Інформаційні технології

УДК 004.051

Нємцов Микита В'ячеславович
студент
Харківського національного університету радіоелектроніки

Каук Віктор Іванович
кандидат технічних наук, доцент кафедри програмної інженерії
Харківський національний університет радіоелектроніки

ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ, ЯКІ ВИКОРИСТОВУЮТЬСЯ У КОМПІЛЯТОРАХ КОДУ

Анотація. *У роботі було досліджено методи оптимізації, які використовуються при компіляції програмного коду, були досліджені методи вибору оптимальних параметрів компіляції, було виявлено їх переваги, недоліки та шляхи оптимізації.*

Ключові слова: *методи оптимізації програмного коду, компілятор, опції компіляції, генетика, алгоритм сходження до вершини, ітеративне перетворення.*

Проблема і актуальність дослідження. В більшості випадків, для налаштування компіляторів з використанням оптимізації достатньо використовувати спеціальні рівні оптимізації. Вони наявні в майже всіх новітніх компіляторах, та представляють собою набір налаштувань, які були визначені розробниками заздалегідь. Чим більший рівень оптимізації, тим сильніше вона змінює кодову базу для отримання вищої продуктивності. Окрім того, що користувач компілятора може використовувати рівні, йому ще доступні окремі опції, які він може налаштовувати і які мають вплив на окремі прийоми оптимізації.

Нажаль, така гнучкість використання, може обернутись проти самого розробника, оскільки не завжди застосування певного правила призводить до оптимізації. Так наприклад, в мові програмування C++ є можливість налаштувати компілятор таким чином, що він буде використовувати динамічну компоновку. Компоновкою називають процес об'єднання декількох модулів з бібліотек в один файл, який потім буде виконуватись. Динамічна компоновка дозволяє компілятору додавати модулі не перед етапом компіляції, а у момент виконання. Це повинно привести до скорочення розміру результативної програми. Однак існує такий спосіб оптимізації простору, який призведе до зниження продуктивності за рахунок необхідності включення бібліотеки перед виконанням функції.

Окрім того, що компілятори з використання оптимізації програмного коду інколи призводять до негативного результату, існує проблема пов'язана з типом системи під яку проходить транслювання. В залежності від типу цільової системи, оптимізації, які використовують компілятори можуть працювати з різним результатом. Саме тому, програмісти часто зустрічаються з ситуацією коли на цільовій і альтернативній платформах, приріст продуктивності програмного коду різний. В таких ситуаціях починають використовувати різні методи динамічного налаштування компілятора коду. Завдяки цьому, розробники програмного забезпечення можуть змінювати налаштування оптимізації компілятора, базуючись на цільовій платформі автоматично. Нажаль, такий підхід має низку проблем. Однією з найважливіших є те, що зазвичай не відомо який набір налаштувань необхідно виконувати на платформі. Тому пошук перетворюється на процес спроб та помилок. Така поведінка може завдати шкоди великому бізнесу, оскільки при переході на інші платформі та операційні системи, може бути зміна в продуктивності, а для знаходження набору підходящих налаштувань, необхідний час та гроші.

Через все зазначене вище, можна стверджувати, що питання вибору методів оптимізації компіляторів, які дають максимальний приріст до продуктивності та можуть бути використані на різних системах досі є відкритим та актуальним.

Огляд поточного стану об'єкту дослідження. Зараз існує декілька способів автоматичного налаштування параметрів оптимізації компілятора. Можна виділити три найбільш розповсюджених.

- Оптимізація з використанням машинного навчання. [1] Зараз, останньою розробкою в цьому напрямі є використання Байєсовської нейронної мережі для визначення функцій. Цей підхід широко використовують в таких інструментах як Cobayn [2]. Однак, даний спосіб налаштування оптимізації, має деякі недоліки. Перш за все, складність такого підходу в декілька разів вища за інші. Це пов'язано з необхідністю тренування та налаштування моделей, необхідністю мати знання в цій області. Також, зазначений підхід має недолік в тому, що час на навчання моделей є дуже високим. Загалом, тренувальні набори даних складаються з великої кількості опцій оптимізації, які ніяк не пов'язані між собою. Це призводить до того, що для навчання моделі, яка буде давати більш реальні та корисні дані про вибір оптимізації, необхідна велика кількість часу.
- Оптимізація з використанням профілів. Даний спосіб полягає у тому, що розробник вбудовує в свій додаток спеціальний код для збору інформації про виконання програми. Такий код може збирати різну інформацію про час виконання програми. До проблем зазначеного методу можна віднести час затрачений на визначення найліпшого профілю оптимізації. Це є результатом того, що компілятор повинен запустити програму та зняти метрики для того, щоб отримати інформацію про якість профілю.

- Ефективна адаптивна компіляція. Однією і мабуть найголовнішою проблемою ітеративного підходу був час, який необхідно було затратити на запуск та порівняння коду після кожної ітерації оптимізації. Це призводило до того, що процес можна було поділити на два однакових за часом процеси: аналіз статистичного звіту та порівняння. Вирішенням для цієї проблеми став віртуальний запуск скомпільованої програми. Без прямого запуску програми, аналізатор підраховує як багато інструкцій має виконуватись при певному сценарії і генерує статистичний звіт по цим даним. Такий підхід вже протягом довгого часу використовувався в аналізі покриття коду тестами. Використання такого додатку для оптимізації опцій компілятора дає зручний спосіб знаходження найбільш оптимального набору за придатний час, однак має шляхи поліпшення. Перш за все, можна покращити час знаходження оптимального набору за рахунок використання гібридних алгоритмів пошуку. Так, генетичний алгоритм пошуку [3] досить швидко знаходить рішення, яке задовольняє потребу, але для знаходження найкращого можливого набору параметрів можна використати інший алгоритм, такий як алгоритм пошуку сходженням до вершини. [4] Таким чином, можна використати спочатку один тип пошуку, для досягнення задовольняючого результату, а потім запустити інший алгоритм.

Мета дослідження. Метою даної роботи є отримання інформації про способи налаштування параметрів оптимізації при компіляції програмного коду, та оптимізація обраного способу. Незважаючи на те, що дослідження в цій темі проходить досить довго, були знайдені способи покращення існуючих рішень. Основною новизною є сформульований підхід щодо зменшення часу, затраченого на знаходження найліпшого набору параметрів оптимізації компілятора за допомогою комбінованого алгоритму пошуку.

Порівняння алгоритмів. Алгоритм сходження до вершини має хороші показники в локальному пошуку. Починаючи з визначення початкової групи, вирішуючи кращу пошукову область до ітерації від рівня до рівня, результати кожного рівня беруться з найкращих і потім порівнюються, щоб вони могли отримати більш оптимальні.

Генетичний алгоритм є потужним і гнучким метаевристичним, а також відносно новим типом алгоритму, приймаючи ідею природного відбору та генетичних змін природним шляхом. Цей алгоритм відомий як інструмент, який може вирішувати комбінаторні методи оптимізації, такі як проблема комівояжера.

Однак не зважаючи на їх можливе використання для вирішення одних проблем, дані алгоритми мають бути використані на різних масивах даних.

Як приклад, візьмемо класичну проблему комівояжера [5]. Всього будем використовувати в нашому тестуванні 8, 16 та 32 міста. Мовою програмування для реалізації алгоритмів оберемо C# і будемо використовувати бібліотеку «GAF», яка надає реалізовані генетичні алгоритми. Тестування проводиться на процесорі Intel Core i7 8750H, за тактовою частотою 2.21 GHz. Для налаштування генетичного алгоритму, будемо використовувати:

- популяцію 10;
- ймовірність перехрестя 60%;
- можливість мутації 50%;
- кількість генерацій 200.

Проведемо 20 тестів для кожної кількості міст. На рисунку 1 можна побачити результати виконання для 8 міст.

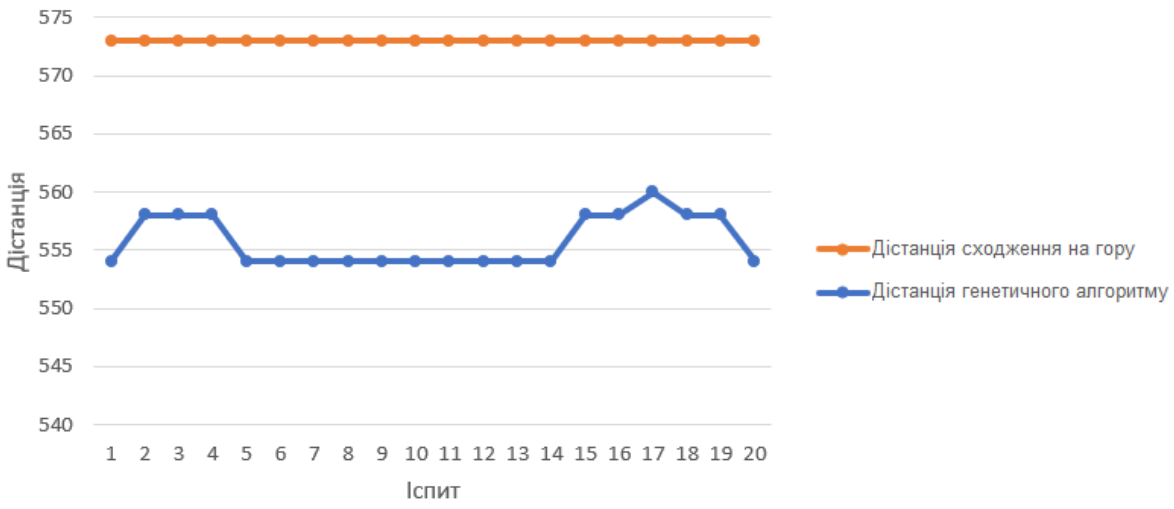

Рис. 1. Результати іспиту для 8 міст

Як можна побачити на першому тесті, при достатньо малому об'ємі даних, алгоритм сходження на гору дає більш оптимальний результат, оскільки всі прогони дають однакову вагу. Однак, варто відмітити, що результат генетичного алгоритму визначає більш дешевий шлях. Час виконання генетичного алгоритму – 820.412 секунд, для алгоритму сходження час склав 4.580. Результати для 16 міст на рисунку 2.

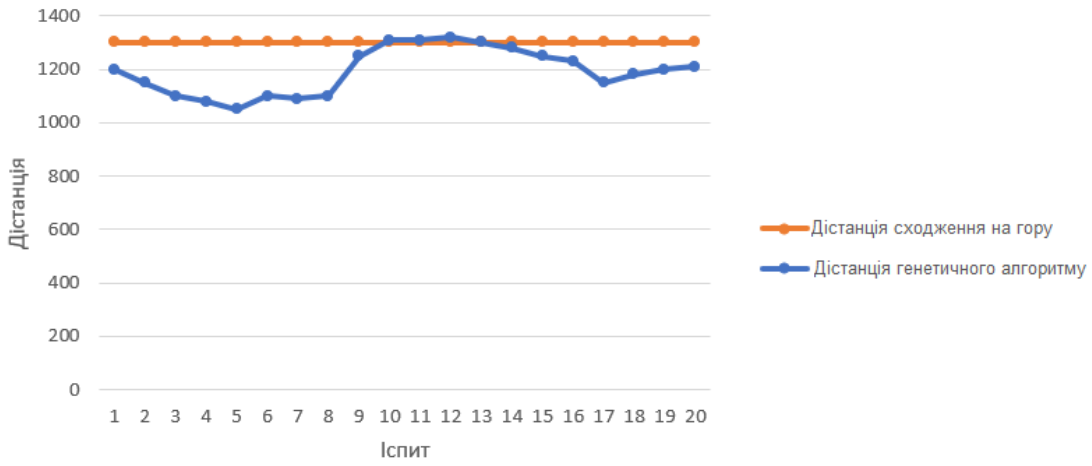

Рис. 2. Результати іспиту для 16 міст

При другому тестуванні, алгоритм сходження на гору знов дає більш оптимальний результат, оскільки всі прогони дають однакову вагу. Цього разу, генетичний алгоритм вже не знайшов більш оптимального шляху на кожному прогоні. Час виконання генетичного алгоритму – 3812.331 секунд,

для алгоритму сходження час склав 73.212 секунд. Результати для 32 міст на рисунку 3.

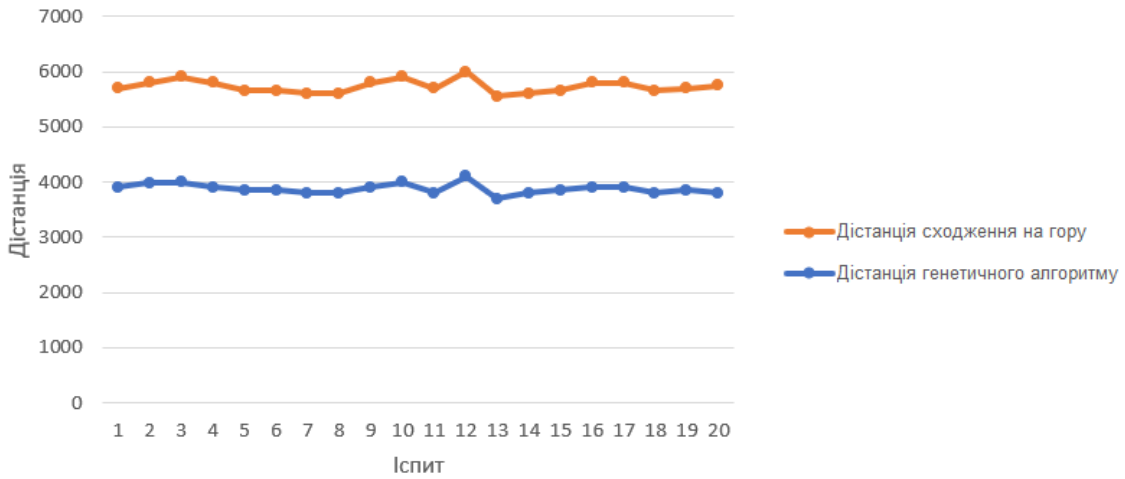

Рис. 3. Результати іспиту для 32 міст

Останній тест показав, що обидва алгоритми дали оптимальний результат. Час виконання генетичного алгоритму – 9875.311 секунд, для алгоритму сходження час склав 132.631 секунд.

Проаналізувавши результати тестувань двох алгоритмів можна зробити висновок, що генетичний алгоритм дає більш вдалі результати, але виконується повільніше за алгоритм сходження. В той час, алгоритм сходження є більш швидким за генетичний, але показав не найбільш оптимальний шлях. Це пов'язано з тим, що останній кожного разу знаходить точку локального мінімуму. Вирішенням цієї проблеми може стати обмеження кількість даних, та їх попередня фільтрація.

Зважаючи на проведений аналіз, можна зробити висновок, що використання алгоритму сходження для знаходження найбільш оптимального набору опцій налаштування компілятору є більш привабливим. Однак, для обмеження кількості вхідних даних та усунення можливих неточностей, слід використовувати генетичний алгоритм, який зможе знайти необхідний мінімум, який можна буде використати в якості вхідних даних до алгоритму сходження. Таким чином можна оптимізувати

алгоритм пошуку, підвищити його точність та мати можливість пошуку за придатний час.

Оптимізація адаптивної компіляції. Я було раніше зазначено, проблемою ефективної адаптивної компіляції є те, що використання одного методу пошуку призводить до різних проблем. Так, наприклад, використання генетичного алгоритму призводить до гарних результатів, проте час пошуку необхідного результату занадто великий і може бути оптимізований.

При використані алгоритму сходження на вершину, результати виконання оптимізації можуть бути отримані за помітно менший час, однак вони можуть бути не точними.

Виходячи з аналізу предметної галузі та використаних алгоритмів, було прийнято рішення використовувати гібридний алгоритм пошуку. Після аналізу двох алгоритмів, було вирішено використовувати генетичний алгоритм на початку пошуку для звуження області. Після того, як буде досягнута необхідна величина кількості параметрів чи буде перевищено час однієї ітерації, алгоритм буде передавати параметри, які залишились на обробку алгоритму сходження до вершини.

Такий підхід до обробки повинен покращити роботу двох алгоритмів. Генетичний алгоритм показує кращу якість результатів на великому об'ємі даних при меншій продуктивності. Як показало дослідження, алгоритм сходження до вершини дає більш неточні дані на великих об'ємах, але є швидшим за генетичний алгоритм.

В якості програмного продукту, який буде виконувати оптимізації, використаємо проект з відкритим кодом «ACOVEA» версії 4.0 [6], розроблений командою ентузіастів. Він дозволяє скомпілювати код з використанням ітеративної оптимізації. Окрім цього, він має можливість використання генетичного алгоритму для знаходження найбільш оптимального набору параметрів оптимізації.

Варто зазначити, що дане програмне забезпечення розповсюджується за ліцензією «GNU General Public License». Мета GNU GPL — надання користувачеві прав на копіювання, зміни й розповсюдження програми та зобов'язань, згідно з якими користувачі всіх похідних від неї програм теж отримають ці права.

Оскільки «ACOVEA» вже використовує генетичний алгоритм, нам необхідно реалізувати алгоритм сходження до вершини. Після того, як алгоритм сходження до вершини буде завершений, необхідно буде змінити додаток для того, щоб він мав змогу запуститись два рази. При першому прогоні, використовуємо генетичний алгоритм для знаходження «задовільного» рішення. Після отримання деяких даних, які зменшують область пошуків, додаток повинен використати алгоритм сходження для знаходження максимуму.

Тестування та аналіз результатів. Після завершення оптимізації бібліотеки, необхідно провести ряд тестувань для порівняння результатів оригінального коду та модифікованого. Тестування будемо проводити використовуючи наступні бібліотеки:

– fpppp. Тест квантової хімії, який вимірює продуктивність за одним стилем обчислення (двома інтегральними похідними електронів), що відбувається в серії програм GaussianXX. Він використовує дуже мало операцій вводу/виводу. Вхід до програми - кількість атомів. Кількість обчислень, які необхідно виконати, пропорційна четвертій потужності числа атомів.

– tomcatv. Tomcatv - це 200-лінійна програма породження сітки з набору тестів з плаваючою комою SPEC92. Tomcatv містить кілька петельних гнізд, які мають залежності по рядах масивів та інші петлі гнізда, які не мають залежності. Крім того, він має низьку ефективність кешу в гнізді циклу, що залежить від рядків, оскільки дані, до яких звертається кожен процесор, не є суміжними у спільному адресному просторі.

– fft. Швидке перетворення Фур'є (FFT) - алгоритм, який обчислює дискретне перетворення Фур'є (DFT) послідовності або його зворотну (IDFT). Найвідоміші алгоритми FFT залежать від факторизації N, але є FFT з O (N log N) складністю для всіх N, навіть для простих N.

Виконання оригінальної бібліотеки для зазначених вище алгоритмів, показало такі результати:

- fppp – середній час 4742.1 секунди;
- tomcat – середній час 2196.6 секунди;
- ft – середній час склав 2178.1 секунди;

Результати модифікованої версії для алгоритму «fpppp» представлені на рисунку 4.

Рис. 4. Результат fpppp тестування модифікованої версії

У результаті модифікації, алгоритм показав прискорення пошуку оптимальних опцій. Середній час складає 4406.1 секунди, що на 7.08% відсотків менше за попередній час. Результат «tomcatv» представлений на рисунку 5.

Рис. 5. Результат tomcatv тестування модифікованої версії

Як можна помітити з результатів тестування, при використанні комбінованого алгоритму, пошук оптимальних параметрів було прискорено на 7.97%. Середній час складає 2021.6 секунд. Результат для алгоритму «ft» представлений на рисунку 6.

Рис. 6. Результат ft тестування модифікованої версії

Останній тест також показав приріст продуктивності, скоротивши середній час пошуку до 2010.1 секунд, що становить 7.71% прискорення. Проаналізувавши отримані результати, можна стверджувати, що проведений аналіз існуючих алгоритмів є достовірним і використання комбінованого алгоритму прискорює пошук найкращих можливих параметрів оптимізації.

Згідно протестованим додаткам, середній приріст продуктивності складає 7-8%. Оскільки тестування було обмежене 100 ітераціями, приріст продуктивності не так сильно помітно, однак при використанні такого методу на реальних алгоритмах та додатках, економія часу буде помітною.

Однак для подальшого дослідження, використаний спосіб прискорення ітеративної компіляції, може бути випробуваний разом з нейронними мережами. Вже існують праці в яких досліджують та прототипують використання байєсовських мереж разом з ітеративними компіляторами. Оскільки ітеративний компілятор використовується в них для тренування моделей та підготовки набору тестових даних, оптимізація ніяк не вплине на швидкість вихідного компілятора, однак може допомогти в прискоренні навчання мережі.

Висновки. В процесі виконання роботи було досліджено методи оптимізації, які використовуються при компіляції програмного коду. Окрім цього були досліджені методи вибору оптимальних параметрів компіляції для досягнення найліпшої продуктивності.

В результаті, було оптимізовано вже існуюче рішення для тестування ітеративної компіляції. Було реалізовано гібридний алгоритм пошуку, який використовував генетичний алгоритм для звуження простору шуканих параметрів, а потім передавав результати роботи на обробку алгоритму сходження до вершини, який знаходив найоптимальніший набір опцій.

Результати отриманої роботи можна використовувати в двох напрямках. По-перше, описаний спосіб підвищення продуктивності може бути використаний в нових версіях ітеративних компіляторів. Іншим і більш перспективним напрямом використання результатів даної роботи є адаптація описаного алгоритму до використання разом з нейронними мережами.

Література

1. ACME: Adaptive Compilation Made Efficient / Computer Science Department of Rice University. URL: http://www.cs.cmu.edu/afs/cs/academic/class/15745-f09/www/papers/p69-cooper.pdf – 02.02.2005 р. – Назва з екрана.
2. COBAYN / Github. URL: https://github.com/amirjamez/COBAYN/ – 26.04.2020 р. Назва з екрана.
3. Голдберг Д. Genetic algorithms in search, optimization, and machine learning [Текст] / Голдберг Д – Addison-Wesley Professional, 2005. 432 с.
4. Поиск восхождением к вершине / Вікіпедія. URL: https://uk.wikipedia.org/. 12.12.2019 р. Назва з екрана.
5. Кук В. The Traveling Salesman Problem: A Computational Study [Текст] / Кук В. – Princeton University Press, 2007. 608 с.
6. ACOVEA / Github. URL: https://github.com/Acovea/libacovea/. 12.04.2020 р. Назва з екрана.