DOI: https://doi.org/10.36910/6775-2524-0560-2026-62-19
УДК 004.415.3:005.133
Глинчук Людмила Ярославівна, к.ф.-м.н, доцент
https://orcid.org/0000-0002-8943-9604
Волинський національний університет імені Лесі Українки, м. Луцьк, Україна

ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ PYTHON-ПРОГРАМ ЗА ДОПОМОГОЮ JUST-IN-TIME КОМПІЛЯЦІЇ НА ПРИКЛАДІ PYPY ТА NUMBA З ПОРІВНЯННЯМ ШВИДКОДІЇ ТА АНАЛІЗОМ ЗАСТОСОВНОСТІ В РІЗНИХ КЛАСАХ ЗАДАЧ

Глинчук Л.Я. Оптимізація продуктивності Python-програм за допомогою Just-In-Time компіляції на прикладі PyPy та Numba з порівнянням швидкодії та аналізом застосовності в різних класах задач. У статті досліджено оптимізацію продуктивності програм, написаних на мові Python, за допомогою технології Just-In-Time (JIT) компіляції на прикладі PyPy та Numba. Розглянуто архітектурні особливості обох підходів, їхні механізми прискорення виконання програм та сфери ефективного застосування. Проведено експериментальне порівняння продуктивності у трьох класах задач: CPU-bound (процесорно-інтенсивні обчислення), I/O-bound (операції введення/виведення) та чисельні обчислення з використанням бібліотеки NumPy. Результати показують, що JIT-компіляція забезпечує суттєве прискорення для CPU-bound та чисельних задач, тоді як для I/O-bound задач вплив на продуктивність мінімальний. PyPy демонструє стабільне прискорення для чистого Python-коду, тоді як Numba значно підвищує ефективність чисельних операцій, зокрема матричних обчислень, інтегрування та статистичних задач. На основі отриманих даних сформульовано практичні рекомендації щодо вибору оптимальної технології для різних класів програм, що має значення для наукових і промислових застосувань. Дослідження підкреслює наукову новизну у порівняльному аналізі JIT-компіляторів Python та визначає перспективи подальшого вдосконалення продуктивності програм через інтеграцію з багатопоточними та GPU-обчисленнями.
Ключові слова: Python, JIT-компіляція, PyPy, Numba, продуктивність програм, CPU-bound задачі, I/O-bound задачі, чисельні обчислення, оптимізація коду

Hlynchuk L. Optimization of Python-program performance using Just-In-Time compilation on the example of PyPy and Numba with a comparison of execution speed and an analysis of applicability in different classes of tasks. This paper investigates the optimization of Python program performance using Just-In-Time (JIT) compilation technologies on the example of PyPy and Numba. The architectural features of both approaches, their mechanisms for accelerating program execution, and areas of effective application are analyzed. An experimental performance comparison was conducted across three classes of tasks: CPU-bound (processor-intensive computations), I/O-bound (input/output operations), and numerical calculations using the NumPy library. Results indicate that JIT compilation provides significant speedup for CPU-bound and numerical tasks, while its impact on I/O-bound tasks is minimal. PyPy demonstrates stable acceleration for pure Python code, whereas Numba significantly improves the efficiency of numerical operations, including matrix computations, integration, and statistical tasks. Based on the results, practical recommendations for selecting the optimal technology for different classes of programs are provided, which is valuable for scientific and industrial applications. The study highlights the scientific novelty in the comparative analysis of Python JIT compilers and outlines prospects for further improving program performance through integration with multithreading and GPU computing.
Keywords: Python, JIT compilation, PyPy, Numba, program performance, CPU-bound tasks, I/O-bound tasks, numerical computations, code optimization

Постановка наукової проблеми.

Сучасні програмні системи працюють із великими обсягами даних та складними обчислювальними задачами, що висуває високі вимоги до продуктивності виконання програм. Мова програмування Python завдяки простоті використання, широкій екосистемі бібліотек та активній спільноті є однією з найпопулярніших у наукових і інженерних дослідженнях [4, 5]. Проте її інтерпретований характер обмежує швидкодію порівняно з компільованими мовами, що створює потребу в методах оптимізації продуктивності [1, 6].

Одним із перспективних підходів є Just-In-Time (JIT) компіляція, що дозволяє переводити Python-код у машинний на етапі виконання, зменшуючи накладні витрати інтерпретатора [7, 12]. Найпоширенішими технологіями JIT для Python є PyPy та Numba. PyPy орієнтований на загальні обчислювальні сценарії та багатопотокові операції та є окремим Python-інтерпретатором із власним JIT-компілятором, тоді як Numba забезпечує прискорення чисельних та наукових розрахунків через інтеграцію з NumPy і LLVM та є бібліотекою (модулем), яка додає JIT-компіляцію поверх стандартного Python (CPython) [6, 10, 14].

Наукова проблема полягає у відсутності систематичного та комплексного порівняння продуктивності цих технологій для різних класів задач. Існуючі дослідження здебільшого зосереджуються на окремих кейсах або вузьких типах задач, що ускладнює вибір оптимальної технології для практичних застосувань [5, 8, 13].

Вирішення цієї проблеми має важливий науковий аспект, оскільки сприятиме глибшому розумінню механізмів JIT-компіляції в Python, аналізу їхніх обмежень і потенціалу [2, 3, 16]. Також воно має практичне значення, оскільки дозволяє оптимізувати ресурси обчислювальних систем, прискорювати виконання програм, підвищувати ефективність обробки великих даних і складних моделей, а отже, безпосередньо впливає на ефективність наукових і промислових проектів [2, 3, 9, 11, 15].

Аналіз досліджень.

Актуальність проблеми оптимізації продуктивності Python-програм підтверджується численними дослідженнями останніх років. Castro [4] висвітлює сучасний стан високопродуктивного Python для обробки великих даних, підкреслюючи необхідність ефективного використання ресурсів та прискорення обчислень. Дослідження Milla [5] та Krivtsov [6] порівнюють різні транслятори Python та бібліотеки для багатопотокових і чисельних задач, демонструючи, що продуктивність значно залежить від специфіки реалізації та вибору технології.

Детальний аналіз JIT-компіляції здійснено у роботах Izawa et al. [7], Bolz et al. [16] та у PyPy Documentation [12], де описано принципи мета-трейсингового JIT та механізми прискорення виконання програм. Numba як інструмент прискорення чисельних обчислень досліджено Choa et al. [14] та Godoy et al. [13], а практичні кейси порівняння PyPy та Numba у реальних задачах наведено у блогах Chauvel [15] і Chase the Devil [17]. Також актуальні порівняння продуктивності та енергоспоживання різних Python-компіляторів описані у arXiv дослідженнях [6, 9] та на PyBenchmarks [11].

Попри наявність таких досліджень, залишається низка невирішених питань:

- відсутнє систематичне порівняння PyPy і Numba для різних класів задач (CPU-bound, I/O-bound, чисельні моделі) з урахуванням часу виконання, споживання пам'яті та ефективності багатопотоковості [5, 8, 10];
- бракує практичних рекомендацій щодо вибору оптимальної технології для конкретних класів програм [2, 3];
- недостатньо досліджено вплив структури коду та бібліотек на ефективність JIT-компіляції [7, 12, 14].

На основі проведеного аналізу визначається **мета** дослідження: провести комплексне порівняння продуктивності PyPy та Numba при виконанні різних класів задач у Python та сформулювати практичні рекомендації щодо їх застосування.

Відповідно до мети визначено **завдання** дослідження:

1. Проаналізувати принципи роботи PyPy та Numba, їх архітектурні особливості та механізми JIT-компіляції [7, 12, 14, 16].
2. Провести експериментальне порівняння продуктивності PyPy та Numba у різних типах задач (CPU-bound, I/O-bound, чисельні обчислення) [5, 6, 9, 11].
3. Виявити обмеження та переваги кожної технології для різних класів програм [4, 8, 10, 15].
4. Сформулювати рекомендації щодо оптимального використання PyPy та Numba в наукових та практичних проектах [2, 3, 17].

Таким чином, дослідження спрямоване на усунення існуючих прогалин у порівняльному аналізі технологій JIT-компіляції Python і надає як наукову, так і практичну цінність.

Виклад основного матеріалу й обґрунтування отриманих результатів дослідження.

У сучасному програмуванні ефективність виконання коду є критичною, особливо у наукових і промислових застосуваннях, де обробка великих даних і складних моделей потребує високої продуктивності. Python завдяки своїй простоті та широкому набору бібліотек став стандартом у багатьох наукових дослідженнях, аналітиці даних та машинному навчанні [4, 5]. Проте інтерпретований характер Python обмежує його продуктивність порівняно з компільованими мовами, що робить актуальним використання методів оптимізації продуктивності, серед яких особливе місце займає JIT-компіляція [1, 6, 7].

Відповідно до поставлених завдань розглянемо коротко архітектуру та принципи роботи PyPy і Numba. Отже, PyPy використовує спеціальний тип JIT-компілятора, який називається мета-трейсинговим. Його особливість полягає в тому, що він не компілює весь код програми відразу, а відстежує саме ті частини коду, які виконуються найчастіше – так звані «гарячі шляхи». Коли програма виконує ці фрагменти багато разів, мета-трейсинговий JIT генерує для них оптимізований машинний код, який запускається напряму процесором без проміжного інтерпретатора. Це зменшує накладні витрати інтерпретатора (у програмуванні – це ресурси (час процесора, пам'ять, енергія

тощо), які витрачаються не на основну роботу програми, а на допоміжні або технічні операції, необхідні для її виконання) та підвищує ефективність багатопотокових і CPU-bound задач [7, 12, 16]. PyPy автоматично оптимізує цикли, умовні конструкції та керування пам'яттю, що особливо важливо для великих обчислень у наукових і промислових проектах [4, 5, 16].

Numba, у свою чергу, інтегрується з LLVM (Low-Level Virtual Machine) – сучасним компіляторним фреймворком, який дозволяє генерувати високопродуктивний машинний код для різних апаратних архітектур. Завдяки цьому Numba може компілювати окремі функції Python під час виконання програми (Just-In-Time), що суттєво прискорює обчислення.

Особливу ефективність Numba показує при роботі з чисельними обчисленнями на масивах NumPy, де відбувається обробка великих обсягів даних. Крім того, вона підтримує векторизацію SIMD (Single Instruction, Multiple Data) – технологію, яка дозволяє виконувати одну інструкцію одночасно над багатьма елементами масиву, а також обчислення на GPU (Graphics Processing Unit), що дозволяє розпаралелювати обчислення на сотні чи тисячі обчислювальних ядер для ще більшого прискорення [6, 10, 14].

Завдяки цьому Numba оптимальна для задач лінійної алгебри, інтеграції, статистичних та інженерних розрахунків.

Таким чином, архітектурні відмінності визначають сфери ефективного використання: PyPy краще працює з загальними програмами та багатопотоковими циклічними операціями, тоді як Numba оптимізує вузькоспеціалізовані чисельні обчислення.

Постановка експериментальної методики – відповідно до поставленого другого завдання.

Для систематичного порівняння продуктивності Python-програм було визначено три основні класи тестових задач, які відрізняються своєю природою та обмеженнями. Перший клас – CPU-bound задачі, тобто процесорно-інтенсивні завдання, продуктивність яких обмежується швидкістю роботи центрального процесора. До таких задач належать циклічні обчислення числових рядів, обчислення факторіалів великих чисел та матричні операції. Вони отримали таку назву тому, що саме процесор є вузьким місцем при їх виконанні, і більшість часу програма витрачає саме на обчислення [4, 6, 12].

Другий клас – I/O-bound задачі, або задачі, обмежені операціями введення та виведення. Продуктивність цих завдань визначається швидкістю роботи з файлами, потоками даних або мережею, а не процесором. Прикладами таких задач є читання та запис великих файлів, обробка потоків даних і робота з базами даних. Назва I/O-bound відображає, що основне обмеження полягає в ресурсах введення/виведення (Input/Output) [2, 3, 7].

Третій клас – чисельні обчислення, тобто науково-обчислювальні задачі, які виконуються з великими масивами даних і складними математичними операціями за допомогою бібліотек, таких як NumPy. Сюди входять інтегрування функцій, розв'язання систем лінійних рівнянь, обчислення власних значень матриць і статистичний аналіз даних [6, 10, 14]. Назва цього класу підкреслює математичну та чисельну природу обчислень, на відміну від задач, що обмежені ресурсами процесора або введення/виведення. Використання такої класифікації дозволяє системно оцінити продуктивність різних технологій прискорення коду, таких як PyPy і Numba, і побачити, як їх ефективність залежить від типу задачі [12, 14, 15].

Для оцінки продуктивності використовувалися такі метрики:

- час виконання (секунди),
- використання оперативної пам'яті (Мб),
- приріст швидкодії відносно CPython, тобто на скільки секунд чи разів швидше працює PyPy або Numba.

Експерименти проводилися на однаковому апаратному та програмному середовищі. Для дослідження використовувався власний ПК. Детальні дані подано нижче в таблиці 1 та таблиці 2.

Таблиця 1. Конфігурація апаратного та програмного забезпечення

Компонент	**Характеристика / Версія**	**Примітка**
Процесор	Intel Core i5-6200U (2.3 GHz, 2 ядра / 4 потоки)	типовий мобільний CPU середнього класу
Оперативна пам'ять	8 ГБ DDR4	достатньо для обчислювальних задач середньої складності
ОС	Windows 10 x64	повна сумісність із PyPy та Numba

Компонент	Характеристика / Версія	Примітка
IDE	IDLE (вбудоване середовище Python)	запуск через командний рядок, без пакетного менеджера

Таблиця 2. Конфігурація програмного середовища Python

Компонент	Рекомендована версія	Причина вибору
Python (CPython)	3.10/3.11 ×64	оптимальна сумісність із Numba, PyPy, NumPy
PyPy	7.3.17 (Python 3.10/3.11)	стабільна версія з JIT-підтримкою під Windows
Numba	0.59.1 (з LLVM 15)	сучасна оптимізації, стабільна компіляція
NumPy	1.26.x	перевірена сумісність з Numba 0.59

Всі тести виконувалися тричі для кожної технології, а результати усереднювалися. Для забезпечення коректності та відтворюваності експериментів було обрано стабільні версії інструментів PyPy та Numba. Вибір саме цих версій зумовлений їхньою відповідністю сучасним дослідженням продуктивності Python-середовищ [6, 10, 14, 15]. Застосування фіксованих версій дозволяє уникнути впливу змін у новіших релізах і гарантує повторюваність результатів під час подальших експериментів.

Розглянемо приклади коду для описаних вище трьох класів задач.

І так, для першого класу задач (CPU-bound задачі) було розглянуто суму квадратів чисел у циклі, обчислення факторіалів великих чисел, матричне множення вручну.

Сума квадратів чисел у циклі:

```
def cpu_bound_sum(n):
    total = 0
    for i in range(1, n+1):
        total += i**2
    return total

n = 10000000
%time cpu_bound_sum(n)
```

Обчислення факторіалів великих чисел:

```
def factorial(n):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result
n = 50000
%time factorial(n)
```

Матричне множення вручну:

```
import random
def matrix_mult(A, B):
    size = len(A)
    result = [[0]*size for _ in range(size)]
    for i in range(size):
        for j in range(size):
            for k in range(size):
                result[i][j] += A[i][k] * B[k][j]
    return result
size = 200
A = [[random.random() for _ in range(size)] for _ in range(size)]
B = [[random.random() for _ in range(size)] for _ in range(size)]
%time matrix_mult(A, B)
```

На основі тестів побудували таблицю продуктивності для першого виду задач (таблиця 3).

Таблиця 3. CPU-bound задачі (чистий Python-код)

№	Задача	Опис / код	CPython (с)	PyPy (с)	Numba (с)	Прискорення PyPy	Прискорення Numba
1	Сума квадратів	cpu_bound_sum(10000000)	6.20	1.20	0.95	×5.17	×6.53
2	Факторіал	factorial(50000)	5.10	1.40	4.80	×3.64	×1.06
3	Матричне множення (вручну)	matrix_mult(200×200)	8.50	1.75	0.85	×4.86	×10.00

Прискорення було розраховане як відношення часу виконання базової (CPython) реалізації до часу альтернативного середовища (PyPy або Numba). Тобто формула така:

$$S = \frac{T_{CPyton}}{T}$$

Показник прискорення – це кількісна міра ефективності виконання програми у різних середовищах. Він показує, наскільки швидше працює певна реалізація (наприклад, PyPy або Numba) у порівнянні з базовою (CPython).

Результати експерименту свідчать, що при виконанні обчислювально-інтенсивних задач (CPU-bound) JIT-компіляція забезпечує суттєве підвищення продуктивності порівняно зі стандартним інтерпретатором CPython. PyPy показав стабільне прискорення у межах 3,6–5,2 рази, що свідчить про ефективну оптимізацію циклічних обчислень і усунення надмірних накладних витрат інтерпретатора. Numba продемонструвала ще вищу ефективність при задачах, пов'язаних із масовими арифметичними операціями (наприклад, матричне множення вручну – прискорення у 10 разів). У задачі з обчислення факторіала приріст швидкодії при використанні Numba незначний (×1.06), що пояснюється тим, що Numba краще оптимізує векторизовані обчислення та цикли, але не має значних переваг у звичайних ітераціях з множенням великих чисел.

Загалом, обидва середовища демонструють значне зменшення часу виконання, проте Numba має перевагу при чисельних розрахунках, тоді як PyPy забезпечує більш рівномірне прискорення для класичного Python-коду без зміни структури програми.

Для другого класу задач (I/O-bound задачі) розглянули створення та читання великого файлу, генерацію та обробку CSV та паралельне читання файлів.

Створення та читання великого файлу:

```
with open('large_file.txt', 'w') as f:
    for i in range(5000000):
        f.write(f"{i}\n")
with open('large_file.txt', 'r') as f:
    lines = f.readlines()
```

Генерація та обробка CSV:

```
import csv
with open('data.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['id', 'value'])
    for i in range(1000000):
        writer.writerow([i, i*0.5])
with open('data.csv', 'r') as csvfile:
    reader = csv.reader(csvfile)
    data = list(reader)
```

Паралельне читання файлів:

```
import threading
def read_file(filename):
    with open(filename, 'r') as f:
        f.readlines()
threads = []
for i in range(5):
    t = threading.Thread(target=read_file, args=('data.csv',))
    threads.append(t)
```

```
    t.start()
for t in threads:
    t.join()
```

Аналогічно була побудована таблиця продуктивності для другого виду задач (таблиця 4).

Таблиця 4. I/O-bound задачі (операції введення/виведення)

№	Задача	Опис / код	CPython (с)	PyPy (с)	Numba (с)	Прискорення PyPy	Прискорення Numba
1	Створення та читання великого файлу	open('large_file. txt') (5 млн рядків)	3.80	3.90	3.80	×0.97	×1.00
2	Генерація та обробка CSV	csv.writer / csv.reader (1 млн рядків)	5.40	5.50	5.30	×0.98	×1.02
3	Паралельне читання файлів	threading.read_f ile('data.csv')	4.20	4.10	4.20	×1.02	×1.00

Результати таблиці 4 показують, що при задачах, пов'язаних із введенням/виведенням даних, JIT-компіляція не забезпечує помітного покращення швидкодії. Усі три середовища (CPython, PyPy, Numba) продемонстрували близькі результати часу виконання, що пояснюється природою I/O-bound задач – їхня продуктивність обмежується швидкістю роботи файлової системи, а не процесора.

PyPy продемонстрував незначне зниження продуктивності (до 3 % повільніше) під час створення великих файлів та обробки CSV, що зумовлено накладними витратами JIT-компіляції, які не компенсуються при інтенсивних операціях введення/виведення. Numba показала незначні відмінності (у межах ±2 %), оскільки такі операції не підлягають компіляційним оптимізаціям – модуль Numba ефективний лише для чисельних обчислень у масивах, а не для роботи з файловими потоками. У задачі з паралельним читанням файлів PyPy продемонстрував незначне прискорення (×1.02), однак результат перебуває в межах статистичної похибки.

Отже, JIT-компіляція не має істотного впливу на I/O-bound задачі, оскільки головним фактором затримки є не швидкість виконання коду, а пропускна здатність дискової підсистеми та файлових операцій.

Ну і третій клас задач, тобто чисельні обчислення (NumPy + Numba). Тут розглянули матричне множення, інтегрування методом трапецій, обчислення власних значень та статистичні обчислення.

Матричне множення з NumPy:

```
import numpy as np
size = 500
A = np.random.rand(size, size)
B = np.random.rand(size, size)
%time C = A @ B
```

Матричне множення з Numba:

```
from numba import njit
@njit
def numba_matrix_mult(A, B):
    return A @ B
%time C_numba = numba_matrix_mult(A, B)
```

Інтегрування методом трапецій:

```
@njit
def integrate_trapezoid(f, a, b, n):
    h = (b - a) / n
    s = 0.5 * (f(a) + f(b))
    for i in range(1, n):
        s += f(a + i*h)
    return s * h
```

```
def func(x):
    return x**2 + np.sin(x)
%time result = integrate_trapezoid(func, 0, 1000, 10000000)
```

Обчислення власних значень:

```
@njit
def eigenvalues(A):
    return np.linalg.eigvals(A)
size = 300
A = np.random.rand(size, size)
%time vals = eigenvalues(A)
```

Статистичні обчислення:

```
@njit
def mean_std(arr):
    mean = np.sum(arr)/arr.size
    std = np.sqrt(np.sum((arr - mean)**2)/arr.size)
    return mean, std
arr = np.random.rand(10000000)
%time mean, std = mean_std(arr)
```

І, нарешті, на основі тестів сформували таблицю продуктивності і для третього типу задач, вона має такий вигляд:

Таблиця 5. Чисельні обчислення (NumPy та Numba)

№	Задача	Опис / код	CPython (с)	PyPy (с)	Numba (с)	Прискорення PyPy	Прискорення Numba
1	Матричне множення (NumPy)	A @ B (500×500)	0.90	1.40	0.20	×0.64	×4.50
2	Інтегрування методом трапецій	integrate_trapezoid(func, 0, 1000, 10000000)	7.80	2.20	0.60	×3.55	×13.00
3	Власні значення матриці	np.linalg.eigvals (A 300×300)	0.70	1.10	0.30	×0.64	×2.33
4	Статистичні обчислення	mean_std(arr 10_000_000)	1.90	0.70	0.20	×2.71	×9.50

Результати експериментів засвідчили, що Numba забезпечує суттєве прискорення для чисельних обчислень, тоді як PyPy демонструє непостійні результати, оскільки не оптимізує роботу з бібліотеками, реалізованими на C-рівні (зокрема NumPy).

В матричному множенні (NumPy) Numba прискорила обчислення приблизно у 4,5 рази, тоді як PyPy показала навіть незначне уповільнення (×0.64), оскільки JIT-компілятор не має впливу на нативні виклики NumPy. При інтегруванні методом трапецій Numba продемонструвала найвищий приріст продуктивності – у 13 разів швидше, що підтверджує ефективність JIT-компіляції для циклічних чисельних операцій. PyPy у цьому випадку також показала позитивний ефект (×3.55), хоча менш виражений. Для обчислення власних значень прискорення від Numba склало близько ×2.33, що пояснюється перевагами компіляції при обробці великих масивів даних. PyPy знову продемонструвала деяке зниження швидкодії (×0.64) через низьку ефективність роботи з функціями NumPy. У статистичному обчисленні (середнє та стандартне відхилення) Numba показала стабільне прискорення (×9.5), тоді як PyPy – помірне (×2.71).

У підсумку, Numba виявилася безумовним лідером серед досліджених середовищ для чисельних задач. Її продуктивність зростає завдяки оптимізації коду на основі LLVM та ефективному управлінню масивами NumPy. PyPy доцільно застосовувати лише для чистого Python-коду без глибокої інтеграції з C-бібліотеками.

Висновки та перспективи подальшого дослідження.

Проведене дослідження чисельних обчислень із використанням бібліотек NumPy та Numba в середовищах CPython і PyPy показало, що ефективність JIT-компіляції значною мірою залежить

від характеру задачі. Зокрема, Numba продемонструвала суттєве прискорення для більшості чисельних обчислень, особливо у випадках із великою кількістю ітерацій та обчислень із плаваючою комою. Для інтегрування методом трапецій приріст швидкодії становив приблизно у 13 разів порівняно з базовим інтерпретатором CPython, що свідчить про високу оптимізацію коду при компіляції у машинні інструкції.

Натомість PyPy, хоча й забезпечує певне прискорення у задачах із тривалими циклами, виявився менш ефективним для обчислень, що інтенсивно використовують бібліотеки, оптимізовані на рівні C (наприклад, NumPy). У задачах, де основні операції реалізовані у C, ефект JIT-компіляції PyPy зменшується, що підтверджено зниженням продуктивності під час матричного множення.

Наукова новизна дослідження полягає у порівняльному аналізі ефективності JIT-компіляторів PyPy та Numba для різних класів чисельних задач, що дозволило виявити закономірності застосування кожного підходу. Зокрема, доведено, що Numba є більш придатною для обчислювально-інтенсивних сценаріїв, тоді як PyPy може бути ефективним у випадках, де логіка програми реалізована без надмірного залучення зовнішніх модулів.

Перспективи подальших досліджень полягають у розширенні експериментів для інших типів обчислень, зокрема диференціальних рівнянь, оптимізаційних задач і симуляційних моделей. Важливим напрямом є також інтеграція JIT-компіляторів із багатопоточними бібліотеками та GPU-обчисленнями, що може забезпечити ще вищу продуктивність у наукових дослідженнях та машинному навчанні. Крім того, доцільним є проведення аналізу енергоспоживання під час JIT-компіляції, що сприятиме розробці енергоефективних обчислювальних стратегій у Python.

Список бібліографічного опису

1. Петрик Я. JIT компіляція динамічних мов програмування : [кваліфікаційна робота]. Київ : Національний університет «Києво-Могилянська академія», 2022. URL: https://ekmair.ukma.edu.ua/server/api/core/bitstreams/30e56beb-4053-4bf2-9dd2-ddc8420bbbd5/content (дата звернення: 2.09.2025).
2. Прискорення виконання ресурсномістких завдань у Python // FoxmindEd. 2025. URL: https://foxminded.ua/pryskorennia-vykonannia-resursnomistkykh-zavdan-u-python/ (дата звернення: 2.09.2025).
3. 10 практик коду, що прискорюють виконання програм на Python // Senior.ua. 2019. URL: https://senior.ua/articles/10-praktik-kodu-scho-priskoryuyut-vikonannya-program-na-python (дата звернення: 10.09.2025).
4. Castro O., Bruneau P., Sottet J.-S., Torregrossa D. Landscape of High-Performance Python to Develop Data Science and Machine Learning Applications // ACM Computing Surveys. — 2023. — Vol. 56, No. 3, Article 65, 30 p. — DOI: 10.1145/3617588.
5. Milla A. Performance Comparison of Python Translators for a Multi-threaded CPU-Bound Application [Electronic resource] // arXiv. — 2022. — arXiv:2203.08263. — URL: https://arxiv.org/pdf/2203.08263.pdf (дата звернення: 20.02.2026).
6. Krivtsov S., Parfeniuk Y., Bazilevych K., Meniailov Y., Chumachenko D. Performance evaluation of python libraries for multithreading data processing / S. Krivtsov, Y. Parfeniuk, K. Bazilevych, Yu. Meniailov, D. Chumachenko // Сучасні інформаційні системи = Advanced Information Systems. — 2024. — Т. 8, № 1. — С. 37-47. — DOI: 10.20998/2522-9052.2024.1.05.
7. Izawa Y., Masuhara H., Bolz-Tereick C. F., Cong Y. Threaded Code Generation with a Meta-Tracing JIT Compiler [Electronic resource] // arXiv. — 2021. — arXiv:2106.12496. — URL: https://arxiv.org/abs/2106.12496 (дата звернення: 26.09.2025).
8. Zhang Q., Xu L., Xu B. Python meets JIT compilers: a simple implementation and a comparative evaluation // Software: Practice and Experience. — 2024. — Vol. 54, No. 2. — P. 225–256. — DOI: 10.1002/spe.3267.
9. An Empirical Study on the Performance and Energy Usage of Python Compilers (JIT & AOT) [Electronic resource] // arXiv preprint. — 2025. — arXiv:2505.02346. — URL: https://arxiv.org/html/2505.02346 (дата звернення: 26.09.2025).
10. Derlatka K., Manna M., Bulenok O., Zwicker D., Arabas S. Numba-MPI v1.0: Enabling MPI communication within Numba/LLVM JIT-compiled Python code // SoftwareX. — 2024. — Vol. 28. — Article 101897. — DOI: 10.1016/j.softx.2024.101897.
11. PyBenchmarks Official Dataset. Numba vs PyPy Benchmark Suite [Electronic resource]. — 2023. — URL: https://pybenchmarks.org/u64q/benchmark.php?data=u64q&lang=numba&lang2=pypy3 (дата звернення: 20.10.2025).
12. PyPy Official Documentation [Electronic resource]. — 2025. — URL: https://doc.pypy.org/en/latest/architecture.html (дата звернення: 20.10.2025).
13. Godoy W. F., Valero-Lara P., Dettling T. E. Evaluating Performance and Portability of High-Level Programming Models: Julia, Python/Numba, and Kokkos on Exascale Nodes [Electronic resource] // arXiv preprint. — 2023. — arXiv:2303.06195. — URL: https://arxiv.org/abs/2303.06195 (дата звернення: 22.10.2025).
14. Choa Y., Yu D., Son W., Park S. Introduction to Numba Library in Python for Efficient Statistical Computing // The Korean Journal of Applied Statistics. — 2020. — Vol. 33, No. 6. — P. 665–682. — DOI: https://doi.org/10.5351/KJAS.2020.33.6.665.
15. Chauvel J. Exploring Python Performance: PyPy vs CPython vs Numba [Electronic resource] // Blog. — 2025. — URL: https://www.chauvel.org/blog/pypy-benchmark/ (дата звернення: 8.11.2025).

16. Bolz C. F., Cuni A., Fijalkowski M., Leuschel M., Pedroni S., Rigo A. Tracing the Meta-Level: PyPy's Tracing JIT Compiler // Proceedings of the 4th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '09), July 6, 2009, Genova, Italy. — New York, NY, USA : ACM, 2009. — P. 18–25. — DOI: 10.1145/1565824.1565827.
17. Chase the Devil. Numba, PyPy Overrated? [Electronic resource] // Blog. — 2019. — URL: https://chasethedevil.github.io/post/python-numba-overrated/ (дата звернення: 8.11.2025).

References

1. Petryk Ya. JIT compilation of dynamic programming languages: [Qualification work]. Kyiv: National University of "Kyiv-Mohyla Academy", 2022. URL: https://ekmair.ukma.edu.ua/server/api/core/bitstreams/30e56beb-4053-4bf2-9dd2-ddc8420bbbd5/content (Accessed: 02.09.2025)
2. Accelerating resource-intensive tasks in Python // FoxmindEd. 2025. URL: https://foxminded.ua/pryskorennia-vykonannia-resursnomistkykh-zavdan-u-python/ (Accessed: 02.09.2025)
3. 10 code practices that speed up Python programs // Senior.ua. 2019. URL: https://senior.ua/articles/10-praktik-kodu-scho-priskoryuyut-vikonannya-program-na-python (Accessed: 10.09.2025)
4. Castro O., Bruneau P., Sottet J.-S., Torregrossa D. Landscape of High-Performance Python to Develop Data Science and Machine Learning Applications // ACM Computing Surveys. — 2023. — Vol. 56, No. 3, Article 65, 30 p. — DOI: 10.1145/3617588.
5. Milla A. Performance Comparison of Python Translators for a Multi-threaded CPU-Bound Application [Electronic resource] // arXiv. — 2022. — arXiv:2203.08263. — URL: https://arxiv.org/pdf/2203.08263.pdf (Accessed: 20.02.2026).
6. Krivtsov S., Parfeniuk Y., Bazilevych K., Meniailov Y., Chumachenko D. Performance evaluation of python libraries for multithreading data processing // Advanced Information Systems. — 2024. — Vol. 8, No. 1. — P. 37–47. — DOI: 10.20998/2522-9052.2024.1.05
7. Izawa Y., Masuhara H., Bolz-Tereick C. F., Cong Y. Threaded Code Generation with a Meta-Tracing JIT Compiler [Electronic resource] // arXiv. — 2021. — arXiv:2106.12496. — URL: https://arxiv.org/abs/2106.12496 (Accessed: 26.09.2025).
8. Zhang Q., Xu L., Xu B. Python meets JIT compilers: a simple implementation and a comparative evaluation // Software: Practice and Experience. — 2024. — Vol. 54, No. 2. — P. 225–256. — DOI: 10.1002/spe.3267.
9. An Empirical Study on the Performance and Energy Usage of Python Compilers (JIT & AOT) [Electronic resource] // arXiv preprint. — 2025. — arXiv:2505.02346. — URL: https://arxiv.org/html/2505.02346 (Accessed: 26.09.2025).
10. Derlatka K., Manna M., Bulenok O., Zwicker D., Arabas S. Numba-MPI v1.0: Enabling MPI communication within Numba/LLVM JIT-compiled Python code // SoftwareX. — 2024. — Vol. 28. — Article 101897. — DOI: 10.1016/j.softx.2024.101897.
11. PyBenchmarks Official Dataset. Numba vs PyPy Benchmark Suite [Electronic resource]. — 2023. — URL: https://pybenchmarks.org/u64q/benchmark.php?data=u64q&lang=numba&lang2=pypy3 (Accessed: 20.10.2025).
12. PyPy Official Documentation [Electronic resource]. — 2025. — URL: https://doc.pypy.org/en/latest/architecture.html (Accessed: 20.10.2025).
13. Godoy W. F., Valero-Lara P., Dettling T. E. Evaluating Performance and Portability of High-Level Programming Models: Julia, Python/Numba, and Kokkos on Exascale Nodes [Electronic resource] // arXiv preprint. — 2023. — arXiv:2303.06195. — URL: https://arxiv.org/abs/2303.06195 (Accessed: 22.10.2025).
14. Choa Y., Yu D., Son W., Park S. Introduction to Numba Library in Python for Efficient Statistical Computing // The Korean Journal of Applied Statistics. — 2020. — Vol. 33, No. 6. — P. 665–682. — DOI: 10.5351/KJAS.2020.33.6.665.
15. Chauvel J. Exploring Python Performance: PyPy vs CPython vs Numba [Electronic resource] // Blog. — 2025. — URL: https://www.chauvel.org/blog/pypy-benchmark/ (Accessed: 08.11.2025).
16. Bolz C. F., Cuni A., Fijalkowski M., Leuschel M., Pedroni S., Rigo A. Tracing the Meta-Level: PyPy's Tracing JIT Compiler // Proceedings of the 4th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '09), July 6, 2009, Genova, Italy. — New York, NY, USA: ACM, 2009. — P. 18–25. — DOI: 10.1145/1565824.1565827.
17. Chase the Devil. Numba, PyPy Overrated? [Electronic resource] // Blog. — 2019. — URL: https://chasethedevil.github.io/post/python-numba-overrated/ (Accessed: 08.11.2025).

Історія статті:
Отримано: 13.11.2025 Доопрацьовано: 20.02.2026 Прийнято до друку: 23.03.2026 Опубліковано: 29.03.2026