
          b[(-2*t3+t4)] = 0.33333 * (a[(-2*t3+t4)-1] + a[(-2*t3+t4)] + a[(-2*t3+t4) + 1]);;
          a[(-2*t3+t4-1)] = b[(-2*t3+t4-1)];;
        }
        a[(N-2)] = b[(N-2)];;
      }
      for (t3=max(ceild(32*t2-N+33,2),32*t1);t3<=min(min(T-1,32*t1+31),16*t2-2);t3++) {
        for (t4=32*t2;t4<=32*t2+31;t4++) {
          b[(-2*t3+t4)] = 0.33333 * (a[(-2*t3+t4)-1] + a[(-2*t3+t4)] + a[(-2*t3+t4) + 1]);;
          a[(-2*t3+t4-1)] = b[(-2*t3+t4-1)];;
        }
      }
      if (N >= 5) {
        for (t3=max(32*t1,16*t2-1);t3<=min(min(floord(32*t2-N+32,2),T-1),32*t1+31);t3++) {
          b[2] = 0.33333 * (a[2 -1] + a[2] + a[2 + 1]);;
          for (t4=2*t3+3;t4<=2*t3+N-2;t4++) {
            b[(-2*t3+t4)] = 0.33333 * (a[(-2*t3+t4)-1] + a[(-2*t3+t4)] + a[(-2*t3+t4) + 1]);;
            a[(-2*t3+t4-1)] = b[(-2*t3+t4-1)];;
          }
          a[(N-2)] = b[(N-2)];;
        }
      }
      for (t3=max(max(ceild(32*t2-N+33,2),32*t1),16*t2-1);t3<=min(min(T-1,32*t1+31),16*t2+14);t3++) {
        b[2] = 0.33333 * (a[2 -1] + a[2] + a[2 + 1]);;
        for (t4=2*t3+3;t4<=32*t2+31;t4++) {
          b[(-2*t3+t4)] = 0.33333 * (a[(-2*t3+t4)-1] + a[(-2*t3+t4)] + a[(-2*t3+t4) + 1]);;
          a[(-2*t3+t4-1)] = b[(-2*t3+t4-1)];;
        }
```

Рисунок 3.5 – jacobi-1d після розбиття на тайли без оптимізації розміру

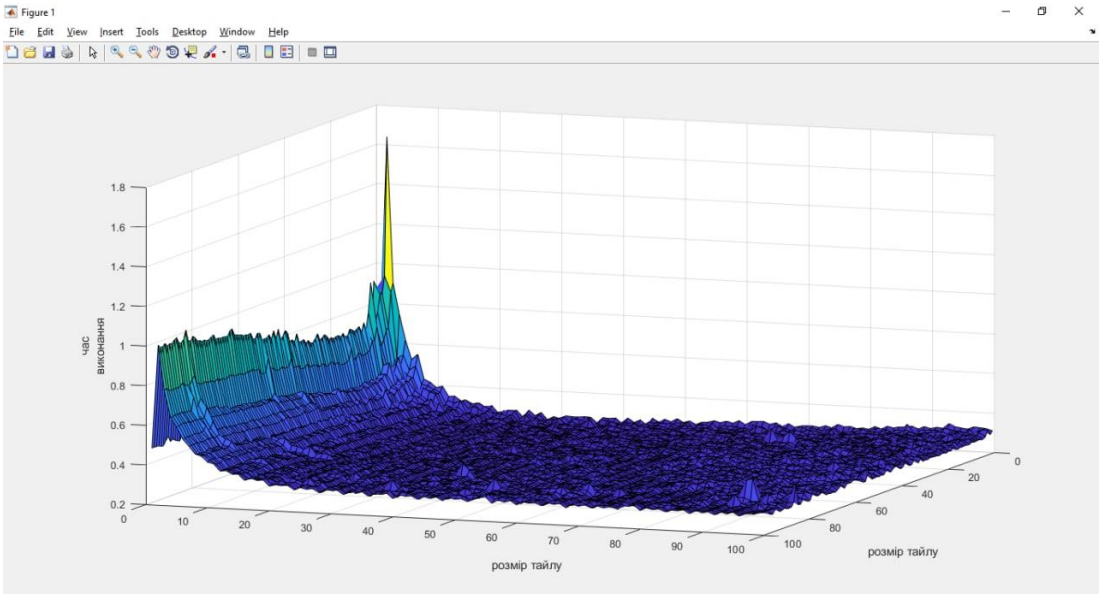


Рисунок 3.6 – залежність часу виконання програми від розміру тайлу для алгоритму Jacobi-1D

3.3 Trisolv

На рисунку 3.7 зображено алгоритм trisolv з пакету PolyBench без оптимізацій.

```
for (i=0;i<=N-1;i++) {
    for (j=0;j<=N-1;j++) {
        for (k=0;k<=j-1;k++) {
            B[j][i]=B[j][i]-L[j][k]*B[k][i];

        }
        B[j][i]=B[j][i]/L[j][j]; // S2 ;
    } // for j
} // for i
```

Рисунок 3.7 – фрагмент коду з пакету Pluto, trisolv

На рисунку 3.8 зображено згенерований код після застосування методу тайліенгу.

Для отримання залежності часу виконання від розміру тайла була використана програма, що була описана вище. З її допомогою були зібрані дані часу виконання для кожного розміру тайлу з діапазону від 1 до 50. На основі зібраних даних був побудований графік залежності часу від розміру тайлів, що зображений на рисунку 3.9

```
if (N >= 1) {
  for (t1=0;t1<=floord(N-1,32);t1++) {
    for (t2=0;t2<=floord(N-1,32);t2++) {
      for (t3=0;t3<=t2;t3++) {
        if (t2 >= t3+1) {
          for (t4=32*t2;t4<=min(N-1,32*t2+31);t4++) {
            for (t5=32*t3;t5<=32*t3+31;t5++) {
              lbv=32*t1;
              ubv=min(N-1,32*t1+31);
              for (t6=lbv;t6<=ubv;t6++) {
                B[t4][t6]=B[t4][t6]-L[t4][t5]*B[t5][t6];;
              }
            }
          }
        }
        if (t2 == t3) {
          lbv=32*t1;
          ubv=min(N-1,32*t1+31);
          for (t6=lbv;t6<=ubv;t6++) {
            B[32*t2][t6]=B[32*t2][t6]/L[32*t2][32*t2];;
          }
        }
        if (t2 == t3) {
          for (t4=32*t2+1;t4<=min(N-1,32*t2+31);t4++) {
            for (t5=32*t2;t5<=t4-1;t5++) {
              lbv=32*t1;
              ubv=min(N-1,32*t1+31);
              for (t6=lbv;t6<=ubv;t6++) {
                B[t4][t6]=B[t4][t6]-L[t4][t5]*B[t5][t6];;
              }
            }
            lbv=32*t1;
            ubv=min(N-1,32*t1+31);
            for (t6=lbv;t6<=ubv;t6++) {
              B[t4][t6]=B[t4][t6]/L[t4][t4];;
            }
          }
        }
      }
    }
  }
}
```

Рисунок 3.8 – trisolv після розбиття на тайли без оптимізації розміру

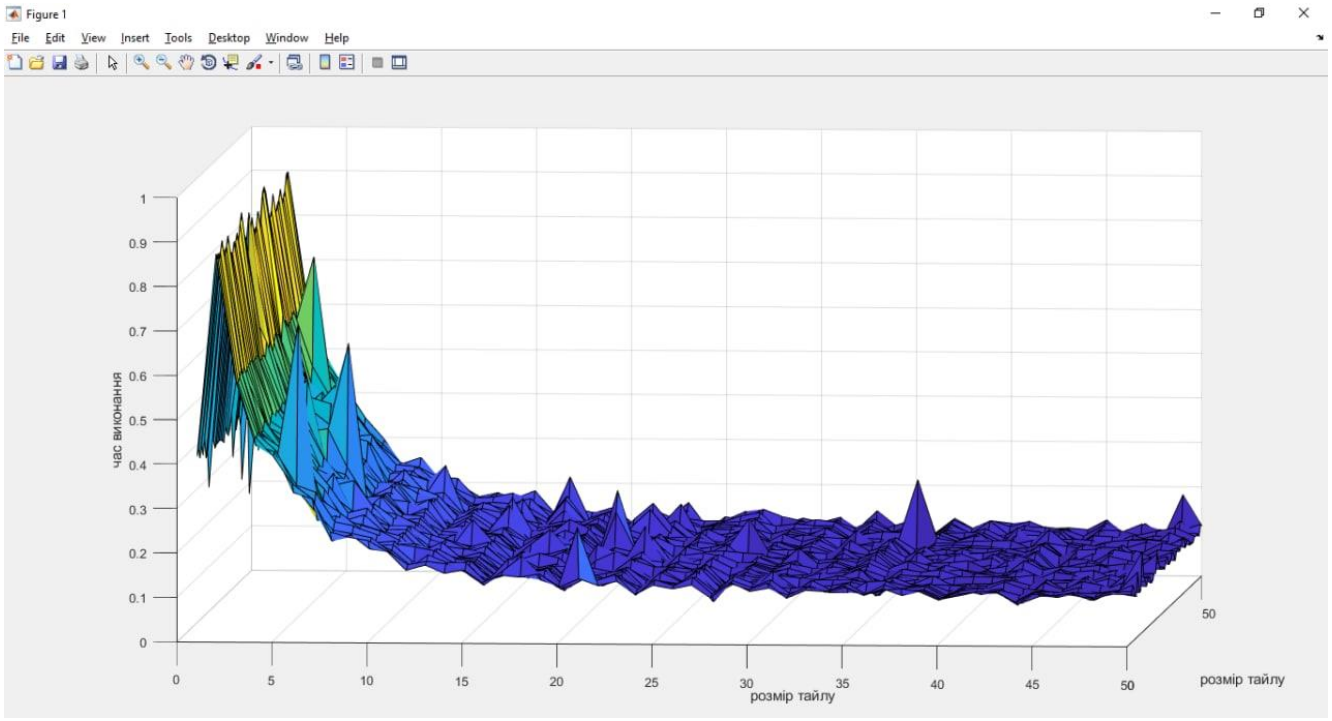


Рисунок 3.9 – залежність часу виконання програми від розміру тайлу для алгоритму Trsolv

3.4 Symm

На рисунку 3.10 зображено алгоритм jacobi-1d, з пакету PolyBench без оптимізацій.

```
for (i=0; i<NMAX; i++) {
  for (j=0; j<NMAX; j++) {
    for (k=0; k<j-1; k++) {
      c[i][k] += a[j][k] * b[i][j];
      c[i][j] += a[j][j] * b[i][j];
    }
    c[i][j] += a[j][j] * b[i][j];
  }
}
```

Рисунок 3.10 – ssymm

На рисунку 3.11 зображено згенерований код після застосування методу тайліенгу.

```
for (t2=0;t2<=floord(NMAX-1,32);t2++) {
  for (t3=0;t3<=floord(NMAX-1,32);t3++) {
    for (t4=0;t4<=min(floord(NMAX-3,32),t3);t4++) {
      for (t5=32*t2;t5<=min(NMAX-1,32*t2+31);t5++) {
        for (t6=max(32*t3,32*t4+2);t6<=min(NMAX-1,32*t3+31);t6++) {
          for (t7=32*t4;t7<=min(32*t4+31,t6-2);t7++) {
            c[t5][t6] += a[t6][t6] * b[t5][t6];;
          }
        }
      }
    }
  }
}
for (t2=0;t2<=floord(NMAX-1,32);t2++) {
  for (t3=0;t3<=floord(NMAX-1,32);t3++) {
    for (t4=32*t2;t4<=min(NMAX-1,32*t2+31);t4++) {
      lbv=32*t3;
      ubv=min(NMAX-1,32*t3+31);
      for (t5=lbv;t5<=ubv;t5++) {
        c[t4][t5] += a[t5][t5] * b[t4][t5];;
      }
    }
  }
}
for (t2=0;t2<=floord(NMAX-1,32);t2++) {
  for (t3=0;t3<=floord(NMAX-3,32);t3++) {
    for (t4=t3;t4<=floord(NMAX-1,32);t4++) {
      for (t5=32*t2;t5<=min(NMAX-1,32*t2+31);t5++) {
        for (t6=max(32*t4,32*t3+2);t6<=min(NMAX-1,32*t4+31);t6++) {
          lbv=32*t3;
          ubv=min(32*t3+31,t6-2);
          for (t7=lbv;t7<=ubv;t7++) {
            c[t5][t7] += a[t6][t7] * b[t5][t6];;
          }
        }
      }
    }
  }
}
```

Рисунок 3.11 – ssymm після розбиття на тайли без оптимізації розміру

Для отримання залежності часу виконання від розміру тайла була використана програма, що була описана вище. З її допомогою були зібрані дані часу виконання для кожного розміру тайлу з діапазону від 1 до 50. На основі зібраних даних був побудований графік залежності часу від розміру тайлів, що зображений на рисунку 3.12

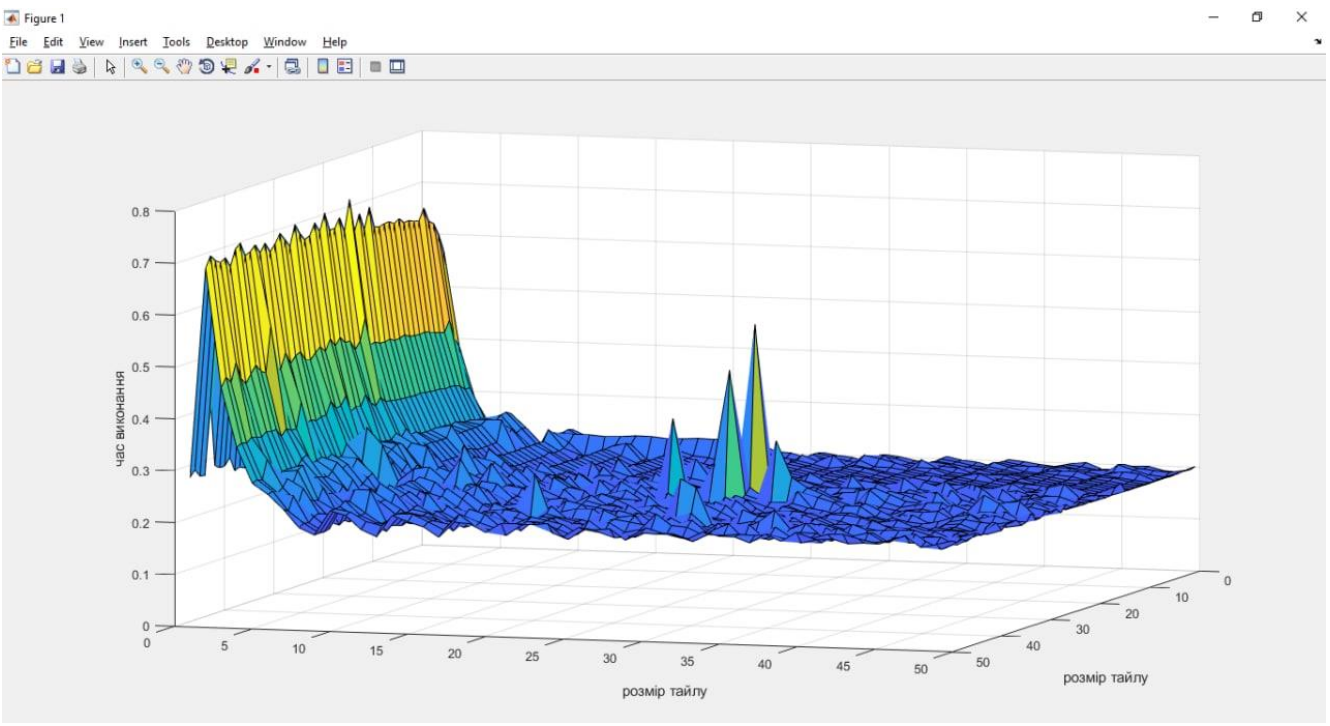


Рисунок 3.12 – залежність часу виконання програми від розміру тайлу для алгоритму ssymm

3.5 Висновки

З огляду на графіки вище можна бачити, що час виконання дуже варіюється від розміру тайлів. З чого можна зробити висновок про непередбачуваність та відсутність будь-якого відомого розподілу.

Таким чином було явно продемонстровано NP повноту прблеми. Отже, логічно буде вірушувати її за допомогою одного з еволюційних алгоритмів

4 ЕВОЛЮЦІЙНІ АЛГОРИТМИ

Методи засновані на популяції — це ітераційна техніка, що обробляє популяцію осіб і змушує їх еволюціонувати спираючись на деякі правила що повинні бути чітко прописані. На кожній ітерації, етапи само-адаптації змінюють етапи кооперації.

Самоадаптація означає, що індивіди розвиваються незалежно, а співпраця передбачає обмін інформацією між особами. Наступна схема (рисунок 4.1) підсумовує представлений вище опис.

Багато різних алгоритмів може бути описано за представленою вище схемою. Наприклад, операторі відбору та схрещування генетичних алгоритмів можуть бути розглянуті як операціїї кооперації, в той час як оператор мутації може бути як частина процесу самоадаптації.

Для порівняння різний еволюційних алгоритмів, потрібно визначити основні характеристики, що описуютть кожний еволюційний алгоритм окремо.

Для того щоб проіліструвати різні форми які можуть приймати ці характеристики, розглянемо деякі алгоритми, що були запропоновані в якості вирішення проблеми k-кольорів та проблему коммівояжера.

Задача k-кольорів заключається в тому, щоб зафарбувати вершини графу з заданою кількістю k кольорів так, щоб два сусудніх вузли завжди мали різний колір.

Задача коммівояжера заключається в тому, визначити маршрут за яким коммівояжеру потрібно обійти всі вершини у найбілльш ефективний спосіб.

```
Згенерувати початкову популяцію осіб
ПОКИ не викоані умови зупинки ВИКОНУВАТИ
    кооперація;
    самоадаптація;
КІНЕЦЬ
```

Рисунок 4.1 – Загальний алгоритм усіх методів заснованих на популяції

4.1 Опис осіб

Поки техніки локального пошуку намагаються покращувати одна рішення, еволюційні алгоритми обробляють популяцію осіб. Ці особи не обов’язково є рішенням поставленої задачі. Вони можуть бути частиною рішення. Більш того, вони можуть представляти будь-який тип об’єкту з певною властивістю, а потім бути трансформованими в рішення за певною наперед розробленою процедурою.

Для ілюстрації концепту розглянемо спершу проблему комівояжера. Більшість генетичних алгоритмів що були представлені в якості рішення даної задачі визначають осіб як рішення проблеми, вірогідно непридатними. Однак, в роботі [21] запропонували алгоритм, в якому особи були маршрутами. Спеціальні процедури були розроблені для того щоб створювати придатні рішення комбінуючи різні маршрути.

Другим прикладом буде марувїна система, що описана в роботі [22] для задачі k-кольорів. Особи визначені як зафарбовані мурахи що рухаються по ребрам графу, а процедура для генерації зафарбованого графу розроблена на основі розташування мурах на графі.

В той час як особи які можуть бути окремими частинами рішення, які ще потрібно зібрати для того щоб сформувати повне рішення, в деяких випадках особи також можуть представляти цілу множину рішень.

Для прикладу візьмемо острівкові генетичні алгоритми, в яких рішення згруповані разом і формують острови рішень.

Під час етапу кооперації такого алгоритму, кожен острів розкриває всю інформацію, що містить сам. Рішення можуть переходити з одного острову до іншого.

Під час періоду само адаптації кожен острів модифікується окремо

4.2 Процес оцінки

Популяція еволюційного алгоритму ітераційно еволюціонує спираючись на деякі конкретні правила. На кожній ітерації нові особи створюються і потрібно визначити які увійдуть до популяції. Більш того, деякі особи повинні бути відкинуті з популяції задля уникнення вимирання популяції.

Від однієї популяції до іншої популяції може бути абсолютно змінена у тому сенсі, що тільки нові особини залишаться в популяції. З іншого боку стабільна Еволюція означає, що тільки частина популяції може бути змінена між двома наступними та ітераціями.

Більшість еволюційних алгоритмів працюють з популяціями фіксованого розміру. Однак розмір популяції може варіюватися протягом процесу пошуку. Розглянемо наступний приклад в якому популяція містить старі особи так само як і щойно згенеровані. Рішення які особи з популяції будуть залишені а які ні може бути прийнято випадково.

Такі рішення на основі Імовірностей продукують непередбачуваний результат, тому Можна вирішити залишити особину в популяції якщо її значення яке вимірюється деякою Функцією Достатньо близьке до найкращого значення в популяції.

Важливо також зазначити що генетичні алгоритми паралельні. Це звичайно пояснює чому реалізації генетичних алгоритмів на паралельних комп’ютерах так добре дослідження. Паралельне програмування дозволяє асинхронну еволюція. В такому випадку кожна особина неприривно модифікується без перевірки чи зміни вже були зроблені на інших особах.

4.3. Структура сусідства

Дві особини не обовязково можуть обмінюватись інформацією протягом етапу кооперації. В дійсності, можливо зазначити що кожна особина знаходиться в певній підмножині особин, що можуть обмінюватись інформацією про їхні

значення, їхній зміст, тощо. В такому випадку кажуть, що популяція структурована. Ця структура може бути узагальнена певною функцією сусідів N, шо назначає правильно підмножину популяції для кожного її члена.

Визначаючи цю функцію N, в такий спосіб що, для кожної особи i, вираз $N(i) \cup \{i\}$ еквівалентно дорвінює всій популяції загалом. Популяція називається неструктурованою, якщо кожна особа може коммунікувати з будь-якою іншою особиною. [23]

4.4. Джерела інформації

Коли особина i має велику множину сусідів $N(i)$ з якоюю вона може кооперувати, це ще не означає що вона отримає переваги від такого великого обсягу інформації. Вона мабуть обере тільки 1 сусіда у випадковому порядку з яким буде оперувати для того щоб створити нову особину. Таке трапляється у генетичних алгоритмах де нащадок особини створюється на основі лише двох предків. Однак, кількість особин що комунікують для того щоб створити нову особину може бути більше ніж два. Це випадок розкиданого пошуку. Дійсно, така техніка оптимізації не накладає обмеження у вигляді тільки двох осіб які можуть бути об’єднані.

У тому ж дусі, Алгоритм з адаптивної пам’яттю був запропонований у роботі для задачі комівояжера. Він не накладає ніяких обмежень на кількість маршрутів що можуть бути об’єднані для створення нового рішення. Цей еволюційний алгоритм Враховує кожен маршрут як особи. Об’єднання деяких маршрутів призводить до створення рішення яке покращується методами локального пошуку. Потім це покращене рішення декомпозується у маршрути щоб були взяті як Кандидати для створення популяції.

Слід зазначити що коли множина особин спілкується для того щоб створити нову це не означає що особини передають інформацію. Наприклад, односторонні обміни виникають в генетичних алгоритмах на базі островів. Дійсно, цей

еволюційний алгоритм визначає особини як множини рішень що називається островами. Ці острови розташовані у направленому кільці І вони постійно передають їхні найкращі рішення до їх нащадків. Таким чином, коли 2 особини Взаємодіють 1 з них передає інформацію а інша відповідно модифікується. В такому випадку лише 1 предок був використаний для модифікації особини.

З іншого боку вся популяція може робити внесок для створення Нової особи. Навіть більше, всі популяції що були створені протягом попередніх ітерації можуть брати участь в цьому інформаційному обміні. В такому випадку створення нової особини називається заснований на історії популяції. Під історією мається на увазі Інформація що не може бути отримана через аналіз особин у поточній популяції. Історичний облік популяції на попередніх ітераціях Необхідний для доступності такої інформації. Мурав'їні системи засновані на цьому концепті так як слід феромонів використовується для навігації мурах і є аналогом сумарної інформації що трапилось за весь процес від початку пошуку.

4.5. Нежиттєздатність

Кожен еволюційний алгоритм базується на точному визначенні особин. Коли об'єднуються інформація із множини особин, може скластися така ситуація що з твориться особина яка є нежиттєздатною. Розглянемо приклад з двома зафарбованим графами. Новий спосіб за фарбування може бути отриманий випадковим вибором з обох варіантів і присвоєний до кожної вершини в графі. Таке об'єднання може призвести до нежиттєздатних варіантів У яких сусідні вершини мають той самий колір.

Той самий концепт може бути проілюстрований і на прикладі задачі комівояжера. Якщо два життєздатних рішення обмінюються частинами їхніх маршрутів то може скластися така ситуація що комівояжер відвідає вузли більше ніж 1 раз а деякі може не відвідати взагалі. Тому важливо зазначити як обробляється нежиттєздатність особин.

Як було зазначено у роботі [24] існує щонайменше 3 стратегії. Найпростіший спосіб результативність якого не була доведена, заключать я в тому щоб просто відкидати нежиттєздатні особини що були отримані на етапі кооперації. Друга стратегія заключається в тому щоб приймати нежиттєздатність але штрафувати особу у функції яка вимірює її якість. 3-я стратегія заключається у розробці спеціальної процедури яка відповідає за об'єднання особин і якаб створювала лише життєздатні варіанти. Остання стратегія часто називається відновлення.

4.6. Стратегія посилення

Багато еволюційних алгоритмів можуть бути значно покращення використовуючи різні алгоритми покращення під час етапу само адаптації. Під алгоритмом покращення мається на увазі будь-яка процедура яка намагається покращити значення особини без використання інформації від інших особин. Використання алгоритму покращення має на меті покращити пошук в деяких областях простору рішень. Вбудовування локального пошуку в еволюційний алгоритм є успішним доповненням у багатьох додатках як це було показано в роботі [25]. Воно й не дивно. В дійсності, використання популяції осіб запевнює дослідження великої частини про строк рішень, а алгоритм покращення допомагає визначити найкращі особини вже знайдених регіонах простору рішень.

Успішність еволюційних алгоритмів що не використовують ніякі алгоритми покращень інколи може бути пояснено передачею доречно інформації під час етапу взаємодії. Насправді, для деяких задач, можливо Співвіднести гарну якість особин з певною частиною інформації яку вона містить. В такому випадку нові особини високої якості можуть бути легко згенеровані Шляхом змішування певних частин інформації з найкращих особин. Наприклад, можна припустити, що гарне рішення проблеми комівояжера має маршрути, що є частиною оптимальних. Таким чином об'єднання маршрутів з найкращих рішень можуть

призвести до створення ще кращого рішення. Це може пояснити успіх генетичних алгоритмів у яких особини визначені як маршрути, а не готове рішення проблеми.

Найбільш складною є ситуація для проблеми з кольорами. В дійсності складно оцінити чому 1 рішення краще за інше. В такому випадку алгоритми покращення є необхідними для забезпечення успіху еволюційного алгоритму.

4.7. Стратегія диверсифікації

Одна із головних складнощів з якими стикаються еволюційні алгоритми являється передчасна збіжність алгоритму в локальний оптимум. Для того щоб віддалити особини від локального оптимуму може бути застосована процедура шуму яка випадковим чином змінює особини. Така процедура модифікує кожну особину незалежно. На відміну від алгоритмів покращення вона має непередбачуваний результат на значення особини, в тому сенсі що вона не обов’язково покращить її значення.

Існує інший спосіб який може допомогти відтягнути особини що скупчилися навколо локального оптимуму. Замість того щоб використовувати шум який продукує не передбачуваний результат, стратегія диверсифікації може бути застосована для того щоб систематично генерувати нові особини які знаходяться в нових регіонах простору рішень. Розсіяний пошук якраз використовує таку стратегію.

4.8. Короткі висновки по головним характеристикам еволюційних алгоритмів

Для того щоб підсумувати вище зазначені концепти розглянемо Будь-який еволюційний алгоритм для вирішення конкретної задачі оптимізації. Філософія цього генетично алгоритму може бути описана в наступними характеристиками.

Особини. Повинно буди визначене відношення між особами і задачею. Особи можуть бути життєздатними та не життєздатними рішеннями задачі. Вони також можуть бути частинами рішення, множинами рішень або будь-чим іншим.

Процес еволюції. Популяція може бути як фіксованого розміру так і динамічного. Повинно бути зазначено чи популяція еволюціонує спираючись на заміщення чи стабільність. У випадку реалізації генетичного алгоритму на паралельній машині може з’явитись асинхронна еволюція.

Сусідства. Обмін інформацію може бути заснований на структурований та не структуровані популяції. В першому випадку може бути зазначена спеціальна структура що використовується наприклад кільце, решітка, тощо.

Джерела інформації. Кількість батьків що необхідні для створення нового індивіду слід зазначити заздалегідь. Це число може бути будь-яким більше 0. Нові особи також можуть бути створені спираючись на історію популяції.

Нежиттєздатність. Потрібно визначити спосіб у який будуть оброблятися нежиттєздатні особи.

Покращення. Під час етапу само адаптації для кожної особи в популяції може бути застосований алгоритм покращення.

Диверсифікація. Для того щоб уникнути завчасного збігу в локальному оптимальні для кожного індивідуума може бути застосована процедура шуму. Також може бути застосована стратегія диверсифікації.

4.9 Опис еволюційних алгоритмів

Тепер розглянемо використання наведених вище понять для опису різних еволюційних алгоритмів.

4.9.1. Генетичні алгоритми

Генетичні алгоритми були створені базуючись на біології, а саме на тих біологічних процесах що дозволяють популяції організмів адаптуватися до

їхнього середовища. Генетичні алгоритми працюють з трьома операторами які послідовно виконується для того щоб змусити популяцію еволюціянувати.

Оператор відбору вирішує які особи виживуть в популяції. Гарні особи зазвичай мають більше шансів бути вибраним.

Оператор схрещення об’єднує 2 особи для того щоб створити нову яка називається нащадок. Оператори відбору та схрещування відбувається під час етапу кооперації.

Операція мутації застосовується під час етапу само адаптації. Оператор мутації запроваджує деякій шум у популяцію для того щоб запобігти завчасному збігу популяції в локальний оптимум. [26]

Генетичний алгоритм представлено у вигляді псевдо коду на рисунку 4.2, а його властивості описані в таблиці 4.1

```
Обрати парне чило p>=2
Згенерувати початкову популяцію P0, що містить p осіб
i := 0
ПОКИ не виконана умови зупинки ВИКОНУВАТИ
    i := i + 1
    Ініціалізувати популяцію Pi порожньою множиною
    ПОКИ Pi містить менше ніж p осіб ВИКОНУВАТИ
        Обрати дві особи Ii та I2 із множини P(i-1)
        Застосувати оператор схрещення до I1 та I2
        Отримати нащадків O1, O2
        Додати O1, O2 у множину Pi
    КІНЕЦЬ
    Застосувати оператор мутації для кожної особи із множини Pi
КІНЕЦЬ
```

Рисунок 4.2 – псевдокод генетичного алгоритму

Таблиця 4.1 – опис властивостей

Особи	Зазвичай життєздатні рішення
Процес еволюції	Замінний. Популяція константного розміру

Продовження таблиці 4.1

Сусідство	Ймовірно структуроване
Джерела інформації	Двоє батьків
Нежиттєздатність	Ніколи не відбувається
Покращення	Відсутнє
Диверсифікація	Процедура шуму. Мутація

4.9.2. Розсіяний пошук

Даний еволюційний алгоритм являється стратегією пошуку що системно генерує множину розсіяних точок із множини опорних точок. Це відбувається шляхом лінійних комбінацій підмножин по точних опорних точок. Точки що були отримані в результаті такого об’єднання називається перевірочна ми точками. Замість того щоб використовувати лише невід’ємні лінійні комбінації точок росіяни пошук використовує більш загальні лінійній комбінації що включають від’ємні ваги.

До тих пір доки точки відповідають життєздатного рішення задачі, перевірочні точки можуть порушувати деякі обмеження. Ці перевірочні точки модифікується процедурою відновлення, що трансформує їх на підходяще рішення. Алгоритм покращення застосовується для кожної нової точки за для того щоб спробувати згенерувати точки ще вищої якості. Ці нові точки формують множину розсіяних точок. Нова множина опорних точок що будуть використані на наступній ітерації обирається із поточних опорних і розсіяних точок. Цей відбір так само як і створення перевірочних точок відбувається на етапі кооперації а процедура відновлення і алгоритм покращення застосовується на етапі самоадаптації. [25]

На рисунку 4.3 наведений псевдокод методу розсіяного пошуку, а його властивості описані в таблиці 4.2

```
Згенерувати початкову множину R0 опорних точок
i := 0
ПОКИ не виконана умова зупинки ВИКОНУВАТИ
    i := i + 1;
    Визначити множину Ti перевірочних точок,
        створюючи лінійні комбінації точок з R(i-1);
    Трансформувати перевірочни точки з Ti
        в множину життєздатних точок Fi;
    Покращити точки з Fi, для того щоб
        отримати множину розсіяних точок Di
    Обрати точки з обєднання R(i-1) та Di та
        додати їх до множини опорних точок Ri
КІНЕЦЬ
```

Рисунок 4.3 – розсіяний пошук

Таблиця 4.2 – властивості розсіяного пошуку

Особи	Життєздатні рішення
Процес еволюції	Постійний. Популяція зазвичай константного розміру
Сусідство	Неструктуроване
Джерела інформації	Мінімум двоє батьків
Нежиттєздатність	Відновлюється
Покращення	Алгоритм покращення
Диверсифікація	Об’єднання осіб

4.9.3. Муравїна система

Даний метод базується на біологічних дослідженнях поведінки колонії мурах. В даному еволюційному алгоритми прості агенти, які називається мурахи, Досліджують область простору рішень і розповсюджують зібрану інформацію.

У базові системі мурах, кожна мураха являє собою процедуру що здатна згенерувати рішення поставленої задачі. На кожному кроці побудови кожна мураха повинна вирішити як зробити наступний крок на шляху до виконання частини проблеми. Такий вибір базується на двох факторах.

Фактор трасування навігує мураху до попередніх виборів, що призвели до гарних результатів. Інтенсивність даного фактору повідомляє мурахам про якість рішення, що було згенеровано зробивши даний вибір.

Фактор бажаності веде мураху до виборів які призводять до найкращих значень функції що вимірює якість рішення.

Коли всі мурахи закінчили їх завдання, фактор трасування оновлюється. Вибір що призвів до гарного рішення призводить до збільшення відповідних факторів трасування. Погані вибори призводять до зменшення цього фактору. Рішення щоб були сформовані мурахами передаються в алгоритм покращення. Фактор трасування відбувається на етапі кооперації, фактор бажаності і алгоритм покращення виконується на етапі само адаптації.[22]

На рисунку 4.6 приведено псевдокод даного методу, а його властивості занесені в таблицю 4.5

```
Ініціалізувати фактори трасування і обрати кількість мурах - n
ПОКИ не виконана умова зупинки ВИКОНУВАТИ
    Побудувати n нових рішень,
        використовуючи фактори бажаності та трасування;
    Оновити фактори трасування;
    Застосувати алгоритм покращення
        для кожного нового рішення;
КІНЕЦЬ
```

Рисунок 4.6 – псевдокод базового алгоритму системи мурах

Таблиця 4.5 – Муравїна система

Особи	Життєздатні рішення
Процес еволюції	Замінний. Популяція константного розміру
Сусідство	Неструктуроване
Джерела інформації	Історія еволюції
Нежиттєздатність	Ніколи не відбувається
Покращення	Алгоритм покращення
Диверсифікація	Відсутня

4.9.4. Адаптивна пам'ять

Центральна пам’ять є відповідальною за зберігання найкращих компонент рішення що були знайдені під час пошуку. Ці компоненти об’єднуються для того щоб створити нові рішення. Якщо вирішення не є життєздатними, вони передаються в процедуру відновлення. Потім вони передаються алгоритму покращення. Нарешті компоненти нових рішень визначаються як особи, що можуть бути обрані.

Алгоритм покращення використовується нас фазі само адаптації, а процедура об’єднання відбір відбувається на фазі кооперації.

Під час першої ітерації центральна пам’ять містить дуже різні компоненти і процедура об’єднання буде прагнути створити розмаїття нових рішень. Таким чином диверсифікація відбувається протягом перших кроків. Потім компоненти що знаходяться в центральній пам’яті будуть прагнути бути частинами дуже маленьких множин рішень що знаходяться в обмеженій кількості областей простору рішень. Процес пошуку плавно рухається від диверсифікації до посилення [25]

Псевдокод на рисунку 4.7, а його властивості в занесені в таблицю 4.6

```
Згенерувати множину рішень і представити їхні компоненти в центральну память
ПОКИ не виконана умова зупинки ВИКОНУВАТИ
    Обєднати компоненти в центральній памяті
        для того щоб створити нові рішення;

    Застосувати процедуру відновлення
        для нежиттєздатних рішень;

    Застосувати алгоритм покращення
        для кожного нового рішення;

    Оновити центральну память, видаливши деякі компоненти
        і представивши новоутворені з життєздатних рішень;
КІНЕЦЬ
```

Рисунок 4.7 – псевдокод базового алгоритму адаптивної пам’яті

Таблиця 4.6 – опис властивостей алгоритм адаптивної пам’яті

Особи	Компоненти життєздатного рішення
Процес еволюції	Постійний. Популяція константного розміру
Сусідство	Неструктуроване
Джерела інформації	Як мінімум двоє батьків
Нежиттєздатність	Процедура відновлення
Покращення	Алгоритм покращення
Диверсифікація	Неявна протягом першої ітерації

4.9.5. Генетичний алгоритм на базі островів

Як вже зазначалося вище, еволюційні алгоритми можуть оперувати особами що не завжди відповідають життєздатним рішенням. Особи можуть бути частиною рішень, множинами рішень або будь-якою інформацією, що може бути об’єднана для створення життєздатного рішення.

Генетичні алгоритми на базі островів демонструють випадок де особи являється множинами рішень. Даний генетичний алгоритм працює на 2 різних рівнях.

На нижньому рівні особи визначаються як життєздатні рішення задачі. Ці особи еволюціонують на засадах звичайного генетичного алгоритму.

На верхньому рівні особи визначені як множини рішень, які називаються островами, і знаходяться на направленому кільці. Кожен острів постійно передає його найкраще рішення до його нащадка. [27]

Блок схема зображена на рисунку 4.8, а властивості алгоритму занесені до таблиць 4.7 та 4.8

```
Обрати число островів - k
Обрати розмір кожно острову - p
Згенерувати множину потужністю k*p початковими життєздатними рішеннями,
    та розділити її на k островів P1, ..., Pk;
i:=0;
ПОКИ не виконана умова зупинки ВИКОНУВАТИ
    i := i + 1
    ДЛЯ КОЖНОГО острову Pi ВИКОНУВАТИ
        Обрати дві особи Ii та I2 із множини P(i-1);
        Застосувати оператор схрещення до I1 та I2;
        Отримати нащадків O1, O2;
        Застосувати оператор мутації та алгоритм покращення
            для O1, O2;
        Вирішити чи варто додавати O1 та O2 в множину Pi
            з заміною попередніх рішень;
    КІНЕЦЬ

    ЯКЩО i ділиться націло на задане число n ТОДІ
        Перемістити найкраще рішення з кожного острову Pi
            до острову P(i mod k)+1
КІНЕЦЬ
```

Рисунок 4.8 – блок схема алгоритму на базі островів

Таблиця 4.8 – опис властивостей на нижньому рівні

Особи	Життєздатне рішення
Процес еволюції	Постійний. Популяція константного розміру
Сусідство	Неструктуроване
Джерела інформації	Двоє батьків

Продовження таблиці 4.8

Нежиттєздатність	Ніколи не відбувається
Покращення	Алгоритм покращення
Диверсифікація	Процедура шуму

Таблиця 4.9 – опис властивостей на верхньому рівні

Особи	Множини життєздатного рішення
Процес еволюції	Змінний. Популяція константного розміру
Сусідство	Структуроване по кільцю
Джерела інформації	Один нащадок
Нежиттєздатність	Ніколи не відбувається
Покращення	Відсутній
Диверсифікація	Відсутня

5 ОПТИМІЗАЦІЯ РОЗМІРУ ТАЙЛУ ЗА РАХУНОК ВИКОРИСТАННЯ ГЕНЕТИЧНОГО АЛГОРИТМУ

5.1 Генетичні алгоритми

Генетичні алгоритми (ГА) – це адаптивні методи, які можна використовувати для вирішення завдань пошуку та оптимізації. Вони засновані на генетичних процесах біологічних організмів. Протягом багатьох поколінь природні популяції розвиваються відповідно до принципів природного відбору та «виживання найсильніших. Імітуючи цей процес, генетичні алгоритми можуть «розвивати» рішення проблем реального світу, якщо вони були належним чином закодовані. Основні принципи ГА були вперше суворо визначені в роботі [26].

ГА працюють із сукупністю «індивідів», кожен з яких представляє можливе рішення даної проблеми. Кожній особині призначається «оцінка» відповідно до того, наскільки гарне рішення проблеми він проедставляє. Високопридатним особинам надається можливість «розмножуватися» шляхом «схрещування» з іншими особинами в популяції. Це породжує нових особин як «нащадків», які мають деякі риси, взяті від кожного «предка». Найменш придатні члени популяції мають менше шансів бути відібраними для розмноження і тому «вимирають».

Таким чином, створюється ціла нова популяція можливих рішень шляхом вибору найкращих особин із поточного «покоління» та їх спарювання, щоб створити новий набір особин. Це нове покоління містить більшу частку характеристик, якими володіли хороші представники попереднього покоління. Таким чином, протягом багатьох поколінь хороші характеристики поширюються на все населення. Віддаючи перевагу спарюванню більш підібраних особин, досліджуються найбільш перспективні області пошуку. Якщо ГА було розроблено добре, популяція наблизиться до оптимального рішення проблеми.

Основною перевагою генетичних алгоритмів є їхня гнучкість і надійність як глобального методу пошуку. Це «слабкі методи», які не використовують

інформацію про градієнт і роблять відносно мало припущень щодо проблеми, що вирішується. Вони можуть мати справу з дуже нелінійними задачами та недиференційованими функціями, а також функціями з кількома локальними оптимумами. Вони також легко піддаються паралельній реалізації, що дозволяє використовувати їх у реальному часі.

Основним недоліком генетичних алгоритмів є їхня гнучкість. Дизайнер повинен придумати схеми кодування, які дозволяють ГА використовувати переваги базових блоків. Потрібно переконатися, що функція оцінки призначає значущі характеристики придатності до ГА. Не завжди зрозуміло, як можна сформулювати функцію оцінки.

5.2 Загальний огляд алгоритму

Функція оцінки, або цільова функція, забезпечує вимірювання ефективності щодо певного набору параметрів. Функція фітнесу перетворює цей показник продуктивності в розподіл репродуктивних можливостей. Оцінка рядка, що представляє набір параметрів, не залежить від оцінки будь-якого іншого рядка. Придатність цього рядка, однак, завжди визначається щодо інших членів поточної сукупності. У генетичному алгоритмі придатність визначається як: f_i / f_A, де f_i — оцінка, пов’язана з рядком i, а f_A — середня оцінка всіх рядків у сукупності.

Придатність також може бути визначена на основі рангу рядка в сукупності або за допомогою методів вибірки, наприклад відбору турніру. Генетичний алгоритм є двоетапним процесом. Починається з ініціації популяції. Відбір застосовується до поточної популяції для створення проміжної популяції. Потім рекомбінація та мутація застосовуються до проміжної популяції для створення наступної популяції. Процес переходу від поточної популяції до наступної популяції становить одне покоління у виконанні генетичного алгоритму.

Стандартний ГА можна представити таким чином (рисунок 5.1):

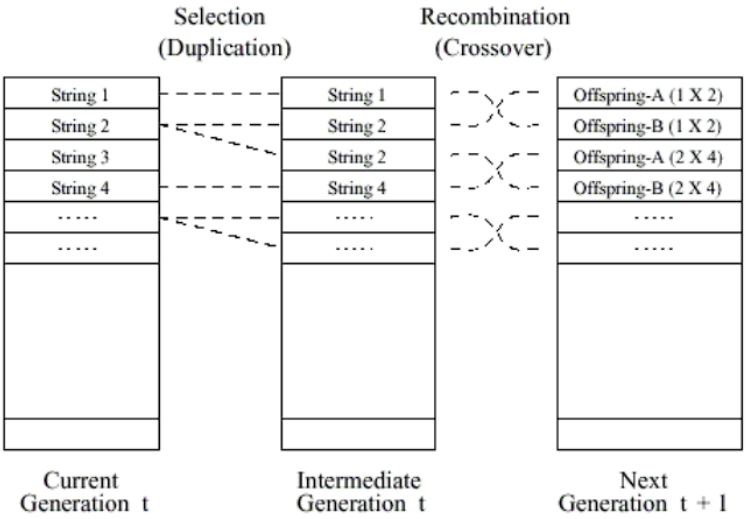


Рисунок 5.1 – генетичний алгоритм, популяції

У першому поколінні поточна популяція також є початковою. Після обчислення f_i / f_A для всіх рядків у поточній популяції виконується відбір. Імовірність того, що рядки в поточній сукупності копіюються (тобто дублюються) і розміщуються в проміжному поколінні, пропорційна їх придатності.

Особи обираються за допомогою «стохастичної вибірки із заміною» для заповнення проміжної популяції. Процес відбору, який більш точно відповідатиме очікуваним значенням придатності, — це «залишок стохастичної вибірки». Для кожного рядка i, де f_i/f_A більше 1,0, ціла частина цього числа вказує, скільки копій цього рядка безпосередньо розміщено в проміжній популяції. Потім усі рядки (включаючи ті з f_i/f_A менше 1,0) розміщують додаткові копії в проміжній популяції з імовірністю, що відповідає дробовій частині f_i/f_A. Наприклад, рядок з f_i/f_A = 1:36 розміщує 1 копію в проміжній сукупності, а потім отримує шанс 0:36 на розміщення другої копії. Рядок із відповідністю f_i/f_A = 0:54 має шанс 0:54 помістити один рядок у проміжну популяцію. Остаточна стохастична вибірка найбільш ефективно реалізується за допомогою методу, відомого як стохастична універсальна вибірка.

Припустимо, що популяція розміщена у випадковому порядку, як у круговій діаграмі, де кожній людині призначається місце на круговій діаграмі пропорційно пристосованості. Навколо пирога розміщено зовнішнє колесо рулетки з N

рівномірно розташованих покажчиків. Одне обертання колеса рулетки тепер одночасно вибирає всіх N членів проміжної попоуляції.

Після проведення відбору побудова проміжної популяції завершена і може відбутися рекомбінація. ЇЇ можна розглядати як створення наступної популяції з проміжної. Кросовер застосовується до випадково об'єднаних рядків з імовірністю, що позначається p_c. (Популяція вже має бути достатньо перемішаною в процесі випадкового відбору.) Вибір пари рядків з ймовірністю p_c «рекомбінує» ці рядки, щоб утворити два нових рядки, які вставляються в наступну популяцію.

Розглянемо наступний двійковий рядок: 1101001100101101. Рядок представляє можливе рішення проблеми оптимізації параметрів. Нові вибіркові точки в просторі генеруються шляхом повторного об’єднання двох батьківських рядків. Розглянемо цей рядок 1101001100101101 та інший двійковий рядок, ухуухуххуууухуххху, у яких значення 0 і 1 позначаються х та у. Використовуючи одну випадково вибрану точку рекомбінації, 1-точковий кросовер відбувається наступним чином (рисунок 5.2):

11010 \/ 01100101101
ухуух /\ уххууухуххy

Рисунок 5.2 – точковий кросовер

Обмін фрагментами між двома предками дає наступне потомство (рисунок 5.3):

11010уххууухуххy and ухуух01100101101

Рисунок 5.3 – нащадки після точкового кросоверу

Після рекомбінації ми можемо застосувати оператор мутації. Для кожного біта в популяції мутувати з деякою низькою ймовірністю p_m. Зазвичай рівень мутації застосовується з імовірністю 0,1%-1%. Після завершення процесу відбору, рекомбінації та мутації можна оцінити наступну популяцію. Процес оцінки, відбору, рекомбінації та мутації формує одне покоління у виконанні генетичного алгоритму [26]

5.3 Вирішення завдання

Перш ніж запускати ГА, необхідно розробити відповідне кодування (або подання) для проблеми. Також потрібна функція придатності, яка призначає показник якості кожному закодованому розв’язку. Під час запуску необхідно відібрати батьків для розмноження та рекомбінувати, щоб отримати потомство

В рамках даної роботи хромосомою буде виступати розмір тайлу. Кодування хромосоми було виконано вектором $size = (a_1\ a_2\ \cdots a_n)$, де $n -$ це кількість вкладених циклів, тобто розмірність.

5.3.1 Початкова популяція

Початкова популяція хромосом створюється за допомогою генератора випадкових чисел на основі нормального розподілу.

5.3.2 Фітнес функція

Для кожної задачі, яку потрібно розв’язати, необхідно придумати відповідну функцію. Для конкретної хромосоми, функція пристосованості повертає єдине число «придатність», який, як передбачається, пропорційний «корисності» або «здатності» особи, яку ця хромосома представляє.

В рамках даної роботи фітнес функцією, яку ми намагаємось мінімізувати, є час виконання програми. Який в свою чергу можна приблизно оцінити як суму 5 компонент [28]

$$T_{cost} = p_{nn}C_{I/O} + \left\lceil \frac{V_{comm}}{K} \right\rceil C_{comm} + V_{comm}t_{comm} + (V_{comm} - M)^2 \quad (5.1)$$

де $C_{I/O}$ – час на запуск операції читання/запису, $t_{I/O}$ – час читання/запису з/в файл на диску, C_{comm} – час запуску взаємодії між тайлами, t_{comm} – час взаємодії між елементами, V_{comm} – час взаємодії між тайлами, K – максимальна довжина повідомлення.

Загальна ціль – мінімізувати T_{cost}, таким чином потрібно мінімізувати чотири підцілі з яких складається T_{cost}.

Перша підціль – час читання/запису – визначається кількістю звертань до файлу. Припускаючи рядкову розмітку, для того щоб мінімізувати кількість операції з файлом кількість рядків що необхідно зчитати, потрібно зменшити. Це не завжди можливо на практиці, тому що зменшення кількості звертань до файлу може призвести до неправильних тайлів, або деякі вимоги щодо взаємодії можуть призвести до нерівних меж на деяких гранях простору ітерацій. На рисунку 5.4 блок (1) і блок (2) мають таку саму кількість вузлів, однак час читання/запису, тобто кількість підрядників, блоку (2) – 4, в той час як блоку (1) – 2. Все інше однакове. Блок (1) є кращим вибором ніж блок (2).

Друга підціль заключається в мінімізації часу необхідного на завантаження блоку в головну пам’ять. Час завантаження визначається за формулою $V_{comp}t_{I/O}$, де $t_{I/O}$час для читання чи запису елемента з або в файл і V_{comp} – кількість ітерації.

Третя підціль заключається в мінімізації загального часу запуску комунікації між блоком і його сусідом. Час комунікації обраховується добутком кількості повідомлень які необхідно передати між блоками і часу на запуск коммунікації.

Четверта підціль заключається в зменшення загального часу комунікації між блоком і його сусідами. Час комунікації визначається добутком часу комунікації на час комунікації кожного елементом.

П’ята пілціль заключається в тому, щоб вмістити тайли в локальну пам'ять паралельного процесору.

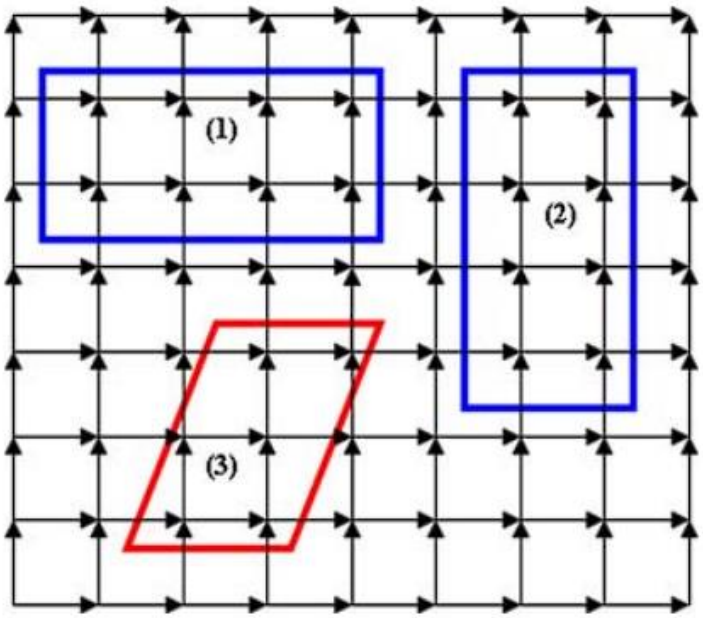


Рисунок 5.4 – різні фігури тайлів

5.3.3 Репродукція

Можливо, кращі особи будуть обрані декілька разів в одній генерації, гірші ж можуть не обрані взагалі. Обравши двох батьків, їхні хромосоми рекомбінуються, зазвичай використовуючи механізм кросоверу та мутації. Кросовер може не застосовуватись до всіх пар, що були обрані для схрещення. Особи обираються випадково. Зазвичай шанс застосування кросоверу для особи складає 60-100%. Якщо кросовер не застосовується, то нащадки продукуються просто дублюючи батьків.

Схрещування було представлене у вигляді комбінування відповідних компонент хромосоми за наступною формулою:

$$a_i = l_i * \lambda + r_i * (1 - \lambda) \quad (5.2)$$

де $a_i - i$ компонента вектору розміру тайлу, $l_i, r_i - i$ компонента векторів хромосом, що схрещуються, $\lambda -$ коефіцієнт, що обирається виподковим чином, при чому $\lambda \in [0; 1)$

В даній роботи операція мутації міняє місцями дві випадкові компоненти.

5.4 Методи відбору

Розглянемо основні методи селекції в генетичних алгоритмах. А саме: the roulette wheel selection (RWS), the stochastic universal sampling (SUS), the linear rank selection (LRS), the exponential rank selection (ERS), the tournament selection (TOS), and the truncation selection (TRS) [29]

5.4.1 Roulette Wheel Selection (RWS)

Помітною характеристикою цього методу відбору є той факт, що він дає кожній окремій людині і з поточної популяції ймовірність $p(i)$ бути обраною, пропорційну її придатності

$$p(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)} \quad (5.3)$$

Де n позначає чисельність популяції за кількістю особин. RWS може бути реалізований відповідно до наступного псевдокоду (рисунок 5.5)

{

- Calculate the sum $S = \sum_{i=1}^{n} f(i)$;

```
• For each individual 1≤i≤n do {
   - Generate a random number α∈[0,S];
   - iSum=0;   j=0;
   - Do {
       - iSum←iSum+f(j);
       - j←j+1;
     } while(iSum<α and j<n)
   -   Select the individual j;}
 }
```

Рисунок 5.5 – псевдокод алгоритму Roulette Wheel Selection

Зауважимо, що відомим недоліком цієї методики є ризик передчасного зближення ГА до локального оптимуму через можливу присутність домінантної особини, яка завжди перемагає в змаганнях і вибирається в якості батька.

5.4.2 Stochastic Universal Sampling (SUS)

SUS є варіантом RWS, спрямованим на зниження ризику передчасної конвергенції. Його можна реалізувати відповідно до наступного псевдокоду (рисунок 5.6):

```
{
• Calculate the mean f̄=1/n∑_{i=1}^{n} f(i);
• Generate a random number α∈[0,1];
• Sum=f(1);  delta=α×f̄;  j=0;
• Do {
   - If (delta<Sum) {
       - select the jth individual;
       - delta=delta+Sum;
     }
     else {
       - j=j+1;
       - Sum=Sum+f(j);
     }
 } while (j<n)
 }
```

Рисунок 5.6 – псевдокод алгоритму Stochastic Universal Sampling

5.4.3 Linear Rank Selection (LRS)

LRS також є варіантом RWS, який намагається подолати недолік передчасної конвергенції GA до локального оптимуму. Він заснований на ранзі людей, а не на їхньому фізичному стані. Ранг n присвоюється найкращій людині, тоді як найгірша людина отримує ранг 1. Таким чином, на основі свого рангу, кожна особа і має ймовірність бути обраною, заданою виразом

$$p(i) = \frac{rank(i)}{n \times (n-1)} \tag{5.3}$$

Після ранжирування всіх осіб поточної популяції процедуру LRS можна реалізувати відповідно до наступного псевдокоду (рисунок 5.7):

```
{
• Calculate the sum v = 1/(n − 2.001);
• For each individual 1 ≤ i ≤ n do {
    - Generate a random number α ∈ [0, v];
    - For each 1 ≤ j ≤ n do {
        - If ( p(j) ≤ α ) {
            - Select the j^th individual;
            - Break;
          }
      }
  }
}
```

Рисунок 5.7 – псевдокод алгоритму Linear Rank Selection

5.4.4 Exponential Rank Selection (ERS)

ERS заснований на тому ж принципі, що і LRS, але він відрізняється від LRS ймовірністю вибору кожної особини. Для ERS ця ймовірність визначається виразом:

$$p(i) = 1.0 * \exp(-\frac{rang(i)}{c}) \quad (5.4)$$

$$c = \frac{n*2*(n-1)}{6*(n-1)+n} \quad (5.5)$$

Після обчислення n ймовірностей решту методу можна описати наступним псевдокодом (рисунок 5.8):

```
{
•  For each individual 1≤i≤n do {
     -  Generate a random number α∈[1/9 c, 2/c];
     -  For each 1≤j≤n do {
          -  If ( p(j)≤α ) {
               -  Select the jth individual;
               -  Break;
             } // end if
        } // end for j
   }// end for i
```

Рисунок 5.8 – псевдокод алгоритму Exponential Rank Selection

5.4.5 Tournament Selection (TOS)

Турнірний відбір є різновидом методів відбору на основі рангів. Його принцип полягає у випадковому виборі набору з k осіб. Потім ці особини класифікуються відповідно до їх відносної придатності, і для відтворення

вибирається найпридатніша особина. Весь процес повторюється n разів для всієї сукупності. Отже, ймовірність вибору кожної особини визначається виразом:

$$p(i) = \begin{cases} \frac{C_{n-1}^{k-1}}{C_n^k} \text{ якщо } i \in [1, n-k-1] \\ 0 \text{ якщо } i \in [n-k, n] \end{cases} \quad (5.6)$$

Технічно, реалізацію TOS можна виконати за псевдокодом (рисунок 5.9):

```
{
• Create a table t where the n individuals are placed in
  a randomly chosen order
• For i=1 to n do {
  - for j=1 to n do {
    - i1 = t(j);
    - For m=1 to n do {
     - i2 = t(j+m);
     - If (f(i1)>f(i2)) the select i1 else select i2;
    }// end for m
    - j = j+k;
   } // end for j
```

Рисунок 5.9 – псевдокод алгоритму Tournament Selection

5.4.6 Truncation Selection (ERS)

Truncation Selection — це дуже проста техніка, яка впорядковує рішення-кандидати для кожної сукупності відповідно до їх придатності. Потім лише певна частина p найпридатніших особин відбирається і відтворюється p разів. Він рідше використовується на практиці, ніж інші методики, за винятком дуже великої популяції. Псевдокод методики такий (рисунок 5.10):

```
{
• Order the n individuals of P(t) according to their
  fitness;
• Set the portion p of individuals to select
  (e.g. 10% ≤ p ≤ 50%);
• sp = int(n × p) // selection pressure;
• Select the first sp individuals;
}
```

Рисунок 5.10 -- псевдокод алгоритму Truncation Selection

5.5 Порівняння методів вібдору на практиці

Для того щоб забезпечити найвищу ефективність пошуку розміру тайла, були виконані заміри часу для двох методів, а саме турнірний та стохастичний.

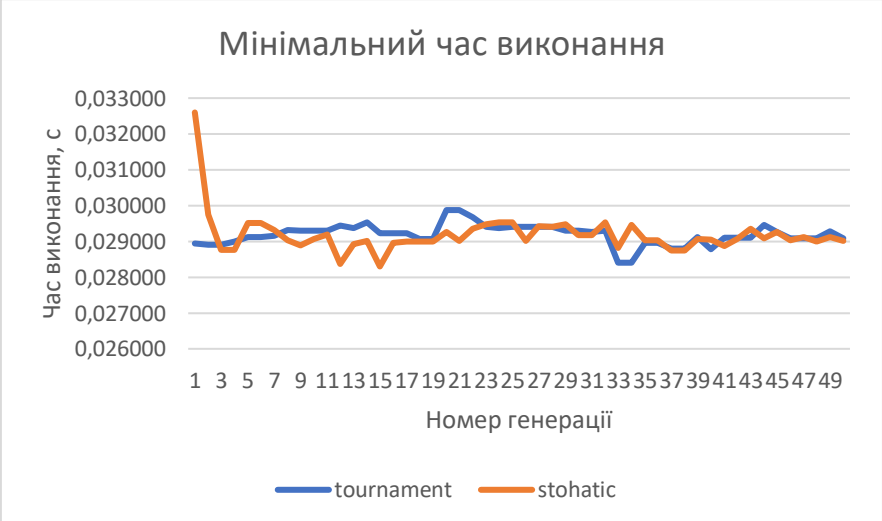
Ефективність алгоритму вибору була виміряна у вигляді швидкості сходження часу виконання цільової програми.

В якості цільової програми був обраний один з алгоритмів з пакету PolyBench, що зображений на рисунку 5.11.

```
for (t = 0; t < _PB_TSTEPS; t++)
  {
    for (i = 1; i < _PB_N - 1; i++)
      B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
    for (i = 1; i < _PB_N - 1; i++)
      A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
  }
```

Рисунок 5.11 – фрагмент коду програми з пакету PolyBench

На рисунку 5.12 зображено графік найкращого часу виконання для кожної генерації



Риусунок 5.12 – графік найкращого часу виконання для генерації

На рисунку 5.13 зображений графік, що показує залежність середнього відхилення від порядкового номеру генерації, тобто ефективність сходження.

Рисунок 5.13 – графік стандартного відхилення

На рисунку 5.14 зображено графік середнього часу в популяції. Показує виродження популяції та прагнення до оптимуму

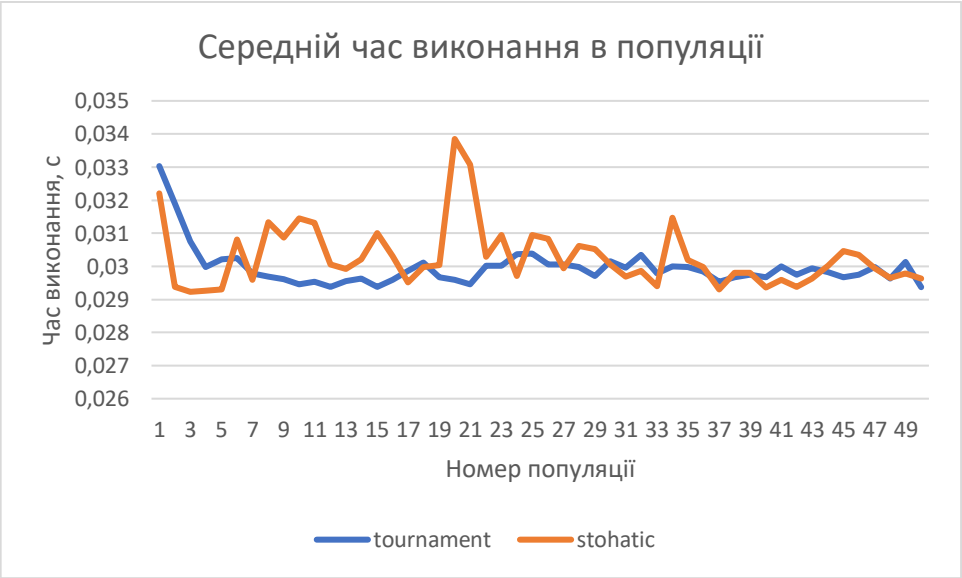


Рисунок 5.14 – графік середнього часу в кожній популяції

Таким чином можна бачити, що метод турнірного відбору більш стійкий до випадкових сплесків та збігається швидше для даної задачі. Тому він і буде використаний в подальших дослідженнях.

6 РЕАЛІЗАЦІЯ ПРОГРАМНОЇ СИСТЕМИ АВТОМАТИЧНОЇ ОПТИМІЗАЦІЇ ДОДАТКІВ

6.1 Deap framework

DEAP (Distributed Evolutionary Algorithms in Python) — це нова еволюційна обчислювальна платформа для швидкого створення прототипів і тестування ідей. Його дизайн відрізняється від більшості інших існуючих фреймворків тим, що він прагне зробити алгоритми явними, а структури даних прозорими, на відміну від більш поширених фреймворків типу чорного ящика. Він також включає легкий паралелізм, коли користувачам не потрібно турбуватися про деталі реалізації, такі як синхронізація та балансування навантаження. Кілька прикладів ілюструють численні властивості DEAP [30]

Основна архітектура DEAP побудована навколо різних компонентів, які визначають конкретні частини еволюційного алгоритму. На рисунку 6.1 показані основні модулі, що утворюють каркас. Ядро DEAP складається з трьох модулів: BASE, CREATOR та TOOLS.

Базовий модуль містить об'єкти та структури даних, які часто використовуються в ГА, які ще не реалізовані в стандартній бібліотеці Python. Python, що забезпечує більшість необхідних структур даних, цей модуль фактично реалізує лише три класи: загальну пристосованість, дерево з префіксом і набір інструментів.

Панель інструментів — це контейнер для інструментів (операторів), які користувач хоче використовувати у своєму ГА. Наприклад, якщо користувачеві потрібна мутація в його алгоритмі, але він має доступ до кількох конструкцій мутації, він вибере одну модель, яка найкраще підходить для його поточної проблеми, назве «MutationXYZ», і зареєструє її в наборі інструментів. Таким чином, він зможе будувати алгоритми, які не пов’язані з наборами операторів. Якщо пізніше він вирішить, що якась інша мутація краще підходить, його

алгоритм залишиться незмінним, йому потрібно буде лише оновити набір інструментів, який використовує алгоритм.

Концепцію, що лежить в основі панелі інструментів, можна знайти в більшості фреймворків типу blackbox. Однак набір інструментів DEAP відрізняється двома аспектами.

По-перше, інші фреймворки зазвичай примушують окремий підпис для кожного типу операції, щоб забезпечити можливість заміни операторів. У DEAP підпис не застосовується, отже, дозволяє користувачеві реалізувати якийсь спеціальний оператор, як він хоче, без необхідності обходити тонкощі реалізації.

По-друге, чорні скриньки зазвичай надають значення за замовчуванням для більшості параметрів, щоб спростити виклик оператора. Ця практика суперечить першій гіпотезі, і тому тут не визначаються значення за замовчуванням для жодного параметра оператора. Панель інструментів виконує це завдання спрощення, дозволяючи користувачеві реєструвати значення параметрів у оператора. Таким чином, користувач повинен розуміти кожен оператор, який він використовує, а значення параметрів завжди чітко вказуються перед визначенням алгоритму, таким чином уникаючи будь-якої можливої двозначності.

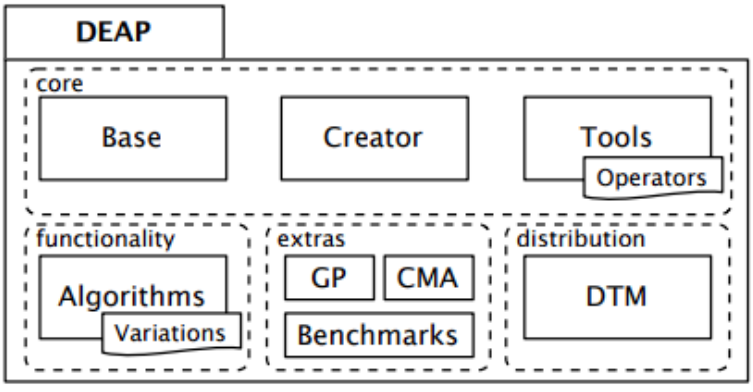


Рисунок 6.1 –архітектура фреймворку DEAP

Модуль creator — це мета-фабрика, яка дозволяє створювати класи як шляхом успадкування, так і композиції з використанням парадигми

функціонального програмування, таким чином звільняючи користувача від тягаря визначення класу. Атрибути, як дані, так і функції, можуть бути динамічно додані для створення нових класів об’єктів, уповноважених користувачем для надання функціональних можливостей ГА [30]

Модуль інструментів містить часто використовувані оператори. Він також надає об'єкти, які полегшують різні завдання аналізу, такі як контрольні точки, обчислення статистики або генеалогію.

Основні функції DEAP керуються модулем алгоритмів, який містить чотири часто використовувані алгоритми:

- Генераційний
- (μ, λ)
- (μ + λ)
- Запитай-і-розкажи [31]

Однак DEAP жодним чином не обмежується цими чотирма. Вони є лише відправною точкою для користувачів для розробки власних налаштованих алгоритмів.

Оператори та інструменти, які не надаються основними модулями, мають власний модуль, наприклад, оператори генетичного програмування і структури даних можна знайти в модулі gp, а CMA-ES — у модулі cma. Модуль контрольних показників включає різні найсучасніші функції тесту, які можна використовувати для оцінки ефективності алгоритму.

Останній модуль фреймворка, названий dtm для диспетчера розподілених завдань, обробляє паралелізм.

6.2 Деталі реалізації

Був розроблений наступний алгоритм (рисунок 6.2)

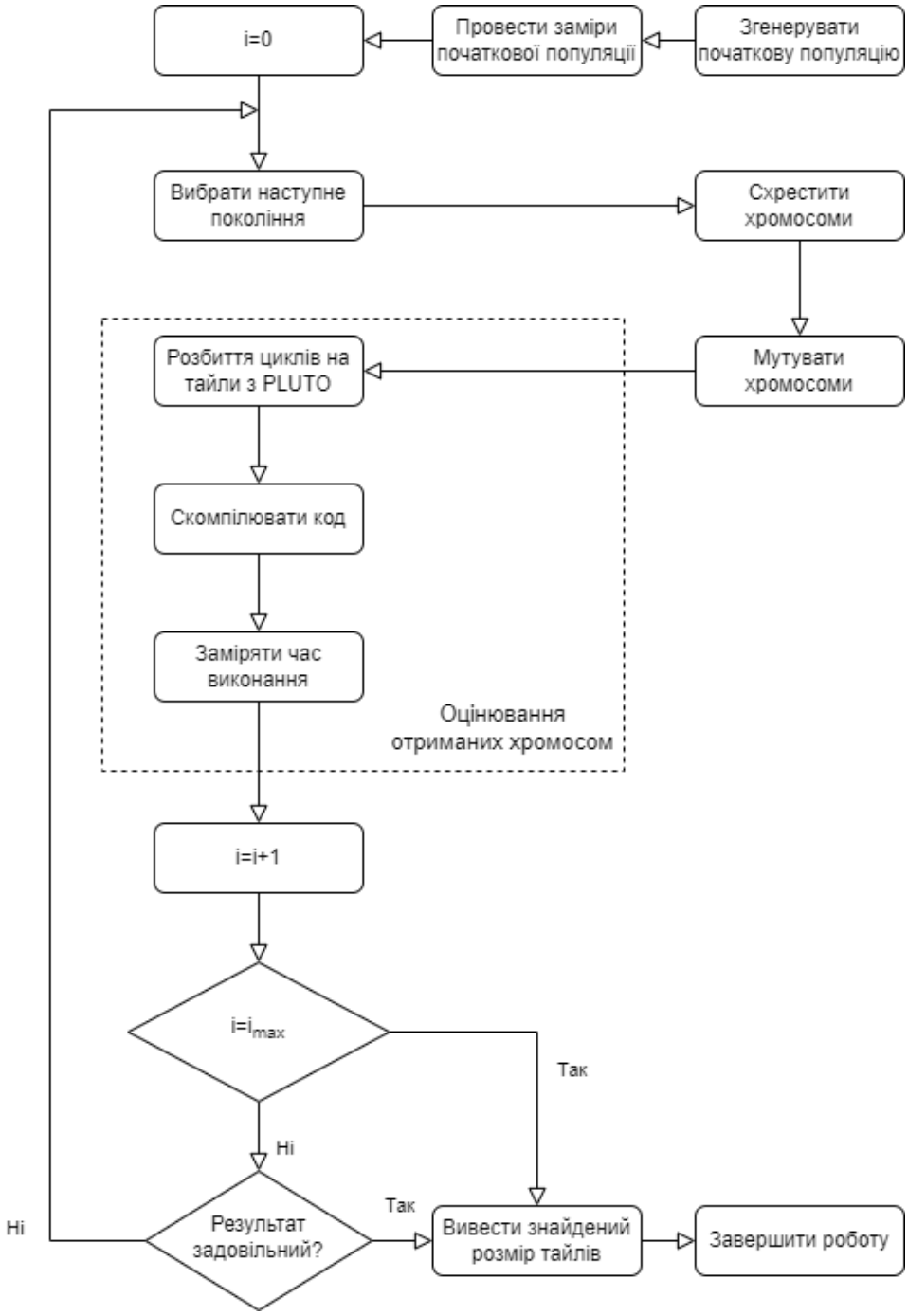


Рисунок 6.2 –алгоритм системи автоматичної оптимізації додатків

6.2.1 Налаштування фреймворку DEAP

Спочатку визначимо клас FitnessMin, що буде наслідувати клас Fitness з модулю deap.base (рисунок 6.3). Додатково він отримає атрибут weights, що дозволить фреймворку розуміти максимізувати чи мінімізувати значення кандидатів.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

Рисунок 6.3 – визначення класу FitnessMin

Клас кандидату буде представлений як наслідник класу list, а додатковий атрибут fitness буде визначений щойно створеним класом FitnessMin (рисунок 6.4).

```
creator.create("Individual", list, fitness=creator.FitnessMin)
```

Рисунок 6.4 – визначення класу Individual

Таким чином можна використати щойно створені класи для представлення окремих кандидатів, так само як і всю популяцію.

Всі обєкти, що будуть використовуватись далі, окремий кандидат, популяція, так само як і функції, оператори і аргументи будуть зберігатись в DEAP контейнері, що називається Toolbox. В ньому є два методи register та uregister для того щоб додавати та видаляти вміст контейнера.

Далі реєструється (рисунок 6.5) функція-генератор toolbox.attr_int() і дві функції-ініціалізації individual() та population().

toolbox.attr_int() генерує випадкове число від одиниці до CMaxTileSize. CMaxTileSize – константа, що в залежності від цільової архітектури.

Реєстрація інструменту в тулбоксі лише асоціює аліас до вже ісуючої функції і заморожує частину її аргументів. Це дозволяє зафіксувати будь-яку кількість аргументів з конкретними значеннями так, що потрібно зазначити лише ті, яких не вистачає.

Конкретні кандидати будуть генеруватись за допомогою функції initRepeat(). Її першим аргументом є клас-контейнер. Контейнер буде заповнений за допомогою методу attr_int(), який передається другим аргументом і буде мати CTileDimension чисел типу integer. Ця інформація передається третім аргументом.

Коли individual() викликають, то повернеться ініціалізований кандидат, шляхом послідовних викликів attr_int у кількості CTileDimension.

```
toolbox = base.Toolbox()
# Attribute generator
#                      define 'attr_int' to be an attribute ('gene')
#                      which corresponds to integers sampled uniformly
#                      from the range [1,CMaxTileSize] (e.g. 0 or 255 with equal
#                      probability)
toolbox.register("attr_int", random.randint, 1, CMaxTileSize)

# Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_int, CTileDimension)

# define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Рисунок 6.5 – реєстрація обєкту популяції в контейнері Toolbox

Так як оцінка розміру тайлу спряжена з використанням програмного пакету PLUTO, то була розроблена окрема функція evalTileSize(individual) (рисунок 6.6). В ній можна виділити такі основні компоненти:

- Збереження розміру тайлу на диск, для його подальшого використанням PLUTO
- Видалення попередніх результатів роботи PLUTO
- Запуск PLUTO для відповідної програми з відповідними розмірами тайлів. В результаті чого генерується і зберігається трансформований код
- Отриманий вихідний код компілюється за допомогою gcc
- Запуск згенерованої програми. Після виконання тестова програма друкує в стандартний потік виводу час, за який вона виконалась
- Зчитування та часу виконання та повернення його як результат оцінки кандидата

```
def getTime(path):
    result = subprocess.run([path], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    stdOut = result.stdout.decode('utf-8')
    stdErr = result.stderr.decode('utf-8')
    output = stdOut if len(stdOut) > 0 else stdErr
    t = float(output[:-2])
    return t

def evalTileSize(individual):
    tileSize = individualToTileSize(individual)
    print('size =', tileSize)

    if min(tileSize) == 0:
        print('t = 9999')
        return 9999,

    with open('./tile.sizes', 'w') as f:
        for size in map(lambda x: str(x)+'\n', tileSize):
            f.write(size)

    files = glob.glob('./*.tiled.c')
    subprocess.run(['rm'] + files, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
    try:
        subprocess.run(['make'], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL, timeout=3)
    except:
        subprocess.run(['killall', 'pluto'], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL
        print('t = 9999')
        return 9999,

    t = getTime('./tiled')

    print('t =', t)
    return t,
```

Рисунок 6.6 – розроблена окрема функція evalTileSize(individual)

6.2.2 Згенерувати початкову популяцію

Перш за все потрібно згенерувати початкову популяцію. З фреймворком DEAP це можна зробити за допомогою вище зареєстрованого методу population() (рисунок 6.7)

```
# create an initial population of 50 individuals (where
# each individual is a list of integers)
pop = toolbox.population(n=50)
```

Рисунок 6.7 – використання методу population

В результаті створиться змінна pop, в яку буде записана початкова популяція з 50 кандидатів. Так як параметр n не був визначений під час реєстрації методу в тулбоксі, його слід зазначити під час виклику.

Наступник кроком буде проведення оцінки початкової популяції

6.2.3 Провести заміри початкової популяції

Для проведення замірів використовується зареєстрована функція evalTileSize, що була описана вище.

Застосована функція map(), що дозволяє асоціювати функцію оцінки для кожного кандидату в популяції (рисунок 6.8). Результат запишеться у змінну fitness

```
# Evaluate the entire population
fitnesses = list(map(toolbox.evaluate, pop))
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit
```

Рисунок 6.8 – обраховування значень популяції

Перед тим як переходити до головного циклу в алгоритмі визначимо константи для ймовірностей мутації та схрещування (рисунок 6.9).

```
# CXPB  is the probability with which two individuals
#       are crossed
#
# MUTPB is the probability for mutating an individual
CXPB, MUTPB = 0.5, 0.5
```

Рисунок 6.9 – визначення вірогідностей мутції та схрещування

6.2.4 Вибрати наступне покоління

Для вибору покоління була використана функція toolbox.selTournament(tournSize=3) (рисунок 6.10). Дана функція реалізує вибір кандидатів згідно стратегії турніру.

Турнірний відбір – це такий метод відбору осіб з популяції, що залучає виконання деяких турнірів серед кількох осіб, котрі обираються випадковим чином з усієї популяції. В даному випадку обирається три особи. Переможець кожного турніру, тобто той, чиї показники оцінки найбільш оптимальні, обирається для операції кросоверу.

```
# Select the next generation individuals
offspring = toolbox.select(pop, len(pop))
# Clone the selected individuals
offspring = list(map(toolbox.clone, offspring))
```

Рисунок 6.10 – етап відбору

Таким чином були створений список нащадків який є точною копією відібраних осіб. Метод toolbox.clone() гарантує, що ми використовуємо не посилання на окремих осіб, а повністю незалежний екземпляр. Це надзвичайно важливо, оскільки генетичні оператори в наборі інструментів будуть змінювати надані об’єкти на місці.

Далі ми виконаємо як кросовер і мутацію.

6.2.5 Схрестити хромосоми

Кросовер отриманих нащадків відбувається з певною ймовірністю CXPB (рис. 6.11). Оператор del скасує придатність модифікованого потомства.

```
# Apply crossover and mutation on the offspring
for child1, child2 in zip(offspring[::2], offspring[1::2]):

    # cross two individuals with probability CXPB
    if random.random() < CXPB:
        toolbox.mate(child1, child2)

        # fitness values of the children
        # must be recalculated later
        del child1.fitness.values
        del child2.fitness.values
```

Рисунок 6.11 – кросовер

Наступним кроком буде мутація отриманих нащадків.

6.2.6 Мутувати хромосоми

Мутація відбувається з певною ймовірністю MUTPB (рис. 6.12).

Оператори схрещування (або спарювання) і мутації, надані в DEAP, зазвичай приймають відповідно 2 або 1 особину як вхід і повертають 2 або 1 модифіковану особину(и). Крім того, вони змінюють цих осіб у контейнері інструментів, і нам не потрібно перепризначати їхні результати.

```
for mutant in offspring:

    # mutate an individual with probability MUTPB
    if random.random() < MUTPB:
        toolbox.mutate(mutant)
        del mutant.fitness.values
```

Рисунок 6.12 – мутація

Оскільки вміст деяких наших нащадків змінився під час останнього кроку, тепер нам потрібно повторно оцінити їхню придатність (рисунок 6.13)

```
fitnesses = map(toolbox.evaluate, invalid_ind)
for ind, fit in zip(invalid_ind, fitnesses):
    ind.fitness.values = fit
```

Рисунок 6.13 – оцінка нащадків

Останнім ключовим етапом в циклі генетичного алгоритму є заміна популяції на нащадків (рисунок 6.14)

```
# The population is entirely replaced by the offspring
pop[:] = offspring
```

Рисунок 6.14 – заміна популяції нащадками

Щоб перевірити ефективність еволюції, ми обчислимо та надрукуємо мінімальні, максимальні та середні значення придатності всіх індивідуумів нашої популяції, а також їх стандартні відхилення. (рисунок 6.15)

```
# Gather all the fitnesses in one list and print the stats
fits = [ind.fitness.values[0] for ind in pop]

length = len(pop)
mean = sum(fits) / length
sum2 = sum(x*x for x in fits)
std = abs(sum2 / length - mean**2)**0.5

print("  Min %s" % min(fits))
print("  Max %s" % max(fits))
print("  Avg %s" % mean)
print("  Std %s" % std)
```

Рисунок 6.15 – обрахунки стандратного відхилення, тощо

7 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ РОБОТИ СИСТЕМИ НА ПРИКЛАДІ ПРОГРАМ З ПАКЕТУ POLYBENCH

Проведемо експериментальні дослідження роботи системи. Для цього візьмемо тестові програми з пакету PolyBench та запустимо на них дану систему. На рисунку 7.1 зображений проміжний етап роботи системи.

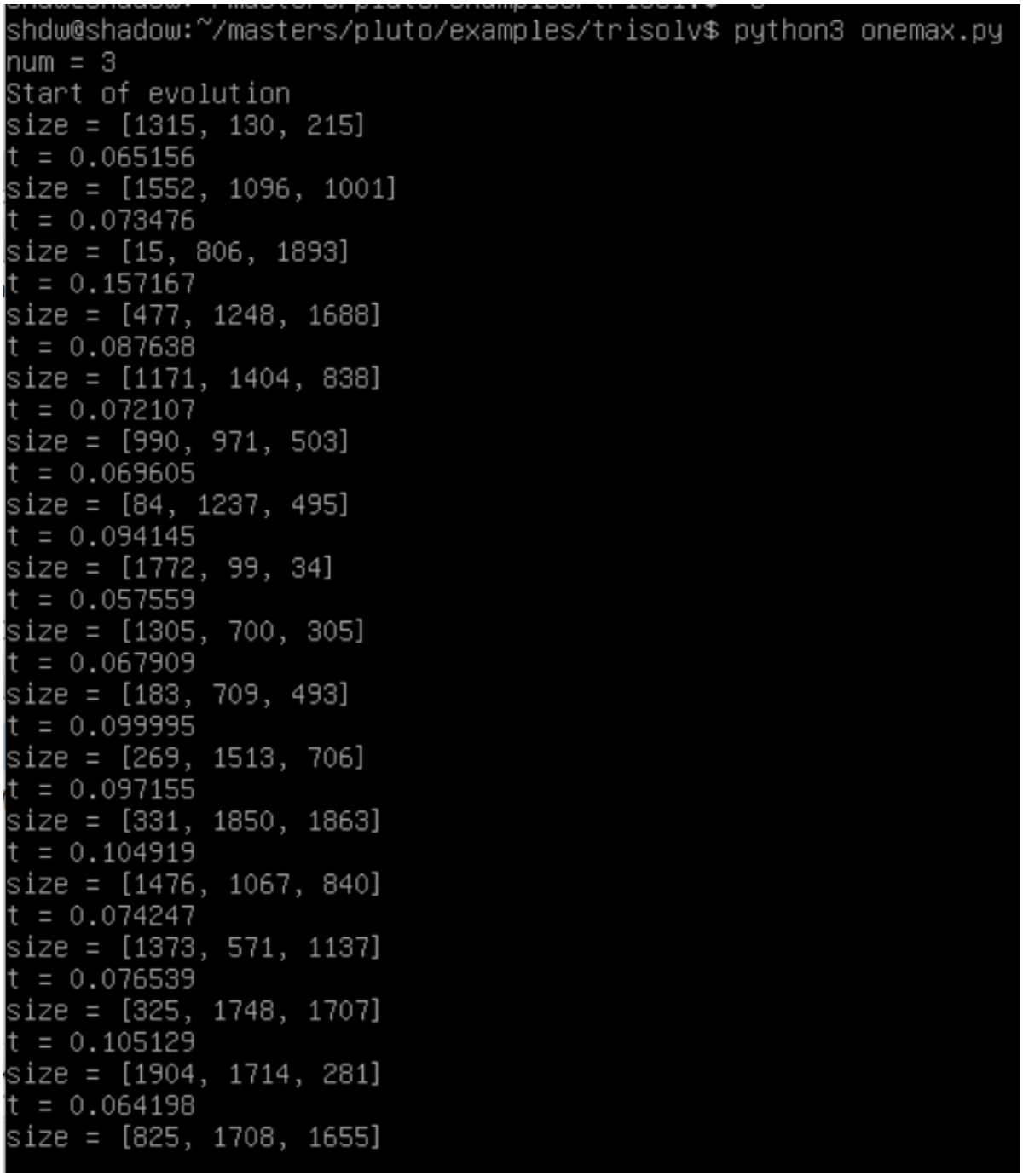


Рисунок 7.1 – проміжний етап роботи системи

В експерименті брали участь такі тестові програми:

- tmm – Множення трикутних матриць. Фрагмент коду зображений на рисунку 7.2
- dsyrk – метод симетричного рангу k. Фрагмент коду зображений на рисунку 7.4
- ssymm – операції над матрицями. Фрагмент коду зображений на рисунку 7.6
- strmm – операції над матрицями. Фрагмент коду зображений на рисунку 7.8
- strsm – вирішення системи рівнянь. Фаргмент коду зображений на рисунку 7.10
- gemver – множення векторів та додавання матриць. Фрагмент коду зображений на рисунку 7.12
- trisolv –вирішувач рівнянь з трикутними матрицями. Фрагмент коду зображений на рисунку 7.14

```
for(i = 0; i < NMAX; i++) {
        for(j=i; j < NMAX; j++) {
                for(k=i;  k< NMAX; k++) {
                        C[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

Рисунок 7.2 – tmm

На рисунку 7.3 зображений фрагмент коду алгоритму tmm, що був згенерований методом тайліенгу.

```
if (NMAX >= 1) {
  for (t1=0;t1<=floord(NMAX-1,32);t1++) {
    for (t2=t1;t2<=floord(NMAX-1,32);t2++) {
      for (t3=t1;t3<=floord(NMAX-1,32);t3++) {
        for (t4=32*t1;t4<=min(NMAX-1,32*t1+31);t4++) {
          for (t5=max(32*t3,t4);t5<=min(NMAX-1,32*t3+31);t5++) {
            lbv=max(32*t2,t4);
            ubv=min(NMAX-1,32*t2+31);
            for (t6=lbv;t6<=ubv;t6++) {
              C[t4][t6] += A[t4][t5] * B[t5][t6];;
            }
          }
        }
      }
    }
  }
}
```

Рисунок 7.3 – оптимізований tmm

```
for (i=0; i<NMAX; i++) {
  for (j=0; j<NMAX; j++) {
    for (k=j; k<NMAX; k++) {
      c[j][k] += a[i][j] * a[i][k];
    }
  }
}
```

Рисунок 7.4 – dsyrk

На рисунку 7.5 зображений фрагмент коду алгоритму dsyrk, що був згенерований методом тайліенгу

```
if (NMAX >= 1) {
  for (t1=0;t1<=floord(NMAX-1,32);t1++) {
    for (t2=t1;t2<=floord(NMAX-1,32);t2++) {
      for (t3=0;t3<=floord(NMAX-1,32);t3++) {
        for (t4=32*t1;t4<=min(NMAX-1,32*t1+31);t4++) {
          for (t5=32*t3;t5<=min(NMAX-1,32*t3+31);t5++) {
            lbv=max(32*t2,t4);
            ubv=min(NMAX-1,32*t2+31);
            for (t6=lbv;t6<=ubv;t6++) {
              c[t4][t6] += a[t5][t4] * a[t5][t6];;
            }
          }
        }
      }
    }
  }
}
```

Рисунок 7.5 – оптимізований dsyrk

```
for (i=0; i<NMAX; i++) {
  for (j=0; j<NMAX; j++) {
    for (k=0; k<j-1; k++) {
      c[i][k] += a[j][k] * b[i][j];
      c[i][j] += a[j][j] * b[i][j];
    }
    c[i][j] += a[j][j] * b[i][j];
  }
}
```

Рисунок 7.6 – ssymm

На рисунку 7.7 зображений фрагмент коду алгоритму ssymm, що був згенерований методом тайліенгу

```
for (t2=0;t2<=floord(NMAX-1,1171);t2++) {
  for (t3=0;t3<=floord(NMAX-1,1404);t3++) {
    for (t4=0;t4<=min(floord(702*t3+700,419),floord(NMAX-3,838));t4++) {
      for (t5=1171*t2;t5<=min(NMAX-1,1171*t2+1170);t5++) {
        for (t6=max(1404*t3,838*t4+2);t6<=min(NMAX-1,1404*t3+1403);t6++) {
          for (t7=838*t4;t7<=min(838*t4+837,t6-2);t7++) {
            c[t5][t6] += a[t6][t6] * b[t5][t6];;
          }
        }
      }
    }
  }
}
for (t2=0;t2<=floord(NMAX-1,1171);t2++) {
  for (t3=0;t3<=floord(NMAX-1,1404);t3++) {
    for (t4=1171*t2;t4<=min(NMAX-1,1171*t2+1170);t4++) {
      lbv=1404*t3;
      ubv=min(NMAX-1,1404*t3+1403);
      for (t5=lbv;t5<=ubv;t5++) {
        c[t4][t5] += a[t5][t5] * b[t4][t5];;
      }
    }
  }
}
for (t2=0;t2<=floord(NMAX-1,1171);t2++) {
  for (t3=0;t3<=floord(NMAX-3,1404);t3++) {
    for (t4=ceild(702*t3-417,419);t4<=floord(NMAX-1,838);t4++) {
      for (t5=1171*t2;t5<=min(NMAX-1,1171*t2+1170);t5++) {
        for (t6=max(838*t4,1404*t3+2);t6<=min(NMAX-1,838*t4+837);t6++) {
          lbv=1404*t3;
          ubv=min(1404*t3+1403,t6-2);
          for (t7=lbv;t7<=ubv;t7++) {
            c[t5][t7] += a[t6][t7] * b[t5][t6];;
          }
        }
      }
    }
  }
}
```

Рисунок 7.7 – оптимізований ssymm

```
for (i=1; i<NMAX; i++) {
  for (j=0; j<NMAX; j++) {
    for (k=0; k<i; k++) {
      b[j][k] += a[i][k] * b[j][i];
    }
  }
}
```

Рисунок 7.8 – strmm

На рисунку 7.9 зображений фрагмент коду алгоритму strmm, що був згенерований методом тайліенгу

```
if (NMAX >= 2) {
  for (t1=0;t1<=floord(NMAX-1,32);t1++) {
    for (t2=0;t2<=floord(NMAX-1,32);t2++) {
      for (t3=0;t3<=min(floord(NMAX-2,32),t2);t3++) {
        for (t4=32*t1;t4<=min(NMAX-1,32*t1+31);t4++) {
          for (t5=max(32*t2,32*t3+1);t5<=min(NMAX-1,32*t2+31);t5++) {
            for (t6=32*t3;t6<=min(32*t3+31,t5-1);t6++) {
              b[t4][t6] += a[t5][t6] * b[t4][t5];;
            }
          }
        }
      }
    }
  }
}
```

Рисунок 7.9 – оптимізований strmm

```
for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
                for (k=i+1; k<N; k++) {
                        if (k == i+1) b[j][i] /= a[i][i];
        b[j][k] -= a[i][k] * b[j][i];
    }
}
```

Рисунок 7.10 – strsm

На рисунку 7.11 зображений фрагмент коду алгоритму strsm, що був згенерований методом тайліенгу

```
if (N >= 2) {
  for (t1=0;t1<=floord(N-1,32);t1++) {
    for (t2=0;t2<=floord(N-2,32);t2++) {
      for (t3=t2;t3<=floord(N-1,32);t3++) {
        if (t2 == t3-1) {
          for (t4=32*t1;t4<=min(N-1,32*t1+31);t4++) {
            for (t5=32*t2;t5<=32*t2+30;t5++) {
              for (t6=32*t2+32;t6<=min(N-1,32*t2+63);t6++) {
                b[t4][t6] -= a[t5][t6] * b[t4][t5];;
              }
            }
            b[t4][(32*t2+31)] /= a[(32*t2+31)][(32*t2+31)];;
            for (t6=32*t2+32;t6<=min(N-1,32*t2+63);t6++) {
              b[t4][t6] -= a[(32*t2+31)][t6] * b[t4][(32*t2+31)];;
            }
          }
        }
        if (t2 == t3) {
          for (t4=32*t1;t4<=min(N-1,32*t1+31);t4++) {
            for (t5=32*t2;t5<=min(N-2,32*t2+30);t5++) {
              b[t4][t5] /= a[t5][t5];;
              for (t6=t5+1;t6<=min(N-1,32*t2+31);t6++) {
                b[t4][t6] -= a[t5][t6] * b[t4][t5];;
              }
            }
          }
        }
        if (t2 <= t3-2) {
          for (t4=32*t1;t4<=min(N-1,32*t1+31);t4++) {
            for (t5=32*t2;t5<=32*t2+31;t5++) {
              for (t6=32*t3;t6<=min(N-1,32*t3+31);t6++) {
                b[t4][t6] -= a[t5][t6] * b[t4][t5];;
              }
            }
          }
        }
      }
    }
  }
}
```

Рисунок 7.11 – оптимізований strsm

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        B[i][j] = A[i][j] + u1[i]*v1[j] + u2[i]*v2[j];

for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        x[i] = x[i] + beta* B[j][i]*y[j];


for (i=0; i<N; i++)
    x[i] = x[i] + z[i];

for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        w[i] = w[i] + alpha* B[i][j]*x[j];
```

Рисунок 7.12 – gemver

На рисунку 7.13 зображений фрагмент коду алгоритму gemver, що був згенерований методом тайліенгу.

```
for (i=0;i<=N-1;i++) {
    for (j=0;j<=N-1;j++) {
        for (k=0;k<=j-1;k++) {
            B[j][i]=B[j][i]-L[j][k]*B[k][i];  //S1 ;

        }
        B[j][i]=B[j][i]/L[j][j]; // S2 ;
    } // for j
} // for i
```

Рисунок 7.13 – trisolv

```
for (t2=0;t2<=floord(N-1,256);t2++) {
  for (t3=0;t3<=floord(N-1,256);t3++) {
    for (t4=8*t2;t4<=min(floord(N-1,32),8*t2+7);t4++) {
      for (t5=8*t3;t5<=min(floord(N-1,32),8*t3+7);t5++) {
        for (t6=32*t5;t6<=min(N-1,32*t5+31);t6++) {
          lbv=32*t4;
          ubv=min(N-1,32*t4+31);
          for (t7=lbv;t7<=ubv;t7++) {
            B[t6][t7] = A[t6][t7] + u1[t6]*v1[t7] + u2[t6]*v2[t7];;
            x[t7] = x[t7] + beta* B[t6][t7]*y[t6];;
          }
        }
      }
    }
  }
}
for (t2=0;t2<=floord(N-1,256);t2++) {
  for (t3=8*t2;t3<=min(floord(N-1,32),8*t2+7);t3++) {
    lbv=32*t3;
    ubv=min(N-1,32*t3+31);
    for (t4=lbv;t4<=ubv;t4++) {
      x[t4] = x[t4] + z[t4];;
    }
  }
}
for (t2=0;t2<=floord(N-1,256);t2++) {
  for (t3=0;t3<=floord(N-1,256);t3++) {
    for (t4=8*t2;t4<=min(floord(N-1,32),8*t2+7);t4++) {
      for (t5=8*t3;t5<=min(floord(N-1,32),8*t3+7);t5++) {
        for (t6=32*t4;t6<=min(N-1,32*t4+31);t6++) {
          for (t7=32*t5;t7<=min(N-1,32*t5+31);t7++) {
            w[t6] = w[t6] + alpha* B[t6][t7]*x[t7];;
          }
        }
      }
    }
  }
}
```

Рисунок 7.14 – оптимізований gemver

На рисунку 7.15 зображений фрагмент коду алгоритму trisolv, що був згенерований методом тайліенгу.

```
if (N >= 1) {
  for (t1=0;t1<=floord(N-1,32);t1++) {
    for (t2=0;t2<=floord(N-1,32);t2++) {
      for (t3=0;t3<=t2;t3++) {
        if (t2 >= t3+1) {
          for (t4=32*t2;t4<=min(N-1,32*t2+31);t4++) {
            for (t5=32*t3;t5<=32*t3+31;t5++) {
              lbv=32*t1;
              ubv=min(N-1,32*t1+31);
              for (t6=lbv;t6<=ubv;t6++) {
                B[t4][t6]=B[t4][t6]-L[t4][t5]*B[t5][t6];;
              }
            }
          }
        }
        if (t2 == t3) {
          lbv=32*t1;
          ubv=min(N-1,32*t1+31);
          for (t6=lbv;t6<=ubv;t6++) {
            B[32*t2][t6]=B[32*t2][t6]/L[32*t2][32*t2];;
          }
        }
        if (t2 == t3) {
          for (t4=32*t2+1;t4<=min(N-1,32*t2+31);t4++) {
            for (t5=32*t2;t5<=t4-1;t5++) {
              lbv=32*t1;
              ubv=min(N-1,32*t1+31);
              for (t6=lbv;t6<=ubv;t6++) {
                B[t4][t6]=B[t4][t6]-L[t4][t5]*B[t5][t6];;
              }
            }
            lbv=32*t1;
            ubv=min(N-1,32*t1+31);
            for (t6=lbv;t6<=ubv;t6++) {
              B[t4][t6]=B[t4][t6]/L[t4][t4];;
            }
          }
        }
      }
    }
  }
}
```

Рисунок 7.15 – оптимізовний trisolv

В результаті роботи системи були зібрані дані по часу роботи тестових програм без використання оптимізацій, з використанням розмірів тайлів замовчуванням та з оптимальним розміром тайлів. На рисунку 7.16 зображений графік роботи тестових програм. По осі абсцис відкладені тестові програми та варіації оптимізацій. По осі ординат відкладений відносний час роботи тестової програми.

Відносний час вираховується згідно формули

$$T = \frac{T_{optimized}}{T_{orignal}} * 100\%$$

де $T_{optimized}$ – час виконання оптимізованої програми, $T_{original}$ – час виконання тестової програми без оптимізацій.

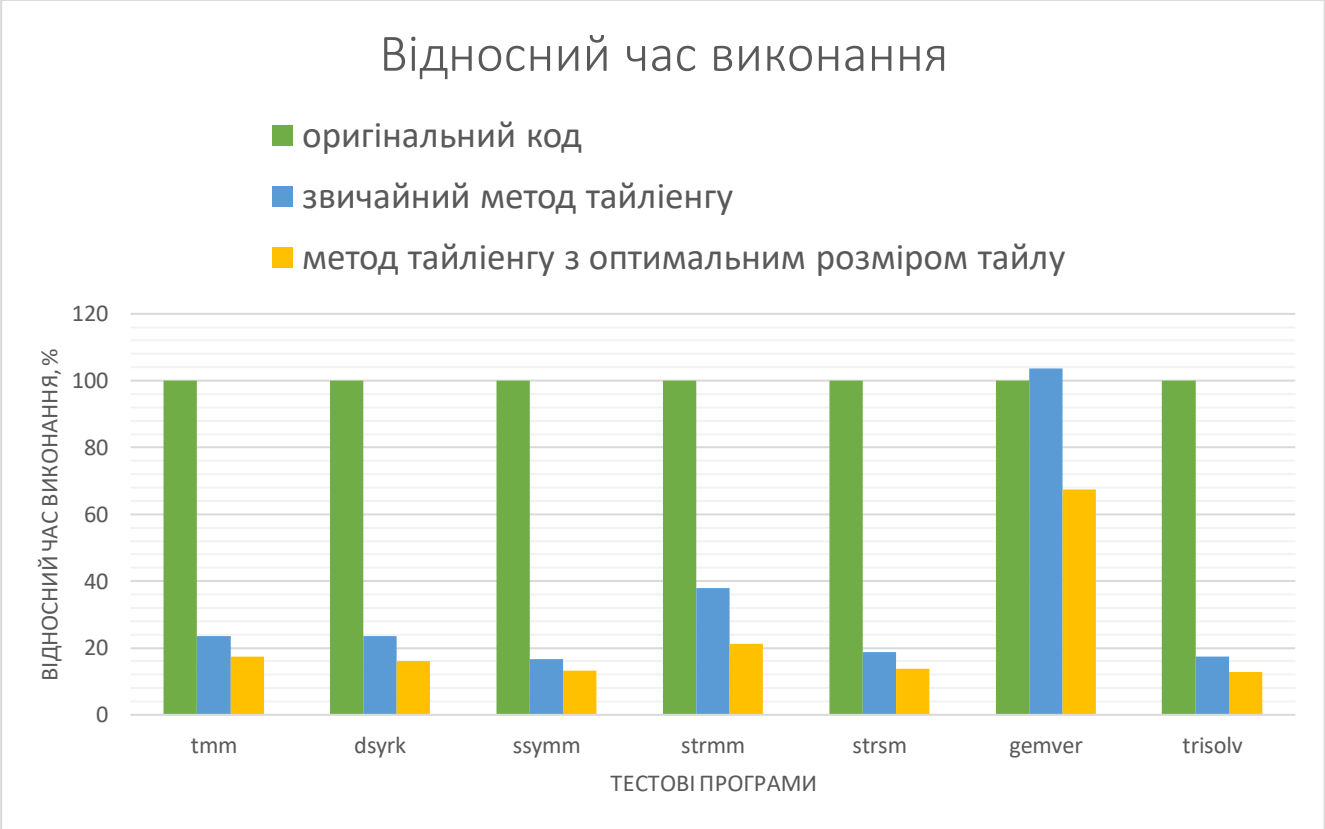


Рисунок 7.16 – порівняння часів виконання програм до і після оптимізацій

В таблиці 7.1 зображений відносний час виконання програм до і після оптимізації.

Таблиця 7.1 – відносний час виконання програми

Назва	Звичаний метод тайліенгу, %	З оптимальним розміром тайлу, %
Tmm	23,56263	17,39522
Dsyrk	23,55187	16,14898
Ssymm	16,59173	13,1425
Strmm	37,85709	21,2802

Продовження таблиці 7.1

Назва	Звичаний метод тайліенгу, %	З оптимальним розміром тайлу, %
Strsm	18,82737	13,85996
gemver	103,7182	67,5371
trisolv	17,4305	12,79438

З огляду на графік, можна дійти висновку, що метод тайліенгу дає близько 60% приросту в швидкодії. В то час, як оптимальний розмір тайлу дозволяє ще більше покращити ці показники, в середньому покращення сягає 75%

Для ще більшої оптимізації були застосовані техніки паралелізації OpenMP. Графіки відносного часу паралельних програм зображено на рисунку 7.17.

Так час виконання програм з увімкненою оптимізацією OpenMP і з використанням оптимальних розмірів тайлів дозволяє зменшити початковий час виконання на 77%.

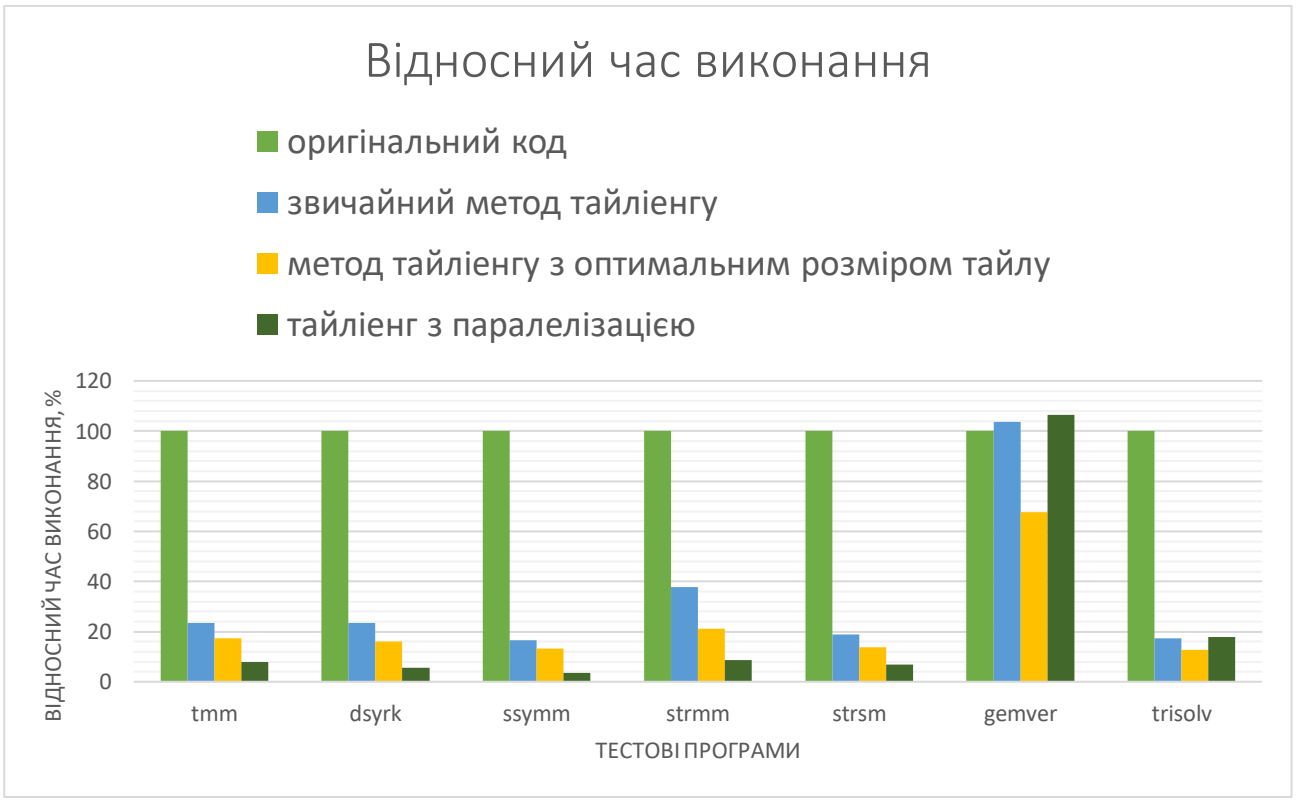


Рисунок 7.17 – порівняння відносного часу виконання

В таблиці зображений відносний час виконання програм до і після оптимізації.

Таблиця 7.2 -- відносний час виконання програми

Назва	Звичаний метод тайліенгу, %	З оптимальним розміром тайлу, %	Тайліенг з паралелізацією, %
Tmm	23,56263	17,39522	7,838792
Dsyrk	23,55187	16,14898	5,490657
Ssymm	16,59173	13,1425	3,548592
Strmm	37,85709	21,2802	8,541408
Strsm	23,56263	17,39522	6,893843
gemver	23,55187	16,14898	106,4235
trisolv	16,59173	13,1425	17,80805

ВИСНОВКИ

В результаті виконання магістерської дисертації було підвищено ефективність оптимізації додатків методом тайліенгу, за допомогою використання генетичного алгоритму, а саме:

- розроблено систему на базі програмного пакету Pluto, що запроваджує перетворення вихідного коду
- розроблено генетичний алгоритм, що розширює можливості Pluto, а саме пошук оптимального розміру тайлу
- виконано дослідження швидкодії програм з пакету PolyBench

Проведено дослідження розробленої системи та порівняння швидкодії програм, що не використовують оптимальний розмір тайлу.

Як результат, було досягнуто зменшення часу виконання програми, в середньому на 11%.

Крім того, що була розроблена система для автоматизації оптимізації додатків, було виконане ще:

- Написано програмний код, що дозволяє фіксувати найкращий варіант розмірів тайлів під час їх пошуку генетичним алгоритмом
- Написано програмний код, що дозволяє фіксувати середнє відхилення часу виконання програми під час роботи генетичного алгоритму
- Написано програмний код, що дозволяє фіксувати середній час виконання програми серед популяції під час роботи генетичного алгоритму
- Написано програмний код, що дозволяє провести дослідження розробленої системи одразу на багатьох варіантах програм
- Проведено дослідження та порівняння методів відбору генетичного алгоритму

Ключовою особливістю використання генетичного алгоритму є можливість застосовувати його для будь яких додатків без їх попереднього глибинного

аналізу. Також немає необхідності модифікувати програмний пакет Pluto. Данихй підхід використовує Pluto як незалежний модуль, що дає йому гнучкість.

За результатами проведених досліджень було опубліковано статтю:

Чемерис О. А., Душабаєв Р. Т. Оптимізація розміру тайлу при розпаралелюванні вкладених циклів за рахунок використання генетичного алгоритму // Міжнародний науковий журнал "Інтернаука". — 2022. — №5

ЛІТЕРАТУРА

1. Efficient Pipelining of Nested Loops: Unroll-and-Squash [Електронний ресурс] - Режим доступу: https://ieeexplore.ieee.org/document/1015491/

2. Smart Tiling for Program Optimization and Parallelization[Електронний ресурс] - Режим доступу: https://link.springer.com/chapter/10.1007/978-3-030-98883-8_2

3. TMTLS: Combine TM with TLS to Limit the Memory Contentions and Exploit the Parallelism in the Long-Running Transactions [Електронний ресурс] - Режим доступу: https://ieeexplore.ieee.org/document/6005433

4. Loop Selection for Multilevel Nested Loops Using a Genetic Algorithm[Електронний ресурс] - Режим доступу: https://www.hindawi.com/journals/mpe/2021/6643604/

5. Improving Locality and Parallelism in Nested Loops[Електронний ресурс] - Режим доступу: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.9865&rep=rep1&type=pdf

6. M. Griebl , C. Lengauer, “The Loop Parallelizer LooPo-Announcement”, In Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing

7. The Cache Complexity of Multithreaded Cache Oblivious Algorithms [Електронний ресурс] - Режим доступу: https://www.csd.uwo.ca/~mmorenom/CS433-CS9624/Resources/p271-frigo.pdf

8. Hybrid Hexagonal/Classical Tiling for GPUs [Електронний ресурс] - Режим доступу: https://dl.acm.org/doi/10.1145/2581122.2544160

9. Code generation for multiple mappings [Електронний ресурс] - Режим доступу: https://ieeexplore.ieee.org/document/380437

10. Iteration space tiling for memory hierarchies [Електронний ресурс] - Режим доступу:

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.578.5640&rep=rep1&type=pdf

11. effective automatic parallelization of stencil computations [Електронний ресурс] - Режим доступу: https://www.csa.iisc.ac.in/~udayb/publications/uday-pldi07.pdf

12. On the Scalability of Loop Tiling Techniques [Електронний ресурс] - Режим доступу:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.298.2615&rep=rep1&type=pdf

13. Tiling stencil computations to maximize parallelism [Електронний ресурс] - Режим доступу:https://ieeexplore.ieee.org/document/6468470

14. Scanning polyhedra with do loops [Електронний ресурс] - Режим доступу:https://hal-mines-paristech.archives-ouvertes.fr/hal-00752774/document

15. Parameterized loop tiling [Електронний ресурс] - Режим доступу: http://web.cs.ucla.edu/~pouchet/doc/cpc-article.10.pdf

15. PLUTO: An automatic polyhedral parallelizer and locality optimizer for multicores [Електронний ресурс] - Режим доступу: http://pluto-compiler.sourceforge.net

16. PC Project. Par4All initiative for automatic parallelization. [Електронний ресурс] - Режим доступу: https://pips4u.org/doc/meetings/2010-pips-developer-day-25-october-2010/PIPS-day-Par4All-RK-expose.pdf/at_download/file

17. Polyhedral parallel code generation for CUDA [Електронний ресурс] - Режим доступу: https://www.researchgate.net/publication/256121128_Polyhedral_Parallel_Code_Generation_for_CUDA

18. Parameterized tiling revisited [Електронний ресурс] - Режим доступу:https://www.researchgate.net/publication/220799099_Parameterized_Tiling_Revisited

19. Parametric multi-level tiling of imperfectly nested loops [Електронний ресурс] - Режим доступу:

https://www.researchgate.net/publication/221235799_Parametric_Multi-Level_Tiling_of_Imperfectly_Nested_Loops

20. DynTile: Parametric tiled loop generation for parallel execution on multicore processors [Електронний ресурс] - Режим доступу: https://www.researchgate.net/publication/220950296_DynTile_Parametric_tiled_loop_generation_for_parallel_execution_on_multicore_processors

21. An adaptive memory heuristic for a class of vehicle routing problems with minmax objective. Computers and Operations Research [Електронний ресурс] - Режим доступу: https://www.sciencedirect.com/science/article/pii/S0305054896000652

22. Coloration de graphes a l'aide de fourmis [Електронний ресурс] - Режим доступу: https://www.researchgate.net/publication/220937444_Optimisation_par_colonies_de_fourmis_pour_la_programmation_logique_etendue

23. Intelligent structural operators for the k-way graph partitioning problem [Електронний ресурс] - Режим доступу: https://www.researchgate.net/publication/46399834_Intelligent_Structural_Operators_for_the_k-way_Graph_Partitioning_Problem

24. A genetic algorithm approach to multi-fault diagnosis. In: Davis, L. (Ed.), Handbook of Genetic Algorithms.

25. A framework for the description of evolutionary algorithms [Електронний ресурс] - Режим доступу: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.513.4151&rep=rep1&type=pdf

26. Goldberg, D., 1989. Genetics Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, MA. Golden, B.L

27. A summary of research on parallel genetic algorithms. [Електронний ресурс] - Режим доступу: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=953C33197190C26A9A9D930E005AEC15?doi=10.1.1.49.4669&rep=rep1&type=pdf

28. A New Genetic Algorithm for Loop Tiling [Електронний ресурс] - Режим доступу: https://link.springer.com/article/10.1007/s11227-006-6367-9

29. Selection Methods for Genetic Algorithms [Електронний ресурс] - Режим доступу: https://www.researchgate.net/publication/259461147_Selection_Methods_for_Genetic_Algorithms

30. DEAP: A Python Framework for Evolutionary Algorithms [Електронний ресурс] - Режим доступу: https://www.researchgate.net/publication/235707002_DEAP_A_Python_framework_for_Evolutionary_Algorithms

ДОДАТОК А

Технічні науки

УДК 32.973.3

Чемерис Олександр Анатолійович

д.т.н., с.н.с, професор факультету інформатики та обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського»

Чемерис Александр Анатольевич

д.т.н., с.н.с, професор факультета информатики и вычислительной техники Национального технического университета Украины «Киевский политехнический институт имени Игоря Сикорского»

Alexander Chemeris

D.Sc, professor of the Faculty of Informatics and Computer Science of the National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"

Душабаєв Рустам Толкинбайович

студент 6 курсу факультету інформатики та обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського»

Душабаев Рустам Толкынбаевич

студент 6 курса факультета информатики и вычислительной техники Национального технического университета Украины «Киевский политехнический институт имени Игоря Сикорского »

Rustam Dushabaiev

student of the 6th year Faculty of Informatics and Computer Science of the National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"

ОПТИМІЗАЦІЯ РОЗМІРУ ТАЙЛУ ПРИ РОЗПАРАЛЕЛЮВАННІ ВКЛАДЕНИХ ЦИКЛІВ ЗА РАХУНОК ВИКОРИСТАННЯ ГЕНЕТИЧНОГО АЛГОРИТМУ

ОПТИМИЗАЦИЯ РАЗМЕРА ТАЙЛА ПРИ РАЗПАРАЛЛЕЛИВАНИИ ВЛОЖЕННЫХ ЦИКЛОВ ЗА СЧЕТ ИСПОЛЬЗОВАНИЯ ГЕНЕТИЧЕСКОГО АЛГОРИТМА

OPTIMIZATION OF THE TILE SIZE FOR PARALLELIZING NESTED LOOPS USING THE GENETIC ALGORITHM

Анотація. *Розглядається метод розбиття вкладених циклів на тайли та проблема пошуку оптимального розміру тайлу за допомогою генетичного алгоритму. Пропонується використання програмного пакету PLUTO – інструмент для трансформації вкладених циклів. Даний пакет не запроваджує механізми для пошуку оптимальних розмірів тайлів. В ході розробки системи оптимізації були розглянуті існуючі рішення даної проблеми. Приведено результати роботи системи на. В якості результатів роботи системи були представлені заміри часу виконання тестових програм до оптимізації та після. Програми були взяті на базі перевірочної колекції широко використовуваних алгоритмів різних класів PolyBench.*

Ключові слова: *тайлінг, генетичний алгоритм, PLUTO, PolyBench*

Аннотация. *Рассматривается метод разбиения вложенных циклов в тайлы и проблема поиска оптимального размера тайла с помощью генетического алгоритма. Предлагается использование программного пакета PLUTO – инструмент для трансформации вложенных циклов. Данный пакет не вводит механизмы поиска оптимальных размеров тайлов. В ходе разработки системы оптимизации были рассмотрены существующие решения данной проблемы. Приведены результаты работы системы. В качестве результатов работы системы были представлены замеры времени выполнения тестовых программ до оптимизации и после. Программы были взяты на базе проверочной коллекции широко используемых алгоритмов разных классов PolyBench.*

Ключевые слова: *тайлинг, генетический алгоритм, PLUTO, PolyBench*

Abstract. *The method of dividing nested cycles into tiles and the problem of finding the optimal tile size using a genetic algorithm are considered. It is proposed to use the software package PLUTO - a tool for transforming nested loops. This package does not introduce mechanisms for finding the optimal tile size. During the development of the optimization system, the existing solutions to this problem were considered. The results of the system on. As a result of the system, measurements of the execution time of test programs before optimization and after were presented. The programs were taken on the basis of a test collection of widely used algorithms of different classes of PolyBench.*

Key words*: tailing, genetic algorithm, PLUTO, PolyBench*

Вступ

Проблема автоматичного створення ефективних програм користувача є дуже актуальною. Найбільш гостро питання покращення ефективності виконання додатків користувача стоїть для вбудованих та мобільних систем, особливо для тих, що працюють на базі багатоядерних процесорів. На сьогоднішній день такі системи отримали широке розповсюдження завдяки розвитку програмного та апаратного забезпечення. Компілятори програмного забезпечення розвиваються та постійно покращують техніки оптимізації програмного коду, однак існує множина задач, що потребує додаткових оптимізацій. Для таких задач є алгоритмічне рішення, що потребує використання багатовимірних циклів. Одним із методів оптимізації таких частин алгоритму є метод розбиття вкладених циклів на тайли.

Особливістю даного методу являється покращення ефективності виконання програм користувача без необхідності залучення сторонніх бібліотек, а також не використовує паралелізм, тож він може бути застосований для програм, що будуть виконуватись на одноядерних процесорах.

Так як пошук оптимального розміру тайлу є NP повною задачею [1], то буде логічним використати певний нечіткий алгоритм. В даній роботі пропонується

використати генетичний алгоритм. Хоча існують і інші погляди щодо вирішення даної задачі.

В [2] автори обраховують розмір тайлу для двох вкладених циклів в компіляторі IBM XL Fortran мінімізуючи функцію, що залежить від рядків кешу, що не перекривають одна одну та сторінок в буфері асоціативної трансляції, до яких звертається розбиті на тайли цикл. Для більших порядків вони використовували ітераційний підхід. Так вони знаходили розміри тайлів для зовнішніх циклів, а потім для двох останніх вирішувалась задача мінімізації.

Пізніше були створені фреймворки багатогранної автоматичні трансформації, такі як Pluto та PPCG, які ефективно вирішують задачу розбиття на тайли. Однак, вони не мають жодної моделі вибору розміру тайлу. Підходи, що були описані в [3, 4, 5] націлені на пошук розміру тайла для подальшого їх використання разом зі згаданими вище фреймворками. Однак, в них налічуються наступні недоліки:

- знаходять однакові розміри тайлів для всіх вимірів
- задають обмеження на простір пошуку розмірів тайлів

Існує ще пласт рішень вибору розміру тайлу для окремих випадків. Наприклад, такі бібліотеки як OpenBLAS, MKL, Eigen [6, 7, 8] покладаються на вручну оптимізовані реалізації, в яких розмір тайлів підлаштований інженерами під різний розмір об’єму оброблюваних даних. Існують спроби автоматизувати дані оптимізації, наприклад, LAPACK [9], PHiPAC [10], ATLAS [11]. Вони ітеративно підбирають розмір тайла та порядок циклів намагаючись знайти варіант, який буде давати максимальну швидкодію. Всі ці бібліотеки мають високу швидкодію на великих масивах даних, але можуть давати гірші результати для невеликих матриць як показано в роботі [12].

Існують також заточені під конкретні задачі мови програмування та компілятор під конкретні задачі. Так, FLAME [13] розроблений з метою оптимізувати суто обрахунки лінійної алгебри. Нажаль він вимагає від програміста розбивати складні матричні операції на більш прості блоки, які потім

замінюються на оптимальну бібліотечну реалізацію. До того ж відповідальність за те, щоб забезпечити оптимальний розмір тайлу та повністю використати потенціал даних бібліотек повністю лягає на програміста.

Оптимізація розміру тайлу

Метод тайлінгу заключається в тому, щоб фрагментувати цикли на блоки меншого розміру. Такий підхід забезпечує краще використання ресурсів обчислювальної техніки, а саме більш оптимальне застосування кеш памяті.

Аби краще зрозуміти проблему та суть методу, пропонується переглянути приклад двох вкладених циклів (рис. 1), що проходять по всім елементам масиву без використання будь яких оптимізацій.

```
for (i=1; i<NMAX; i++) {
  for (j=0; j<NMAX; j++) {
    for (k=0; k<i; k++) {
      b[j][k] += a[i][k] * b[j][i];
    }
  }
}
```

Рис. 1. Три вкладених цикли до розбиття на тайли

Джерело: авторська розробка.

Результат роботи оптимізації методом тайлінгу (рис. 2) призводить до локалізації доступу до пам’яті, що покращує загальну ефективність виконання фрагменту коду.

```
for (t1=0;t1<=floord(NMAX-1,32);t1++) {
  for (t2=0;t2<=floord(NMAX-1,32);t2++) {
    for (t3=0;t3<=min(floord(NMAX-2,32),t2);t3++) {
      for (t4=32*t1;t4<=min(NMAX-1,32*t1+31);t4++) {
        for (t5=max(32*t2,32*t3+1);t5<=min(NMAX-1,32*t2+31);t5++) {
          for (t6=32*t3;t6<=min(32*t3+31,t5-1);t6++) {
            b[t4][t6] += a[t5][t6] * b[t4][t5];;
          }
        }
      }
    }
  }
}
```

Рис.2. Фрагмент коду після оптимізації методом тайлингу

Джерело: авторська розробка.

Даний фрагмент коду був згенерований засобами пакету PLUTO. PLUTO — це інструмент автоматичного розпаралелювання, заснований на багатогранній моделі. В термінах оптимізацій, що робить компілятор, даний метод запроваджує абстракцію для виконання високорівневих трансформацій, наприклад, оптимізацію вкладених циклів. За допомогою Pluto можна трансформувати код написаний на мові C. Трансформації в основному використовуються для ефективного розбиття циклів на тайли. [14]

Для деяких алгоритмів PLUTO пропонує евристично підібрані розміри тайлів. За замовчуванням використовується розмір тайлу рівний 32 для кожного виміру. Форма приймається за прямокутник.

Був заміряний час виконання оригінального фрагменту коду та розбитого на тайли. Для того щоб оцінити відносний приріст в швидкості виконання, час за який виконується оригінальний код був прийнятий за 100%. Тоді відносний час виконання оптимізованого коду обраховується за наступною формулою:

$$T = \frac{T_{tiled}}{T_{orignal}} * 100\%$$

де T – відносний час виконання оптимізованого коду, T_{tiled} – час виконання оптимізованого коду, $T_{original}$ – час виконання оригінального фрагменту коду.

Провівши заміри часу виконання обох фрагментів коду (рис. 3) та порівнявши результати, можемо бачити значне покращення часу.

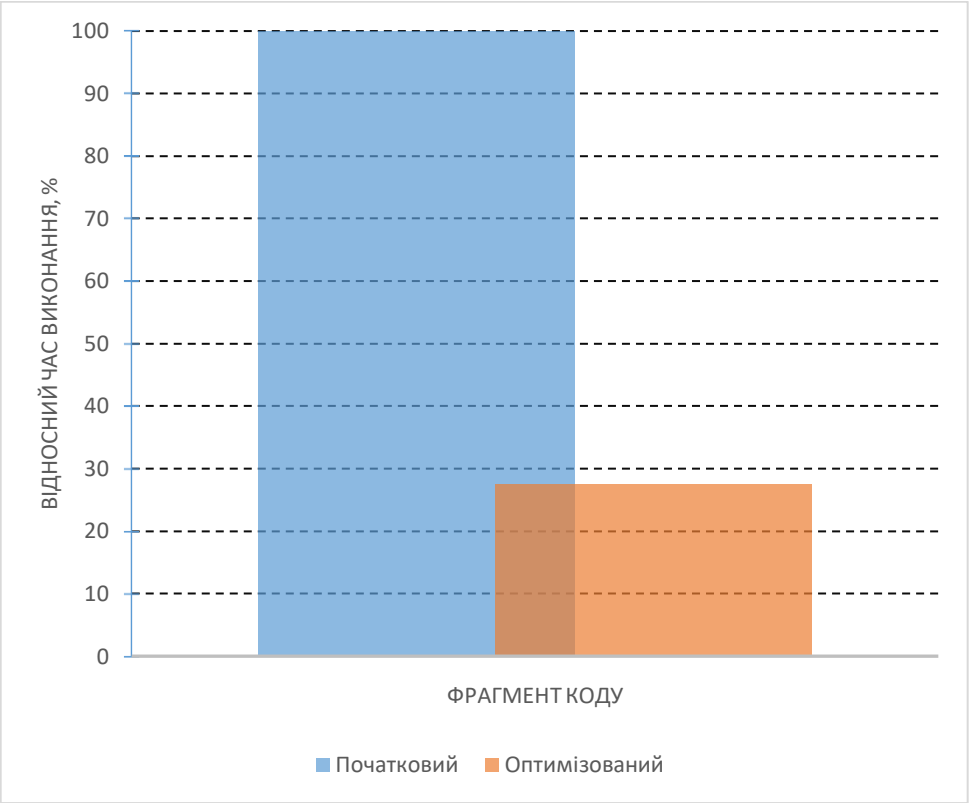


Рис. 3. Результати замірів роботи програм до і після оптимізації

Джерело: авторська розробка.

Так як проблема пошуку оптимального розміру тайлу є NP-повною, то було б доречно використати один з еволюційних алгоритмів. В даному випадку був використаний генетичний алгоритм (рис. 4).

Хромосомою буде виступати розмір тайлу. Кодування хромосоми було виконано вектором $size = (a_1\ a_2\ \cdots a_n)$, де $n -$ це кількість вкладених циклів, тобто розмірність.

Схрещування була представлена у вигляді комбінування відповідних компонент хромосоми за наступною формулою [15]:

$$a_i = l_i * \lambda + r_i * (1 - \lambda)$$

де $a_i - i$ компонента вектору розміру тайлу, $l_i, r_i - i$ компонента векторів хромосом, що схрещуються, $\lambda -$ коефіцієнт, що обирається виподковим чином, при чому $\lambda \in [0; 1)$

Мутація міняє місцями дві випадкові компоненти.

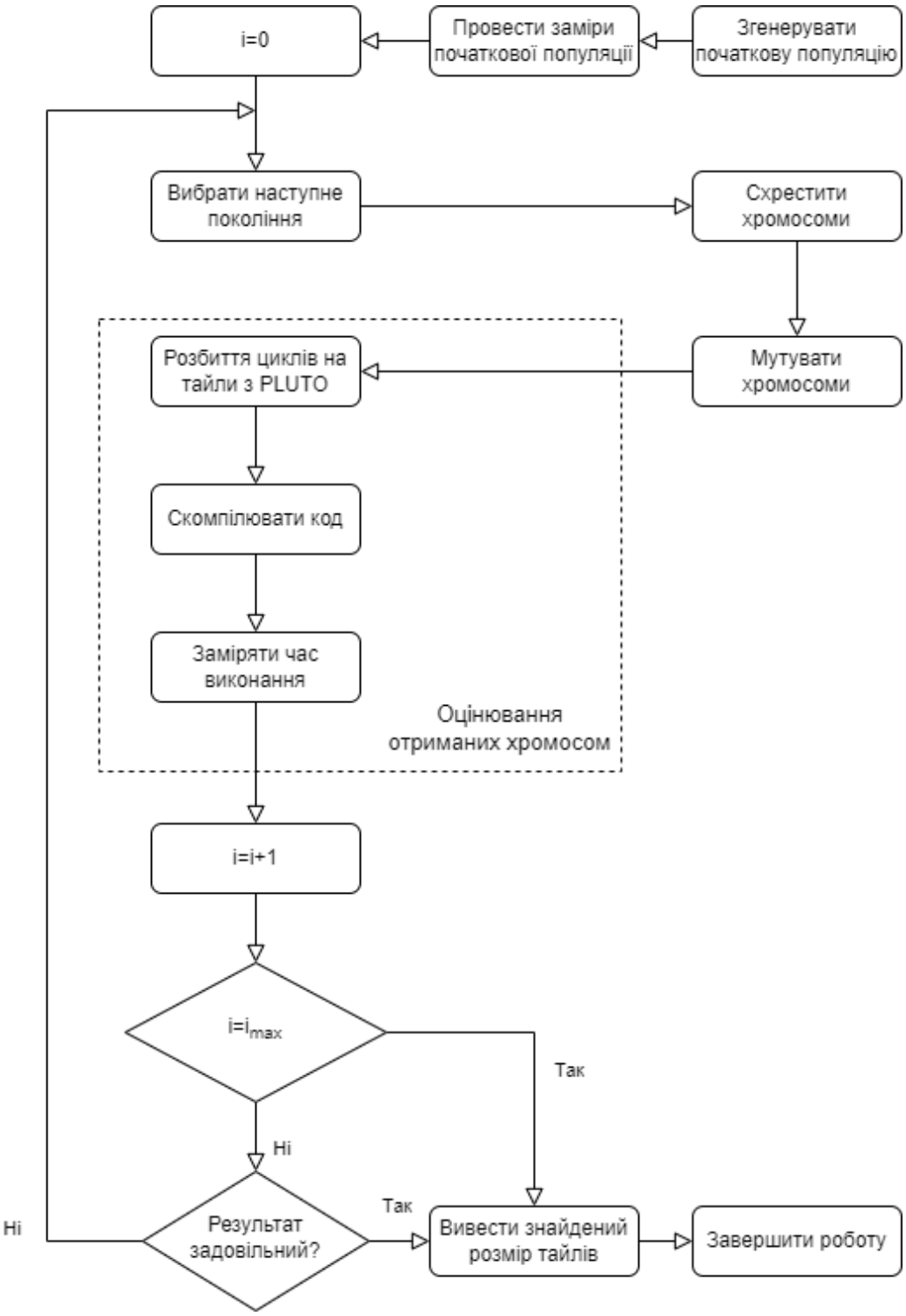


Рис. 4. Генетичний алгоритм

Джерело: авторська розробка.

В результаті роботи алгоритму був згенерований остаточний варіант розбиття (рис. 5) із найліпшим знайденим розміром тайлів.

```
for (t1=0;t1<=floord(NMAX-1,237);t1++) {
  for (t2=0;t2<=floord(NMAX-1,134);t2++) {
    for (t3=0;t3<=min(floord(67*t2+66,80),floord(NMAX-2,160));t3++) {
      for (t4=237*t1;t4<=min(NMAX-1,237*t1+236);t4++) {
        for (t5=max(134*t2,160*t3+1);t5<=min(NMAX-1,134*t2+133);t5++) {
          for (t6=160*t3;t6<=min(160*t3+159,t5-1);t6++) {
            b[t4][t6] += a[t5][t6] * b[t4][t5];;
          }
        }
      }
    }
  }
}
```

Рис. 5. Зненерований фрагмент коду з більш оптимальним розміром тайлу

Джерело: авторська розробка.

На графіку із замірами часу (рис. 6) видно покращення відносно випадкового розміру.

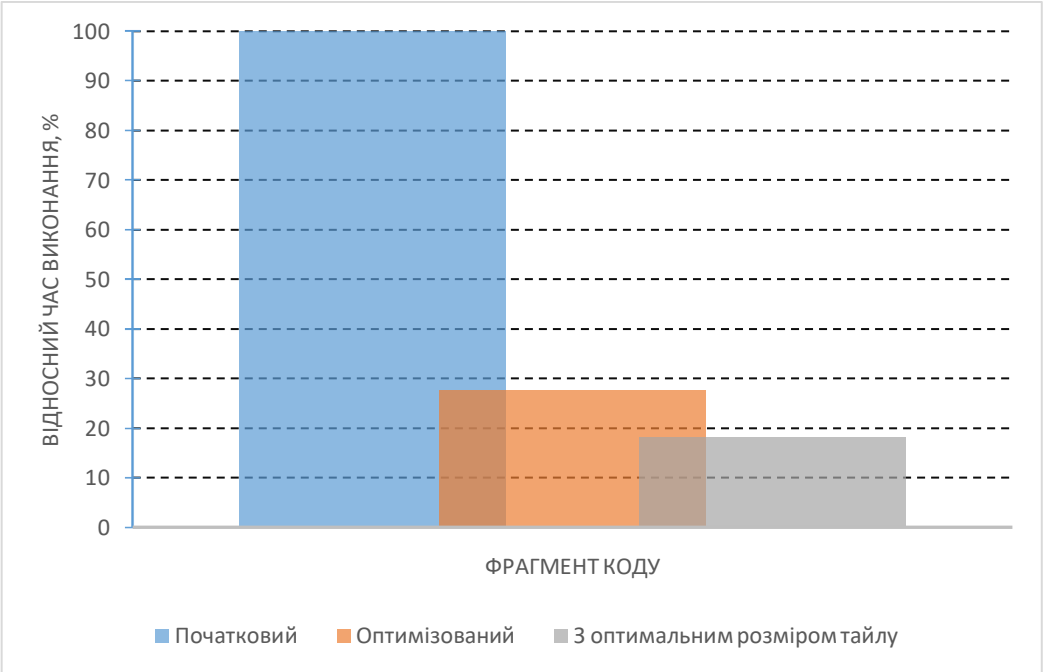


Рис. 6. Порівняння часу виконання оптимізованих фрагментів коду з розміром тайлу за замовчуванням та оптимальнім

Джерело: авторська розробка.

Таблиця 1

Результати замірів у відсотковому співвідношенні

Без оптимізацій	Розмір тайлу за замовчуванням	Оптимальний розмір тайлу
100	**27.53**	**18.11**

Джерело: авторська розробка.

З огляду на результати, можна дійти висновку, що генетичний алгоритм пошуку дозволив покращити ефективність роботи. Результати експериментів наведені в Таблиці 1.

Виходячи із специфіки проблеми і алгоритму, очевидно, що даний підхід не є вичерпним та можливі подальші оптимізації. Наразі алгоритм не змінює форму тайлу, щоб розширити множину пошуку. Це може збільшити час пошуку, але також може привести до кращих результатів.

Висновки

В даній науковій роботі запропоновано використання генетичного алгоритму для пошуку оптимального розміру тайлу при автоматичній трансформації та розпаралелюванні програм користувача для багатопроцесорних обчислювальних систем. Проведені експерименти, при яких визначався час виконання програми і проводився аналіз швидкодії програм до та після тайлінгу з порівнянням запропонованого методу оптимізованого тайлінгу, показують значний приріст швидкодії виконання програм користувача. Так, з послідовною програмою прискорення складає 5.5 разів, а в порівнянні зі звичайним тайліенгом – 1.5 рази.

Результати отримані в ході дослідження можуть використовуватися надалі у якості бази для подальших модифікацій та покращень методу розбиття на тайли.

Література

1. P. Boulet, D. Dongarra, Y. Robert, and F. Vivien, Tiling for heterogeneous computing platforms. 1998 [Електронний ресурс].

2. V. Sarkar and N. Megiddo. 2000. An Analytical Model for Loop Tiling and Its Solution

3. Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. 2013. Tile Size Selection Revisited

4. Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. 2012. Analytical Bounds for Optimal Tile Size Selection.

5. Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. 2010. Automatic Creation of Tile Size Selection Models

6. Eigen [n.d.]. A C++ template library for linear algebra. http://eigen.tuxfamily.org/ [Електронний ресурс].

7. MKL [n.d.]. Intel math kernel library (MKL). http://software.intel.com/en-us/intel-mkl. [Електронний ресурс].

8. Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs.

9. E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A Portable Linear Algebra Library for High-performance Computers

10. Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 2014. Optimizing Matrix Multiply Using PHiPAC: A Portable, Highperformance, ANSI C Coding Methodology

11. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2000. Automated Empirical Optimization of Software and the ATLAS Project.

12. Jaewook Shin, Mary Hall, Jacqueline Chame, Chun Chen, and Paul D. Hovland. 2009. Autotuning and Specialization: Speeding up Matrix Multiply for Small Matrices with Compiler Technology.

13. F. G. V. Zee, E. Chan, R. A. v. d. Geijn, E. S. Quintana-OrtÃŋ, and G. Quintana-OrtÃŋ. 2009. The libflame Library for Dense Matrix Computations.

14. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model / Uday Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. International Conference on Compiler Construction (ETAPS CC), Apr 2008, Budapest, Hungary.

15. D. E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.