UNIWERSYTET GDAŃSKI
WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI

Krzysztof Wiśniewski
numer albumu: 274276

Kierunek studiów: Bioinformatyka
Specjalność: Ogólna

Optymalizacja oprogramowania do detekcji splątania kwantowego

Praca licencjacka
wykonana
pod kierunkiem
dr hab. Marcin Wieśniak, prof. UG

Gdańsk 2023

Spis treści

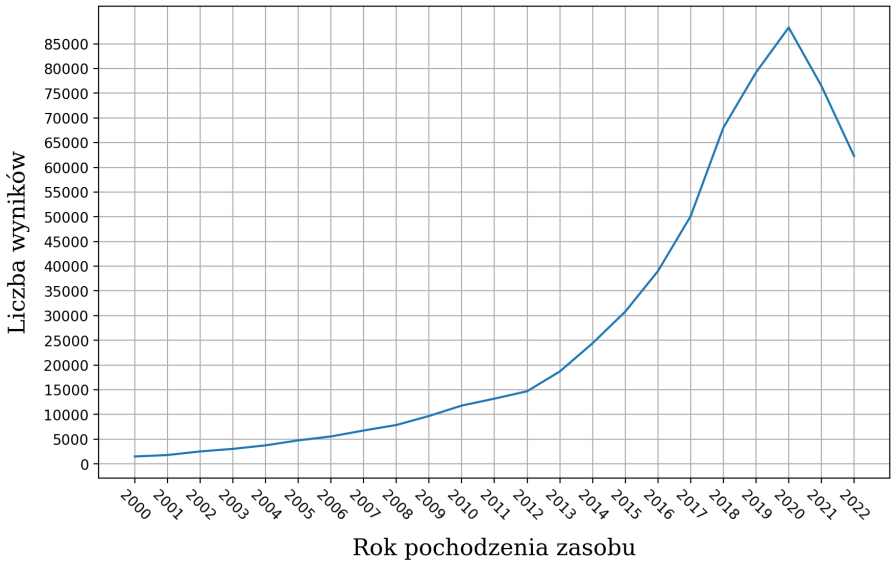
Streszczenie

W tej pracy przeprowadzam analizę efektywności metod optymalizacji czasu pracy programu CSSFinder służącego do detekcji splątania kwantowego. Podejmowane przeze mnie wysiłki skupiają się w głównej mierze na doborze lepszych narzędzi pozwalających na wydajniejsze prowadzenie dużych ilości obliczeń macierzowych. Rozważane będą skutki re-implementacji w języku Python z wykorzystaniem bibliotek NumPy[9][13], Numba[6][8], Cython[4][18] oraz re-implementacja w języku Rust[32] z wykorzystaniem skrzyni[1]. Każda z implementacji została przetestowana na specjalnie dobranym zestawie danych, a wyniki są na bieżąco omawiane. Pod koniec podsumowuję wady i zalety poszczególnych rozwiązań i rozważam w jakich scenariuszach najlepiej się one sprawują.

1 Wstęp

1.1 Dlaczego Python?

Na przestrzeni ostatnich 20 lat język Python, stworzony przez Guido van Rossum, zanotował intensywny wzrost popularności. Pokazują to liczne zestawienia, w tym zestawienie najczęściej wykorzystywanych języków programowanie na GitHub'ie[14], w którym Python w roku 2022 zajął 2 miejsce, czy też zestawienie TIOBE Index[34], uznające ten język za obecnie najbardziej rozpowszechniony pośród doświadczonych programistów (Maj 2023).



Rysunek 1: Ilość wyników zwróconych przez wyszukiwarkę Google Scholar dla zapytania 'python language' z podziałem na rok opublikowania zasobu w przestrzeni publicznej.

Niestety, interpretowany kod, napisany w Pythonie, pomimo licznych zalet, posiada również dotkliwą wadę - pod względem wydajności znacząco odstaje od kompilowanych języków programowania (C[29], C++[30], Rust[31]). Jednak, podejmując odpowiednie wysiłki, możliwe jest aby programy, których kluczowa logika została napisana w języku Python, zbliżały się

[1] ang. crate, określenie na bibliotekę-pakiet w ekosystemie języka Rust.

wydajnością do odpowiedników przetłumaczonych na kod maszynowy. Taki stan rzeczy czyni z języka Python bardzo wygodny język do prototypownia w procesie wytwarzania nowych rozwiązań programistycznych.

1.2 Cel pracy

Praca ta ma na celu eksplorację wybranych metod optymalizacji czasu wykonania oprogramowania CSSFinder oraz weryfikację uzyskanych zmian wydajności programu. W dalszej jej części zaprezentuję wyniki testów wydajności i przeanalizuję specyfikę poszczególnych metod maksymalizacji wydajności. Podejście do redukcji czasu wykonywania programu zaprezentowane w tej pracy można zastosować dla większości oprogramowania, napisanego w języku Python, które koncentruje się na wykonywaniu dużych ilości obliczeń macierzowych.

Do przeprowadzenia takich analiz konieczne było wielokrotne re-implementowanie części obliczeniowej programu. Funkcjonalny kod opisywany w tej pracy dostępny jest w repozytoriach Gita[26] w serwisie GitHub [38][39][41]. W skutek prac projektowych utworzyłem również grupę publicznych pakietów, które można pobrać z serwisu PyPI:

- cssfinder[37]
- cssfinder_backend_numpy[40]
- cssfinder_backend_rust[42]

Zainstalowanie ich jest możliwe przy pomocy menadżera pakietów języka Python, np. pip[21]. Pakiety są kompatybilne z implementacją CPython w wersjach 3.8 - 3.10 i były testowane na systemach Windows (10), Linux (Ubuntu 22.04) oraz macOS (12).

1.3 Program CSSFinder

Program CSSFinder, którego autorem jest dr hab. Marcin Wieśniak, prof. UG, z wydziału Matematyki, Fizyki i Informatyki Uniwersytetu Gdańskiego. Kod implementuje wyspecjalizowany wariant algorytmu zaproponowanego przez E. Gilberta[3] który służy do znajdowania odległości pomiędzy punktem, a zbiorem wypukłym. Oprogramowanie jest przeznaczone do detekcji splątania kwantowego[10][12][11] poprzez analizę macierzy gęstości opisujących układy kubitów[2]. Algorytm ten wielokrotnie, z sukcesem, był wykorzystany do analizy problemów z dziedziny fizyki kwantowej[10] przy okazji również pokonując rozwiązania bazujące na uczeniu maszynowym[15].

Oryginalna implementacja wykorzystuje język Python oraz bibliotekę NumPy. Posiada ona 4 różne tryby pracy, dedykowane do rozwiązywania różnych problemów, oznaczane kolejno cyframi:

1. pełna separowalność stanu n ku*d*itów[3],

[2]Potencjalnie również ku*d*itów, natomiast tego typu dane nie będą analizowane w tej pracy.

[3]ang. full separability of an n qu*d*it state

2. separowalność stanu dwudzielnego[4]

3. rzeczywiste 3-częściowe uwikłanie stanu trzech ku*d*itów [5]

4. rzeczywiste 4-częściowe uwikłanie stanu trzech ku*d*itów [6]

Dodatkowo pozwala na podanie macierzy symetrii oraz macierzy projekcji układu.

1.4 Modularyzacja

Podczas procesu optymalizacji planowałem wypróbować liczne rozwiązania, które wymagały zasadniczych zmian w algorytmie. Jednocześnie część programu odpowiadająca za interakcję z użytkownikiem i ładowanie zasobów miała pozostawać taka sama. Zdecydowałem więc że tworzony przeze mnie kod musi być modularny, aby uniknąć duplikacji wspólnych elementów. Tak też program został podzielony na dwie części: główną (core), z interfejsem użytkowników i narzędziami pomocniczymi oraz część implementującą algorytm (backend). Korpus jest w całości napisany w języku Python i wykorzystuje wbudowany w ten język mechanizm importowania bibliotek w celu wykrywania i ładowania implementacji algorytmu. Dane macierzowe w obrębie korpusu przechowywane są jako obiekty ndarray z biblioteki NumPy, ze względu na uniwersalność w świecie bibliotek do obliczeń tensorowych. Pozwala to na proste podmiany implementacji o dowolnie różnym pochodzeniu, w tym implementacje w językach kompilowanych. Uprościło to znacznie proces weryfikacji zmian w zachowaniu programu i przyspieszyło proces tworzenia kolejnych implementacji, jako że kod interfejsu programistycznego jest mniej pracochłonny niż kod pozwalający na interakcję z użytkownikiem.

[4] ang. separability of a bipartite state
[5] ang. genuine 3-partite entablement of a 3-quDit state
[6] ang. genuine 4-partite entablement of a 3-quDit state

2 Metody

2.1 Kompilacja AOT

Kompilacja AOT (Ahead Of Time) to proces tłumaczenia jednej reprezentacji programu (na przykład w języku programowania wysokiego poziomu) na inną (na przykład kod maszynowy) przed rozpoczęciem pracy kompilowanego programu.

Obecnie najpowszechniej używana implementacja języka Python, CPython, posiada możliwość korzystania z bibliotek współdzielonych (.so - Linux, .dll/.pyd - Windows) które powstały w skutek kompilacji kodu wysokiego poziomu. Dostęp do funkcji zawartych w takich bibliotekach można uzyskać na kilka sposobów:

1. Przy pomocy API modułu ctypes[24]. Pozwala ono opisać interfejs funkcji obcej (tj. takiej która została napisana w języku niższego poziomu i skompilowana do kodu maszynowego) i wywołać tak opisaną funkcję.

2. Poprzez zawarcie w bibliotece odpowiednio nazwanych symboli, automatycznie rozpoznawanych przez interpreter języka Python. Takie biblioteki określa się mianem modułów rozszerzeń [25]. W tym przypadku warto dodać, że pomimo, że oficjalna dokumentacja wspomina tylko o językach C i C++, natomiast powstały biblioteki które pozwalają wykorzystać w łatwy sposób wiele innych języków programowania, takich jak Rust przy pomocy Py03[22] lub GO z użyciem biblioteki gopy[17].

3. Wykorzystując bibliotekę Cython[18][4]. Oferuje ona dedykowany język, o tej samej nazwie, który jest nadzbiorem języka Python, który rozszerza jego składnię o możliwość statycznego typowania. Biblioteka zawiera transpilator, zdolny przetłumaczyć dedykowany język na C/C++, a następnie, wykorzystując osobno zainstalowany kompilator, skompilować do kodu maszynowego.

4. Kompilując kod pythona z użyciem biblioteki mypyc[35]. Ta, podobnie do biblioteki Cython, również zawiera transpilator, natomiast zamiast korzystać z dedykowanego języka, opiera się on na dodanych w Pythonie 3.5[5] (PEP 484[28] i PEP 483[27]), adnotacjach typów. Jest on rozwijany obok projektu mypy - pakietu do statycznej analizy typów dla języka Python, również opartej na adnotacjach typów[33].

Ponieważ w każdym z wymienionych przypadków, kod niższego poziomu jest kompilowany przed dostarczeniem do użytkownika, pozwala to na wykorzystanie zaawansowanych możliwości automatycznej optymalizacji dostarczanych przez współczesne kompilatory, na przykład LLVM, które jest sercem implementacji clang (język C++) oraz rustc (język Rust). Wiele bibliotek korzysta z mieszanek wymienionych powyżej metod, w tym cieszące się dużą popularnością NumPy, CuPy, Tensorflow, czy PyTorch. Dwie ostatnie biblioteki koncentrują się w głównej mierze na uczeniu maszynowym i głównie pod tym kontem są optymalizowane. Ich interfejsy są bardzo zbliżone do NumPy i CuPy, ale brakuje w nich niektórych narzędzi, które nie

znajdują zastosowania w dziedzinie sztucznej inteligencji. W dalszej części pracy intensywnie wykorzystywana będzie biblioteka NumPy. Niestety, ze względu na ograniczenia czasowe oraz wstępne przewidywania dotyczące wydajności[7] biblioteka CuPy nie wzięta pod uwagę.

Ponadto w zestawieniu pojawi się implementacja napisana w języku Rust, wykonująca operacje macierzowe w oparciu o bibliotekę Ndarray[23]. Komunikacja pomiędzy interpreterem Pythona, a biblioteką oparta została o rozwiązanie opisane w punkcie drugim, dzięki wspomnianej tam bibliotece PyO3[22]. Jest to wymiernie reprezentatywny przykład tego jaką wydajność można uzyskać tworząc kod w typowym niskopoziomowym języku programowania, posiadającym relatywnie niskopoziomową kontrolę nad pamięcią. Dodatkowo w zestawieniach pojawi się wariant tej implementacji który dodatkowo będzie wykorzystywał bibliotekę OpenBLAS[43][23] do operacji mnożenia macierzowego.

2.2 Kompilacja JIT

Kompilacja JIT to proces tłumaczenia jednej reprezentacji programu (na przykład w języku programowania wysokiego poziomu) na inną (na przykład kod maszynowy) po rozpoczęciu pracy programu. Zazwyczaj wymaga to aby program rozpoczynał pracę w trybie interpretowanym, a następne kompilował sam siebie i przechodził w tryb wykonywania skompilowanego kodu.

W momencie pisania tej pracy istnieją dwa szeroko dostępne i aktywnie utrzymywane narzędzia oferujące kompilację JIT dla języka Python.

Pierwszym z nich jest pełna alternatywna implementacja języka Python - PyPy[36]. Wykonywana przez nią kompilacja JIT działa on na podobnej zasadzie do uprzednio wymienionych - śledzi cały kod który wykonuje i automatycznie decyduje które fragmenty skompilować do kodu maszynowego[2]. Niestety, posiada ona zasadniczą wadę - jej interfejs binarny[8] oraz programistyczny[9] różni się od CPythona, a większość pakietów które normalnie wykorzystują moduły rozszerzeń nie oferuje pre-kompilowanych pakietów dla PyPy. Powoduje to że instalacje takich pakietów są bardzo czasochłonne i obecności kompilatora na urządzeniu docelowym. Dodatkowo, pre-kompilowany kod nie czerpie żadnych korzyści z kompilatora JIT zawartego w PyPy. Problemy te powodują, że PyPy nadaje się głównie do wykonywania aplikacji napisanych w czystym języku Python.

Drugim narzędziem jest biblioteka Numba[6][8]. Ona, w przeciwieństwie do PyPy, wymaga aby fragmenty kodu, które mają być skompilowane, miały postać funkcji oznaczonych dedykowanymi dekoratorami[10]. Została ona również zaprojektowana aby dobrze współgrać z biblioteką NumPy. Jej zastosowanie z założenia ma generować wzrost wydajności nawet w

[7] CuPy jest odpowiednikiem NumPy który wykorzystuje do obliczeń GPU. Z tego względu radzi sobie wyśmienicie z operacjami na dużych macierzach, natomiast najprawdopodobniej macierze tutaj rozważane są zbyt małe aby uzyskać wzrost wydajności[7]. Jednocześnie pomimo podobieństwa do NumPy, biblioteka ta różni się i posiada problematyczne zależności (CUDA) co czyni adaptację kodu czasochłonną.

[8] ang. ABI - Application Binary Interface

[9] ang. API - Application Programming Interface.

[10] Obecnie dostępny jest też dekorator pozwalający na kompilację klas, niestety jest on niestabilny i nie radzi sobie w wielu sytuacjach.

sytuacjach gdy kod programu bardzo mocno eksploatuje możliwości biblioteki NumPy.

Z uprzednio wymienionych względów dotyczących preferowanych zastosowań powyższych rozwiązań w dalszej części będę próbował wykorzystać bibliotekę Numba, natomiast pominę możliwość skorzystania z PyPy.

2.3 Dane testowe

Podczas pomiarów konsekwentnie wykorzystywałem ten sam zestaw macierzy gęstości, aby móc wygodnie porównywać wyniki wydajności poszczególnych implementacji. W dalszej części pracy będę wielokrotnie odnosił się do tych macierzy posługując się symbolem ρ z liczbą w indeksie dolnym. Liczba ta będzie wskazywać na konkretną z wymienionych poniżej macierzy.

$$\rho_0 = \begin{bmatrix}
0.25 & 0 & 0.25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0 & 0 & 0 & -0.25 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0.25 & 0 & 0.25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0 & 0 & 0 & -0.25 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0.25 & 0 & 0.25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0 & 0 & 0 & -0.25 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-0.25 & 0 & -0.25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.25 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25
\end{bmatrix}$$

Rysunek 2: Macierz ρ_0.

Pierwsza wymieniana macierz opisuje układ 5 kubitów i posiada wymiary 32×32. Pomimo że nie zawiera ona wartości, podczas analizy zawsze będzie reprezentowana przez macierze zawierające liczby zespolone, ponieważ szczególnie kosztowne obliczeniowo części algorytmu wymagają aby części urojone były obecne, co znaczy że usuwanie ich w wybranych miejscach nie niesie wymiernych zysków wydajnościowych.

Następnie w zbiorze macierzy wykorzystywanych jako dane wejściowe znajduje się pięć macierzy reprezentujących układy od 2 do 6 kubitów, które przyjmują rozmiary od 4×4 do 64×64. Są one wypełnione zerami poza pierwszym i ostatnim elementem w pierwszej i ostatniej kolumnie - te przyjmują wartość 0.5.

$$\rho_n = \begin{bmatrix}
0.5 & 0 & \dots & 0 & 0.5 \\
0 & 0 & \dots & 0 & 0 \\
\vdots & \vdots & & \vdots & \vdots \\
0 & 0 & \dots & 0 & 0 \\
0.5 & 0 & \dots & 0 & 0.5
\end{bmatrix}_{(2^n \times 2^n)}$$

Rysunek 3: Ogólna postać macierzy $\rho_2 - \rho_6$.

W tekście macierze te będą oznaczane jako ρ_2 do ρ_6, w zależności od reprezentowanej liczby kubitów[11]. Macierze te stanowią wygodny zestaw danych do weryfikacji ogólnej charakterystyki zachowania alternatywnych implementacji algorytmu, pomimo, że wyniki przy ich pomocy uzyskiwane tak bardzo odbiegają od tych uzyskiwanych przy pomocy ρ_0.

[11]Tak więc macierz ρ_2 ma wymiary 4×4 i reprezentuje 2 kubity, macierz ρ_3 ma wymiary 8×8 i reprezentuje 3 kubity, macierz ρ_4 ma wymiary 16×16 i reprezentuje 4 kubity, itd. aż do ρ_6, 64×64 .

2.4 Środowisko testowe

Podczas pomiarów wydajności wykorzystywałem każdorazowo to samo środowisko testowe. Do chłodzenia CPU wykorzystywane było chłodzenie wodne typu AIO, temperatura w pokoju oscylowała w okolicy 25°C, procesor podczas testów wydajności nie doświadczał temperatur powyżej 80°C.

OS	Ubuntu 22.04.2 LTS 64-bit
Kernel	5.19.0-42-generic
Python	3.10.6 64-bit
NumPy	1.23.5
Numba	0.56.4
Cython	3.0.0b1
GCC	11.3.0 64-bit
Rust	1.68.2 64-bit
CPU	AMD Ryzen 9 7950X
RAM	64GB DDR5 5600MHz CL40
DRIVE	512GB SSD GOODRAM CX400 (SATA)

Tablica 1: Konfiguracja środowiska testowego.

2.5 Profilowanie

Podczas prac nad optymalizacją czasu pracy programu kluczowym było stałe zbieranie informacji na temat tego które fragmenty kodu pochłaniają najwięcej czasu. Standardowo proces zbierania takich danych określa się mianem profilowania i technologie po które sięgałem podczas re-implementacji algorytmu posiadają gotowe narzędzia pozwalające na skuteczne pozyskiwanie takich danych oraz ich wizualizację.

Dla kodu w języku Python, implementacja CPython tego języka posiada w bibliotece standardowej dwa dedykowane moduły oferujące funkcjonalność profilowania: 'profile' i 'cProfile'. Pierwszy jest zaimplementowany w języku Python, drugi w C. Ponieważ drugi z nich posiada mniejszy dodatkowy narzut na procesor, zdecydowałem żeby to na nim oprzeć moje analizy. W celu wizualizacji uzyskanych wyników posłużyłem się otwartoźródłowym programem snakeviz[19].

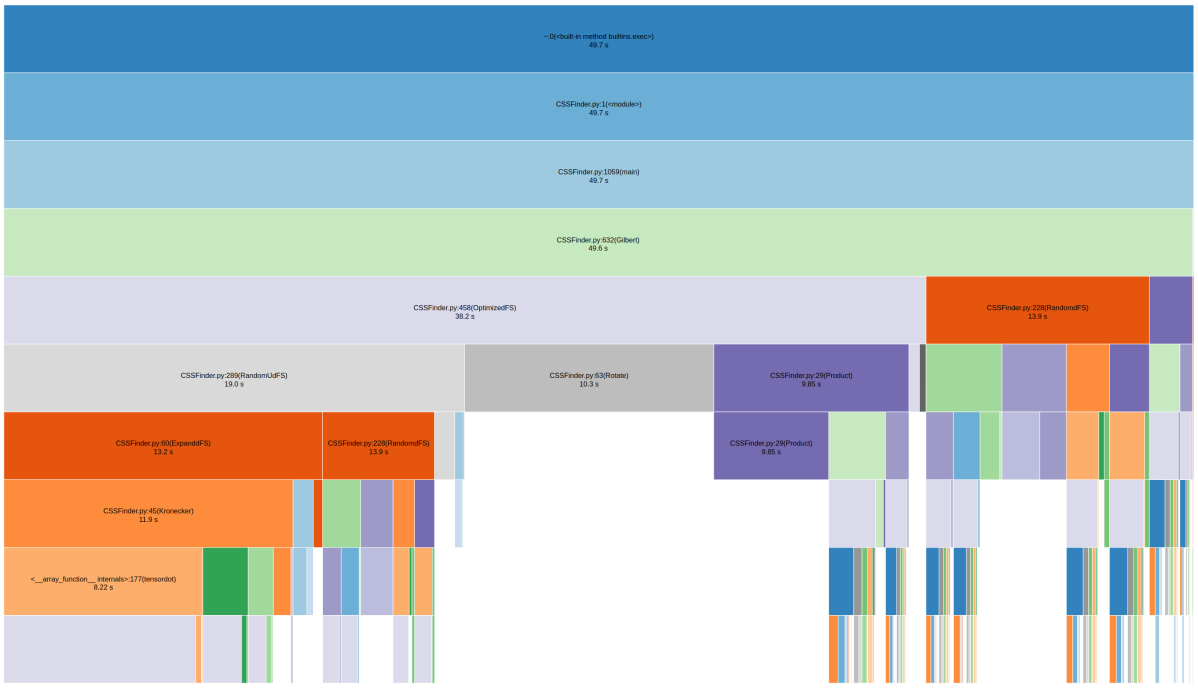
Do zbierania informacji na temat charakterystyki pracy kodu napisanego w języku Rust wykorzystałem narzędzie perf pochodzące z pakiety linux-tools-5.19.0-42-generic pobranego przy pomocy menadżera pakietów apt-get. Do wizualizacji uzyskanych wyników wykorzystałem jedno z otwartoźródłowych narzędzi funkcjonujące pod nazwą hotspot[16].

3 Wyniki

3.1 Wstępne profilowanie

Prace nad optymalizacją kodu rozpocząłem od wstępnego profilowania pracy programu w trybie 1 (ang. full separability of an n-quDit state) przekazując do obliczeń układ 5 kubitów opisany macierzą ρ_0 (Patrz rysunek 2).

Program wykonywał proces analizy stanu aż do uzyskania 1000 korekcji. Przekazany limit liczby iteracji wynosił 2.000.000 i nie został osiągnięty. Podczas pomiarów, program wykorzystywał domyślny globalny generator liczb losowych biblioteki NumPy (PCG64[20]) z ziarnem ustawionym na wartość 0.



Rysunek 4: Diagram podsumowujący pracę programu wygenerowany przez program snakeviz.

Rysunek 4 przedstawia diagram, typu Icicle, obrazujący udział czasu, pochłoniętego przez wykonywanie poszczególnych funkcji, w całkowitym czasie pracy programu. Pierwszy blok od góry to pierwsze wywołanie pochodzące z interpretera. Następnie bloki, których opisy zaczynają się od 'CSSFinder.py' to wywołania w kodzie programu. Najniższe bloki to wywołania do funkcji bibliotek, głównie NumPy. Snakeviz automatycznie podejmuje decyzję o nie adnotowaniu bloku gdy opis nie ma szansy zmieścić się w obrębie bloku. Aby usunąć z diagramu zbędny szum informacyjny, funkcje których wykonywanie zajęło mniej niż 1% czasu programu były pomijane.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	1.431e-05	1.431e-05	49.72	49.72	CSSFinder.py:1(<module>)
1	7.526e-05	7.526e-05	49.68	49.68	CSSFinder.py:1059(main)
1	0.3098	0.3098	49.63	49.63	CSSFinder.py:632(Gilbert)
1028	0.5381	0.0005234	38.2	0.03716	CSSFinder.py:458(OptimizedFS)
411200	0.8332	2.026e-06	19.03	4.627e-05	CSSFinder.py:289(RandomUdFS)
595516	0.67	1.125e-06	13.88	2.331e-05	CSSFinder.py:228(RandomdFS)
411200	0.384	9.338e-07	13.17	3.203e-05	CSSFinder.py:60(ExpanddFS)
822400	0.7256	8.823e-07	11.94	1.452e-05	CSSFinder.py:45(Kronecker)
849257	10.3	1.213e-05	10.3	1.213e-05	CSSFinder.py:63(Rotate)
1068026	6.535	6.118e-06	9.85	9.223e-06	CSSFinder.py:29(Product)
1332780	2.17	1.628e-06	4.502	3.378e-06	CSSFinder.py:21(Normalize)
1332780	2.247	1.686e-06	3.802	2.853e-06	CSSFinder.py:33(Generate)
737264	0.4225	5.73e-07	2.548	3.456e-06	CSSFinder.py:18(Outer)
595516	0.4642	7.794e-07	2.361	3.964e-06	CSSFinder.py:26(Project)
1233601	0.8998	7.294e-07	1.165	9.447e-07	CSSFinder.py:39(IdMatrix)
1	3.046e-06	3.046e-06	0.05277	0.05277	CSSFinder.py:96(readmtx)
1	1.752e-06	1.752e-06	0.05277	0.05277	CSSFinder.py:552(Initrho0)
1	4.597e-06	4.597e-06	0.002477	0.002477	CSSFinder.py:1049(DisplayLogo)
1	5.189e-06	5.189e-06	0.0004394	0.0004394	CSSFinder.py:954(DetectDim0)
1	1.628e-05	1.628e-05	2.526e-05	2.526e-05	CSSFinder.py:556(Initrho1)
1	1.903e-06	1.903e-06	5.671e-06	5.671e-06	CSSFinder.py:599(DefineSym)
40	3.038e-06	7.595e-08	3.038e-06	7.595e-08	CSSFinder.py:192(writemtx)
1	1.102e-06	1.102e-06	2.846e-06	2.846e-06	CSSFinder.py:624(DefineProj)
2	2.3e-07	1.15e-07	2.3e-07	1.15e-07	CSSFinder.py:845(makeshortreport)

Tablica 2: Dane dotyczące pracy oryginalnej implementacji programu CSSFinder uzyskane przy pomocy programy cProfile. Tabela posiada oryginalne nazwy kolumn, nadane przez program snakeviz. Znaczenia kolumn, kolejno od lewej: `ncalls` - ilość wywołań funkcji. `tottime` - całkowity czas spędzony w ciele funkcji bez czasu spędzonego w wywołaniach do podfunkcji. `percall` - `totime` dzielone przez `ncalls`. `cumtime` - całkowity czas spędzony w wewnątrz funkcji i w wywołaniach podfunkcji. `percall` - `cumtime` dzielone przez `ncalls`. `filename:lineno(function)` - Plik, linia i nazwa funkcji.

Z uzyskanych danych wynika że znakomitą większość (77%[12]) czasu pracy programu zajmuje funkcja `OptimizedFS()`. W jej wnętrzu 38% czasu pochłania proces generowania losowych macierzy unitarnych, który w dużej mierze wykorzystuje mnożenia tensorowe (26%). Poza funkcją `OptimizedFS()`, znaczący wpływ na czas wykonywania ma też funkcja ‘rotate()‘, która pochłania około 21% czasu działania programu. Kolejne 20% czasu zajmuje funkcja `product()`, obliczająca odległość Hilberta-Schmidta pomiędzy dwoma stanami. Pozostałe wywołania mają stosunkowo marginalny wpływ na czas pracy i ich analiza na tym etapie nie niesie za sobą znaczących korzyści.

Takie wyniki wskazują jednoznacznie że kluczowa dla czasu pracy programu jest tu maksymalizacja wydajności operacji macierzowych. Ten pozornie oczywisty wniosek wyznacza prosty kurs dalszych prac nad programem. W uzyskanych danych nie widać problemów z operacjami I/O[13], a dla wielu aplikacji potrafią one stanowić poważne ograniczenie wydajności. W tym wypadku nie jest konieczne sięganie po rozwiązania takie jak asyncio czy wielowątkowość, które stosuje się w razie problemów z operacjami I/O.

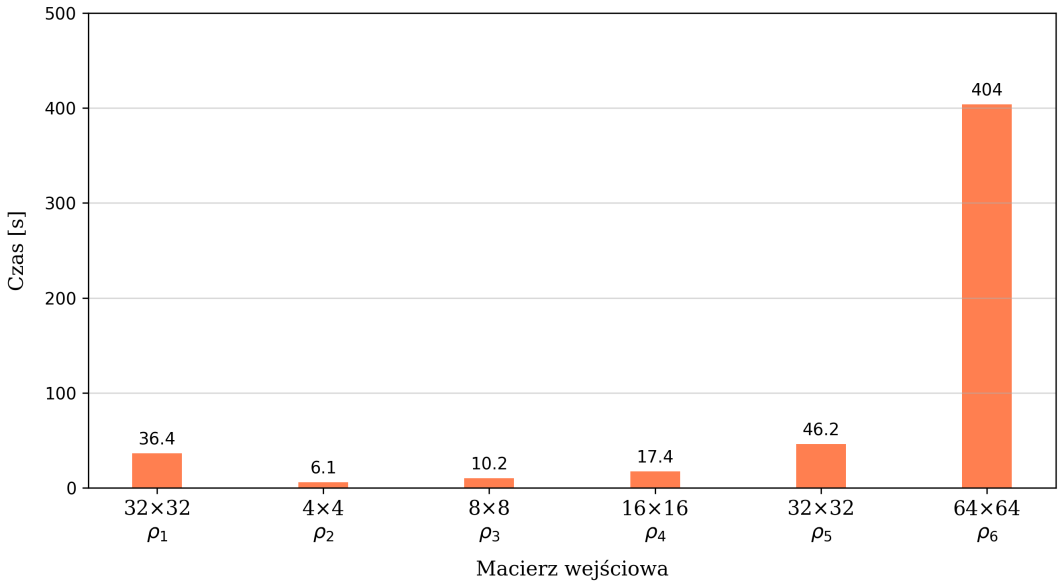
[12]Wartość 77% jak i wartości procentowe dalszej części tego akapitu zostały zaokrąglone do jedności, ze względu na małe znaczenie rzeczowe części ułamkowych.

[13]I/O - operacje wejścia wyjścia, w tym wypadku odczyt z i pisanie do plików.

3.2 Wstępne pomiary wydajności

Aby uzyskać dobrą bazę porównawczą, wykonałem serię pomiarów czasu pracy programu na macierzach $\rho_0 1$, ρ_2 - ρ_6, przedstawionych na rysunkach 2 i 3.

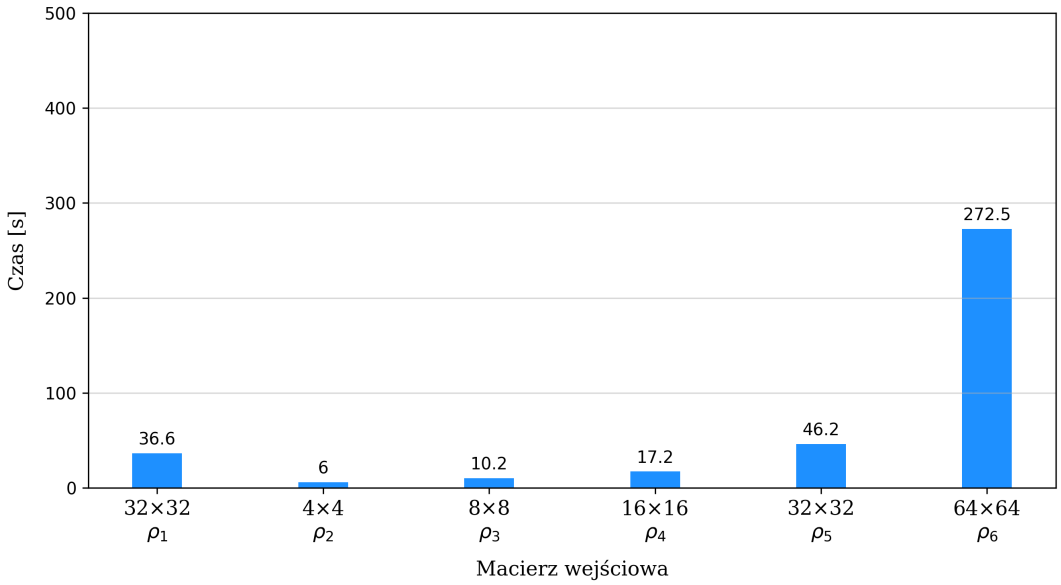
Dane przekazywałem kolejno do programu z poleceniem działania w trybie 1 (full separability of an n-quDit state) do osiągnięcia 1000 korekcji lub do 2.000.000 iteracji algorytmu, w zależności od tego co nastąpi szybciej. Dla wszystkich macierzy algorytm uzyskał 1000 korekcji i w żadnym przypadku nie osiągnął maksymalnej liczby iteracji. Dla każdej macierzy pomiar był powtarzany pięciokrotnie, a wyniki z pomiarów zostały uśrednione. Podczas obliczeń ziarno globalnego generatora liczb losowych biblioteki NumPy było ustawione na 0. Pomiary czasu pracy dotyczyły wyłącznie samego algorytmu[14].



Rysunek 5: Wyniki wstępnych testów wydajności oryginalnego kodu dla macierzy ρ_2 - ρ_6.

Podczas testów zaobserwowałem interesujące zjawisko dotyczące wydajności dla macierzy 64×64. W przypadku takich rozmiarów danych biblioteka NumPy automatycznie decyduje o wykorzystaniu wielowątkowej implementacji mnożenia macierzowego. Niestety, daje to efekt odwrotny do zamierzonego - obliczenia zamiast przyspieszać zwalniają. Na rysunku 5 zostały przedstawione czasy obliczeń dla macierzy ρ_2 - ρ_6 z domyślnym zachowaniem biblioteki.

[14] tj. funkcji 'Gilbert()', nie biorą więc pod uwagę czasu pochłoniętego przez importowanie modułów, ładowanie danych itp. natomiast operacje pisania do plików które były wykonywane w obrębie tej funkcji są wliczane w czas pracy.



Rysunek 6: Wyniki wstępnych testów wydajności oryginalnego kodu z zablokowaną liczbą wątków obliczeniowych dla macierzy ρ_2 - ρ_6.

Jeśli przy pomocy zmiennych środowiskowych ustawimy ilość wątków wykorzystywanych do obliczeń na 1 uzyskujemy znaczące skrócenie czasu obliczeń dla macierzy 64×64. Wyniki testów w takich warunkach zostały przedstawione na rysunku 6. Dla macierzy w mniejszych rozmiarach nie odnotowałem różnicy w wydajności pomiędzy konfiguracją domyślną, a manualnie dostosowywaną. Warto dodać że ilość iteracji wykonywanych przez program nie zmienia się, różnica wynika wyłącznie z czasu trwania operacji arytmetycznych. Taki stan rzeczy najprawdopodobniej jest wynikiem dodatkowego obciążenia ze strony komunikacji i/lub synchronizacji między wątkami.

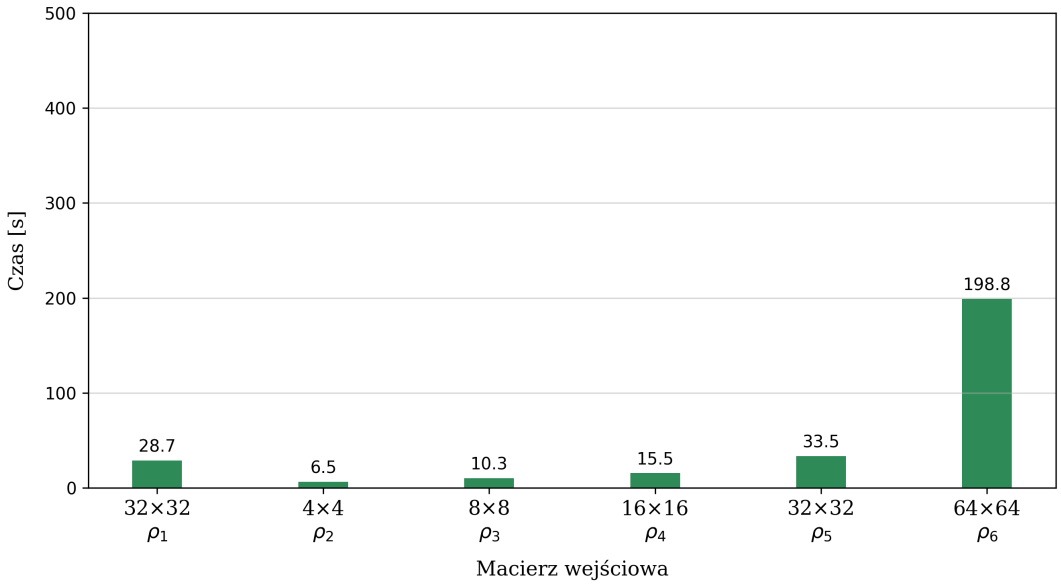
3.3 Re-implementacje

3.3.1 Python i NumPy

Pierwsza wykonana przeze mnie re-implementacja algorytmu, została napisana w języku Python, a do realizowania obliczeń na macierzach liczb zespolonych wykorzystywała bibliotekę NumPy. Podczas przepisywania podjąłem jednak dodatkowe wysiłki aby zastępować kod Pythona wywołaniami do funkcji zawartych w bibliotece NumPy. Ponieważ kluczowe dla wydajności fragmenty kodu tego pakietu są zaimplementowane w języku niższego poziomu, a następnie skompilowane kompilatorem optymalizującym, oferują znacznie wyższą wydajność niż analogiczny kod napisany w języku Python. Proces ten pozwolił mi również zapoznać się lepiej z charakterystyką programu i udoskonalić interfejs służący do komunikacji pomiędzy częścią główną, a samą implementacją (backend'em). Sam algorytm pozostał taki sam, natomiast konstrukcja kodu zmieniła się diametralnie, więc dogłębne analizy różnic byłyby nieczytelne, dlatego nie zostaną tutaj zawarte. W dalszej części pojawią się wyniki pomiarów wydajności.

Pomiary czasu pracy były wykonywane przy użyciu macierzy ρ_2 - ρ_6. Dane przekazywałem

kolejno do programu z poleceniem działania w trybie FSnQd[15] do osiągnięcia co najmniej 1000 korekcji lub do 2.000.000 iteracji algorytmu, w zależności od tego co nastąpi szybciej. Dla wszystkich macierzy algorytm uzyskał co najmniej 1000 korekcji i w żadnym przypadku nie osiągnął maksymalnej liczby iteracji. Dla każdej macierzy pomiar był powtarzany pięciokrotnie a wyniki zostały uśrednione. Podczas obliczeń ziarno domyślnego globalnego generatora liczb losowych biblioteki NumPy było ustawione na 0. Program działał z zablokowaną liczbą wątków obliczeniowych. Pomiary czasu pracy dotyczyły przede wszystkim samego algorytmu[16].



Rysunek 7: Wyniki testów wydajności alternatywnej implementacji Python z użyciem biblioteki NumPy dla macierzy ρ_2 - ρ_6.

Uzyskane wyniki zostały przedstawione na rysunku 7. Wykres został utworzony przy pomocy biblioteki matplotlib[1].

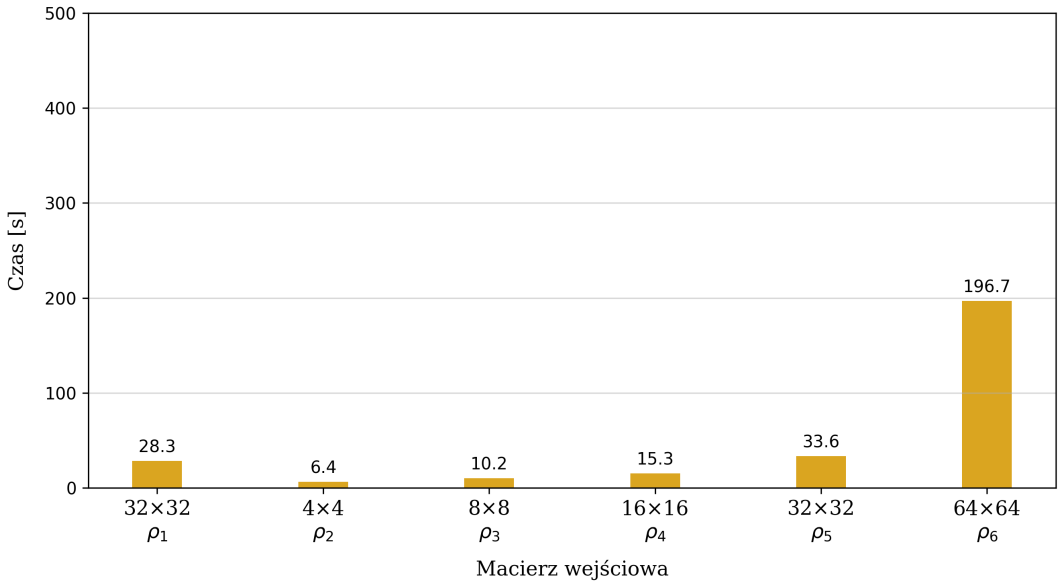
3.3.2 Python i NumPy z AOT

Następnym wykonanym przeze mnie krokiem było skompilowanie mojej implementacji korzystającej z NumPy do kodu maszynowego przy pomocy biblioteki Cython. Kod przeznaczony do takiej kompilacji nie musi być adnotowany dedykowanymi informacjami o typach. Zostanie on w tedy przetłumaczony na odpowiednie operacje w języku C/C++, a potem skompilowany do kodu maszynowego. Brak adnotacji powoduje niestety, że program zachowuje swoją dynamiczną naturę, charakterystyczną dla języka Python. Kompilacja pozwala jednak usunąć dodatkowy narzut na procesor ze strony interpretera. W takim scenariuszu spodziewać należy się, że zyski z kompilacji będą niewielkie, ale mogą wystąpić.

Pomiary czasu pracy były wykonywane w taki sam sposób jak dla implementacji bez AOT.

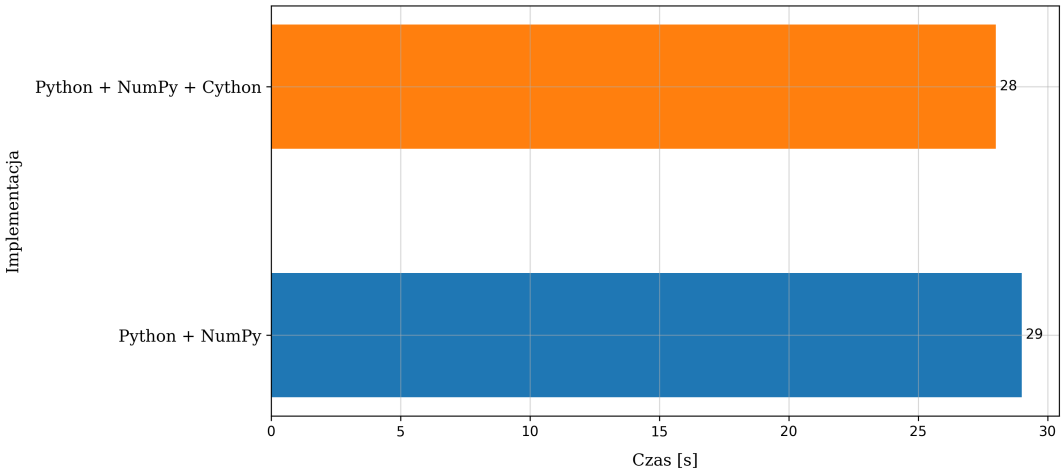
[15]Tryb FSnQd jest odpowiednikiem trybu 1 (full separability of an n-quDit state) z oryginalnego kodu.

[16]Pomiary nie biorą więc pod uwagę czasu pochłoniętego przez importowanie modułów itp., natomiast operacje wczytywania danych i pisania do plików są wliczane w czas pracy, ponieważ wbudowany w program mechanizm pomiaru czasu pracy rozpoczyna pomiar zanim dane zostaną załadowane.



Rysunek 8: Wyniki testów wydajności implementacji Python z użyciem biblioteki NumPy oraz pakietu Cython do kompilacji AOT dla macierzy ρ_2 - ρ_6.

Na rysunku 8 przedstawione zostały wyniki pomiarów czasu pracy skompilowanej wersji w języku Python bazującej na bibliotece NumPy wykorzystujące macierze ρ_2 - ρ_6. Dodatkowa kompilacja nie poskutkowała widocznym skróceniem czasu pracy programu, jedynie wynik dla macierzy 64×64 różni się nieznacznie. Może to być spowodowane usunięciem szczątkowego obciążenia ze strony interpretera, które nie jest mierzalne podczas krótszych testów z mniejszymi macierzami. Natomiast możliwe jest również że ta różnica wynika z korzystniejszych warunków losowo zapewnionych przez system operacyjny.



Rysunek 9: Wyniki testów wydajności implementacji Python i NumPy Z AOT w porównaniu do implementacji bez AOT dla macierzy ρ_0.

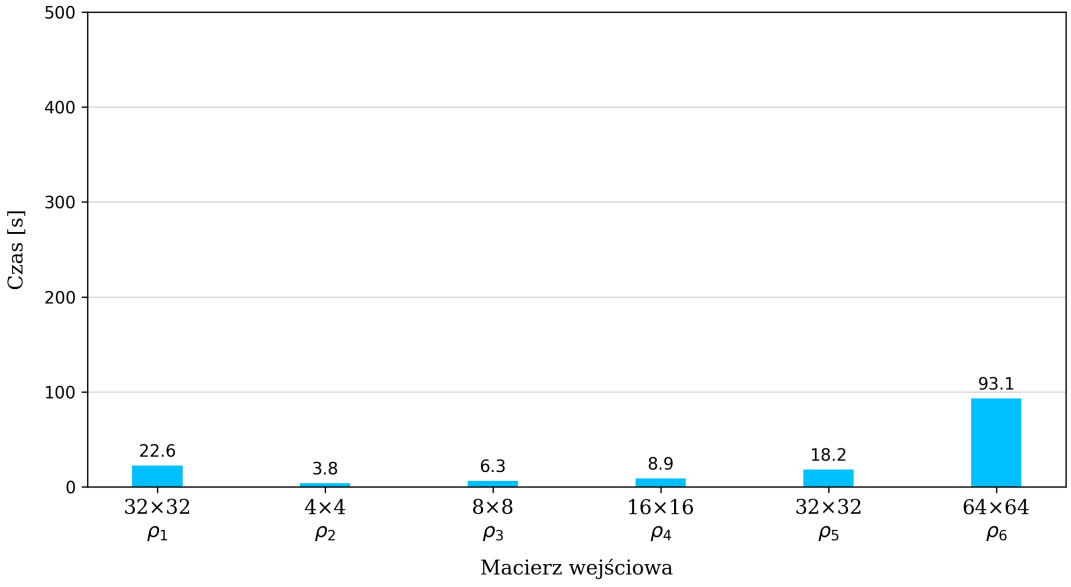
Wyniki dla testów wykonanych z użyciem macierzy ρ_0 widoczne na rysunku 9 prowadzą do analogicznych wniosków co w przypadku macierzy ρ_2 - ρ_6 - pre-kompilacja całości z wykorzystaniem biblioteki Cython nie poprawia znacząco wydajności rozważanego kodu.

3.3.3 Python i NumPy z JIT

Ostatnia stworzona przeze mnie re-implementacja w języku Python bazująca na bibliotece NumPy dodatkowo korzysta z kompilacji JIT. Pakiet Numba, który został wykorzystany do zrealizowania kompilacji JIT, posiada dwa tryby pracy. Pierwszy wykonuje kompilację na podstawie specjalnie dostarczonych przez programistę deklaracji typów dla funkcji podlegających kompilacji i jest wykonywany zaraz po rozpoczęciu pracy programu[17]. Drugi polega na śledzeniu typów wejściowych i wyjściowych funkcji i automatycznie kompiluje funkcję dla tych typów danych które są odpowiednio często używane używane[18].

Ponadto, Numba posiada dodatkowe parametry kompilacji, które można przekazać do funkcji `numba.jit`. Jednym z nich, posiadającym szczególnie duży wpływ na wydajność, flaga `nopython`. Tryb `nopython=True` oferuje znacznie większe możliwości optymalizacji i potencjalnie lepszą wydajność. Niestety nie wszystkie funkcje dostępne w bibliotece NumPy są akceptowane przez kompilator JIT pakietu Numba w trybie `nopython=True`. Do niekompatybilnych należy między innymi funkcja tensordot która implementuje mnożenie tensorowe. Wspomniana funkcja może zostać skompilowana tylko w trybie obiektowym (`nopython=False`), który po kompilacji zachowuje dynamiczną naturę Pythona. Niestety, brak możliwości skompilowania funkcji używającej tensordot powoduje również brak możliwości skompilowania funkcji wyżej w drzewie wywołań. W efekcie znacząca część implementacji używającej JIT musi używać trybu obiektowego.

Pomiary czasu pracy były wykonywane w taki sam sposób jak dla implementacji bez JIT.



Rysunek 10: Wyniki testów wydajności implementacji w języku Python z użyciem biblioteki NumPy i pakietu Numba do kompilacji JIT dla macierzy ρ_2 - ρ_6.

Na rysunku 10 przedstawione zostały wyniki uzyskane podczas pomiarów czasu pracy

[17] ang. eager (compilation) - niecierpliwa (kompilacja).
[18] ang. lazy (compilation) - leniwa (kompilacja).

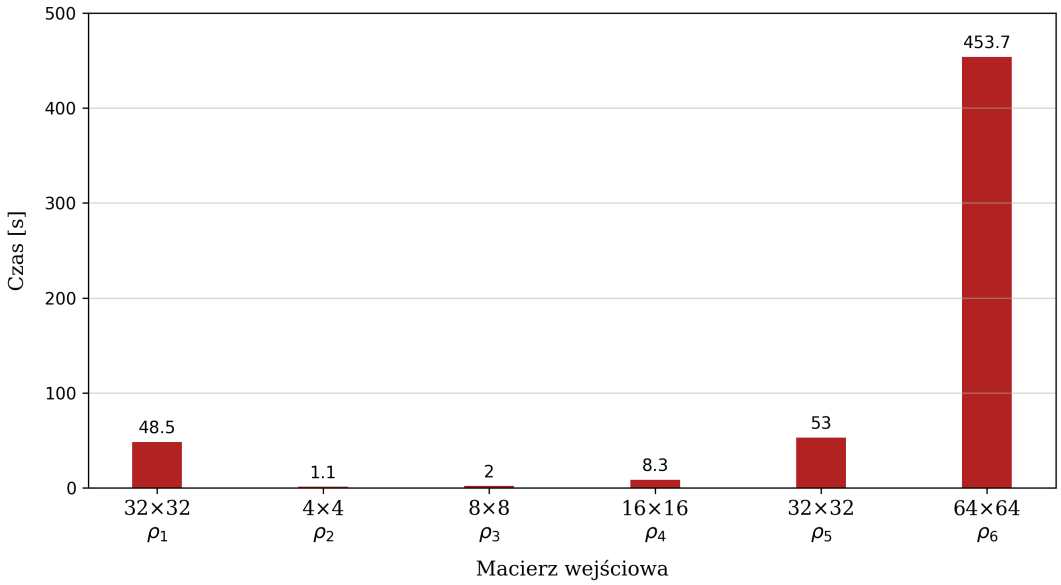
implementacji z JIT, w zależności od rozmiaru macierzy gęstości. Kod który wykorzystywał kompilację JIT oferował podczas testów dwu-czterokrotnie (w zależności od rozmiaru macierzy testowej) większą wydajność niż kod bez niej. Tak znaczącą poprawę implementacja zawdzięcza prawdopodobnie temu, że kompilator JIT może specjalizować kod dla dokładnie jednej platformy, korzystając z całego spektrum jej możliwości. Dotyczy to na przykład instrukcji SIMD, takich jak AVX512, które są dostępne w procesorze użytym do testów, ale wiele wciąż popularnych procesorów ich nie posiada. Wymusza to, przy kompilacji AOT, zastąpienie tych instrukcji innymi szerzej dostępnymi, aby zmaksymalizować przenośność kodu. Dodatkowo kompilator może brać pod uwagę inne charakterystyczne cechy konkretnych architektur. Te dodatkowe informacje i możliwość dodatkowej specjalizacji kodu czynią kompilację JIT bardzo potężnym narzędziem

3.3.4 Rust i Ndaray

Aby uczynić to porównanie jak najpełniejszym, podjąłem również wysiłek zaimplementowania części obliczeniowej programu w języku Rust. Język ten wybrałem z kilku względów. Posiada on pełną infrastrukturę pozwalającą w łatwy sposób kompilować programy i biblioteki wykorzystujące stworzone przez innych programistów rozwiązania. Daje mu to znaczącą przewagę nad językami takimi jak C/C++ które wymagają aby bardziej skomplikowane projekty samodzielnie skompletowały systemy budowania opierającego się na rozwiązaniach podmiotów trzecich, takich jak CMake, szczególnie jeśli chcą być dostępne na wielu platformach. Ponadto konkurencja nie posiada ujednoliconego standardu pozwalającego na łatwe uzyskanie odstępu do bibliotek otwartoźródłowych, Rust natomiast taki system posiada. W efekcie w łatwy sposób mogłem dołączyć gotowe rozwiązania pozwalające na prowadzenie obliczeń na macierzach liczb zespolonych. W efekcie cały proces wstępnej konfiguracji sprowadził się do około godziny, co stanowi wyśmienity wynik, biorąc pod uwagę, że przed podejściem do tego projektu nie miałem żadnej praktycznej styczności z tym językiem programowania. Dodatkową zaletą tego języka jest automatyczny system zarządzania pamięcią oparty na koncepcji własności (ang. ownership), który usuwa konieczność manualnego zarządzania pamięcią, jednocześnie bez konieczności wprowadzania mechanizmu liczenia referencji i dedykowanego automatycznego 'odśmiecacza' (ang. garbage collector) które to są częstym źródłem problemów z wydajnością i użyciem pamięci.

Pomiary czasu pracy implementacji w języku Rust były wykonywane przy użyciu macierzy ρ_2 - ρ_6. Dane przekazywałem kolejno do programu z poleceniem działania w trybie FSnQd do osiągnięcia co najmniej 1000 korekcji lub do 2.000.000 iteracji algorytmu, w zależności od tego co nastąpi szybciej. Dla wszystkich macierzy algorytm uzyskał co najmniej 1000 korekcji i w żadnym przypadku nie osiągnął maksymalnej liczby iteracji. Dla każdej macierzy pomiar był powtarzany pięciokrotnie a wyniki zostały uśrednione. Pomiary czasu pracy dotyczyły przede wszystkim samego algorytmu[19].

[19]Nie biorą więc pod uwagę czasu pochłoniętego przez importowanie modułów itp., natomiast operacje wczytywania danych i pisania do plików są wliczane w czas pracy.



Rysunek 11: Wyniki testów wydajności implementacji w języku Rust dla macierzy ρ_2 - ρ_6.

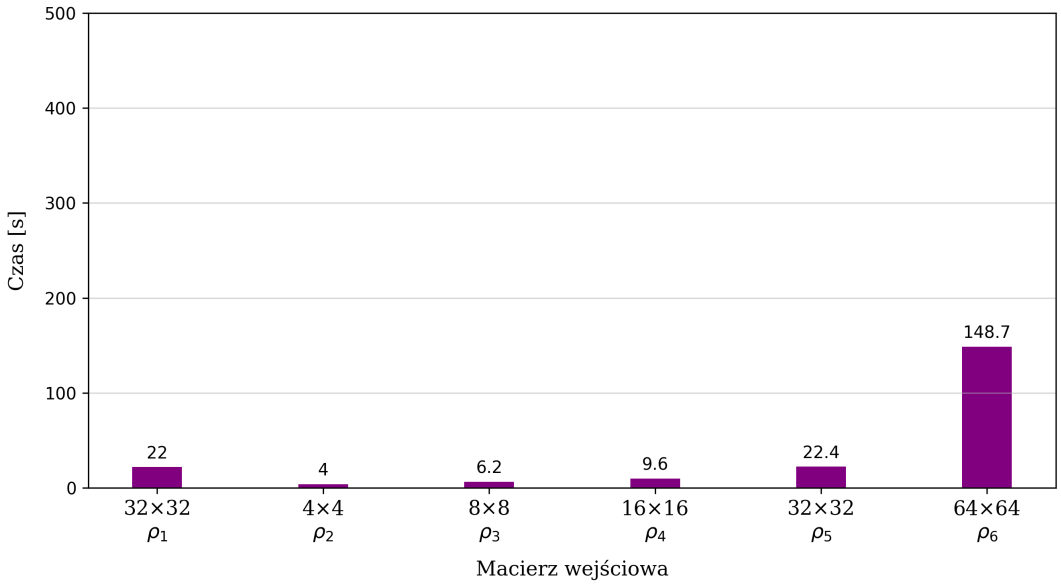
Na rysunku 11 zaprezentowane zostały wyniki pomiarów czasu pracy implementacja w języku Rust. Prezentuje ona znacząco lepsze wyniki dla małych macierzy oraz znacznie gorsze wyniki dla dużych macierzy. Jest to prawdopodobnie spowodowane tym, że sama implementacja mnożenia macierzowego nie jest wyspecjalizowana, aby wykorzystywać maksimum możliwości procesora na którym jest wykonywana, w przeciwieństwie do na przykład biblioteki NumPy, które wewnętrznie wykorzystuje bibliotekę OpenBLAS[13]. W efekcie nie czerpie ona korzyści z instrukcji SIMD, takich jak AVX512.

3.3.5 Rust i Ndaray z OpenBLAS

Biblioteka Ndarray, która jest sercem implementacji w języku Rust, posiada przełącznik funkcjonalności[20] który pozwala wykorzystać funkcje zawarte w bibliotece OpenBLAS jako implementację mnożenia macierzowego. O ile kompilacja dla wszystkich platform które ma wspierać CSSFinder (Windows, Linux i MacOS) jest poza moim zasięgiem, to uznałem, że warto zweryfikować jakie efekty daje wykorzystanie tej funkcjonalności w środowisku laboratoryjnym.

Pomiary czasu pracy były wykonywane w taki sam sposób jak dla implementacji która nie korzystała z OpenBLAS.

[20] ang. feature switch



Rysunek 12: Wyniki testów wydajności implementacji w języku Rust z użyciem biblioteki OpenBLAS dla macierzy ρ_2 - ρ_6.

Wykorzystanie biblioteki OpenBLAS poskutkowało znaczącym wzrostem wydajności, przekraczającym możliwości oryginalnej implementacji. Wyniki te zostały przedstawione na rysunku 12. Kod tutaj omawiany ustępuje jedynie implementacji z sekcji 3.3.3, wykorzystującej JIT. Jednocześnie sprawuje się on gorzej dla małych macierzy niż wariant bez OpenBLAS.

3.4 Precyzja obliczeń

Oryginalny program, jak i re-implementacje które pojawiły się powyżej, posługiwały się liczbami zespolonymi na bazie liczb zmiennoprzecinkowych podwójnej precyzji. Jedna taka liczba zajmuje 64 bity. Jednak w wielu przypadkach taka precyzja obliczeń nie jest konieczna do uzyskania poprawnych wyników. Podstawową zaletą wykorzystania liczb zmiennoprzecinkowych pojedynczej precyzji, czyli 32 bitowych, jest zmniejszenie rozmiaru macierzy. Pozwala na umieszczenie większej części macierzy w pamięci podręcznej procesora. Dodatkowo zwiększa to przepustowość obliczeń wykorzystujących instrukcje SIMD, ponieważ wykorzystują one rejestry o stałych rozmiarach (128, 256, 512 bitów) które mogą na ogół pomieścić dwukrotnie więcej liczb 32 bitowych niż 64 bitowych. Pozwala to oczekiwać że obliczenia wykorzystujące liczby zmiennoprzecinkowe pojedynczej precyzji będą trwały krócej.

Tworzony przeze mnie kod od początku powstawał z zamysłem umożliwienia wykorzystania liczb zmiennoprzecinkowych o różnych precyzjach, dlatego transformacja ta była dość prosta. W języku Python, wykorzystując bibliotekę NumPy przejście na liczby pojedynczej precyzji wymagało prawie każdorazowego deklarowania że wynik operacji ma posiadać typ complex64 (cały czas mówimy o liczbach zespolonych które składają się z dwóch wartości zmiennoprzecinkowych). Nie wszystkie operacje które przyjmują parametr określający typ wejściowy są akceptowane przez kompilator JIT biblioteki Numba gdy jest on przekazywany. To

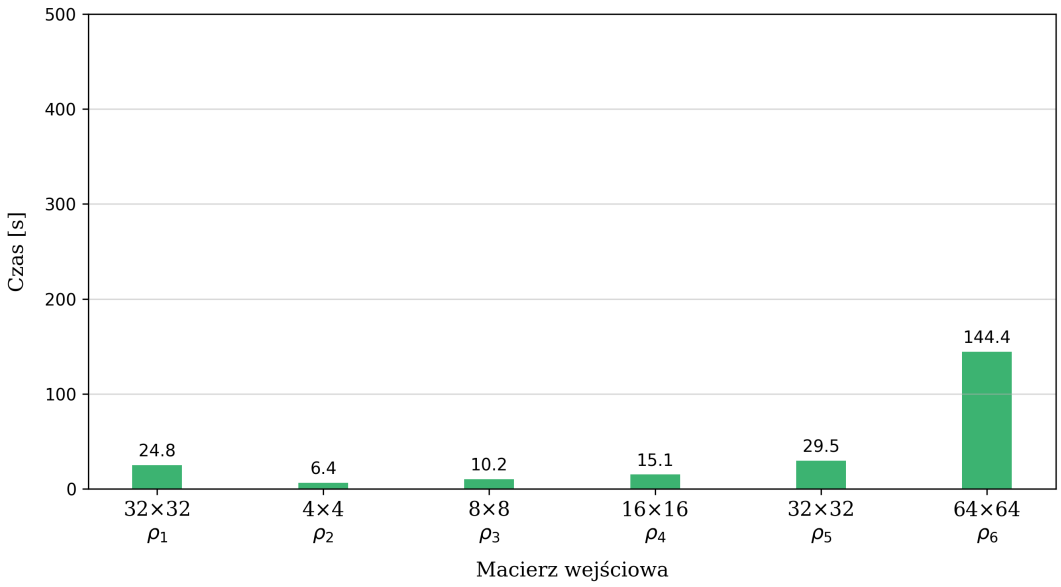
ograniczenie można obejść wykonując zmianę typu jako osobną operację przy pomocy metody `astype()`.

Warto tutaj zaznaczyć że wszystkie implementacje w języku Python powstają ze wspólnego szablony który był ewaluowany przez bibliotekę Jinja2 do różnych wariantów kodu, w zależności od tego jakie parametry były do niego przekazywane. Pozwoliło to uniknąć wielokrotnego pisania wspólnych fragmentów kodu, a elementy unikalne są dodawane warunkowo. Zastosowanie introspekcji do konstruowania odpowiedniego kodu w trakcie wykonywania programu mogłoby w znaczący sposób obniżyć wydajność, dlatego zdecydowałem się sięgnąć po system bardziej statyczny, który na pewno nie wpływał na czas pracy programu.

W przypadku języka Rust, posiada on dedykowany konstrukt składniowy pozwalający na deklarowanie funkcji w oparciu o symbole zastępcze wobec których stawia się zbiór wymagań dotyczących wspieranych interfejsów. W efekcie funkcja może zostać wyspecjalizowana żeby akceptować zarówno liczby zespolone skonstruowane z liczb zmiennoprzecinkowych pojedynczej jak i podwójnej precyzji. Pozwoliło to uniknąć sięgania po zewnętrzne mechanizmy do tworzenia szablonów, tak jak było to konieczne w języku Python.

3.5 Pomiary z pojedynczą precyzją

3.5.1 Python i NumPy

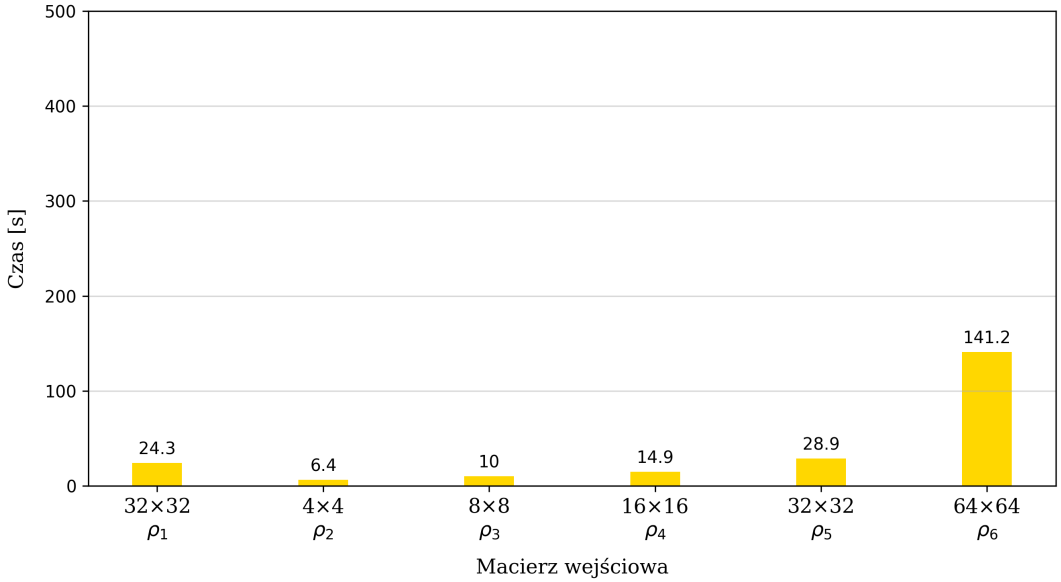


Rysunek 13: Wyniki testów wydajności implementacji w języku Python z użyciem biblioteki NumPy dla macierzy ρ_2 - ρ_6 i obliczeniami pojedynczej precyzji.

Na wykresie 13 przedstawiłem wyniki wydajności dla implementacji napisanej w języku Python wykorzystującej bibliotekę NumPy do przeprowadzania obliczeń na macierzach liczb zespolonych pojedynczej precyzji. W przypadku mniejszych macierzy (4×4, 8×8, 16×16) żadne różnice w czasie pracy, względem wariantu opartego na liczbach podwójnej precyzji, są minimalne,. Dzieje się tak prawdopodobnie dlatego, że macierze te są na tyle niewielkie (do

4KB) że mieszczą się w pamięci cache L1 procesora[21], więc wymnażanie ich jest procesem bardzo szybkim. W momencie kiedy docieramy do macierzy 32×32 co również można wytłumaczyć odwołując się do pojemności pamięci cache procesora. Macierze podwójnej precyzji zajmują dokładnie 16KB ($32 \times 32 \times 2 \times 8$), natomiast dostęp do tej pamięci nie jest ekskluzywny dla jednego procesu, nie może on więc korzystać z całych 16KB. W efekcie część danych przebywa poza pamięcią cache. Natomiast macierze wykorzystujące liczby pojedynczej precyzji zajmują tylko 8KB. Można się więc spodziewać że większość czasu spędzają one w pamięciach L1 i L2, co pozwala przyspieszyć obliczenia. Dodatkowo mniejszy rozmiar liczb pojedynczej precyzji pozwala dwukrotnie zwiększyć przepustowość instrukcji SIMD, co prawdopodobnie odgrywa również bardzo istotną rolę, szczególnie w przypadku macierzy 64×64, dla których obliczenia przyspieszają znacznie bardziej niż w przypadku mniejszych macierzy.

3.5.2 Python i NumPy z AOT

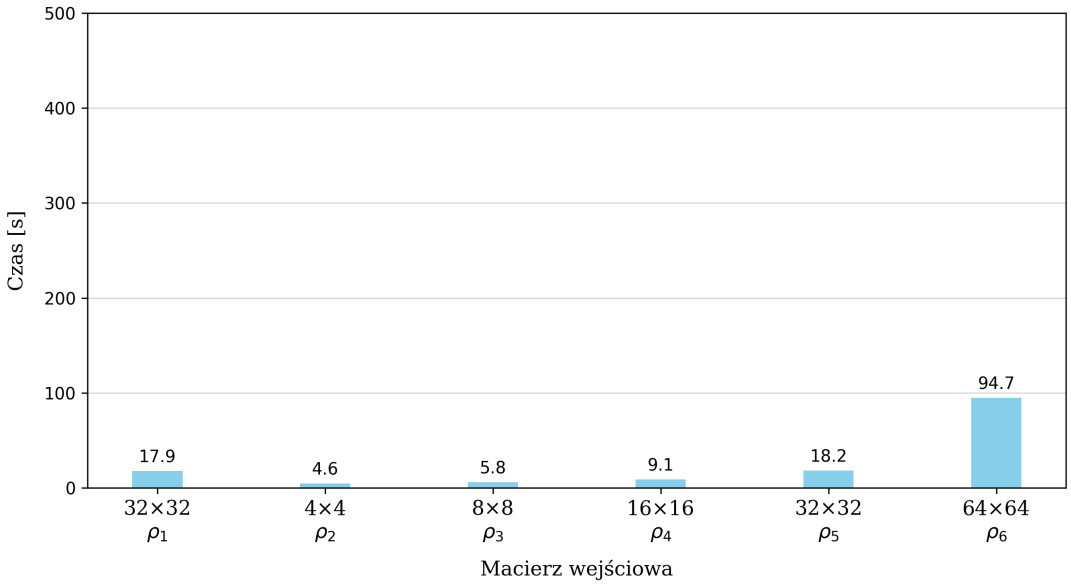


Rysunek 14: Wyniki testów wydajności implementacji w języku Python z użyciem biblioteki NumPy i pakietu Cython do kompilacji AOT dla macierzy ρ_2 - ρ_6 i obliczeniami pojedynczej precyzji.

Wyniki dla wariantu pre-kompilowanego przy pomocy biblioteki Cython nie różnią się znacząco od wariantu nie pre-kompilowanego, podobnie jak w przypadku obliczeń podwójnej precyzji, zostały one zaprezentowane na rysunku 14.

[21] Wykorzystywany tutaj Ryzen 9 7950X posiada 32×16KB cache L1

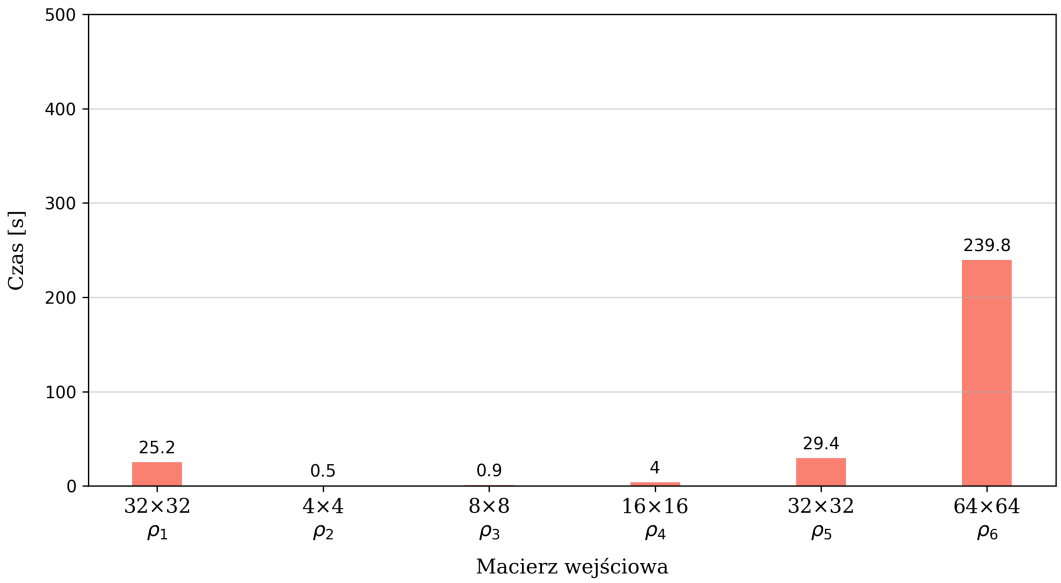
3.5.3 Python i NumPy z JIT



Rysunek 15: Wyniki testów wydajności implementacji w języku Python z użyciem biblioteki NumPy i pakietu Numba do kompilacji JIT dla macierzy ρ_2 - ρ_6 i obliczeniami pojedynczej precyzji.

W przypadku wariantu wykorzystującego kompilację JIT, zysk czasowy jest minimalny lub wręcz nie występuje. Ciężko mi wytłumaczyć dlaczego tak się dzieje, możliwe, że coś powoduje że obliczenia korzystają z tej samej implementacji.

3.5.4 Rust i Ndaray



Rysunek 16: Wyniki testów wydajności implementacji w języku Rust z użyciem biblioteki Ndarray dla macierzy ρ_2 - ρ_6 i obliczeniami pojedynczej precyzji.

Implementacja w języku Rust wykorzystująca bibliotekę Ndarray prezentuje najlepszą wydajność podczas obliczeń na małych macierzach, do 16×16 włącznie. Wynika to najprawdopodobniej z braku dodatkowego obciążenia ze strony interpretera, którego wewnętrzne operacje wprowadzają dodatkowe informacje do pamięci cache, tym samym wypierając z niej macierze na których są prowadzone obliczenia. W przypadku języka interpretowanego, CPU musi wykonywać o znacznie więcej instrukcji, ponieważ każda operacja w języku wysokiego poziomu musi zostać załadowana, po czym odpowiednia akcja musi zostać wybrana i dopiero wykonana. Tak długo jak kluczowe dla wydajności nie jest to ile zajmują mnożenia macierzowe, tak długo implementacja w języku Rust będzie szybsza.

3.5.5 Rust i Ndaray z OpenBLAS

4 Wyniki

5 Dyskusja

Odwołania

[1] J. D. Hunter. „Matplotlib: A 2D graphics environment". W: *Computing in Science & Engineering* 9.3 (2007), s. 90–95. DOI: 10.1109/MCSE.2007.55.

[2] Carl Friedrich Bolz i in. „Tracing the meta-level: PyPy's tracing JIT compiler". W: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. 2009, s. 18–25.

[3] Patrick Lindemann. „The gilbert-johnson-keerthi distance algorithm". W: *Algorithms in Media Informatics* (2009).

[4] Stefan Behnel i in. „Cython: The best of both worlds". W: *Computing in Science & Engineering* 13.2 (2010), s. 31–39.

[5] Yury Selivanov Elvis Pranskevichus. *What's New In Python 3.5*. 2015. URL: https://docs.python.org/3/whatsnew/3.5.html (term. wiz. 14. 05. 2023).

[6] Siu Kwan Lam, Antoine Pitrou i Stanley Seibert. „Numba". W: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, list. 2015. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

[7] Feng Li i in. „CPU versus GPU: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms". W: *Neural Computing and Applications* 31 (2019), s. 4353–4365.

[8] Inc. Anaconda i in. *Numba documentation*. 2020. URL: https://numba.readthedocs.io/en/stable/user/index.html (term. wiz. 12. 05. 2023).

[9] Charles R. Harris i in. „Array programming with NumPy". W: *Nature* 585.7825 (wrz. 2020), s. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[10] Palash Pandya, Omer Sakarya i Marcin Wieśniak. „Hilbert-Schmidt distance and entanglement witnessing". W: *Physical Review A* 102.1 (2020), s. 012409.

[11] Marcin Wieśniak i in. „Distance between bound entangled states from unextendible product bases and separable states". W: *Quantum Reports* 2.1 (2020), s. 49–56.

[12] Mirko Consiglio, Tony JG Apollaro i Marcin Wieśniak. „Variational approach to the quantum separability problem". W: *Physical Review A* 106.6 (2022), s. 062413.

[13] NumPy Developers. *NumPy documentation*. 2022. URL: https://numpy.org/doc/stable/ (term. wiz. 12. 05. 2023).

[14] Inc. GitHub. *The top programming languages*. 2022. URL: https://octoverse.github.com/2022/top-programming-languages (term. wiz. 14. 05. 2023).

[15] Marcin Wieśniak. „Two-Qutrit entanglement: 56-years old algorithm challenges machine learning". W: *arXiv preprint arXiv:2211.03213* (2022).

[16] Klaralvdalens Datakonsult AB. *GitHub - KDAB/hotspot: The Linux perf GUI for performance analysis.* 2023. URL: https://github.com/KDAB/hotspot (term. wiz. 31. 05. 2023).

[17] The go-python Authors. *go-python/gopy: gopy generates a CPython extension module from a go package.* 2023. URL: https://github.com/go-python/gopy (term. wiz. 14. 05. 2023).

[18] *Cython C-Extensions for Python.* 2023. URL: https://cython.org/ (term. wiz. 14. 05. 2023).

[19] Matt Davis. *snakeviz · PyPI.* 2023. URL: https://pypi.org/project/snakeviz/ (term. wiz. 22. 05. 2023).

[20] NumPy Developers. *Random Generator — NumPy v1.24 Manual.* 2023. URL: https://numpy.org/doc/1.24/reference/random/generator.html (term. wiz. 22. 05. 2023).

[21] The pip developers. *pip · PyPI.* 2023. URL: https://pip.pypa.io/en/stable/ (term. wiz. 14. 05. 2023).

[22] The PyO3 developers. *PyO3 user guide.* 2023. URL: https://pyo3.rs/v0.18.3/ (term. wiz. 14. 05. 2023).

[23] Agner Fog. *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs.* 2023. URL: https://docs.rs/ndarray/latest/ndarray/index.html (term. wiz. 22. 05. 2023).

[24] Python Software Foundation. *ctypes — A foreign function library for Python.* 2023. URL: https://docs.python.org/3/library/ctypes.html (term. wiz. 14. 05. 2023).

[25] Python Software Foundation. *Extending Python with C or C++.* 2023. URL: https://docs.python.org/3/extending/extending.html (term. wiz. 14. 05. 2023).

[26] Inc. Free Software Foundation. *Git.* 2023. URL: https://gcc.gnu.org/ (term. wiz. 14. 05. 2023).

[27] Ivan Levkivskyi Guido van Rossum. *PEP 483 - The Theory of Type Hints.* 2023. URL: https://peps.python.org/pep-0483/ (term. wiz. 14. 05. 2023).

[28] Łukasz Langa Guido van Rossum Jukka Lehtosalo. *PEP 484 - Type Hints.* 2023. URL: https://peps.python.org/pep-0484/ (term. wiz. 14. 05. 2023).

[29] hanabi1224. *Programming Language and compiler Benchmarks - C VS Python benchmarks.* 2023. URL: https://programming-language-benchmarks.vercel.app/c-vs-python (term. wiz. 14. 05. 2023).

[30] hanabi1224. *Programming Language and compiler Benchmarks - C++ VS Python benchmarks.* 2023. URL: https://programming-language-benchmarks.vercel.app/cpp-vs-python (term. wiz. 14. 05. 2023).

[31] hanabi1224. *Programming Language and compiler Benchmarks - Rust VS Python benchmarks.* 2023. URL: https://programming-language-benchmarks.vercel.app/rust-vs-python (term. wiz. 14. 05. 2023).

[32] Steve Klabnik i Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[33] Jukka Lehtosalo i mypy contributors. *mypy 1.2.0 documentation*. 2023. URL: https://mypy.readthedocs.io/en/stable/ (term. wiz. 14. 05. 2023).

[34] TIOBE Software. *TIOBE Index for May 2023*. 2023. URL: https://www.tiobe.com/tiobe-index/ (term. wiz. 14. 05. 2023).

[35] mypyc team. *mypyc 1.2.0 documentation*. 2023. URL: https://mypyc.readthedocs.io/en/stable/ (term. wiz. 14. 05. 2023).

[36] The PyPy Team. *PyPy Home Page*. 2023. URL: https://www.pypy.org/ (term. wiz. 14. 05. 2023).

[37] Krzysztof Wiśniewski. *CSSFinder (Core, PyPI)*. 2023. URL: https://pypi.org/project/cssfinder/ (term. wiz. 14. 05. 2023).

[38] Krzysztof Wiśniewski. *CSSFinder (Core)*. 2023. URL: https://github.com/Argmaster/CSSFinder (term. wiz. 12. 05. 2023).

[39] Krzysztof Wiśniewski. *CSSFinder Numpy Backend*. 2023. URL: https://github.com/Argmaster/cssfinder_backend_numpy (term. wiz. 12. 05. 2023).

[40] Krzysztof Wiśniewski. *CSSFinder Numpy Backend (PyPI)*. 2023. URL: https://pypi.org/project/cssfinder-backend-rust/ (term. wiz. 12. 05. 2023).

[41] Krzysztof Wiśniewski. *CSSFinder Rust Backend*. 2023. URL: https://github.com/Argmaster/cssfinder_backend_rust (term. wiz. 12. 05. 2023).

[42] Krzysztof Wiśniewski. *CSSFinder Rust Backend (PyPI)*. 2023. URL: https://pypi.org/project/cssfinder-backend-numpy/ (term. wiz. 12. 05. 2023).

[43] Zhang Xianyi. *OpenBLAS : An optimized BLAS library*. 2023. URL: https://www.openblas.net/ (term. wiz. 31. 05. 2023).