Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Гаврильченка Данила Олександровича*
(ПІБ)

академічної групи *122М-21-2*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

на тему: *Дослідження ефективності впровадження компіляторів мови програмування Python в цикл розробки ПЗ*

_____________________ *Д.О. Гаврильченко*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтингової	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	Проф. Алексєєв М.О.			

Рецензент				

Нормоконтролер	Проф. Лактіонов І.С.			

Дніпро
2022

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_______________ М.О. Алексєєв
(підпис) (прізвище, ініціали)

«   » ____________ 20 22 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122М-21-2* *Гаврильченка Данила Олександровича*
(група) (прізвище та ініціали)

Тема дипломного проекту *Дослідження ефективності впровадження компіляторів мови програмування Python в цикл розробки ПЗ*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 29.11.2022 р. № 2025-л

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес інтерпретації та компіляції програмного коду мови програмування Python.

Предмет досліджень – методи оптимізації коду та підвищення продуктивності програм, створених з використанням мови програмування Python.

Мета роботи – підвищення ефективності виконання програмного коду Python, зменшення часу роботи та пошук потенціальної економічної вигоди при впровадженні розробленої методики у процес розробки програмного забезпечення

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна отриманих результатів дипломної роботи визначається тим, що вперше проведення дослідження актуальних інструментів підвищення продуктивності виконання програм, створених мовою програмування Python.

Практична цінність результатів полягає у тому, що проведено практичне

тестування та оцінка ефективності альтернативних компіляторів, проведено економічне дослідження та сформовані критерії ефективності впровадження технологій у процес розробки ПЗ.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець)**
Аналіз теми та постановка задачі	01.04.2022-30.04.2022
Дослідження принципів внутрішніх процесів інтерпретації та пошук інструментів оптимізації програмного забезпечення	01.05.2022-31.08.2022
Створення програмного забезпечення для проведення тестувань та формування репрезентативних результатів виконання програм	01.09.2022-15.11.202

Завдання видав ____________ *Алексєєв М.О.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання ____________ *Гаврильченко Д.О.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *12.03.2022 р.*

Термін подання дипломного проекту до ЕК *20.12.2022 р.*

РЕФЕРАТ

Пояснювальна записка: 77 стор., 30 Рис. , 4 таблиці, 3 додатка, 40 джерел.

Об'єкт дослідження: процес інтерпретації та компіляції програмного коду мови програмування Python

Предмет дослідження: методи оптимізації коду та підвищення продуктивності програм, створених з використанням мови програмування Python

Мета магістерської роботи: підвищення ефективності виконання програмного коду Python, зменшення часу роботи та пошук потенціальної економічної вигоди при впровадженні розробленої методики у процес розробки програмного забезпечення

Методи дослідження. Для вирішення поставлених задач використані методи: аналізу даних, теорії внутрішнього влаштування інтерпретаторів та компіляторів, об'єктно-орієнтоване програмування.

Наукова новизна отриманих результатів кваліфікаційної роботи визначається тим, що вперше досліджені актуальні інструменти підвищення продуктивності виконання програм, створених мовою програмування Python.

Практична цінність результатів полягає у тому, що отримана оцінка ефективності альтернативних компіляторів та сформовані критерії ефективності впровадження технологій у процес розробки ПЗ.

У розділі «Потенціали економічної вигоди» проведено дослідження ринку розробки програмного забезпечення та наведено практичну вигоду при використанні запропонованих інструментів та методики

Список ключових слів: інтерпретатор, компілятор, продуктивність, Python, CPython, Cython, Numba, Taichi, Amazon Web Services.

ABSTRACT

Explanatory note: 77 pages, 30 pictures, 4 tables, 3 appendices, 40 sources.

The object of research: the process of interpretation and compilation of the program code of the Python programming language

The subject of research: methods of optimizing the code and improving the performance of programs created using the Python programming language.

Purpose of Master's thesis: to increase the efficiency of Python code execution, reduce work time and find the potential for economic benefit when implementing the developed methodology in the software development process.

Research methods. The following methods are used to solve the problems: data analysis, theories of internal configuration of interpreters and compilers, object-oriented programming.

Originality of research is determined by the fact that for the first time conducting a study of relevant tools for improving the performance of programs created in the Python programming language.

The practical value of the results arises from the fact that practical testing and evaluation of the effectiveness of alternative compilers was conducted, an economic study was conducted, and criteria for the effectiveness of the implementation of technologies in the software development process were formed.

In the "Potentials of economic benefit" section, a study of the software development market was conducted and the practical benefit of using the proposed tools and methodology was given.

Keywords: interpreter, compiler, performance, Python, CPython, Cython, Numba, Taichi, Amazon Web Services.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – програмне забезпечення;

ІПЗ – інженер програмного забезпечення;

LLVM – віртуальна машина низького рівня;

GIL – глобальне блокування інтерпретатора;

AWS – веб-сервіси «Амазон»;

HPC – обчислення з високою швидкістю;

ЗМІСТ

ЗМІСТ 7

ВСТУП 9

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ 12

1.1 Поняття інтерпретатора коду 12

1.2 Переваги компіляції над інтерпретацією 13

1.3 Принципи роботи стандартного інтерпретатора CPython 16

1.4 Можливості для оптимізації Python-програм 18

1.5 Завдання дослідження ефективності впровадження компіляторів в розробці ПЗ 24

РОЗДІЛ 2 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА МЕТОДИКИ ДЛЯ ПРОВЕДЕННЯ ПРАКТИЧНИХ ДОСЛІДІВ 26

2.1 Розробка алгоритму для створення лабораторного навантаження... 26

2.2 Опис та визначення складності алгоритму 26

2.3 Створення програми-бенчмарку з використанням CPython 30

2.4 Створення програми-бенчмарку з використанням Cython 32

2.5 Створення програми-бенчмарку з використанням Numba 34

2.6 Створення програми-бенчмарку з використанням Taichi 36

РОЗДІЛ 3 ПРОВЕДЕННЯ ТЕСТІВ З ПРОДУКТИВНОСТІ ТА ПОРІВНЯЛЬНІ РЕЗУЛЬТАТИ РОБОТИ ПРОГРАМ-БЕНЧМАРКІВ 39

3.1 Використовуване обладнання та конфігурація запуску програм 39

3.2 Проведення тестів програми, створеної з використанням CPython . 39

3.3 Проведення тестів програми, створеної з використанням Cython.... 41

3.4 Проведення тестів програми, створеної з використанням Numba.... 44

3.5 Проведення тестів програми, створеної з використанням Taichi 46
3.6 Результати тестів та оцінка ефективності 48
РОЗДІЛ 4 ПОТЕНЦІАЛИ ЕКОНОМІЧНОЇ ВИГОДИ 50
4.1 Дослідження ринку та можливих сфер впровадження 50
4.2 Принципи взаємодії та тарифікації хмарних сервісів 55
4.3 Очікувана вигода при використанні компіляторів в хмарному середовищі.. 57
ВИСНОВКИ 60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 61
ЛІСТІНГ ПРОГРАМИ 65
ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ 77

ВСТУП

Актуальність роботи. У зв'язку з переходом індустрії інформаційних технологій на відкритий світовий ринок та до жорстокої конкуренції з іншими компаніями різних розмірів та з різними можливостями, а також у зв'язку з швидким розвитком як апаратних, так і програмних інструментів, з'явилась необхідність міграції програмних комплексів підприємств з власного обладнання в інфраструктуру хмарних провайдерів.

Хмарні технології являють собою набір програмних інструментів, основний функціонал яких являється у наданні інженерам інтерфейсів для оренди та прямої взаємодії з різноріднім обладнанням. Завдяки подібній взаємодії, можна значно зекономити на купівлі вартісного серверного обладнання, хоча це далеко не єдиний привід для використання хмарних технологій.

Однією перевагою використання орендованої комп'ютерної інфраструктури хмарних провайдерів перед купівлею та обслуговуванням власного парку обчислювальної техніки є тарифікація оренди. Для певних технологій, постачальник обладнання створює можливість щосекундної оплати за використання понад продуктивних комп'ютерів. Наприклад, така можливість є корисною, коли необхідно провести алгоритмічно складні операції, час виконання яких може складати години на звичайній системі. Альтернативно, така монетизація зручна, якщо операція виконується не часто і орендувати окрему машину для розгортки програмного забезпечення невиправдано ані часом, ані вартістю.

Для подібних сценаріїв найчастіше використовують інтерпретовані або скриптові мови програмування, оскільки вони не потребують часу на компіляцію програмного коду у машинні інструкції. До того ж, через простоту використання та розгортки програмного оточення, створення подібного рішення не займає багато людського часу. Але, через необхідність віртуалізації для ітеративного виконання коду або скрипту, загальний час виконання програми зростає, що також веде до збільшення вартості використання послуги провайдера.

Таким чином, важливим науковим завданням є необхідність збільшення продуктивності та зменшення часу виконання програмного забезпечення без збільшення складності створення подібного програмного комплексу для інженера.

У зв’язку з цим, виникає необхідність розробки додаткових інструментів для оптимізації виконавчого коду, а також розвиток самих мов програмування у цьому напрямку.

Найбільш перспективним методом вирішення цієї проблеми є використання альтернативних інтерпретаторів або компіляторів для використовуваної мови програмування. Для мови програмування Python, яка дуже часто використовується у описаних раніше сценаріях, існує багато подібних інструментів з різними сферами застосування та різними умовами експлуатації.

Основні критерії для виправданого використання подібних інструментів є зменшення часу виконання програми без збільшення кількості та складності програмного коду з константними вихідними даними. Представлена робота є спробою такого дослідження для найбільш популярних інтерпретаторів мови Python.

Зв'язок роботи з державними програмами, планами науково-дослідних робіт. Робота виконана відповідно до закону України № 2623-14 від 11.07.2001 "Про пріоритетні напрями розвитку науки і техніки", "Державною комплексною програмою розвитку України".

Метою роботи і завданнями дослідження цієї роботи є формування критеріїв для оцінки інтерпретаторів та отримання об’єктивних даних, отриманих експериментально як результат виконання ідентичного програмного коду, що буде виконуватися відповідно до інструкцій з використання інтерпретаторів, порівняльний аналіз статистичних характеристик оцінок, отриманих різними методами, а також оцінка виправданості використання того чи іншого інтерпретатора в розробці. У роботі досліджені наступні інтерпретатори: “CPython”, “Cython”, “Numba” та “Taichi”.

Об'єкт досліджень – можливості використання мови програмування Python у якості інструменту для розробки високо продуктивного програмного забезпечення.

Предмет досліджень – методи зменшення загального часу виконання програмного коду, написаного мовою програмування Python за допомогою використання інтерпретаторів, створених або сумісних для роботи з цією мовою програмування.

Методи дослідження. При рішенні поставлених завдань виконаний аналіз і наукове узагальнення літературних джерел по початкових посилках досліджень.

Обґрунтованість і достовірність наукових положень, висновків і рекомендацій магістерської роботи обґрунтована коректністю поставлених проблем при аналізі існуючих рішень; обґрунтованістю початкових посилок, витікаючих з потреб сучасного ринку ПЗ та темпів розвитку апаратних та програмних рішень, застосованих в циклі розробки ПЗ.

Особистий внесок здобувача.

Автор самостійно сформулював мету, завдання дослідження, наукові положення і результати, виконав теоретичну і практичну частині роботи.

Структура та обсяг роботи. Робота складається з введення, трьох розділів і висновків. Містить 77 сторінок друкарського тексту, у тому числі 73 сторінки тексту основної частини з 30 малюнками, списку використаних джерел з 40 найменуваннями на 3 сторінках.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ

1.1 Поняття інтерпретатора коду

У комп'ютерних науках інтерпретатор — це комп'ютерна програма, яка безпосередньо виконує інструкції, написані мовою програмування чи мовою сценаріїв, без їх попередньої компіляції в програму машинною мовою. Інтерпретатор зазвичай використовує одну з таких стратегій виконання програми:

1. Розібрати вихідний код та виконати його поведінку безпосередньо;

2. Перетворити вихідний код на деяке ефективне проміжне подання або об'єктний код і негайно виконати його;

3. Явно виконати збережений заздалегідь скомпільований байт-код, створений компілятором і зіставлений з віртуальною машиною інтерпретатора.

Ранні версії мови програмування Lisp[1] та діалекти BASIC для міні-комп'ютерів та мікрокомп'ютерів можуть бути прикладами першого типу. Perl, Raku, Python, MATLAB та Ruby є прикладами другого типу, а UCSD Pascal – прикладом третього типу. Вихідні програми компілюються заздалегідь та зберігаються як машинно-незалежний код, який потім компонується під час виконання та виконується інтерпретатором та/або компілятором для JIT-систем. Деякі системи, такі як Smalltalk та сучасні версії BASIC та Java, також можуть поєднувати два та три. Інтерпретатори різних типів були створені для багатьох мов, традиційно пов'язаних з компіляцією, таких як Algol, Fortran, Cobol, C і C++.

Хоча інтерпретація і компіляція є двома основними засобами реалізації мов програмування, вони виключають одне одного, оскільки більшість інтерпретують систем також виконують деяку роботу з перекладу, як і компілятори. Терміни «інтерпретована мова» або «мова, що компілюється» означають, що канонічною реалізацією цієї мови є інтерпретатор або компілятор відповідно. Мова високого рівня ідеалі є абстракцією, незалежної від конкретних реалізацій.

```
C:\WINDOWS\system32\cmd.
Microsoft Windows [Version 10.0.22621.819]
(c) Microsoft Corporation. All rights reserved.

C:\Users\danho>python3.10
Python 3.10.8 (tags/v3.10.8:aaaf517, Oct 11 2022, 16:50:30) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello world!')
Hello world!
>>> x = 5 * 5 / 2
>>> y = 15 / 7 % 2
>>> print('x = ' + x + ' y = ' + y)
>>> print('x = ', x, ' y = ', y)
x =  12.5  y =  0.1428571428571428
>>>
```

Кожен оператор виконується один за одним, без необхідності збирати увесь пакет

Рис. 1. Приклад послідовного вводу команду у інтерпретуємій мові Python

Інтерпретатор зазвичай складається з набору відомих команд, які може виконувати, і списку цих команд у порядку, у якому програміст бажає їх виконувати. Кожна команда або інструкція містить дані, які програміст хоче змінити та інформацію про те, як змінити дані.

Інтерпретатори мають широкий спектр інструкцій, які спеціалізуються на виконанні різних завдань, але зазвичай можна знайти інструкції інтерпретатора для основних математичних операцій, розгалуження та управління пам'яттю, що робить більшість інтерпретаторів повними за Тюрінгом. Багато інтерпретаторів також тісно інтегровані зі “збирачем сміття” та відладчиком.

1.2 Переваги компіляції над інтерпретацією

Програми, написані мовою високого рівня, або безпосередньо виконуються будь-яким інтерпретатором, або перетворюються на машинний код компілятором, а також асемблером і компонувальником для виконання ЦП.

Хоча компілятори і асемблери зазвичай створюють машинний код, що безпосередньо виконується комп'ютерним обладнанням, вони часто можуть створювати проміжну форму, яка називається об'єктним кодом[2]. По суті, це той же машинно-специфічний код, але доповнений таблицею символів з іменами і тегами, щоб зробити блоки або модулі, що виконуються, ідентифікуються і переміщуються. Скомпільовані програми зазвичай використовують будівельні блоки або функції, що зберігаються у бібліотеці таких модулів об'єктного коду.

Компонувальник використовується для об'єднання попередньо створених файлів бібліотеки з об'єктним файлом програми для формування єдиного файлу, що виконується. Таким чином, об'єктні файли, які використовуються для створення виконуваного файлу, часто створюються в різний час, а іноді навіть різними мовами спроможних генерувати один і той же формат об'єкта.

Простий інтерпретатор, написаний мовою низького рівня (наприклад, мовою програмування Асемблер), може мати аналогічні блоки машинного коду, що реалізують функції мови високого рівня, збережені та виконувані, коли запис функції у таблиці пошуку вказує цей код. Однак інтерпретатор, написаний мовою високого рівня, зазвичай використовує інший підхід, такий як створення та подальший обхід дерева синтаксичного аналізу, або створення та виконання проміжних програмно-визначуваних інструкцій, або те й інше.

Таким чином, і компілятори, і інтерпретатори зазвичай перетворюють вихідний код на «токени», обидва можуть генерувати дерево синтаксичного аналізу і обидва можуть генерувати негайні інструкції (для стікової машини, четверного коду або іншими способами). Основна відмінність у тому, що система компілятора, включаючи компоновщик, генерує автономну програму машинного коду[3], тоді як система інтерпретатора натомість виконує дії, описані високорівневої програмою.

Основною перевагою компілятора є те, що компілятор може зробити майже всі перетворення семантики вихідного коду на машинний рівень раз і назавжди (тобто, поки не потрібно змінити програму), в той час як інтерпретатор повинен виконувати частину цієї роботи з перетворення щоразу, коли виконується оператор чи функція. Однак у ефективному інтерпретаторі більшість роботи з перекладу, включаючи аналіз типів тощо, виділяється і виконується лише за першому запуску програми, модуля, функції і навіть оператора, що дуже схоже на те, як працює компілятор. Однак у більшості випадків скомпільована програма, як і раніше, працює набагато швидше, частково тому, що компілятори призначені для оптимізації коду, і їм може бути надано достатньо часу для цього. Це особливо вірно для простих мов високого

рівня без великої кількості динамічних структур даних, перевірок чи типів.

При традиційній компіляції виконуваний файл компонувальника, який зазвичай є файл формату “.exe”, файли “.dll” або бібліотека, зазвичай переміщається при запуску під загальною операційною системою, як і модулі об'єктного коду, але з тією різницею, що це переміщення виконується динамічно під час виконання, тобто коли програма завантажується для виконання. З іншого боку, скомпіловані та пов'язані програми для невеликих вбудованих систем зазвичай розміщуються статично, часто жорстко закодовані у флеш-пам'яті NOR[4], оскільки в цьому сенсі часто немає вторинного сховища та операційної системи.

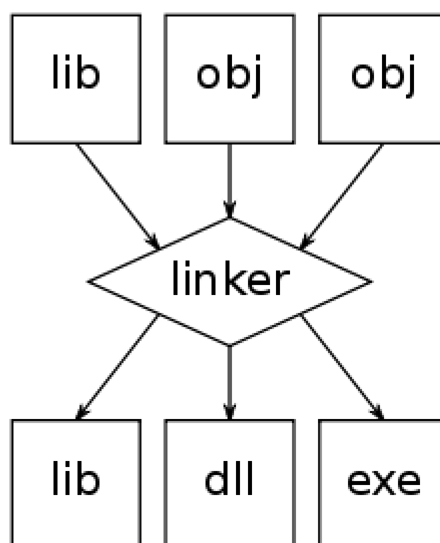


Рис. 2. Схема роботи компонувальника

Історично склалося так, більшість систем інтерпретатора мали вбудований автономний редактор. Це стає все більш поширеним також для компіляторів, хоча деякі програмісти вважають за краще використовувати редактор на свій вибір і запускати компілятор, компонувальник та інші компоненти. інструменти вручну. Компілятори передували інтерпретаторам, тому що апаратне забезпечення на той час не могло підтримувати як інтерпретатор, так і код, що інтерпретується, а типове пакетне середовище того часу обмежувало переваги інтерпретації.

1.3 Принципи роботи стандартного інтерпретатора CPython

CPython - еталонна реалізація мови програмування Python. Написаний на C і Python, CPython є стандартною та найбільш широко використовуваною реалізацією мови Python.

Рис. 3. Логотип мови програмування Python, по сумісності логотип інтерпретатора CPython

CPython можна визначити як інтерпретатор, і як компілятор, оскільки він компілює код Python в байт-код перед його інтерпретацією. Він має зовнішній інтерфейс функцій з кількома мовами, включаючи C, в якому необхідно явно писати прив'язки мовою, відмінною від Python.

Особливістю CPython є те, що він використовує глобальне блокування інтерпретатора (GIL)[5] для кожного процесу інтерпретатора CPython, що означає, що в одному процесі лише один потік може обробляти байт-код Python у будь-який час. Це не означає, що у багатопоточності немає сенсу: найбільш поширений сценарій багатопоточності - це коли потоки в основному очікують завершення зовнішніх процесів.

Це може статися, коли кілька потоків обслуговують окремих клієнтів. Один потік може очікувати відповіді від клієнта, а інший може очікувати на виконання запиту до бази даних, тоді як третій потік фактично обробляє код Python.

Однак GIL означає, що CPython не підходить для процесів, що реалізують алгоритми з інтенсивним використанням ЦП у коді Python, які потенційно

можуть бути розподілені по кількох ядрах.

У реальних додатках ситуації, коли GIL є суттєвим вузьким місцем, досить рідкісні. Це пов'язано з тим, що Python за своєю природою є повільною мовою і зазвичай не використовується для ресурсоємних чи критичних операцій. Python зазвичай використовується на верхньому рівні та викликає функції в бібліотеках для виконання спеціалізованих завдань. Ці бібліотеки зазвичай не написані на Python і код Python в іншому потоці може виконуватися під час виклику одного з цих базових процесів. Бібліотека не-Python, що викликається для виконання завдання, що інтенсивно використовує ЦП, не підпадає під дію GIL і може одночасно виконувати безліч потоків на декількох процесорах без обмежень.

Python-код, який пишеться, компілюється в байт-код Python, який створює файл з розширенням ".pyc". Компіляція байт-коду відбувалася всередині та майже повністю прихована від розробника. Компіляція – це просто етап трансляції, а байт-код – це низькорівневе та незалежне від платформи уявлення вихідного коду. Кожне з вихідних тверджень транслюється до групи інструкцій байт-коду. Ця трансляція байт-коду виконується для прискорення виконання. Байт-код може виконуватися набагато швидше ніж оператори оригінального вихідного коду.

Файл ".pyc", створений етапі компіляції, потім виконується відповідними віртуальними машинами[6]. Віртуальна машина - це просто великий цикл, який перебирає інструкції байт-коду одну за одною для виконання своїх операцій. Віртуальна машина — це механізм виконання Python, який завжди є частиною системи Python і є компонентом, який дійсно запускає сценарії Python. Технічно це лише останній крок того, що називається інтерпретатором Python.

Паралелізм коду Python може бути досягнутий лише за допомогою окремих процесів інтерпретатора CPython, керованих багатозадачною операційною системою. Це ускладнює взаємодію між паралельними процесами Python, хоча модуль багатопроцесорності дещо пом'якшує цю проблему; це означає, що програми, які дійсно можуть виграти від одночасного виконання Python коду, можуть бути реалізовані з обмеженим обсягом накладних витрат.

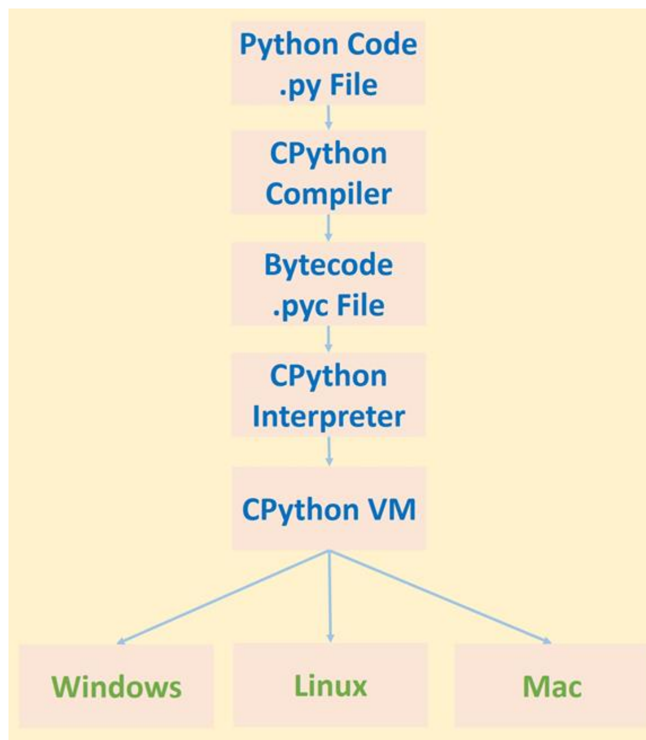


Рис. 4. Схема процесу інтерпретації, при використанні CPython

Наявність GIL спрощує реалізацію CPython та спрощує реалізацію багатопотокових додатків, які не отримують переваг від одночасного виконання коду Python[7]. Однак без GIL багатопроцесорні програми повинні переконатися, що весь загальний код є безпечним.

Хоча було зроблено багато пропозицій щодо скасування GIL, на загальну думку, у більшості випадків переваги GIL переважують недоліки; у тих небагатьох випадках, коли GIL є вузьким місцем, програма повинна будуватися навколо багатопроцесорної структури.

1.4 Можливості для оптимізації Python-програм

Окрім раніше описаної проблеми з GIL, існує також проблема того, як зміні зберігаються у об’єктній моделі Python. Для кращого розуміння, порівняємо процес створення змінної в мові програмування C та Python.

Операція зі створення цілочислової змінної в мові C виглядає наступним чином:

$$int\ number = 117;$$

Коли виконується цей рядок коду, комп'ютер робить наступне:

1. Виділяється достатньо пам'яті для цілого числа за певною адресою (розташування в пам'яті);

2. Призначується значення 117 до місця пам'яті, виділеного на попередньому кроці;

3. Указнику number привласнюється адреса цього значення[8].

Після цього, буде створено зміну, яка виглядатиме як на рисунку 5.

number	
address	*0x1aa2f*
value	117

Рис. 5. Представлення цілочислової змінної під назвою number зі значенням 117 у мові C

Якщо ми призначаємо новий номер number, ми записуємо новий номер на ту саму адресу; перезапис, попереднє значення. Це означає, що змінна є змінною. Зверніть увагу, що адреса (або місце в пам'яті) не змінилося. Розмір нового значення не перевищує ємкість типу об'явленої змінної. Але навіть при зворотному, саме значення змінної було б переповнене, нової змінної або розширення існуючої не відбулося б.

number	
address	*0x1aa2f*
value	256

Рис. 6. При зміні значення змінної, вона зберігає свою адресу

Повторимо той самий процес, але тепер у мові Python.

$$number = 117$$

При інтерпретації цього рядку коду, комп'ютер робить наступне:

1. Створюється PyObject; виділяється достатня кількісті пам’яті для адреси;

2. Встановлюється код типу PyObject на ціле число (як визначено інтерпретатором);

3. Встановлюється для PyObject значення 117;

4. Створюється назва number;

5. Створюється посилання на number;

6. Збільшується лічильник посилань PyObject на 1[9].

Наведені вище дії створюють (спрощено) об’єкти в пам’яті нижче:

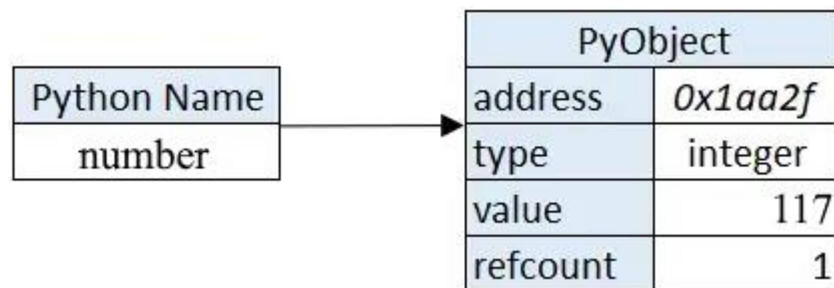


Рис. 7. Представлення цілочислової змінної під назвою number зі значенням 117 у мові Python

Можна відразу помітити, що виконується більше кроків і потребується більше пам’яті для зберігання цілого числа. Окрім типу та значення, віртуальна машина також зберігає “refcount” для збирання сміття. Можна також помітити, що змінна, яку було створено, не володіє блоком пам’яті. Пам'яттю володіє щойно створений PyObject, на який вказує number.

Відтепер, аналогічно призначимо інше значення змінній number, але тепер в мові Python. При цьому, інтерпретатор виконує наступні дії:

1. Створюється новий PyObject за певною адресою, виділяючи достатньо пам’яті;

2. Встановлюється код типу PyObject на ціле число;

3. Встановлюється для PyObject значення 256 (нове значення);

4. Створюється вказівник на нову змінну number;

5. Збільшується refcount нового PyObject на 1;

6. Зменшується підрахунок старого PyObject на 1.

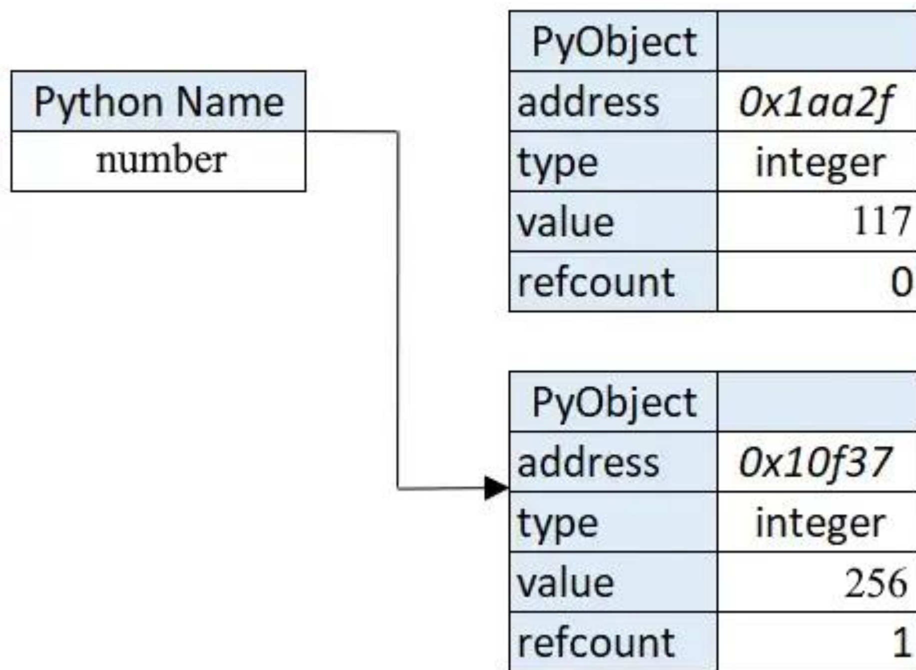


Рис. 8. Нове відображення тієї самої змінної при зміні її значення у Python

Зображення вище демонструє, що замість того, щоб призначати нове значення number, інтерпретатор прив'язує ім'я number до нового об'єкта. Таким чином можна також призначити значення іншого типу, оскільки кожен раз буде створюватися новий PyObject[10]. Number просто вказує на інший PyObject.

Також, необхідно зауважити, що refcount для старого об'єкта дорівнює 0; це гарантує, що збирач сміття його очистить.

Можна зробити висновок, що основними проблемами для швидкості виконання є:

Інтерпретація: компіляція та інтерпретація відбувається під час виконання через динамічну типізацію змінних. З тієї ж причини інженеру, коли потрібно створити новий PyObject, вибирається адреса в пам'яті та виділяється достатньо пам'яті кожного разу, коли створюється або «перезаписується» змінна

для нового PyObject.

Однопоточність: спосіб, яким розроблено збирання сміття, змушує використовувати GIL: обмеження всього виконання одним потоком на одному ЦП.

Саме для вирішення цих проблем було створено альтернативні компілятори, які вирішують ці проблеми. Деякі зовнішні бібліотеки, такі як Numpy, використовують ефективні реалізації C++, які можуть прискорити стандартні завдання. Розглянемо на прикладі цього компілятора, яким чиним підвищується продуктивність.

Numba генерує оптимізований машинний код із чистого коду Python за допомогою інфраструктури компілятора LLVM[11]. Швидкість коду під час використання Numba порівнянна зі швидкістю аналогічного коду на C, C++ або Fortran.

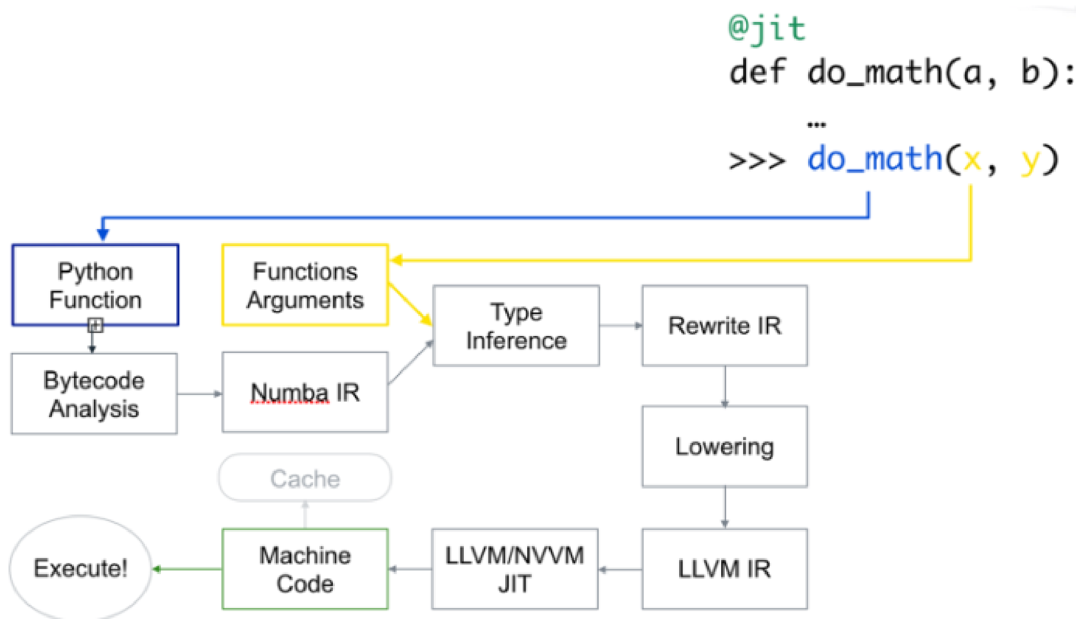


Рис. 9. Схема, яка демонструє принцип роботи компілятора Numba

Спочатку функція Python береться, оптимізується та перетворюється на проміжне представлення Numba. Потім, у результаті визначення типу, він перетворюється на інтерпретований код LLVM. Потім цей код передається до

JIT-компілятора LLVM для видачі машинного коду. JIT-компіляція також може бути виконана в автономному режимі для підвищення продуктивності[12].

Також, поглянемо, як компілятор Cython вирішує оголошені раніше проблеми.

Коли Cython компілюється, він надає модулі розширення CPython. Він забезпечує менші обчислювальні витрати, ніж Python під час виконання. Коди C і C++ можуть бути поміщені в модулі Cython. Cython залежить від інтерпретатора Python та стандартної бібліотеки. Cython використовує оптимістичну оптимізацію, необов'язкове виведення типів, низькі накладні витрати на керуючі структури та низькі накладні витрати на виклики функцій. Його продуктивність залежить від генерації та реалізації кодів C.

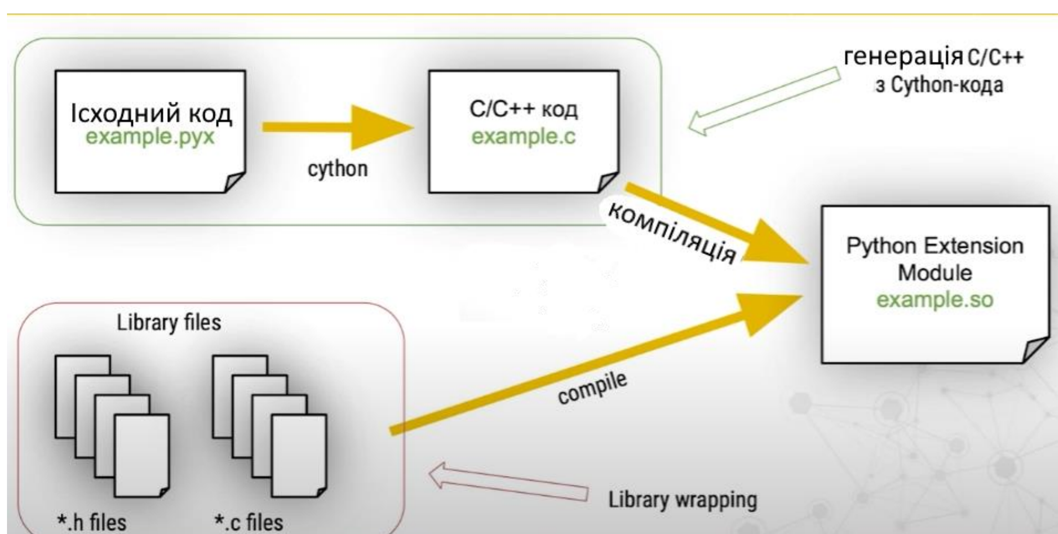


Рис. 10. Схема, яка демонструє принцип роботи компілятора Cython

На першому етапі код Cython перетворюється на еквівалентний оптимізований і незалежний від платформи код C або C++. Звідти вихідний код C або C++ перетворюється на спільний об'єктний файл за допомогою компілятора C або C++. Однак цей загальний файл залежить від платформи. Він має розширення *.so у Linux або Mac OS і розширення *.pyd у Windows[13].

Taichi, це відносно молода розробка, спрямована на розширення можливостей Python завдяки трансляції операторів з CPU на GPU.

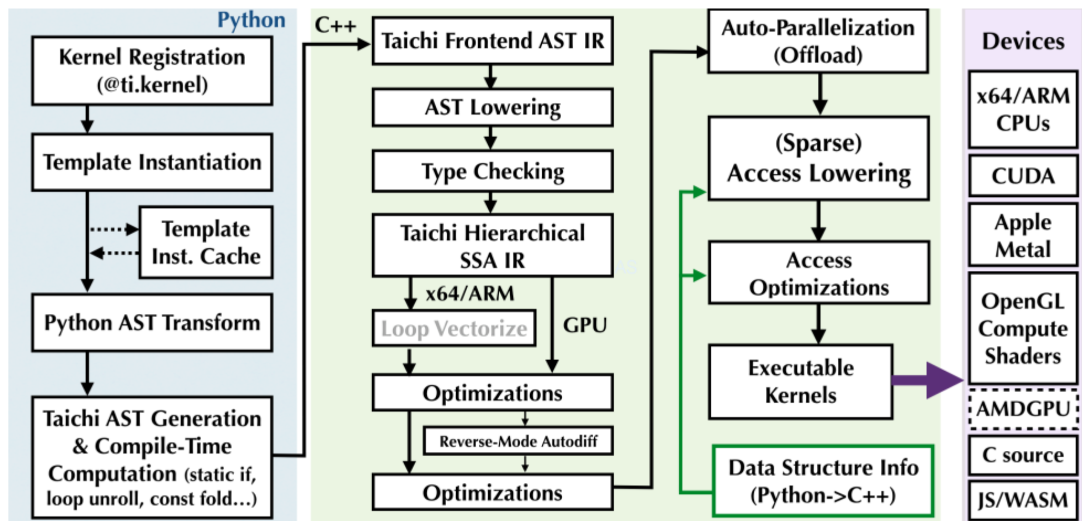


Рис. 10. Схема, яка демонструє життєвий цикл ядра Taichi

Однією з початкових цілей проектування є відокремлення обчислень від структур даних. Механізм, який надає Taichi, є набором універсальних контейнерів даних, званих SNode (/ˈsnoʊd/). SNode можна використовувати для зручного складання ієрархічних, щільних або розріджених багатовимірних полів. Перемикання між макетами "масив структур" і "структура масивів" зазвичай займає не більше 10 рядків коду. Це викликало безліч варіантів використання у чисельному моделюванні. Концепція поділу поширюється і систему типів. Оскільки обсяг пам'яті та пропускна спроможність графічного процесора в даний час стають основними вузькими місцями, важливо мати можливість упаковувати більше даних на одиницю пам'яті. З 2021 року Taichi представила квантовані типи, що настроюються, що дозволяють визначати числа з фіксованою або плаваючою комою з довільними бітами. Це дозволило моделювати MPM понад 400 мільйонів частинок на одному пристрої GPU[14].

1.5 Завдання дослідження ефективності впровадження компіляторів в розробці ПЗ

Незважаючи на значні кошти, витрачені українськими IT-компаніями на внутрішні дослідження та розробки, часто залишається постійний і тривожний розрив між внутрішньою цінністю технології, яку вони розробляють, та їхньою

здатністю ефективно змусити її працювати. За часів жорсткої глобальної конкуренції відстань між технічними обіцянками та реальними досягненнями викликає особливе занепокоєння. Ґрунтуючись на своєму тривалому дослідженні труднощів, з якими зіткнулися менеджери при скороченні цього розриву, можна все ще знайти багато ключових проблем, які мають подолати менеджери, відповідальні за впровадження нових технологій: їх неминуча подвійна роль, різноманітність внутрішніх ринків, які необхідно обслуговувати, законний опір змін, правильний ступінь просування, вибір місця реалізації та необхідність того, щоб одна людина брала на себе загальну відповідальність.

Впровадження технологічних змін до організації ставить перед керівництвом інший набір завдань, ніж робота з компетентного управління проектами. Однак часто менеджери, відповідальні за впровадження технічного нововведення у повсякденне використання, завдяки освіті та досвіду набагато краще підготовлені для управління розвитком цього нововведення, ніж для управління його впровадженням.

Для ефективного впровадження нової технології або методики у процес розробки, необхідно враховувати два основні фактори:

1. Необхідність додаткового інструктажу інженерів стосовно технічної реалізації та особливостей впровадження;

2. Очікувана дохідність та строки реалізації впровадження.

РОЗДІЛ 2
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА МЕТОДИКИ ДЛЯ ПРОВЕДЕННЯ ПРАКТИЧНИХ ДОСЛІДІВ

2.1 Розробка алгоритму для створення лабораторного навантаження

Для створення програми-бенчмарка, необхідно обрати алгоритм, який створить достатнє навантаження на систему та буде репрезентативно демонструвати якісні зміні в продуктивності роботи програми при зміні використуємого інструментарію.

В якості такого алгоритму було обрано побудову фракталу з множини Мандельброта на ірраціональній площині.

Набір Мандельброта — це набір таких точок c на комплексній площині, де повторюване відношення $z_n + 1 = z_n^2 + c$ для $z_0 = 0$ визначає певну послідовність. Іншими словами, це набір c, для якого існує таке дійсне R, що виконується нерівність $|z_n| < R|$ містить усі натуральні числа n. Визначення та назва походять від французького математика Адрієна Дуаді, на честь математика Бенуа Мандельброта[15].

Множина Мандельброта — один із найвідоміших фракталів, і він також не має математичних рамок завдяки своєму кольоровому дизайну. Його фрагменти не зовсім схожі на вихідний набір, але з багаторазовим збільшенням окремі фрагменти стають все більш схожими один на одного.

2.2 Опис та визначення складності алгоритму

Набір Мандельброта генерується шляхом ітерації, що означає повторення процесу знову і знову. У математиці цей процес найчастіше є застосуванням математичної функції. Для множини Мандельброта задіяні функції є одними з найпростіших, які тільки можна собі уявити: усі вони є так званими квадратичними поліномами та мають вигляд $f(x) = x2 + c$, де c — постійне число. По ходу ми точно вкажемо значення c.

Щоб виконати ітерацію $x2 + c$, ми починаємо з початкового числа для ітерації. Це число, яке ми записуємо як $x0$. Застосування функції $x2 + c$ до $x0$ дає нове число

$$x1 = x02 + c$$

Тепер ми виконуємо ітерацію, використовуючи результат попереднього обчислення як вхідні дані для наступного. Тобто:

$$x2 = x12 + c$$
$$x3 = x22 + c$$
$$x4 = x32 + c$$
$$x5 = x42 + c$$

і так далі. Список чисел $x0, x1, x2, \ldots$, створений цією ітерацією, має назву: він називається орбітою $x0$ під час ітерації $x2 + c$.

Теорія ітерованих функцій мотивована запитаннями з реального життя. Прикладом є моделювання зростання популяції тварин. Розмір популяції після одного циклу розмноження залежить від того, скільки тварин є на даний момент, тому математичні моделі зростання популяції зазвичай складаються з функції f(у) змінній x, де x представляє поточний розмір популяції, а $f(x)$ дає очікуваний розмір популяції після одного циклу розмноження. Щоб визначити розмір популяції після будь-якої кількості циклів розмноження, вам потрібно повторити функцію.

Замість того, щоб розглядати лише дійсні значення c, ми також дозволяємо c бути комплексним числом. Давайте розглянемо кілька прикладів ітерації $x2 + c$, коли c є комплексним числом: якщо $c = i$, то орбіта 0 під $x2 + i$ визначається як:

$$x0 = 0$$
$$x1 = 0^2 + i = i$$
$$x2 = i^2 + i = -1 + i$$
$$x3 = (-1 + i)^2 + i = -i$$
$$x4 = (-i)^2 + i = -1 + i$$
$$x5 = (-1 + i)^2 + i = -i$$
$$x6 = (-i)^2 + i = -1 + i$$

і ми бачимо, що ця орбіта з часом змінюється з періодом 2. Якщо ми змінимо с на 2i, тоді орбіта поводиться зовсім по-іншому

$$x0 = 0$$
$$x1 = 02 + 2i = 2i$$
$$x2 = (2i)2 + 2i = -4 + 2i$$
$$x3 = (-4 + 2i)2 + 2i = 12 - 14i$$
$$x4 = (12 - 14i)2 + 2i = -52 - 334i, \dots$$

і ми бачимо, що ця орбіта прагне до нескінченності в комплексній площині (числа, що входять до орбіти, все далі віддаляються від точки 0, яка має координати (0,0))[16].

Таким чином, враховуючі вищесказане, сформуємо псевдо-код майбутньої програми, для обчислення складності алгоритму.

```
for each pixel (Px, Py) on the screen do
    x0 := scaled x coordinate of pixel (scaled to lie in the Mandelbrot X scale (-2.00, 0.47))
    y0 := scaled y coordinate of pixel (scaled to lie in the Mandelbrot Y scale (-1.12, 1.12))
    x := 0.0
    y := 0.0
    iteration := 0
    max_iteration := 1000
    while (x*x + y*y ≤ 2*2 AND iteration < max_iteration) do
        xtemp := x*x - y*y + x0
        y := 2*x*y + y0
        x := xtemp
        iteration := iteration + 1

    color := palette[iteration]
    plot(Px, Py, color)
```

Рис. 11. Псевдокод програми для обчислення та відображення множини Мандельброта

Встановлюючи зв'язок псевдокоду з c, z і P_c:

$$z = x + iy$$
$$z^2 = x^2 + 2ixy - y^2$$
$$c = x_0 + iy_0$$

і тому, як можна побачити в псевдокоді під час обчислення x і y:

$$x = Re(z^2 + c) = x^2 - y^2 + x_0, \text{також } y = \mathrm{Im}(z^2 + c) = 2xy + y_0$$

Щоб отримати барвисті зображення множини, присвоєння кольору кожному значенню кількості виконаних ітерацій можна зробити за допомогою однієї з безлічі функцій. Одним із практичних способів без уповільнення обчислень є використання кількості виконаних ітерацій як запису в палітрі, ініціалізованій під час запуску[17]. Якщо таблиця кольорів має, наприклад, 500 записів, тоді вибір кольорів буде n % 500, де n – кількість ітерацій.

Таким чином, з псевдокоду програми, можна оцінити складність алгоритму:

1. Цикл за координатою x;
2. Вкладений цикл за координатою y;
3. Вкладений цикл визначення приналежності координати до множини Мандельброта;
4. Встановлення кольору пікселя на площині.

Відповідно до складності кожної операції в алгоритмічному уявлені, маємо:

$$O(n) * O(n) * O(1) * O(n) = O(n^3)$$

Отже, складність алгоритму є кубічною.

2.3 Створення програми-бенчмарку з використанням CPython

Для написання програми-бенчмарку стандартного інтерпретатору Python CPython будуть використовуватись наступні бібліотеки:

- “timeit”, для заміру часу виконання функції з пошуку чисел з множини Мандельброта. Слід зауважити, що час, витрачений на відображення результатів вирахувань не включатиметься в загальний час виконання через те, що методи з оптимізації продуктивності використовуються виключно до алгоритмічної частини програми;
- Numpy, для ефективнішої роботи з масивами. NumPy — це бібліотека Python, яка використовується для роботи з масивами. Вона також має функції для роботи в області лінійної алгебри, перетворення Фур’є та матриць. NumPy має на меті надати об’єкт масиву, який у 50 разів швидший за традиційні списки Python.
- Pylab — це процедурний інтерфейс для об’єктно-орієнтованої бібліотеки побудови математичних об’єктів, тобто графіків, площин, тощо. PyLab — це зручний модуль, який масово імпортує matplotlib.pyplot (для побудови графіків) і NumPy (для математики та роботи з масивами) в один

простір імен [18].

```python
from timeit import default_timer as timer
import numpy as np
from pylab import imshow, show

def mandel(x, y, max_iters):
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real * z.real + z.imag * z.imag) >= 2:
            return i

    return max_iters

def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image

image = np.zeros((500 * 2, 750 * 2), dtype=np.uint8)

print('Пошук чисел Мандельброта з використанням CPython')
s = timer()
create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
e = timer()
print('Час виконання: %f секунд' % (e - s,))
imshow(image)
show()
```

Рис. 12. Код програми-бенчмарку для тестування інтерпретатора CPython

У якості масиву для пошуку було обрано діапазон 0-1500 для чисел та 0-1000 для мнимої одиниці. У якості ліміту глибини для обчислення приналежності числа до множини Мандельброта було обрано 20.

Для покращення читаємості коду було вирішено винести цикл визначення приналежності до множини у окрему функцію.

Час виконання функції вимірюється у секундах з точністю до 8-го знаку після коми.

2.4 Створення програми-бенчмарку з використанням Cython

Найпростіший спосіб інсталяції Cython — за допомогою pip: pip install Cython.

Найновіший випуск Cython завжди можна завантажити з офіційного сайту компілятора. Для автономного встановлення, необхідно розпакувати tar-файл або zip-файл, перейти у каталог і запустити: python setup.py install.

Для одноразових збірок, наприклад для CI/тестування, на платформах, які не охоплені одним із пакетів коліс, наданих на PyPI, інсталювати нескомпільовану (повільнішу) версію Cython можна командою: pip install Cython --install-option="--no-cython-compile". [19]

Ключове слово cdef оголошує змінну як статично типізовану змінну C. Змінні C ведуть до швидшого виконання коду, оскільки накладні витрати від динамічного набору тексту Python обходяться. Аргументи функції також можуть бути оголошені як статично типізовані змінні C.

Існує два способи оголошення масивів NumPy як змінних C за допомогою Cython: за допомогою буферів масиву або за допомогою типізованих представлень пам’яті [20]. У цьому випадку використовується введені подання пам’яті.

Види типізованої пам’яті забезпечують ефективний доступ до буферів даних за допомогою синтаксису індексування, подібного до NumPy. Наприклад, ми можемо використовувати int[:,::1], щоб оголосити C-впорядкований 2D-масив NumPy із цілими значеннями, де ::1 означає безперервний макет у цьому вимірі. Види введеної пам’яті можна індексувати так само, як масиви NumPy.

Однак представлення пам’яті не реалізують поелементні операції, такі як NumPy. Таким чином, перегляди пам'яті діють як зручні контейнери даних у вузьких циклах for. Для поелементних операцій, подібних до NumPy, слід використовувати буфери масиву.

Можна було б досягти значного прискорення продуктивності, замінивши

виклик np.abs() швидшим виразом. Причина в тому, що np.abs() — це функція NumPy з невеликими витратами на виклик. Він призначений для роботи з відносно великими масивами, а не зі скалярними значеннями. Ці накладні витрати призводять до значного зниження продуктивності в жорсткому циклі, як тут. Це вузьке місце можна помітити за допомогою анотацій Cython та використанням Jupiter Notebook [21]. Але через те, що такий спосіб сильно погіршує читаємість коду, цієї заміни було вирішено позбутися.

```
6   import cython
7
8   @cython.cdivision(True)
9   def mandel(
10      int w, int h, int max_iters,
11      float min_x, float max_x, float min_y, float max_y
12  ):
13      image = np.zeros((h, w), dtype=np.float32)
14
15      cdef float[:,:] view = image
16      cdef complex z, c, I = 1j
17      cdef int x, y, k
18      cdef float dx, dy, real, im
19
20      dx = <float>(max_x - min_x) / w
21      dy = <float>(max_y - min_y) / h
22
23      for x in range(w):
24          real = min_x + x * dx
25          for y in range(h):
26              im = min_y + y * dy
27              c = real + I * im
28              z = 0
29              for k in range(max_iters):
30                  z = z ** 2 + c
31                  if np.abs(z) > 2:
32                      view[y,x] =  max_iters
33                      break
34
35      return image
36
37  print('Пошук чисел Мандельброта з використанням CPython')
38  s = timer()
39  image = mandel(1500, 1000, 255, -2.0, 1.0, -1.0, 1.0)
40  e = timer()
41  print('Час виконання: %f секунд' % (e - s,))
42  imshow(image)
43  show()
```

Рис. 13. Код програми-бенчмарку для тестування компілятора Cython

Оскільки оптимізації Cython застосуються до окремих функцій, необхідно

об’єднати цикл пошуку числа та трансформування координат для ефективного виклику функцій.

2.5 Створення програми-бенчмарку з використанням Numba

Щоб установити Numba у певному екземплярі Python, використовується pip, як і будь-який інший пакет: pip install numba. Якщо умови розробки дозволяються, варто встановлювати Numba у віртуальне середовище, а не в базову установку Python. [22]

Відповідно до документації, обертаємо функції у декоратор @jit, який, у свою чергу, перетворює функцію на машинний код (або максимально наближений до машинного коду, який може отримати Numba, враховуючи обмеження коду). Найкраща частина використання декоратора @jit — це простота. Можливо досягти значних покращень без будь-яких інших змін у коді.

Зауважимо те, що під час першого запуску функції може бути відчутна затримка, оскільки JIT запускається та компілює функцію. Однак кожен наступний виклик функції повинен виконуватися набагато швидше. Майте це на увазі, якщо планується порівняти JIT-функції з їхніми non-JIT аналогами; перший виклик JIT-функції завжди буде повільнішим. [23]

Найпростіший спосіб використати декоратор jit() — застосувати його до функції та дозволити Numba провести автоматичну оптимізацію. Але декоратор також використовує кілька параметрів, які контролюють його поведінку.

Якщо встановити nopython=True у декораторі, Numba спробує скомпілювати код без жодних залежностей від середовища виконання Python. Це не завжди можливо, але чим більше код складається з чистих числових маніпуляцій, тим вірогідніше, що опція nopython працюватиме. Перевагою цього є швидкість, оскільки JIT-функції без Python не потрібно сповільнюватись, щоб отримувати дані з середовища виконання Python.

```
5   from numba import jit
6
7   @jit(nopython=True, parallel=True, fastmath=True)
8   def mandel(x, y, max_iters):
9       i = 0
10      c = complex(x, y)
11      z = 0.0j
12      for i in range(max_iters):
13          z = z*z + c
14          if (z.real * z.real + z.imag * z.imag) >= 2:
15              return i
16
17      return max_iters
18
19  @jit(nopython=True, parallel=True, fastmath=True)
20  def create_fractal(min_x, max_x, min_y, max_y, image, iters):
21      height = image.shape[0]
22      width = image.shape[1]
23
24      pixel_size_x = (max_x - min_x) / width
25      pixel_size_y = (max_y - min_y) / height
26      for x in range(width):
27          real = min_x + x * pixel_size_x
28          for y in range(height):
29              imag = min_y + y * pixel_size_y
30              color = mandel(real, imag, iters)
31              image[y, x] = color
32
33      return image
34
35  image = np.zeros((500 * 2, 750 * 2), dtype=np.uint8)
36
37  print('Пошук чисел Мандельброта з використанням CPython')
38  s = timer()
39  create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
40  e = timer()
41  print('Час виконання: %f секунд' % (e - s,))
42  imshow(image)
43  show()
```

Рис. 14. Код програми-бенчмарку для тестування компілятора Numba

При встановлені parallel=True у декораторі, Numba скомпілює код Python для використання паралелізму через багатопроцесорність, де це можливо.

З nogil=true Numba звільняє глобальне блокування інтерпретатора (GIL) під час виконання JIT-компільованої функції. Це означає, що інтерпретатор запускатиме інші частини програми Python одночасно, наприклад потоки Python. Варто звернути увагу, що неможливо використовувати nogil, якщо код не компілюється в режимі nopython.

Встановлення cache=True зберігає скомпільований двійковий код у каталозі кешу для сценарію (зазвичай __pycache__). Під час наступних запусків Numba пропустить фазу компіляції та просто перезавантажить той самий код, що

й раніше, припускаючи, що нічого не змінилося. Кешування може трохи пришвидшити час запуску сценарію.

Якщо ввімкнути параметр fastmath=True, параметр fastmath дозволяє використовувати деякі швидші, але менш безпечні перетворення з плаваючою комою. Якщо у коді є операції із плаваючою комою, який гарантовано не генеруватиме NaN (не число) або inf (нескінченність), можна безпечно ввімкнути швидку математику для додаткової швидкості, коли використовуються числа з плаваючою комою, наприклад, в операціях порівняння з плаваючою комою.

Якщо ввімкнути параметр boundscheck=True, параметр boundscheck гарантує, що доступ до масиву не виходить за межі та потенційно призведе до збою програми. Зауважте, що це сповільнює доступ до масиву, тому його слід використовувати лише для налагодження [24].

2.6 Створення програми-бенчмарку з використанням Taichi

Щоб установити Taichi, аналогічно використовується інсталятор пакетів pip, як і будь-яку іншу: pip install taichi.

Модуль ti.types Taichi визначає всі підтримувані типи даних, і вони класифікуються на дві категорії: примітивні типи та складені типи. Примітивні типи відносяться до різних загальновживаних числових типів даних, таких як ti.i32 (int32), ti.u8 (uint8) і ti.f64 (float64). Складені типи відносяться до різних типів даних, подібних до масивів або структур, включаючи ti.types.matrix, ti.types.ndarray і ti.types.struct. Складені типи містять кілька членів примітивних типів або інших складених типів [25].

Taichi від нативного коду Python, у створеній програмі використовуємо два декоратори @ti.kernel і @ti.func:

Функції, прикрашені @ti.kernel, є ядрами Taichi або скорочено ядрами. Це точки входу, де середовище виконання Taichi починає виконувати завдання, і їх потрібно викликати безпосередньо кодом Python. Можливо підготувати свої завдання, як-от читання даних із диска та їх попередня обробка, у рідному Python,

а потім викликати ядра, щоб Taichi узяла на себе ці інтенсивні обчислювальні завдання.

Функції, прикрашені @ti.func, є функціями Taichi. Вони є будівельними блоками ядер і можуть бути викликані лише ядром або іншою функцією Taichi. Так само, як і зі звичайними функціями Python, можна розділити свої завдання на кілька функцій Taichi, щоб покращити читабельність і повторно використовувати їх у різних ядрах.[26]

У створеній програмі complex_sqr() прикрашено @ti.func і є функцією Taichi; create_fractal() прикрашено @ti.kernel і є ядром. Перший (complex_sqr()) викликається другим (create_fractal()). Аргумент і значення, що повертається в create_fractal(), є підказкою типу, тоді як у функції Taichi complex_sqr() – ні.

Підказка типу в Python рекомендована, а не обов’язкова. Taichi робить обов’язковим введення підказки аргументів і значення, що повертається ядром, якщо воно не має аргументу чи оператора повернення.

Ядро розглядає глобальні змінні як константи часу компіляції. Це означає, що він приймає поточні значення глобальних змінних на момент компіляції та не відстежує зміни в них згодом. Потім, якщо значення глобальної змінної оновлюється між двома викликами одного ядра, другий виклик не приймає оновлене значення. Як наслідком цього, функції Taichi є будівельними блоками ядра. Згідно документації, необхідно викликати функцію Taichi зсередини ядра або зсередини іншої функції Taichi.

```
5   import taichi as ti
6
7   ti.init(arch=ti.cpu)
8
9   @ti.func
10  def complex_sqr(z):
11      return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])
12
13  @ti.kernel
14  def create_fractal(
15      min_x: ti.float32, max_x: ti.float32,
16      min_y: ti.float32, max_y: ti.float32,
17      image: ti.ext_arr(), iters: ti.int32
18  ):
19      height, width = image.shape[:2]
20      dx = (max_x - min_x) / width
21      dy = (max_y - min_y) / height
22
23      for y, x in image:
24          c = ti.Vector([min_x + x * dx, min_y + y * dy])
25          z = ti.Vector([0.0, 0.0])
26          for i in range(iters):
27              z = complex_sqr(z) + c
28              if z.norm() >= 2:
29                  break
30              image[y, x] = i
31
32  image = np.zeros((500 * 2, 750 * 2), dtype=np.float64)
33
34  print('Пошук чисел Мандельброта з використанням CPython')
35  s = timer()
36  create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
37  e = timer()
38  print('Час виконання: %f секунд' % (e - s,))
39  print(image)
40  imshow(image)
41  show()
```

Рис. 15. Код програми-бенчмарку для тестування компілятора Taichi

РОЗДІЛ 3
ПРОВЕДЕННЯ ТЕСТІВ З ПРОДУКТИВНОСТІ ТА ПОРІВНЯЛЬНІ РЕЗУЛЬТАТИ РОБОТИ ПРОГРАМ-БЕНЧМАРКІВ

3.1 Використовуване обладнання та конфігурація запуску програм

Для отримання якісних результатів та для їх перевірки на консистентність, виконаємо кожну програму 3 рази та оберемо середнє значення часу виконання.

Усі програми виконуються на одній робочій станції, в одній сесії та з однаковими вхідними параметрами.

У табл. 3.1 можна побачити характеристики обладнання, на якому виконувались програми.

Таблиця 3.1

Характеристики використуємого ПК для запуску тестів

Версія Python	Python 3.10
Операційна система	Microsoft Windows 11
Процесор	AMD Ryzen 5 3500
Графічний процесор	NVidia RTX 2060
Оперативна пам’ять	DDR4 16GB

3.2 Проведення тестів програми, створеної з використанням CPython

Виконаємо запуск програм, створених з використанням штатного інтерпретатора CPython.

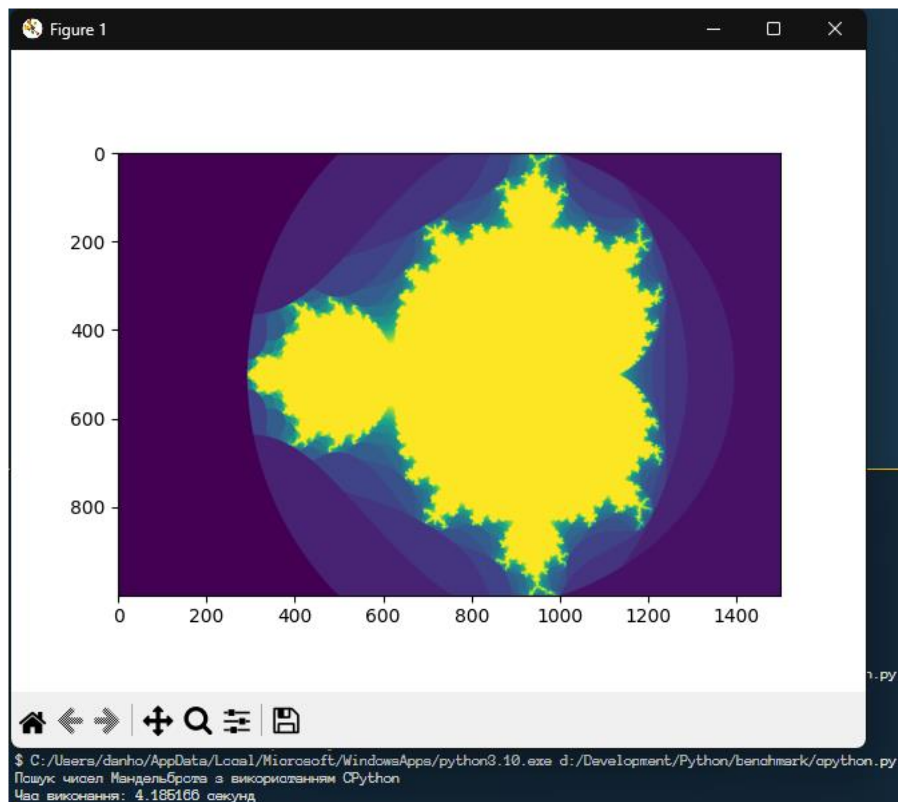


Рис. 16. Перший запуск програми з використанням CPython з результатом у 4 сек. 185 мс.

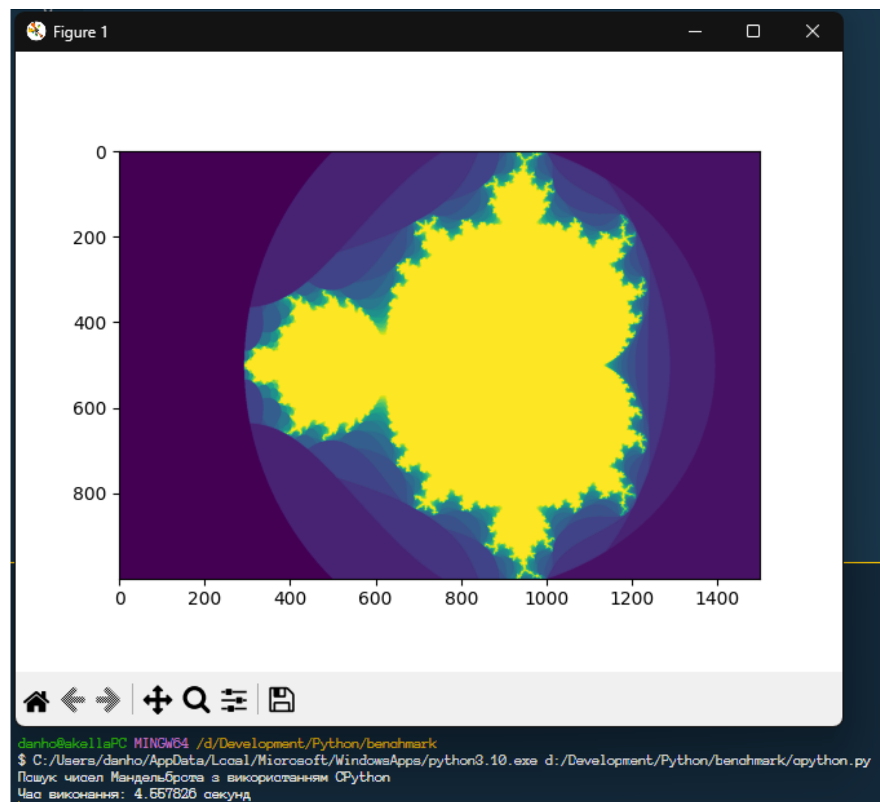


Рис. 17. Другий запуск програми з використанням CPython з результатом у 4 сек. 557 мс.

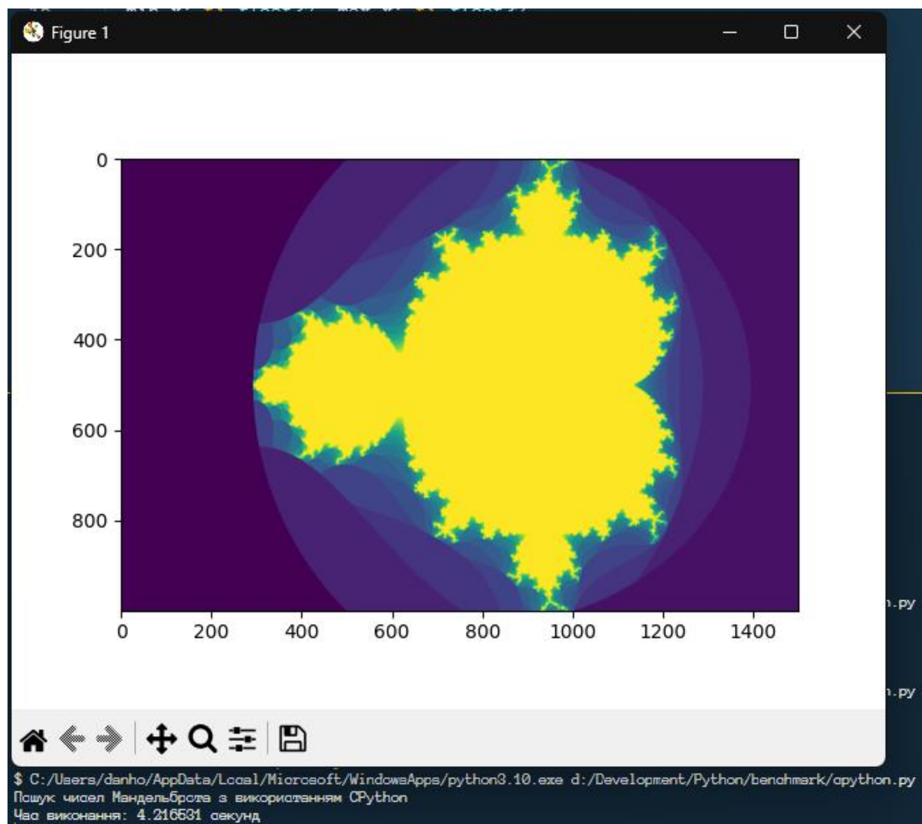


Рис. 18. Третій запуск програми з використанням CPython з результатом у 4 сек. 217 мс.

За результатами запуска, результати усіх тестів були консистентні, помилок або попереджень під час запуску не виникало.

3.3 Проведення тестів програми, створеної з використанням Cython

Щоб використовувати типи даних C у синтаксисі Python, потрібно імпортувати спеціальний модуль Cython у модуль Python, який потрібно скомпілювати. Оскільки Cython може прийняти майже будь-який дійсний вихідний файл Python, одна з найскладніших речей на початку роботи — просто з’ясувати, як скомпілювати розширення.

Для компіляції скрипту, необхідно зберегти його у форматі *.pyx. Тепер нам потрібно створити файл setup.py, який схожий на Makefile, але для Python (див. рисунок 19).

```
cython_setup.py
1   from distutils.core import setup
2   from Cython.Build import cythonize
3
4   setup(ext_modules=cythonize('cy_thon.pyx'))
5
```

Рис. 19. Програма запуску Python-програми з використанням Cython

Щоб використати це для створення файлу Cython, використовуйте параметри командного рядка:

$ python setup.py build_ext --inplace

Це залишить файл у локальному каталозі у форматі *.so в Unix або *.pyd у Windows. Тепер, щоб використати цей файл потрібно запустити інтерпретатор Python і просто імпортувати його, ніби це звичайний модуль Python.

Якщо для модуля не потрібні додаткові бібліотеки C або спеціальні налаштування збірки, можна скористатися модулем pyximport, щоб завантажувати файли .pyx безпосередньо під час імпорту, не запускаючи кожен файл setup.py час зміни коду [27].

```
danho@akellaPC MINGW64 /d/Development/Python/benchmark
$ python3.10 cython_setup.py build_ext --inplace
running build_ext
copying build\lib.win-amd64-cpython-310\cy_thon.cp310-win_amd64.pyd ->

danho@akellaPC MINGW64 /d/Development/Python/benchmark
$ python3.10
Python 3.10.8 (tags/v3.10.8:aaaf517, Oct 11 2022, 16:50:30) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import cy_thon
Пошук чисел Мандельброта з використанням Cython
```

Рис. 20. Процес компіляції Cython скрипту

Виконаємо запуск програм, створених з використанням компілятора Cython.

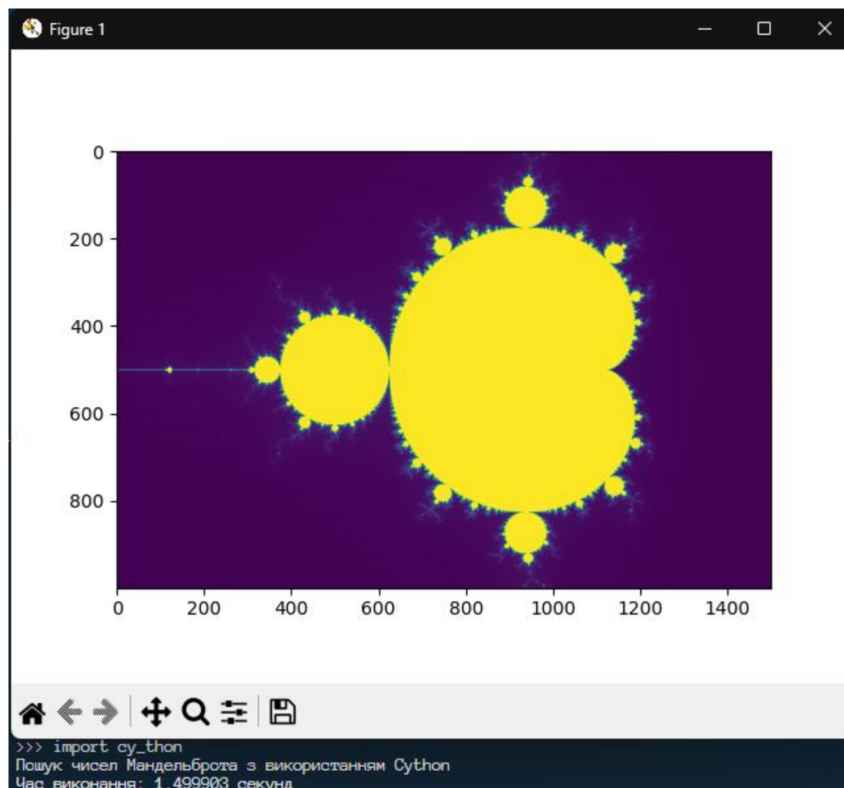


Рис. 21. Перший запуск програми з використанням Cython з результатом у 1 сек. 500 мс.

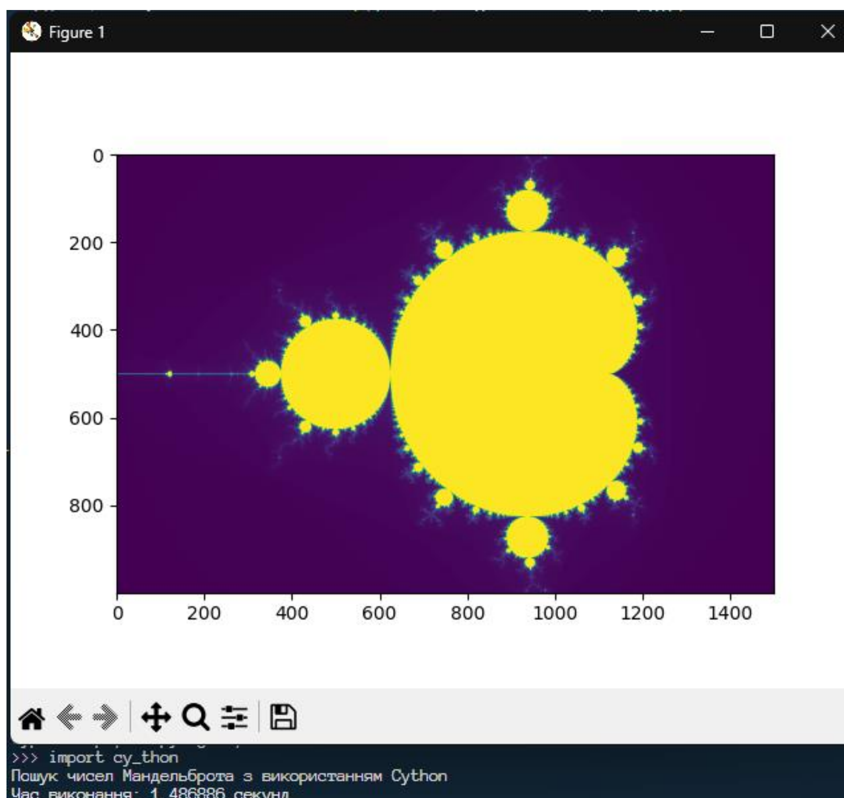


Рис. 22. Другий запуск програми з використанням Cython з результатом у 1 сек. 487 мс.

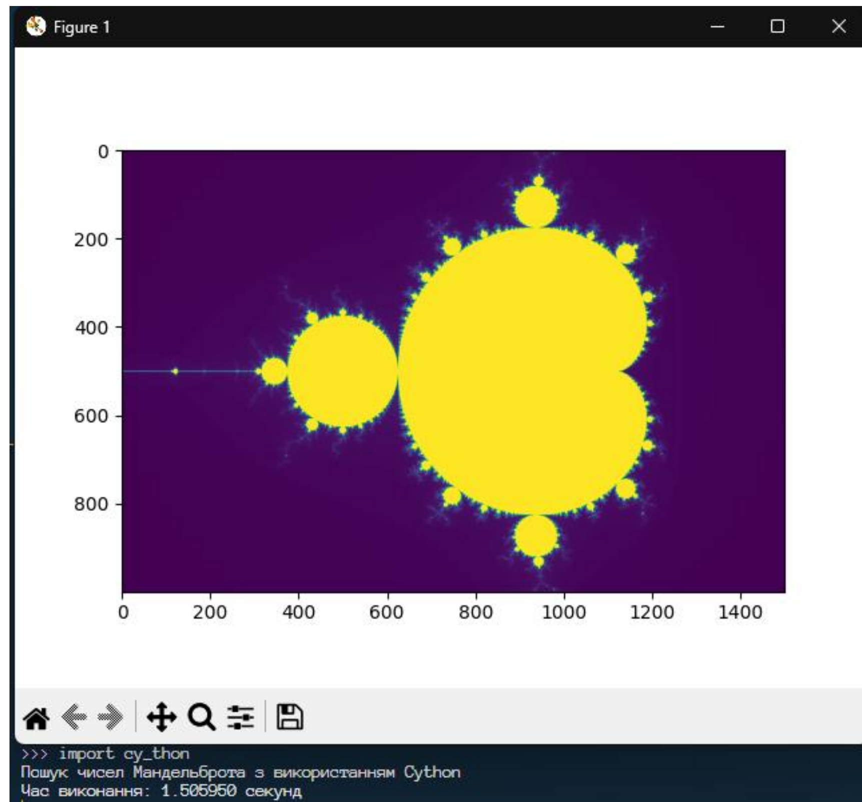


Рис. 23. Третій запуск програми з використанням Cython з результатом у 1 сек. 506 мс.

За результатами запуска, результати усіх тестів були консистентні, помилок або попереджень під час запуску не виникало.

3.4 Проведення тестів програми, створеної з використанням Numba

Виконаємо запуск програм, створених з використанням компілятора Numba.

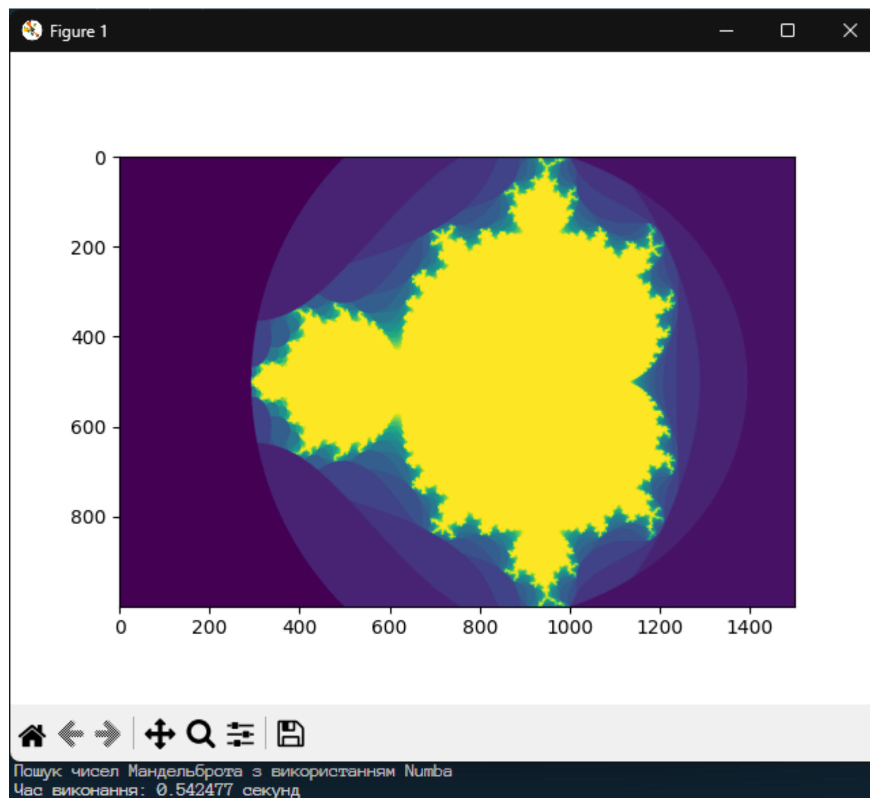


Рис. 24. Перший запуск програми з використанням Numba з результатом у 542 мс.

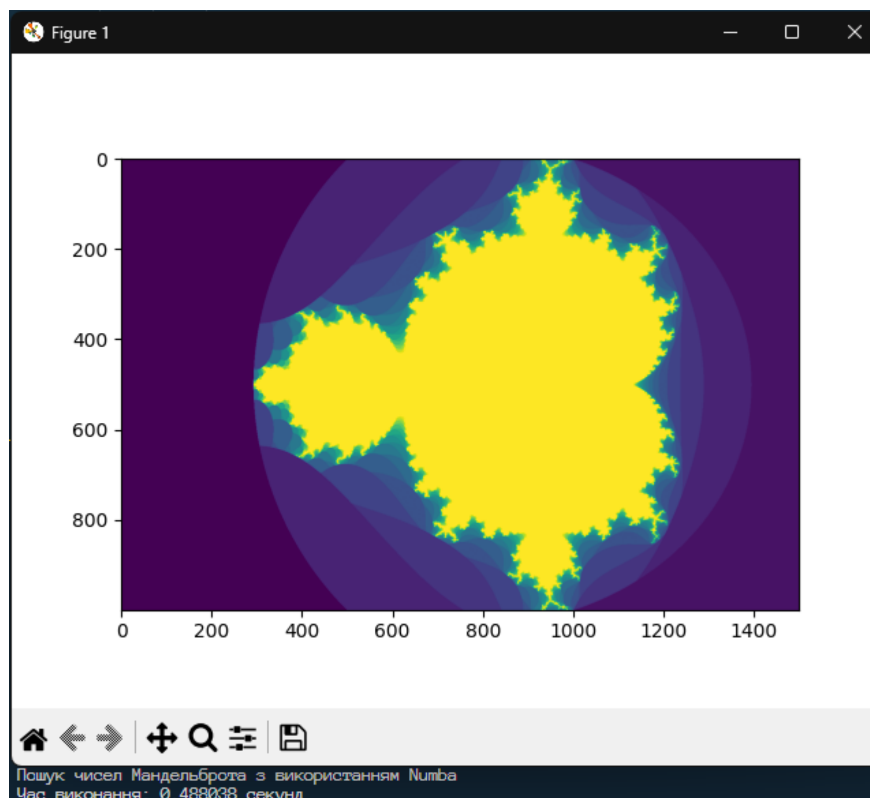


Рис. 25. Другий запуск програми з використанням Numba з результатом у 488 мс.

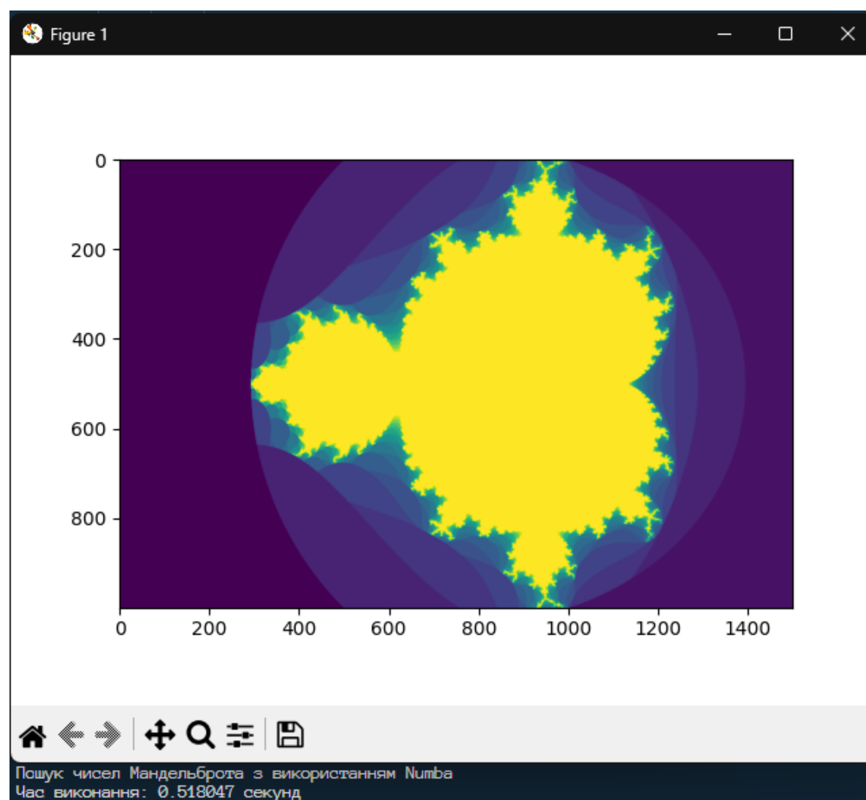


Рис. 26. Третій запуск програми з використанням Numba з результатом у 518 мс.

За результатами запуска, результати усіх тестів були консистентні, помилок або попереджень під час запуску не виникало.

3.5 Проведення тестів програми, створеної з використанням Taichi

Виконаємо запуск програм, створених з використанням компілятора Taichi.

У інтересах прозорості та об’єктивності, тести запускатимуться на центральному процесорі, а не на графічному, оскільки у інтерпретаторі CPython та компіляторі Cython такої можливості не існує. Для цього, у параметрах оточення експліцитно вкажемо:

ti.init(arch=ti.cpu)

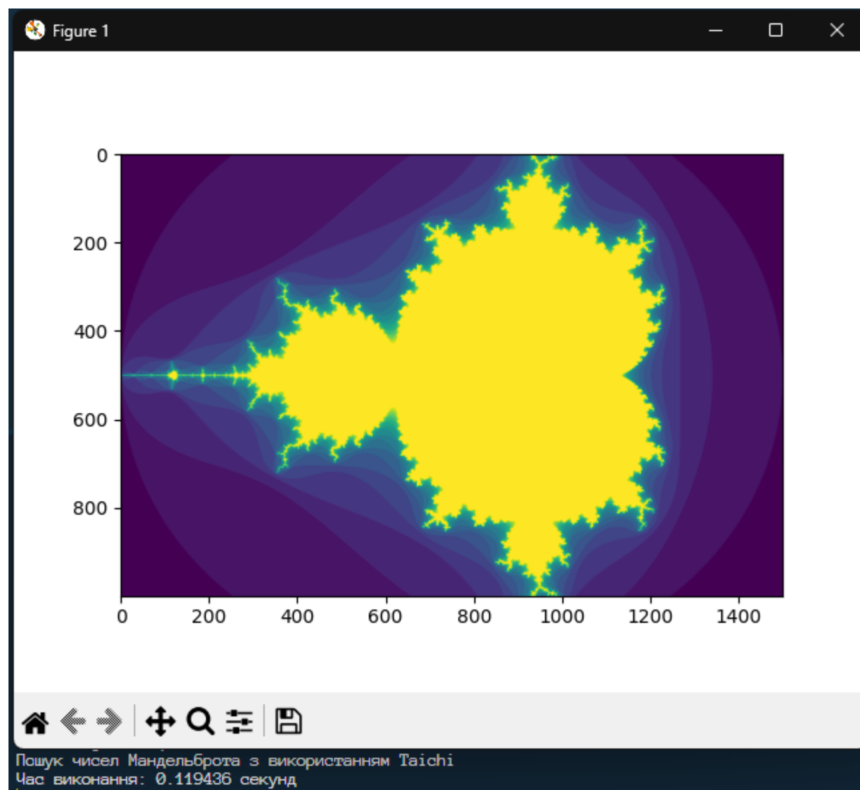


Рис. 26. Перший запуск програми з використанням Taichi з результатом у 119 мс.

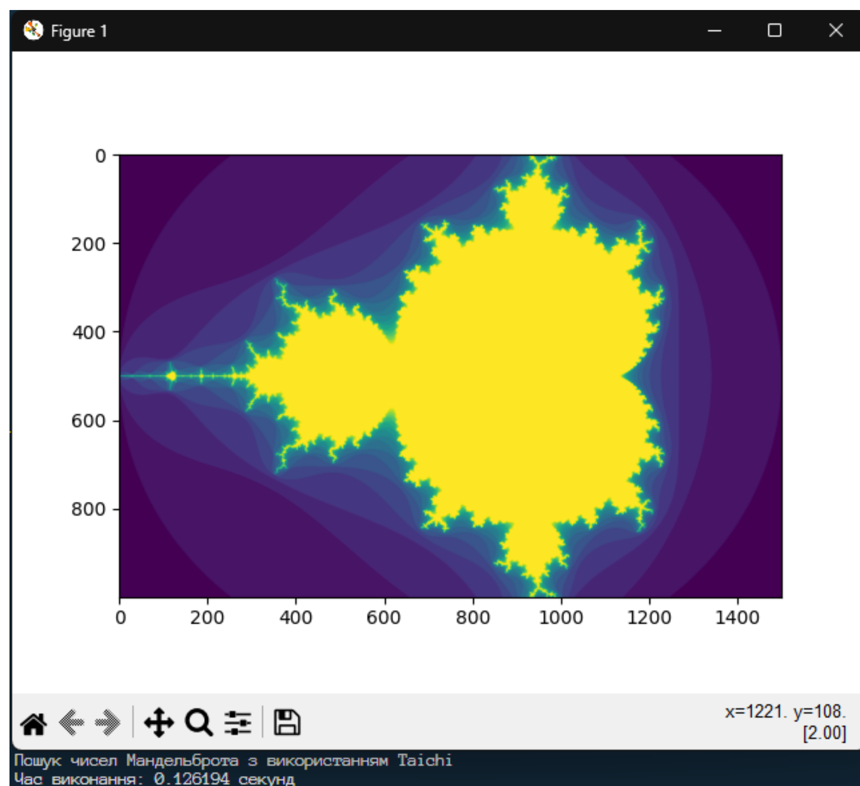


Рис. 27. Другий запуск програми з використанням Taichi з результатом у 126 мс.

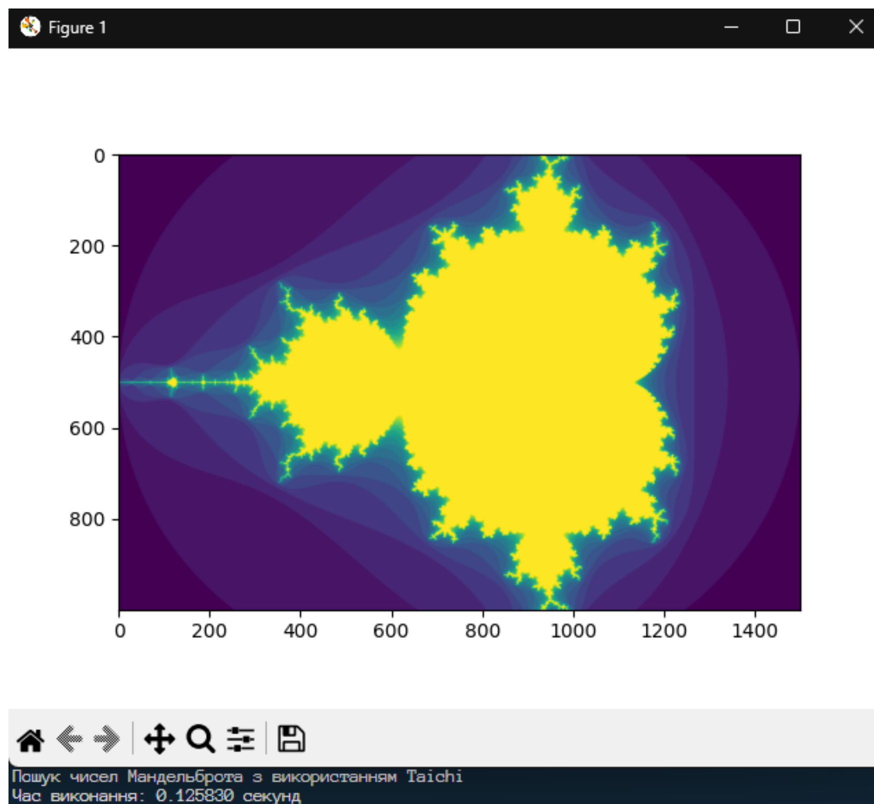


Рис. 28. Третій запуск програми з використанням Taichi з результатом у 126 мс.

За результатами запуска, результати усіх тестів були консистентні, помилок або попереджень під час запуску не виникало.

3.6 Результати тестів та оцінка ефективності

Відповідно до результатів тестів, усі програми показали консистентні результати.

Незначні різності у відображені на графіках зумовлені особливостями використуємого фреймворку для малювання графіків а також методикою інтерпретацій формул.

Варто зауважити, що не лише продуктивність є важливим критерієм для оцінки якості впровадження технологію у виробничий цикл. Підвищена складність програми та погіршення читаємості коду також є важливими у процесі розробки, але об’єктивну сторону цього аспекту складно оцінити. Для кожної окремої задачі різний набір інструментів може генерувати різний

результат.

Порівняльні результати виконання поставленого завдання можна побачити у табл. 3.2.

Таблиця 3.2

Результати тестування та відносний прибуток у продуктивності

Назва інтерпретатора	Результат 1, (мс)	Результат 2, (мс)	Результат 3, (мс)	Середній результат, (мс)	Порівняльний здобуток продуктивності
CPython	4185	4557	4217	4319	0%
Cython	1500	1487	1506	1498	+388%
Numba	542	488	518	516	+937%
Taichi	119	126	126	124	+3500%

РОЗДІЛ 4
ПОТЕНЦІАЛИ ЕКОНОМІЧНОЇ ВИГОДИ

4.1 Дослідження ринку та можливих сфер впровадження

Основна сфера, у якій швидкість виконання програми грає ключову роль є так звана «High Performance Computing».

Високопродуктивні обчислення в основному стосуються практики агрегування обчислювальної потужності таким чином, щоб забезпечити набагато вищу продуктивність, ніж можна отримати від типового настільного комп’ютера або робочої станції, щоб вирішити великі проблеми в науці, інженерії чи бізнесі.

Сьогодні HPC використовується для вирішення складних проблем, що потребують високої продуктивності, і організації все частіше переносять робочі навантаження HPC у хмару [28]. HPC у хмарі змінює економіку розробки продуктів і досліджень, оскільки вимагає менше прототипів, прискорює тестування та скорочує час виходу на ринок.

Деякі робочі навантаження, такі як секвенування ДНК, просто надто великі для будь-якого окремого комп’ютера. HPC або суперкомп’ютерні середовища вирішують ці великі та складні проблеми за допомогою окремих вузлів (комп’ютерів), які працюють разом у кластері (пов’язаній групі), щоб виконувати величезні обсяги обчислень за короткий проміжок часу. Створення та видалення цих кластерів часто автоматизовано в хмарі, щоб зменшити витрати.

HPC можна запускати з багатьма видами робочих навантажень, але два найпоширеніші — це надзвичайно паралельні робочі навантаження та тісно пов’язані робочі навантаження.

Надзвичайно паралельні навантаження – це обчислювальні проблеми, розділені на невеликі, прості та незалежні завдання, які можна виконувати одночасно, часто з невеликим або без зв’язку між ними. Наприклад, компанія може надіслати 100 мільйонів записів кредитних карток до окремих ядер процесора в кластері вузлів. Обробка одного запису кредитної картки є невеликим завданням, і коли 100 мільйонів записів розподілено по кластеру, ці

невеликі завдання можна виконувати одночасно (паралельно) з неймовірною швидкістю. Загальні випадки використання включають моделювання ризиків, молекулярне моделювання, контекстний пошук і моделювання логістики.

Зазвичай велике спільне робоче навантаження розбивається на менші завдання, які постійно взаємодіють [29]. Іншими словами, різні вузли в кластері спілкуються один з одним під час виконання обробки. Загальні випадки використання включають обчислювальну гідродинаміку, моделювання прогнозу погоди, симуляцію матеріалів, емуляцію зіткнення автомобілів, геопросторове моделювання та управління дорожнім рухом.

HPC протягом десятиліть є важливою частиною академічних досліджень та промислових інновацій [30]. HPC допомагає інженерам, дослідникам даних, дизайнерам та іншим дослідникам вирішувати великі складні проблеми за набагато менший час і з меншими витратами, ніж традиційні обчислення.

Серед основних переваг HPC можна зазначити:

1. Зменшене фізичне тестування: HPC можна використовувати для створення симуляцій, усуваючи потребу у фізичних тестах. Наприклад, під час тестування автомобільних аварій набагато простіше та дешевше створити симуляцію, ніж провести краш-тест;

2. Швидкість. Завдяки найновішим ЦП, графічним процесорам (GPU) і мережевим структурам із низькою затримкою, таким як віддалений прямий доступ до пам'яті, у поєднанні з флеш-пам'ятними пристроями та блоковими накопичувачами HPC може виконувати великі обчислення за лічені хвилини, а не тижнів або місяців;

3. Вартість: швидші відповіді означають менше витраченого часу та грошей. Крім того, за допомогою хмарних HPC навіть малі підприємства та стартапи можуть дозволити собі виконувати робочі навантаження HPC, сплачуючи лише за те, що вони використовують, і масштабуючи за потреби;

4. Інновації: HPC стимулює інновації майже в кожній галузі — це сила, що стоїть за новаторськими науковими відкриттями, які покращують якість життя людей у всьому світі.

Найбільші іноземні компанії майже в кожній галузі використовують HPC, і його популярність продовжує зростати. За даними різних досліджень, очікується, що до 2022 року глобальний ринок HPC досягне 44 мільярдів доларів США [31].

Нижче наведено деякі галузі, які використовують HPC, і типи робочих навантажень, які HPC допомагає їм виконувати:

- Аерокосмічна сфера: створення складних симуляцій, як-от повітряний потік над крилами літаків;
- Виробництво: виконання симуляцій, наприклад для автономного водіння, для підтримки проектування, виробництва та тестування нових продуктів, що призводить до безпечніших автомобілів, легших деталей, ефективніших процесів та інновацій;
- Фінансові технології : Виконання комплексного аналізу ризиків, високочастотна торгівля, фінансове моделювання та виявлення шахрайства;
- Геноміка: секвенування ДНК, аналіз взаємодії лікарських засобів і проведення аналізів білка для підтримки досліджень походження;
- Охорона здоров'я: дослідження ліків, створення вакцин і розробка інноваційних методів лікування рідкісних і поширених захворювань;
- Медіа та розваги: створення анімації, відтворення спеціальних ефектів для фільмів, перекодування величезних медіа-файлів і створення захоплюючих розваг;
- Нафта і газ: Виконання просторового аналізу та тестування моделей пластів для прогнозування розташування нафтових і газових ресурсів, а також проведення симуляцій, таких як потік рідини та сейсмічна обробка;
- Роздрібна торгівля: аналіз величезних обсягів даних про клієнтів для надання точніших рекомендацій щодо продуктів і кращого обслуговування клієнтів HPC може виконуватися локально, у хмарі або в гібридній моделі, яка включає деякі з них.

Під час локального розгортання HPC бізнес або дослідницька установа створює кластер HPC із серверами, рішеннями для зберігання даних та іншою

інфраструктурою, якою вони керують і оновлюють з часом.

постачальник хмарних послуг адмініструє та керує інфраструктурою, а організації використовують її за моделлю оплати за використання.

Деякі організації використовують гібридні розгортання, особливо ті, які інвестували в локальну інфраструктуру, але також хочуть скористатися перевагами швидкості, гнучкості та економії коштів хмари. Вони можуть використовувати хмару для виконання деяких робочих навантажень HPC на постійній основі та звертатися до хмарних служб на тимчасовій основі, коли час у черзі стає проблемою на місці.

Організації з локальними середовищами HPC отримують великий контроль над своїми операціями, але їм доводиться боротися з кількома проблемами, зокрема

- Вкладення значних капіталів у обчислювальне обладнання, яке необхідно постійно оновлювати;
- Оплата поточного управління та інших операційних витрат;
- Затримка або час у черзі від днів до місяців, перш ніж користувачі зможуть запустити робоче навантаження HPC, особливо коли попит зростає;
- Відкладення оновлення до більш потужного та ефективного обчислювального обладнання через довгі цикли закупівлі, що уповільнює темпи досліджень і бізнесу.

Частково через витрати та інші труднощі локальних середовищ хмарне розгортання HPC стає все більш популярним, а “Market Research Future” прогнозує зростання світового ринку на 21% з 2017 по 2023 рік [32]. Коли компанії запускають свої робочі навантаження HPC у хмарі, вони платять лише за те, що використовують, і можуть швидко збільшувати чи зменшувати потреби.

Щоб завоювати й утримувати клієнтів, провідні хмарні постачальники підтримують передові технології, спеціально розроблені для робочих навантажень HPC, тому немає загрози зниження продуктивності в міру старіння локального обладнання. Хмарні постачальники пропонують найновіші та найшвидші центральні та графічні процесори, а також флеш-пам’ять із низькою

затримкою, блискавичні мережі RDMA та безпеку корпоративного класу. Послуги доступні цілий день, щодня, без черги.

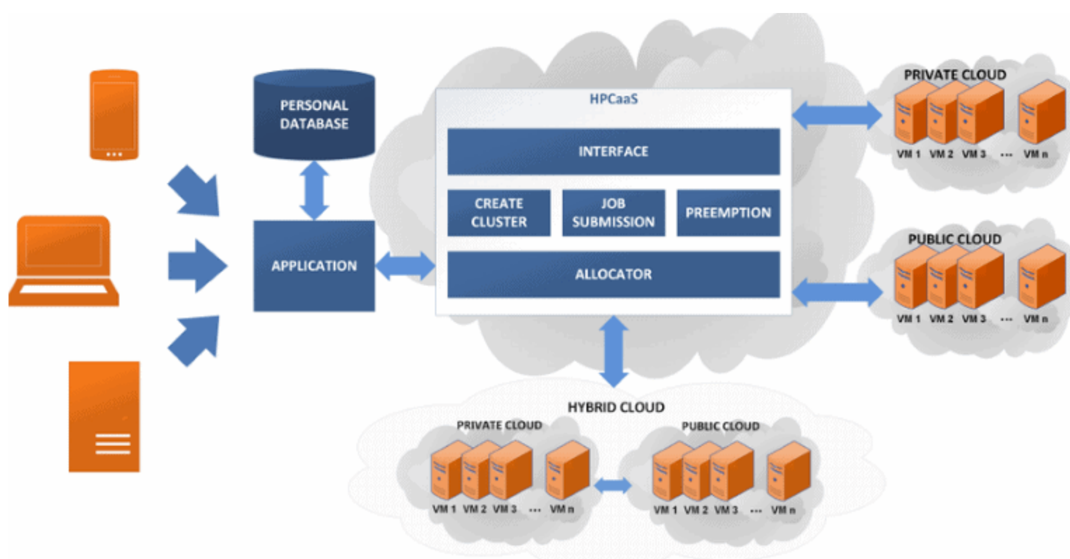


Рис. 29. Схема архітектури кластеру розподіленої високопродуктивної обчислювальної техніки

Не всі хмарні постачальники створені рівними. Деякі хмари не розроблені для HPC і не можуть забезпечити оптимальну продуктивність у періоди пікових навантажень [33]. Чотири риси, які слід враховувати при виборі хмарного провайдера:

- Передова продуктивність: хмарний постачальник повинен мати та підтримувати останнє покоління процесорів, сховищ і мережевих технологій. Переконайтеся, що вони пропонують велику ємність і найвищу продуктивність, яка відповідає або перевищує типові локальні розгортання;
- Досвід роботи з HPC. Вибраний вами хмарний постачальник повинен мати глибокий досвід виконання робочих навантажень HPC для різних клієнтів. Крім того, їхня хмарна служба має бути розроблена так, щоб забезпечити оптимальну продуктивність навіть у періоди пікового навантаження, наприклад, під час запуску кількох симуляцій або моделей. У багатьох випадках голі комп’ютерні екземпляри забезпечують стабільнішу та потужнішу продуктивність порівняно з віртуальними машинами [34];

- Гнучкість для підвищення та переміщення: робочі навантаження HPC повинні виконуватися в хмарі так само, як і на місці. Після того, як переміщується робоче навантаження в хмару «як є» під час операції підйому та переміщення, симуляція, яку запускають наступного тижня, має дати узгоджений результат із тим, який проводили десять років тому. Це надзвичайно важливо в галузях, де щорічні порівняння повинні проводитися з використанням тих самих даних і обчислень. Наприклад, обчислення для аеродинаміки, автомобілів і хімії не змінилися, і результати також не можуть змінитися [35];
- Жодних прихованих витрат: хмарні послуги зазвичай пропонуються за моделлю оплати за використання, тому необхідно переконатися, за що компанія буде платити щоразу, коли користуватиметеся послугою.

4.2 Принципи взаємодії та тарифікації хмарних сервісів

У якості прикладу для розглянення ефективності впровадження, було обрано Amazon Web Services. Amazon Web Services (скорочено AWS) — це набір віддалених обчислювальних служб (також званих веб-службами), які разом складають хмарну обчислювальну платформу, яку Amazon.com пропонує через Інтернет. Найбільш центральними та відомими з цих служб є Amazon EC2, AWS Lambda та Amazon S3.

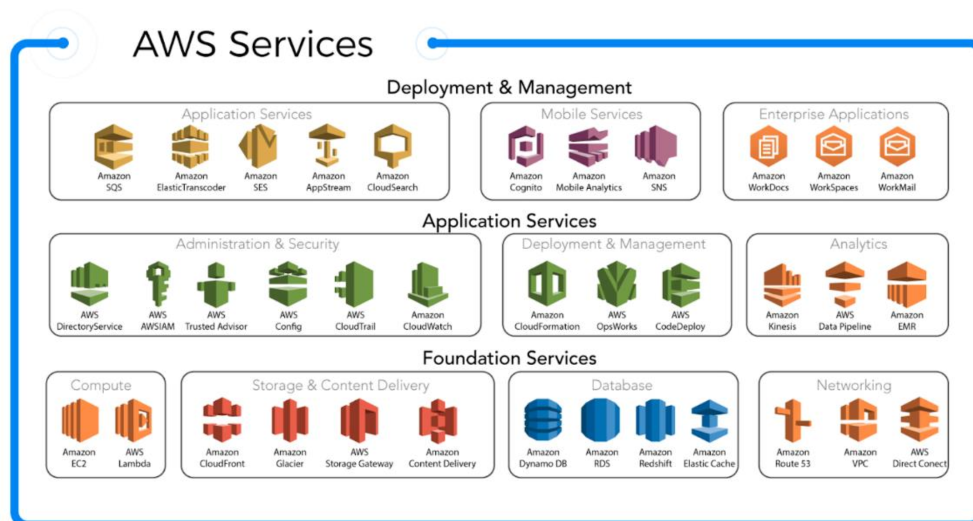


Рис. 30. Перелік основних сервісів Amazon Web Services

AWS Lambda — це безсерверний обчислювальний сервіс, який дає змогу запускати код без підготовки та керування серверами. Створивши логіку масштабування кластера з урахуванням робочого навантаження, можна підтримувати інтеграцію подій і легко керувати максимальним часом виконання. За допомогою Lambda з'являється можливість запускати код практично для будь-якого типу програми чи серверної служби без адміністрування та платити лише за те, що використовуєте. З клієнта стягується плата залежно від кількості запитів на функції та тривалості, необхідної для виконання його коду.

Lambda підраховує запит кожного разу, коли починає виконуватися у відповідь на тригер сповіщення про подію, як-от від Amazon Simple Notification Service (SNS) або Amazon EventBridge, або виклик виклику, як-от від Amazon API Gateway, або через AWS SDK, зокрема тест викликає з консолі AWS [36].

Тривалість обчислюється з моменту початку виконання коду до його повернення або завершення іншим чином, округлюючись до найближчої 1 мс. Ціна залежить від обсягу пам'яті, який виділяється для своєї функції. У моделі ресурсів AWS Lambda обирається потрібний обсяг пам'яті для своєї функції, і розподіляється пропорційна потужність ЦП та інші ресурси. Збільшення обсягу пам'яті викликає еквівалентне збільшення ЦП, доступного для функції.

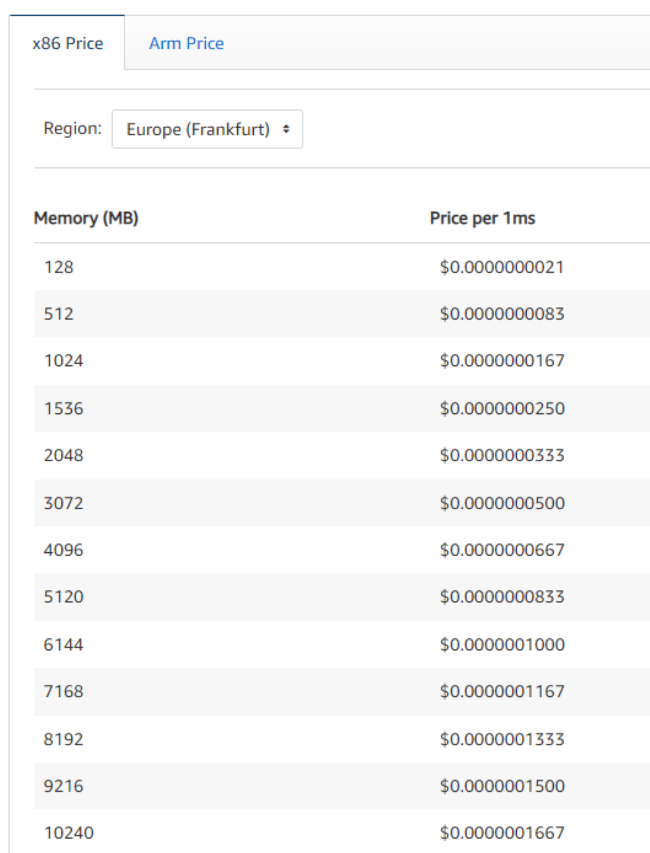
Можливо запускати свої функції Lambda на процесорах, побудованих на архітектурі x86 або Arm [37]. Функції AWS Lambda, що працюють на Graviton2, використовуючи архітектуру процесора на основі Arm, розроблену AWS, забезпечують до 34% кращу цінову продуктивність порівняно з функціями, що працюють на процесорах x86 [38]. Це стосується різноманітних безсерверних робочих навантажень, таких як веб- і мобільні сервери, обробка даних і медіа.

Lambda також пропонує багаторівневі варіанти ціноутворення на вимогу, що перевищує певні місячні порогові значення використання. AWS Lambda бере участь у Compute Savings Plans, гнучкій моделі ціноутворення, яка пропонує низькі ціни на використання Amazon Elastic Compute Cloud (Amazon EC2), AWS Fargate і використання Lambda в обмін на зобов'язання щодо сталого обсягу використання на один або три роки. Завдяки тарифним планам Compute Savings

можна заощадити до 17 відсотків на AWS Lambda [39]. Економія стосується тривалості та передбаченого паралелізму.

4.3 Очікувана вигода при використанні компіляторів в хмарному середовищі

Отримаємо основні розцінки за використання послуги AWS Lambda, актуальних на момент написання роботи [40].


x86 Price | Arm Price

Region: Europe (Frankfurt)

Memory (MB)	Price per 1ms
128	$0.0000000021
512	$0.0000000083
1024	$0.0000000167
1536	$0.0000000250
2048	$0.0000000333
3072	$0.0000000500
4096	$0.0000000667
5120	$0.0000000833
6144	$0.0000001000
7168	$0.0000001167
8192	$0.0000001333
9216	$0.0000001500
10240	$0.0000001667

Рис. 31. Актуальна інформація про тарифікацію у сервісі AWS Lambda

Відповідно до отриманих цін, розрахуємо вартість запуску розроблених програм у цьому сервісі, відповідно до середнього часу виконання (див. табл. 4.1).

Таблиця 4.1

Вартість запуску розроблених програм у середовищі AWS Lambda

Об'єм пам'яті, (МБ)	Вартість запуску CPython	Вартість запуску Cython	Вартість запуску Numba	Вартість запуску Taichi
128	$0.000009069900	$0.000003145800	$0.000001083600	$0.000000260400
512	$0.000035847700	$0.000012433400	$0.000004282800	$0.000001029200
1024	$0.000072127300	$0.000025016600	$0.000008617200	$0.000002070800
1536	$0.000107975000	$0.000037450000	$0.000012900000	$0.000003100000
2048	$0.000143822700	$0.000049883400	$0.000017182800	$0.000004129200
3072	$0.000215950000	$0.000074900000	$0.000025800000	$0.000006200000
4096	$0.000288077300	$0.000099916600	$0.000034417200	$0.000008270800
5120	$0.000359772700	$0.000124783400	$0.000042982800	$0.000010329200
6144	$0.000431900000	$0.000149800000	$0.000051600000	$0.000012400000
7168	$0.000504027300	$0.000174816600	$0.000060217200	$0.000014470800
8792	$0.000575722700	$0.000199683400	$0.000068782800	$0.000016529200
9216	$0.000647850000	$0.000224700000	$0.000077400000	$0.000018600000
10240	$0.000719977300	$0.000249716600	$0.000086017200	$0.000020670800

Відповідно до табл. 4.1, розрахуємо середню економію за один запуск за формулою $\frac{\sum_n^1 c}{n}$:

Таблиця 4.2

Середня економія за запуск розроблених програм у середовищі AWS Lambda

	CPython	Cython	Numba	Taichi
Середня вартість запуску	$0.000316316915	$0.000109711215	$0.000037791046	$0.000009081569
Очікувана економія	0.00%	-65.32%	-88.05%	-97.13%

Таким чином, виходячи з результатів розрахунків у табл. 4.2 використовуючи альтернативні компілятори, можливо зекономити до 97% від вартості послуги AWS Lambda.

ВИСНОВКИ

У дипломній роботі вирішена актуальна науково-технічна задача дослідження ефективності впровадження альтернативних компіляторів мови програмування Python у процес розробки ПЗ.

1. Дослідження такої області розробки ПЗ, як оптимізація коду дозволяє економити значні кошти при користуванні хмарними сервісами.

2. Дослідження показали, що за умови наяви технічної можливості постачати додаткові бібліотеки разом з програмним кодом, обсяг економії значно перевищує затрати на додатковий час, необхідний на написання коду.

3. Використання альтернативних компіляторів не несе за собою ризиків з неконсистентними результатами або огріхами при обчисленнях, за умови правильного використання та додаткових запобіжних заходів у коді.

4. За допомогою впровадження запропонованих технологій у виробничий процес, можна зберегти високу читаємість та простоту коду, присутню мові програмування Python, але за потребою, продуктивність програм можна значно збільшити за умови використання додаткових типів даних, реалізованими у компіляторах «Taichi» або «Cython».

5. Завдяки значним здобуткам у часі виконання, відкриваються нові можливості у вигляді використання мови Python у сферах, у яких раніше його використання було неможливо через його «повільність», таких як програмування промислового обладнання, певні наукові дослідження, зокрема у області біології та аеродинаміки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. John McCarthy History of Lisp / John McCarthy // Artificial Intelligence Laboratory Computer Science Department, Stanford University, Stanford, California – 1987. – P. 3-6.

2. Aho Alfred V. Compilers: principles, techniques, and tools. / Aho Alfred V, Sethi Ravi, Ullman Jeffrey D. // Mark S. Dalton – 1986. – P. 704.

3. Floretin J. John Assemblers and Loaders / Floretin J. John // University of Southampton, Southampton, UK - Elsevier North-Holland Inc. – 1978. – P. 65–66

4. Bez R. Advances in Non-Volatile Memory and Storage Technology / Bez R., Pirovano A. // Woodhead Publishing - 2019. – P. 254

5. Michal Jaworski Expert Python Programming / Michal Jaworski, Tarek Ziade // Packt Publishing – 2019. – P. 101-104

6. Obi Ike-Nwosu Inside The Python Virtual Machin / Obi Ike-Nwosu // Lean Publishing – 2019. – P. 4

7. Michal Jaworski Expert Python Programming / Michal Jaworski, Tarek Ziade // Packt Publishing – 2019. – P. 122

8. B.M. Harwani C Programming Cookbook / B. M. Harwani, Bintu Harwani // Packt Publishing. – 2019. – P.47

9. Anthony Shaw CPython Internals / Anthony Shaw // Real Python (realpython.com). – 2021. – P.85-87

10. Anthony Shaw CPython Internals / Anthony Shaw // Real Python (realpython.com). – 2021. – P.89-90

11. Dr. Gabriele Lanaro Python High Performance / Dr. Gabriele Lanaro // Packt Publishing – 2017. – P. 122

12. Cyrille Rossant IPython Cookbook / Cyrille Rossant // Packt Publishing – 2018. – P. 55

13. Building Cython code URL: https://cython.readthedocs.io/en/stable/src/quickstart/build.html (остання дата звернення 15.11.2022)

14. ‘QuanTaichi’ Quantized Simulation: High Visual Quality With Reduced Memory Cost URL: https://syncedreview.com/2021/07/19/deepmind-podracer-tpu-based-rl-frameworks-deliver-exceptional-performance-at-low-cost-64/ (остання дата звернення 17.11.2022)

15. Mandelbrot Set URL: https://mathworld.wolfram.com/MandelbrotSet.html (остання дата звернення 17.11.2022)

16. What is the Mandelbrot set? URL: https://plus.maths.org/content/what-mandelbrot-set (остання дата звернення 17.11.2022)

17. Mathematical models and heuristic algorithms for pallet building problems with practical constraints / Gabriele Calzavara, Manuel Iori, Marco Locatelli, Mayron C. O. Moreira, Tiago Silveira // Annals of Operations Research. – 2021.

18. Matplotlib - PyLab module for plotting URL: https://www.tutorialspoint.com/matplotlib/matplotlib_pylab_module.htm (остання дата звернення 19.11.2022)

19. Cython 0.29.32 URL: https://pypi.org/project/Cython/ (остання дата звернення 19.11.2022)

20. Numpy Documentation URL: https://numpy.org/doc/stable/reference/generated/numpy.array.html (остання дата звернення 19.11.2022)

21. Jupyter Notebook: An Introduction URL: https://realpython.com/jupyter-notebook-introduction/ (остання дата звернення 19.11.2022)

22. Numba 0.50.1 Documentation URL: https://numba.pydata.org/numba-doc/latest/user/installing.html (остання дата звернення 20.11.2022)

23. Supercharge tensor processing in Python with JIT compilation URL: https://medium.com/starschema-blog/jit-fast-supercharge-tensor-processing-in-python-with-jit-compilation-47598de6ee96 (остання дата звернення 20.11.2022)

24. Numba - Other things of interest URL: https://numba.readthedocs.io/en/stable/user/5minguide.html (остання дата звернення 20.11.2022)

25. Taichi Type System URL: https://docs.taichi-lang.org/docs/type (остання

дата звернення 20.11.2022)

26. Yuanming Hu Life of a Taichi Kernel / Yuanming Hu // Taichi Graphics Team. – 2021. - P.42

27. Compiling Cython code URL: https://neurohackweek.github.io/cython-tutorial/02-compiling/ (остання дата звернення 22.11.2022)

28. Cloud Migration: Why (and How) Companies Are Moving to the Cloud https://www.ntiva.com/blog/cloud-migration-why-companies-are-moving-to-the-cloud (остання дата звернення 25.11.2022)

29. Using clusters for large-scale technical computing in the cloud URL: https://cloud.google.com/architecture/using-clusters-for-large-scale-technical-computing (остання дата звернення 29.11.2022)

30. High Performance Computing in Science and Engineering / Wolfgang E. Nagel, Dietmar H. Kröner, Michael M. Resch // Transactions of the High Performance Computing Center, Stuttgart, Springer Publishing. – 2019. – P. 212

31. HPC Market Five-Year Forecast bumps up to $44 Billion Worldwide URL: https://insidehpc.com/2019/06/hpc-market-five-year-forecast-bumps-up-to-44-billion-worldwide/ (остання дата звернення 29.11.2022)

32. Cloud Engineering Market Research Report - Forecast 2027 URL: https://www.marketresearchfuture.com/reports/cloud-engineering-market-4063 (остання дата звернення 29.11.2022)

33. Peak performance: How retailers used Google Cloud during Black Friday/Cyber Monday URL: https://cloud.google.com/blog/topics/customers/peak-performance-how-retailers-used-google-cloud-during-black-friday-cyber-monday (остання дата звернення 1.12.2022)

34. NETSCOUT advanced network detection and response findings enrich Amazon security lake URL: https://www.zawya.com/en/press-release/companies-news/netscout-advanced-network-detection-and-response-findings-enrich-amazon-security-lake-pisrehkb (остання дата звернення 1.12.2022)

35. Software Reuse in the Emerging Cloud Computing Era / Xiaodong Liu, Edinburgh Napier University / Hongji Yang, University of Leicester // IGI Global. -

2012. – P. 29

36. Amazon API Gateway URL: https://aws.amazon.com/api-gateway/ (остання дата звернення 1.12.2022)

37. Amazon Lambda URL: https://aws.amazon.com/lambda/ (остання дата звернення 1.12.2022)

38. AWS Lambda Functions Powered by AWS Graviton2 Processor URL: https://aws.amazon.com/blogs/aws/aws-lambda-functions-powered-by-aws-graviton2-processor-run-your-functions-on-arm-and-get-up-to-34-better-price-performance/ (остання дата звернення 1.12.2022)

39. Saving Plans On Lambda Functions URL: https://cloudwiry.com/lambda-functions/ (остання дата звернення 1.12.2022)

40. AWS Lambda Pricing URL: https://aws.amazon.com/lambda/pricing/ (остання дата звернення 1.12.2022)

ЛІСТІНГ ПРОГРАМИ

cpython.py

```
from timeit import default_timer as timer
import numpy as np
from pylab import imshow, show

def mandel(x, y, max_iters):
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real * z.real + z.imag * z.imag) >= 2:
            return i

    return max_iters

def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

image = np.zeros((500 * 2, 750 * 2), dtype=np.float64)

print('Пошук чисел Мандельброта з використанням CPython')
s = timer()
create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
e = timer()
print('Час виконання: %f секунд' % (e - s,))
imshow(image)
show()
```

cython.pyx

```
from timeit import default_timer as timer

import numpy as np
from pylab import imshow, show

import cython

@cython.cdivision(True)
```

```
def mandel(
    int w, int h, int max_iters,
    float min_x, float max_x, float min_y, float max_y
):
    image = np.zeros((h, w), dtype=np.float32)

    cdef float[:,:] view = image
    cdef complex z, c, I = 1j
    cdef int x, y, k
    cdef float dx, dy, real, im

    dx = <float>(max_x - min_x) / w
    dy = <float>(max_y - min_y) / h

    for x in range(w):
        real = min_x + x * dx
        for y in range(h):
            im = min_y + y * dy
            c = real + I * im
            z = 0
            for k in range(max_iters):
                z = z ** 2 + c
                if abs(z) >= 2:
                    break
                view[y,x] =  k
    return image

print('Пошук чисел Мандельброта з використанням Cython')
s = timer()
image = mandel(1500, 1000, 255, -2.0, 1.0, -1.0, 1.0)
e = timer()
print('Час виконання: %f секунд' % (e - s,))
imshow(image)
show()
```

cython_setup.py

```
from distutils.core import setup
from Cython.Build import cythonize

setup(ext_modules=cythonize('cy_thon.pyx'))
```

cy_thon.c

```
#include <string.h>
#include <intrin.h>
#include <pythread.h>
#include <complex.h>
#include <cstdlib.h>
#include "pythread.h"
#include <string.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include "pystate.h"
static int __Pyx_setup_reduce(PyObject* type_obj) {
   int ret = 0;
   PyObject *object_reduce = NULL;
   PyObject *object_getstate = NULL;
   PyObject *object_reduce_ex = NULL;
   PyObject *reduce = NULL;
   PyObject *reduce_ex = NULL;
   PyObject *reduce_cython = NULL;
   PyObject *setstate = NULL;
   PyObject *setstate_cython = NULL;
   PyObject *getstate = NULL;
#if CYTHON_USE_PYTYPE_LOOKUP
   getstate = _PyType_Lookup((PyTypeObject*)type_obj, __pyx_n_s_getstate);
#else
   getstate = __Pyx_PyObject_GetAttrStrNoError(type_obj, __pyx_n_s_getstate);
   if (!getstate && PyErr_Occurred()) {
      goto __PYX_BAD;
   }
#endif
   if (getstate) {
#if CYTHON_USE_PYTYPE_LOOKUP
      object_getstate = _PyType_Lookup(&PyBaseObject_Type, __pyx_n_s_getstate);
#else
      object_getstate = __Pyx_PyObject_GetAttrStrNoError((PyObject*)&PyBaseObject_Type,
__pyx_n_s_getstate);
      if (!object_getstate && PyErr_Occurred()) {
         goto __PYX_BAD;
      }
#endif
      if (object_getstate != getstate) {
         goto __PYX_GOOD;
      }
   }
#if CYTHON_USE_PYTYPE_LOOKUP
   object_reduce_ex = _PyType_Lookup(&PyBaseObject_Type, __pyx_n_s_reduce_ex); if
(!object_reduce_ex) goto __PYX_BAD;
#else
   object_reduce_ex = __Pyx_PyObject_GetAttrStr((PyObject*)&PyBaseObject_Type,
__pyx_n_s_reduce_ex); if (!object_reduce_ex) goto __PYX_BAD;
#endif
   reduce_ex = __Pyx_PyObject_GetAttrStr(type_obj, __pyx_n_s_reduce_ex); if
(unlikely(!reduce_ex)) goto __PYX_BAD;
   if (reduce_ex == object_reduce_ex) {
#if CYTHON_USE_PYTYPE_LOOKUP
      object_reduce = _PyType_Lookup(&PyBaseObject_Type, __pyx_n_s_reduce); if
(!object_reduce) goto __PYX_BAD;
#else
      object_reduce = __Pyx_PyObject_GetAttrStr((PyObject*)&PyBaseObject_Type,
__pyx_n_s_reduce); if (!object_reduce) goto __PYX_BAD;
#endif
```

```
    reduce = __Pyx_PyObject_GetAttrStr(type_obj, __pyx_n_s_reduce); if (unlikely(!reduce))
goto __PYX_BAD;
    if (reduce == object_reduce || __Pyx_setup_reduce_is_named(reduce,
__pyx_n_s_reduce_cython)) {
      reduce_cython = __Pyx_PyObject_GetAttrStrNoError(type_obj,
__pyx_n_s_reduce_cython);
      if (likely(reduce_cython)) {
        ret = PyDict_SetItem(((PyTypeObject*)type_obj)->tp_dict, __pyx_n_s_reduce,
reduce_cython); if (unlikely(ret < 0)) goto __PYX_BAD;
        ret = PyDict_DelItem(((PyTypeObject*)type_obj)->tp_dict, __pyx_n_s_reduce_cython);
if (unlikely(ret < 0)) goto __PYX_BAD;
      } else if (reduce == object_reduce || PyErr_Occurred()) {
        goto __PYX_BAD;
      }
      setstate = __Pyx_PyObject_GetAttrStr(type_obj, __pyx_n_s_setstate);
      if (!setstate) PyErr_Clear();
      if (!setstate || __Pyx_setup_reduce_is_named(setstate, __pyx_n_s_setstate_cython)) {
        setstate_cython = __Pyx_PyObject_GetAttrStrNoError(type_obj,
__pyx_n_s_setstate_cython);
        if (likely(setstate_cython)) {
          ret = PyDict_SetItem(((PyTypeObject*)type_obj)->tp_dict, __pyx_n_s_setstate,
setstate_cython); if (unlikely(ret < 0)) goto __PYX_BAD;
          ret = PyDict_DelItem(((PyTypeObject*)type_obj)->tp_dict,
__pyx_n_s_setstate_cython); if (unlikely(ret < 0)) goto __PYX_BAD;
        } else if (!setstate || PyErr_Occurred()) {
          goto __PYX_BAD;
        }
      }
      PyType_Modified((PyTypeObject*)type_obj);
    }
  }
  goto __PYX_GOOD;
__PYX_BAD:
  if (!PyErr_Occurred())
    PyErr_Format(PyExc_RuntimeError, "Unable to initialize pickling for %s",
((PyTypeObject*)type_obj)->tp_name);
  ret = -1;
__PYX_GOOD:
#if !CYTHON_USE_PYTYPE_LOOKUP
  Py_XDECREF(object_reduce);
  Py_XDECREF(object_reduce_ex);
  Py_XDECREF(object_getstate);
  Py_XDECREF(getstate);
#endif
  Py_XDECREF(reduce);
  Py_XDECREF(reduce_ex);
  Py_XDECREF(reduce_cython);
  Py_XDECREF(setstate);
  Py_XDECREF(setstate_cython);
  return ret;
}
```

```
/* PyObjectCallNoArg */
#if CYTHON_COMPILING_IN_CPYTHON
static CYTHON_INLINE PyObject* __Pyx_PyObject_CallNoArg(PyObject *func) {
#if CYTHON_FAST_PYCALL
    if (PyFunction_Check(func)) {
        return __Pyx_PyFunction_FastCall(func, NULL, 0);
    }
#endif
#ifdef __Pyx_CyFunction_USED
    if (likely(PyCFunction_Check(func) || __Pyx_CyFunction_Check(func)))
#else
    if (likely(PyCFunction_Check(func)))
#endif
    {
        if (likely(PyCFunction_GET_FLAGS(func) & METH_NOARGS)) {
            return __Pyx_PyObject_CallMethO(func, NULL);
        }
    }
    return __Pyx_PyObject_Call(func, __pyx_empty_tuple, NULL);
}
#endif

/* CLineInTraceback */
#ifndef CYTHON_CLINE_IN_TRACEBACK
static int __Pyx_CLineForTraceback(CYTHON_NCP_UNUSED PyThreadState *tstate, int c_line)
{
    PyObject *use_cline;
    PyObject *ptype, *pvalue, *ptraceback;
#if CYTHON_COMPILING_IN_CPYTHON
    PyObject **cython_runtime_dict;
#endif
    if (unlikely(!__pyx_cython_runtime)) {
        return c_line;
    }
    __Pyx_ErrFetchInState(tstate, &ptype, &pvalue, &ptraceback);
#if CYTHON_COMPILING_IN_CPYTHON
    cython_runtime_dict = _PyObject_GetDictPtr(__pyx_cython_runtime);
    if (likely(cython_runtime_dict)) {
        __PYX_PY_DICT_LOOKUP_IF_MODIFIED(
            use_cline, *cython_runtime_dict,
            __Pyx_PyDict_GetItemStr(*cython_runtime_dict, __pyx_n_s_cline_in_traceback))
    } else
#endif
    {
      PyObject *use_cline_obj = __Pyx_PyObject_GetAttrStr(__pyx_cython_runtime,
__pyx_n_s_cline_in_traceback);
      if (use_cline_obj) {
        use_cline = PyObject_Not(use_cline_obj) ? Py_False : Py_True;
        Py_DECREF(use_cline_obj);
      } else {
        PyErr_Clear();
        use_cline = NULL;
```

```
    }
  }
  if (!use_cline) {
      c_line = 0;
      (void) PyObject_SetAttr(__pyx_cython_runtime, __pyx_n_s_cline_in_traceback, Py_False);
  }
  else if (use_cline == Py_False || (use_cline != Py_True && PyObject_Not(use_cline) != 0)) {
      c_line = 0;
  }
  __Pyx_ErrRestoreInState(tstate, ptype, pvalue, ptraceback);
  return c_line;
}
#endif

/* CodeObjectCache */
static int __pyx_bisect_code_objects(__Pyx_CodeObjectCacheEntry* entries, int count, int
code_line) {
  int start = 0, mid = 0, end = count - 1;
  if (end >= 0 && code_line > entries[end].code_line) {
      return count;
  }
  while (start < end) {
      mid = start + (end - start) / 2;
      if (code_line < entries[mid].code_line) {
          end = mid;
      } else if (code_line > entries[mid].code_line) {
           start = mid + 1;
      } else {
          return mid;
      }
  }
  if (code_line <= entries[mid].code_line) {
      return mid;
  } else {
      return mid + 1;
  }
}
```

num_ba.py

```
from timeit import default_timer as timer
import numpy as np
from pylab import imshow, show

from numba import jit

@jit(nopython=True, fastmath=True)
def mandel(x, y, max_iters):
  i = 0
  c = complex(x, y)
  z = 0.0j
  for i in range(max_iters):
```

```
        z = z*z + c
        if (z.real * z.real + z.imag * z.imag) >= 2:
            return i

    return max_iters

@jit(nopython=True, fastmath=True)
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

image = np.zeros((500 * 2, 750 * 2), dtype=np.float64)

print('Пошук чисел Мандельброта з використанням Numba')
s = timer()
create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
e = timer()
print('Час виконання: %f секунд' % (e - s,))
imshow(image)
show()
```

tai_chi.py

```
from timeit import default_timer as timer
import numpy as np
from pylab import imshow, show

import taichi as ti

ti.init(arch=ti.cpu)

@ti.func
def complex_sqr(z):
  return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def create_fractal(
    min_x: ti.float32, max_x: ti.float32,
    min_y: ti.float32, max_y: ti.float32,
    image: ti.ext_arr(), iters: ti.int32
):
    height, width = image.shape[:2]
    dx = (max_x - min_x) / width
    dy = (max_y - min_y) / height
```

```
  for y, x in image:
    c = ti.Vector([min_x + x * dx, min_y + y * dy])
    z = ti.Vector([0.0, 0.0])
    for i in range(iters):
      z = complex_sqr(z) + c
      if z.norm() >= 2:
        break
      image[y, x] = i

image = np.zeros((500 * 2, 750 * 2), dtype=np.float64)

print('Пошук чисел Мандельброта з використанням Taichi')
s = timer()
create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
e = timer()
print('Час виконання: %f секунд' % (e - s,))
imshow(image)
show()
```

Додаток А

ВІДГУК

керівника спеціальної частини

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК

Наукового керівника Алексєєва М.О., д.т.н., професор каф.ПЗКС
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

на магістерську роботу

студента Гаврильченко Данило Олександровича
курсу II групи ___122М-21-2
спеціальності 122 Комп'ютерні науки
освітньої програми Інформаційні управляючи системи та технології
на тему Дослідження ефективності впровадження компіляторів мови програмування Python в цикл розробки ПЗ

Актуальність теми. Вирішена актуальна науково-технічна задача підвищення продуктивності програм, створених з використанням мови програмування Python, що дозволить значно зменшити витрати на хмарні сервіси

Мета досліджень. Підвищення ефективності виконання програмного Python, зменшення часу роботи, зменшення застосованих комп'ютерних ресурсів

Коротка характеристика розділів роботи

Практичне значення роботи. Проведено практичне тестування та оцінка ефективності альтернативних компіляторів, проведено економічне дослідження

Висновки та оцінка. Дипломна робота повністю відповідає вимогам, які пред'являються до кваліфікаційних робіт магістрів спеціальності 122 Комп'ютерні науки, має прикладний характер і заслуговує оцінки «відмінно», а Ільченко С.О. присвоєння відповідної кваліфікації.

Науковий керівник Алексєєв М.О., д.т.н., професор каф.ПЗКС
(прізвище, ім'я, по батькові, посада, місце роботи)

«___»______________20__ р. ____________________

Додаток Б

ВІДГУК

рецензента спеціальної частини

РЕЦЕНЗІЯ
на магістерську роботу

студента Гаврильченко Данила Олександровича
(прізвище, ім'я, по батькові)

курсу II групи 122М-21
кафедри програмного забезпечення комп'ютерних систем
спеціальності 122 Комп'ютерні науки
освітньої програми Інформаційни управляючи системи та технології
Тема роботи. Дослідження ефективності впровадження компіляторів мови програмування Python в цикл розробки ПЗ

Стисла характеристика розділів роботи.

Пропозиції, внесені студентом, рівень їх наукового обґрунтування

Практичне значення роботи.

Якість оформлення роботи
Під час виконання магістерської роботи Гаврильченко Д.О. показав хороші навички в аналізі даних, володіння математичними методами, також розуміння, вміння кваліфіковано будувати алгоритми.
Загальний висновок. Магістерська робота Гаврильченко Д.О. повністю відповідає вимогам, які пред'являються до кваліфікаційних робіт магістрів спеціальності 122 Комп'ютерні науки, має прикладний характер і заслуговує оцінки , а Гаврильченко Д.О. присвоєння відповідної кваліфікації.
(підготовленість студента до самостійної роботи як спеціаліста)

Оцінка магістерської роботи ____________________

Рецензент ____________________
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

«___»______________20__ р. ____________________
(підпис)

Додаток Г

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом_ Гаврильченко.doc	Пояснювальна записка до магістерської роботи. Документ Word.
Диплом_ Гаврильченко.pdf	Пояснювальна записка до до магістерської роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програм і результати компіляції
Презентація	
Презентація_Гаврильченко.ppt	Презентація до магістерської роботи