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

1 Wstęp

Closest Separable State Finder (CSSFinder) jest programem pozwalającym na detekcję splątania kwantowego układu oraz określenie jak silnie owe splątanie jest. Bazuje on na dostosowanym algorytmie Elmera G. Gilberta[1], pozwalającym na wyliczenie przybliżonej wartości odległości Hilberta-Schmidta (ang. Hilberta-Schmidta distance, HSD) pomiędzy stanem a zbiorem stanów separowanych. W literaturze algorytm ten pojawia się pod nazwą 'kwantowy algorytm Gilberta'(ang. quantum Gilbert algorithm, QGA)[2]. Działanie tego algorytmu zostało opisane w pracy 'Hilbert-Schmidt distance and entanglement witnessing' której autorami byli Palash Pandya, Omer Sakarya i Marcin Wieśniak[3].

Dr hab. Marcin Wieśniak, prof. UG, utworzył implementację algorytmu QGA w języku Python, wykorzystując bibliotekę NumPy do przeprowadzania koniecznych obliczeń macierzowych. Wybór ten był podyktowany możliwościami oferowanymi przez taki zestaw narzędzi. Pozwalały one w szybki sposób stworzyć prosty kod, zdolny by relatywnie wydajnie przeprowadzać obliczenia na wszystkich najpopularniejszych systemach dla komputerów stacjonarnych.

Zalety języka Python są powszechnie dostrzegane zarówno przez środowiska akademickie, jak i komercyjne, co wyraźnie widać w zestawieniach takich jak wydane przez GitHub, Inc. 'The top programming languages' (2022)[4]. Język Python plasuje się w nim na drugim miejscu.

Alternatywy w postaci języków C, C++ czy Fortran wymagałyby większej ilości bardziej skompilowanego kodu, jednocześnie zmuszając do ręcznego skompletowania systemu budowania, bibliotek oraz zastosowania dedykowanych rozwiązań dla każdego systemu operacyjnego, a przeprowadzanie obliczeń byłoby utrudnione.

1.1 Działanie programu

Oryginalny program i jego re-implementacja posiadają praktycznie identyczną zasadę działania i tylko szczegóły dotyczące sposobu interakcji z nim zmieniły się. Z tego względu w dalszej części tekstu sposób działania programu będzie opisywany bez rozgraniczenia na wersję oryginalną i re-implementację.

Program jako dane wejściowe przyjmuje macierz gęstości opisującą pewien stan ρ_0 układu kwantowego. Następnie program w określonych wypadkach jest w stanie wydedukować wymiary podukładów i ich liczbę lub można je podać jawnie. Następnie dobierany jest stan separowalny ρ_1. Następnie program postępuje zgodnie z następującymi krokami:

1. Zwiększ licznik prób c_t o 1. Wylosuj czysty stan produktowy ρ_2, zwany dalej stanem próbnym.

2. Uruchom preselekcję dla stanu próbnego poprzez sprawdzenie funkcji liniowej. Jeśli się nie powiedzie, wróć do punktu 1.

3. W przypadku udanej preselekcji symetryzujemy ρ_1 względem wszystkich symetrii przez ρ_0, które respektują separowalność.

4. Znaleźć minimum $Tr(\rho_0 - p\rho_1 - (1-p)\rho_2)^2$ względem p.

5. Jeśli minimum występuje dla $0 \leq p \leq 1$, zaktualizuj $\rho_1 \leftarrow p\rho_1 - (1-p)\rho_2$, dodać nową wartość $D^2(\rho_0, \rho_1)$ do listy listy i zwiększyć wartość licznika sukcesu c_s o 1.

6. Przejdź do kroku 1, aż spełnione zostanie wybrane kryterium zatrzymania.

Jako dane wyjściowe program zapisuje do plików historię poprawek i stan ρ_1. Dostępnymi kryteriami zatrzymania jest maksymalna ilość korelacji do uzyskania oraz maksymalna ilość iteracji do wykonania - ta z tych wartości która zostanie osiągnięta jako pierwsza decyduje o zatrzymaniu programu. Jeśli wyznaczona przez program odległość HSD jest odpowiednio niewielka (tj. mniejsza niż $1 \cdot 10^{-4}$) stan jest praktycznie separowalny w przeciwnym wypadku może być uważany za splątany.

1.2 Cele pracy

Celami tej pracy sa:

- eksploracja dostępnych metod maksymalizacji wydajności algorytmu QGA,
- implementacja algorytmu QGA wybranymi z metod,
- weryfikacja poszczególnych z tych rozwiązań pod kątem zmian w czasie pracy programu,
- dostosowane postaci i sposób dystrybucji programu do standardów ekosystemu języka Python.

1.3 Przyczyny przystąpienia do optymalizacji

Na ogół rozpatruje się wiele stanów kwantowych, aby zapoznać się z wybranymi obszarami przestrzeni stanów kwantowych. Wymaga to więc wielokrotnego wywoływania programu CSSFinder na wielu różnych macierzach wejściowych. W naturalny sposób preferowanym jest więc aby obliczenia dla jednego stanu trwały jak najkrócej, im mniej czasu zajmą tym więcej stanów zostanie zbadanych w tym samym czasie.

Niestety, język Python wykorzystany do stworzenia oryginalnej implementacji jest powszechnie znany z problemów z wydajnością[5]. Są one pokłosiem faktu że jest to interpretowany język programowania, a więc konieczne jest by specjalny program (tak zwany interpreter) wykonywał instrukcje zawarte w kodzie programu. Dodatkowo jest to język dynamicznie typowany z bardzo rozbudowanymi możliwościami introspekcji, uniemożliwia to zastosowanie wielu z optymalizacji powszechnie wykorzystywanych w innych językach programowania. Cechy te są jednocześnie jednymi z największych zalet Pythona, obok czytelnej składni i rozbudowanego ekosystemu.

Aby zwiększyć wydajność, koniecznym jest więc poczynić pewne kompromisy i zrezygnować z rozwiązań wygodnych na rzecz rozwiązań bardziej optymalnych dla wydajności. Jednocześnie

niekorzystnym byłoby od razu sięgać po język asemblera, ponieważ posiada on najmniejsze dodatkowe obciążenie i oferuje największą kontrolę nad urządzeniem na którym program jest wykonywany. Chociażby ze względu na fakt że program ten będzie:

1. nie przenośny pomiędzy architekturami,
2. czasochłonny do napisania,
3. istnieje bardzo niewielka szansa że będzie wydajniejszy,
4. wielokrotnie bardziej objętościowy.

Z tego względu w pracy tej rozważę kilka rozwiązań, czynią mniej radykalne kompromisy i wymagając różnej ilości dodatkowego wysiłku aby uzyskać sprawny program. Jednocześnie pokażę też, że nie jest konieczne pisanie kodu asemblera własnoręcznie, by uzyskać wysoką wydajność.

2 Narzędzia

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[6]. 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 [7]. 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[8] lub GO z użyciem biblioteki gopy[9].
3. Wykorzystując bibliotekę Cython[10][11]. 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[12]. Ta, podobnie do biblioteki Cython, również zawiera transpilator, natomiast zamiast korzystać z dedykowanego języka, opiera się on na dodanych w Pythonie 3.5[13] (PEP 484[14] i PEP 483[15]), adnotacjach typów. Jest on rozwijany obok projektu mypy - pakietu do statycznej analizy typów dla języka Python, również opartej na adnotacjach typów[16].

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).

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[17]. 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[18].

Drugim narzędziem jest biblioteka Numba[19][20]. Ona, w przeciwieństwie do PyPy, wymaga aby fragmenty kodu, które mają być skompilowane, miały postać funkcji oznaczonych dedykowanymi dekoratorami.

2.3 Selekcja narzędzi

Język programowania	**Biblioteki**	**Nazwa podprojektu**
Python	NumPy	cssfinder_backend_numpy
Python	NumPy, Cython	cssfinder_backend_numpy
Python	NumPy, Numba	cssfinder_backend_numpy
Rust	Ndarray	cssfinder_backend_rust
Rust	Ndarray, OpenBLAS	cssfinder_backend_rust

Tablica 1: Wybrane narzędzia.

Tabela 1 zawiera zestawienie języków programowania i zastosowanych bibliotek użytych do wykonania re-implementacji algorytmu QGA.

2.3.1 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. Był to dokładnie taki same zestaw, jak wykorzystany do oryginalnej implementacji. 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).

2.3.2 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.

2.3.3 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[1]. Drugi polega na śledzeniu typów wejściowych i wyjściowych funkcji i automatycznie kompiluje funkcję dla tych typów danych które są odpowiednio często używane używane[2].

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

[1] ang. eager (compilation) - niecierpliwa (kompilacja).
[2] ang. lazy (compilation) - leniwa (kompilacja).

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.

2.3.4 Rust i Ndarray

Aby uczynić to porównanie jak najpełniejszym, podjąłem również wysiłek zaimplementowania części obliczeniowej programu w języku Rust.

Język ten wybrałem z kilku względów. Przede wszystkim posiada on gotową, rozbudowaną infrastrukturę narzędzi pomocniczych. Do tych narzędzi zaliczyć należy menadżera pakietów cargo, który zarówno pozwala w łatwy sposób kompilować bardziej rozbudowane projekty i tworzyć z nich łatwe do obsługi pakiety, ale również daje możliwość korzystania z pakietów udostępnionych przez innych programistów.

Pozwoliło to w łatwy i szybki sposób skompletować zestaw bibliotek umożliwiających wydajne i wygodne tworzenie kodu implementacji algorytmu QGA. Ponadto istnienie biblioteki PyO3 znacząco uprościło proces tworzenia interfejsu pozwalającemu interpreterowi języka Python na interakcję z tą implementacją.

Jednocześnie język Rust jest językiem:

1. kompilowanym,
2. wykorzystującym zestaw narzędzi kompilatora LLVM,
3. statycznie typowanym,
4. posiadającym automatyczny system zarządzania pamięcią oparty na koncepcji posiadania (ang. ownership), który usuwa konieczność manualnego zarządzania pamięcią, zarazem bez konieczności wprowadzania mechanizmu liczenia referencji i dedykowanego automatycznego ‘odśmiecacza’ (ang. garbage collector).

Cechy te pozwalają oczekiwać, że skompilowany kod będzie osiągał wydajność zbliżoną do kodu C/C++, skompilowanych przy pomocy kompilatora clang, który również wykorzystuje LLVM do optymalizacji kodu.

Cały proces wstępnej konfiguracji sprowadził się do około godziny, co stanowi wyśmienity wynik, a cały proces implementacji zajął niewiele więcej czasu niż implementacja w języku Python. Jednocześnie język Rust posiada system typów który jest w stanie pomieścić bardzo dużo informacji o zamiarach programisty. W efekcie kompilator ma możliwość wychwycić wiele błędów, których nie może zauważyć kompilator języka C++.

2.3.5 Rust i Ndarray z OpenBLAS

Biblioteka Ndarray, która jest sercem implementacji w języku Rust, posiada przełącznik funkcjonalności[3] który pozwala wykorzystać funkcje zawarte w bibliotece OpenBLAS jako implementację mnożenia macierzowego. Powoduje to niestety, że kompilacja programu zaczyna wymagać by biblioteka OpenBLAS była zainstalowana i dostępna podczas kompilacji, co jest trudne do uzyskania w środowisku które wykorzystuję do kompilacji. W efekcie kompilacja dla wszystkich platform które ma wspierać CSSFinder (Windows, Linux i MacOS) jest poza moim zasięgiem, natomiast byłem w stanie przeprowadzić ją na komputerze który wykorzystuję do testów wydajności, więc została ona wzięta pod uwagę w zestawieniu.

3 Metody

3.1 Modularyzacja

Re-implementując program CSSFinder planowałem wypróbować liczne rozwiązania, które wymagały zasadniczych zmian w kodzie algorytmu, w tym przepisania go w innym języku programowania. 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 dostępnych implementacji algorytmu ('backendów'). Dane macierzowe w obrębie korpusu przechowywane są jako obiekty ndarray z biblioteki NumPy, ze względu na uniwersalność w świecie bibliotek do obliczeń tensorowych (wiele bibliotek w innych językach programowania oferuje gotowe narzędzia do transformacji obiektów ndarray na reprezentacje charakterystyczne dla tych bibliotek).

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. W przyszłości może to również pozwolić na łatwiejszy rozwój nowych implementacji oraz dodawanie nowych funkcjonalności do programu, ponieważ będą one mogły być implementowane stopniowo w różnych implementacjach.

3.2 Dane testowe

Podczas pomiarów konsekwentnie wykorzystywałem ten sam zestaw macierzy gęstości, aby móc wygodnie porównywać wyniki wydajności poszczególnych implementacji. W dalszej części pracy

[3] ang. feature switch

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 \\
-0.25 & 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 1: Macierz ρ_1.

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 2: 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[4]. Macierze te stanowią wygodny zestaw danych do weryfikacji ogólnej charakterystyki

---

[4] Tak więc macierz ρ_2 ma wymiary 4×4 i reprezentuje 2 kubity, macierz ρ_3 ma wymiary 8×8 i reprezentuje

zachowania alternatywnych implementacji algorytmu, pomimo, że wyniki przy ich pomocy uzyskiwane tak bardzo odbiegają od tych uzyskiwanych przy pomocy ρ_0.

3.3 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.

Oprogramowanie	**Wersja**
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
Sprzęt komputerowy	
CPU	AMD Ryzen 9 7950X
RAM	64GB DDR5 5600MHz CL40
DRIVE	512GB SSD GOODRAM CX400 (SATA)

Tablica 2: Konfiguracja środowiska testowego.

3.4 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[21].

3 kubity, macierz ρ_4 ma wymiary 16×16 i reprezentuje 4 kubity, itd. aż do ρ_6, 64×64 .

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[22].

3.5 Precyzja obliczeń

Oryginalny program, jak i pierwsze stworzone przezemnie re-implementacje 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.

3.6 Wykresy

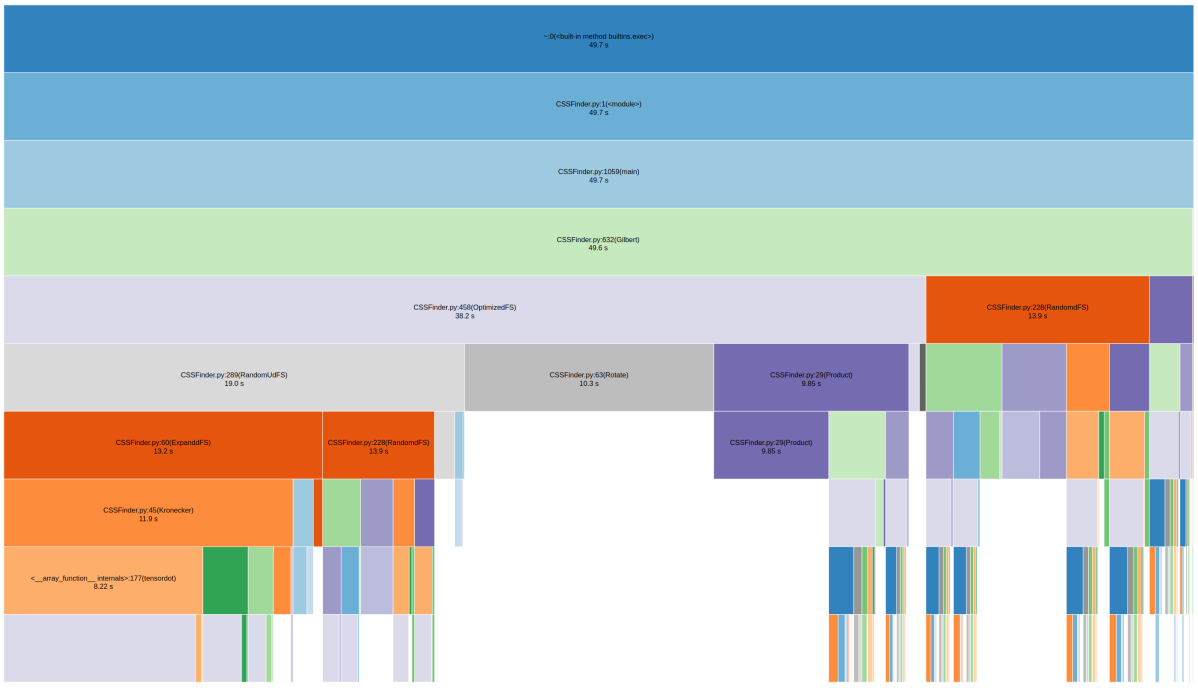
Wszystkie wykresy zamieszczone w tej pracy zostały utworzone przy pomocy skryptów w języku Python z wykorzystaniem biblioteki matplotlib[23].

4 Wyniki

4.1 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_1 (Rysunek 1).

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[24]) z ziarnem ustawionym na wartość 0.



Rysunek 3: Diagram podsumowujący pracę programu wygenerowany przez program Snakeviz.

Pozwoliło mi to wstępnie przyjrzeć się charakterystyce pracy programu i ocenić czy powszechnie dostępne narzędzia mogą zostać wykorzystane w tym wypadku. Rysunek 3 przedstawia diagram, typu Icicle, obrazujący udział czasu, pochłoniętego przez wykonywanie poszczególnych funkcji, w całkowitym czasie pracy programu. Pierwszy blok od góry (`:0(<built-in method builtins.exec>)`) to wywołanie funkcji wykonującej kod programu. Następne bloki, których opisy zaczynają się od 'CSSFinder.py' to wywołania w kodzie programu. Bloki umieszczone najniżej, w większości pozbawione opisów, to wywołania do funkcji bibliotek, głównie NumPy, ale również modułów wbudowanych Pythona. 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 3: 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%[5]) 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 pętli optymalizacyjnej, w tym zawartych w niej operacji macierzowych. Najprostszym sposobem na na uzyskanie takich efektów jest zastąpienie dynamicznego systemu typów i kodu bajtowego algorytmu wykonywanego przez interpreter pythona na statyczny system typów i kod maszynowy. Dodatkowo, niezastąpione są biblioteki

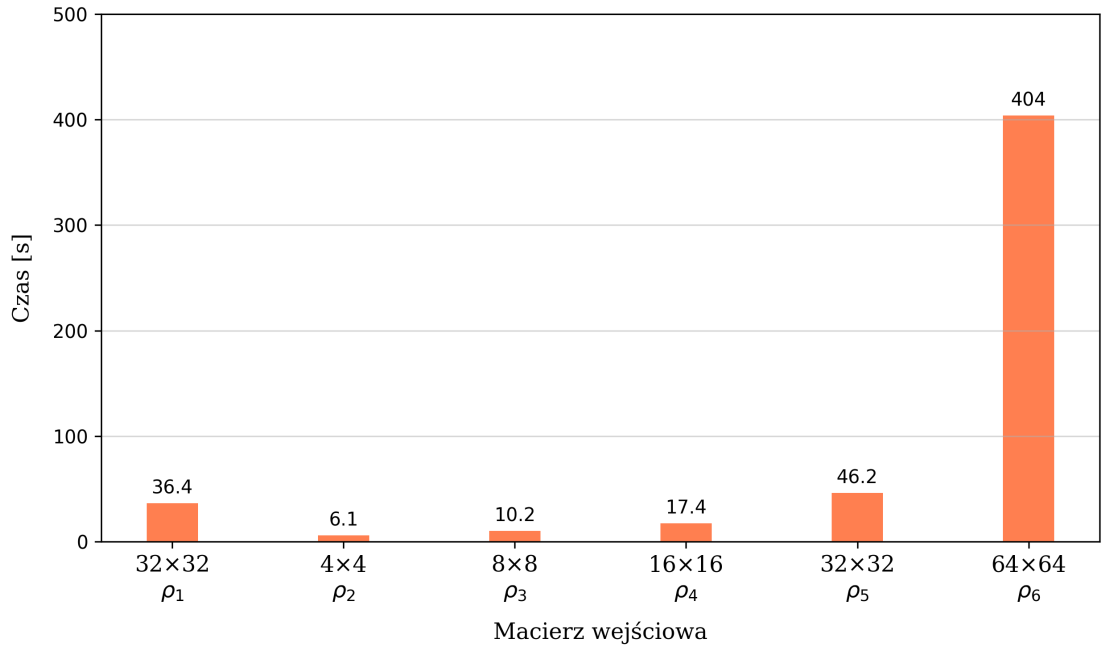
[5]Wartość 77% jak i wartości procentowe dalszej części tego akapitu zostały zaokrąglone do jedności, ze względu na małe znaczenie rzeczowe części ułamkowych.

zawierające wyspecjalizowane implementacje operacji macierzowych, takie jak OpenBLAS. Profilowanie pozwoliło również wykluczyć problemy z operacjami zapisu/odczytu plików oraz inne niespodziewane zjawiska.

4.2 Wstępne pomiary wydajności

Aby uzyskać dobrą bazę porównawczą, wykonałem serię pomiarów czasu pracy programu na macierzach ρ_1, ρ_2 - ρ_6, przedstawionych na rysunkach 1 i 2.

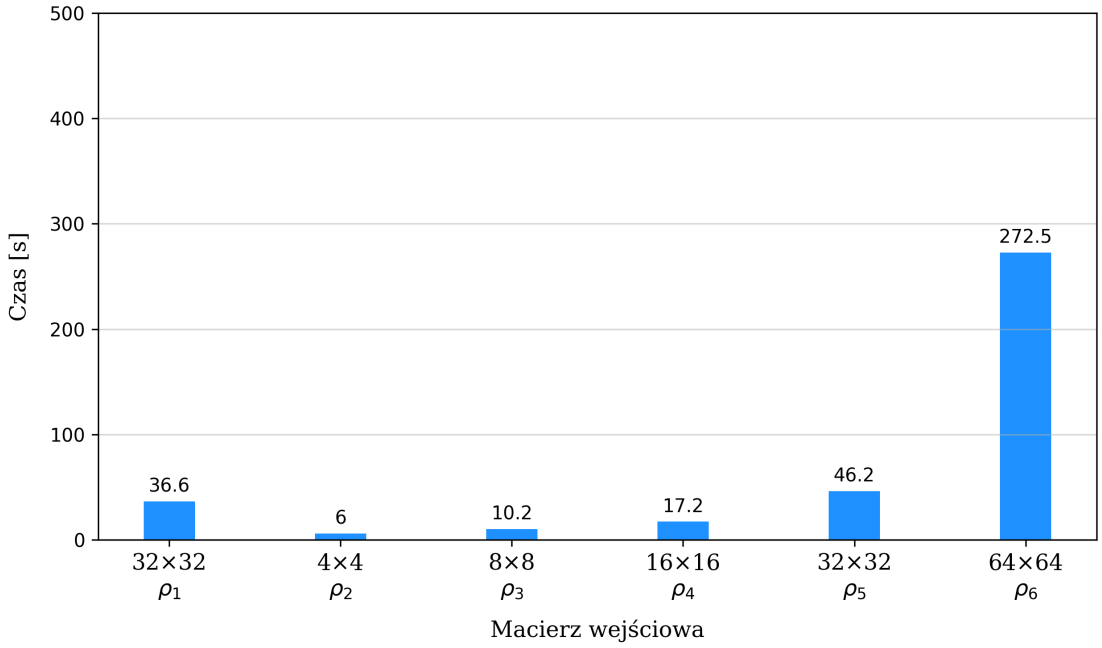
Dane przekazywałem kolejno do programu z poleceniem działania w trybie 1 (full separability of an n-quDit state) do osiągnięcia 1000 korekcji lub do 2.000.000 iteracji algorytmu, w zależności od tego co nastąpi szybciej. Dla wszystkich macierzy algorytm uzyskał 1000 korekcji i w żadnym przypadku nie 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[6].



Rysunek 4: Wyniki wstępnych testów wydajności oryginalnego kodu dla macierzy ρ_1 - 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 4 zostały przedstawione czasy obliczeń dla macierzy ρ_1 - ρ_6 z domyślnym zachowaniem biblioteki.

[6]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 5: Wyniki wstępnych testów wydajności oryginalnego kodu z zablokowaną liczbą wątków obliczeniowych dla macierzy ρ_1 - 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 5. 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.

Chciałbym uściślić, że w dalszej części pracy, mówiąc o wynikach oryginalnego kodu, będę miał na myśli wersję bez zablokowanej ilości wątków, a więc tę której wyniki umieszczone są na rysunku 4, jako że to była pierwotna postać kodu, natomiast zablokowanie ilości wątków wymagało już jego modyfikacji.

4.3 Pomiary z podwójną precyzją

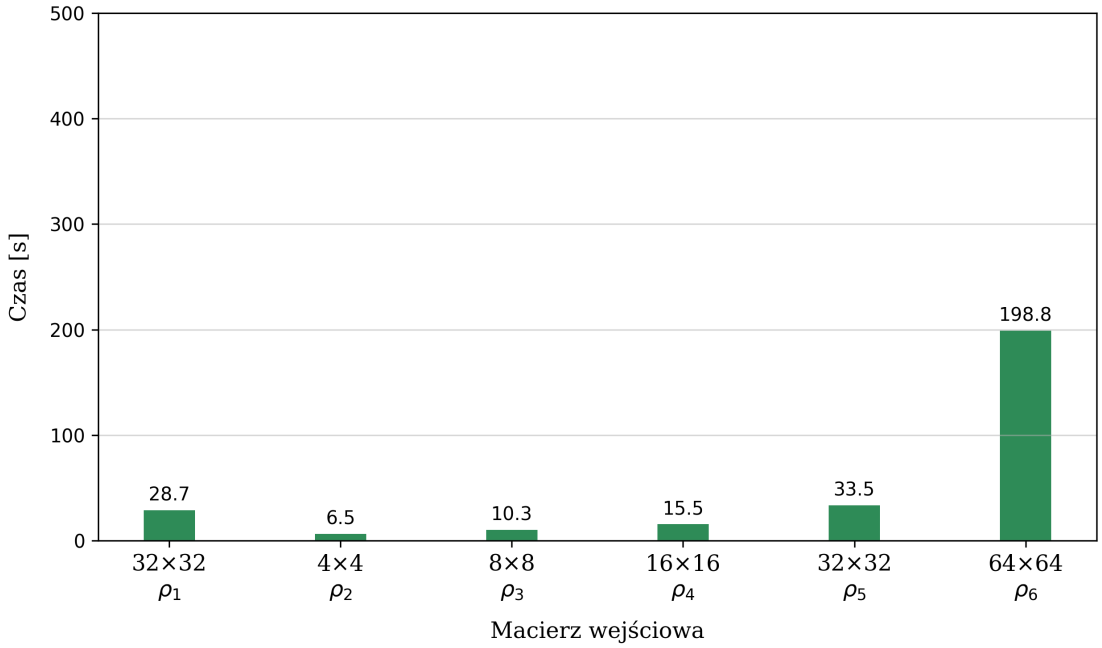
W dalszej części pracy prezentuje wyniki pomiarów czasu pracy re-implementacji algorytmu QGA wykorzystujących liczby zmiennoprzecinkowe podwójnej precyzji.

4.3.1 Python i NumPy

Pomiary czasu pracy były wykonywane przy użyciu macierzy ρ_1 - ρ_6. Dane przekazywałem kolejno do programu z poleceniem działania w trybie FSnQd[7] do osiągnięcia co najmniej 1000

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

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[8].



Rysunek 6: Wyniki testów wydajności alternatywnej implementacji Python z użyciem biblioteki NumPy dla macierzy ρ_1 - ρ_6.

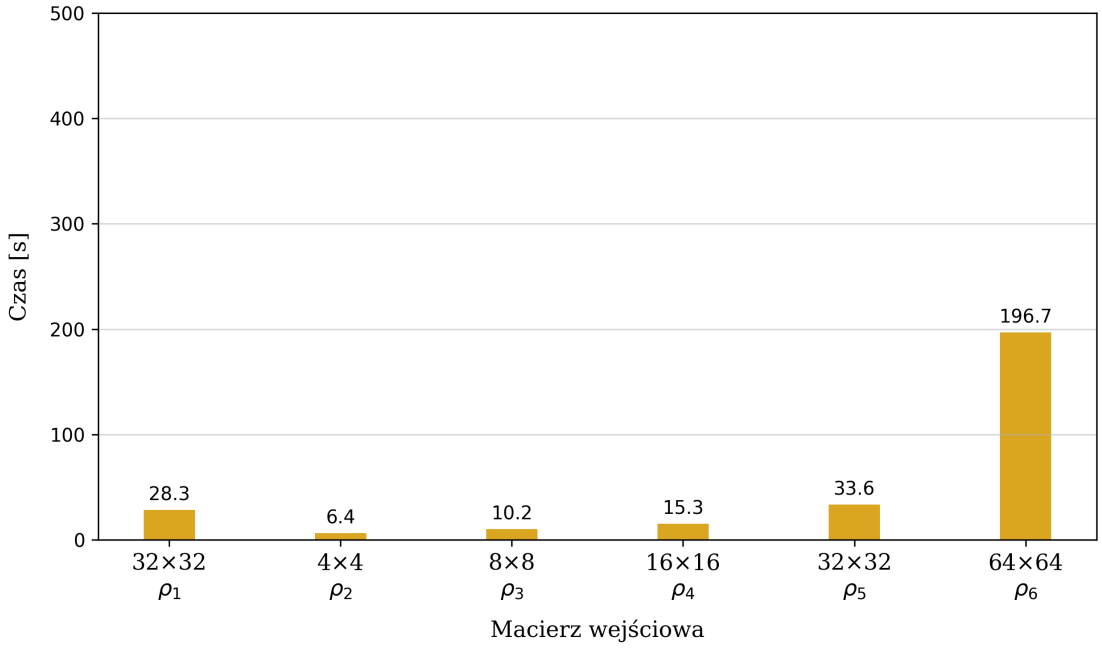
Uzyskane wyniki zostały przedstawione na rysunku 6. W przypadku małych macierzy wyniki są bardzo zbliżone, natomiast w przypadku macierzy 32×32 i 64×64 występuje znacząca poprawa wydajności, odpowiednio $7.9s$ ($\approx 21\%$) dla ρ_1, $12.7s$ ($\approx 27\%$) dla ρ_5 i $205.2s$ ($\approx 50\%$) dla ρ_6.

4.3.2 Python i NumPy z AOT

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

Na rysunku 7 przedstawione zostały wyniki pomiarów czasu pracy skompilowanej wersji w języku Python bazującej na bibliotece NumPy wykorzystujące macierze ρ_1 - ρ_6. kompilacja nie poskutkowała istotnym skróceniem czasu pracy programu względem wariantu bez AOT, różnice wynoszą koło 1%. Uzysk ten może być spowodowany usunięciem szczątkowego obciążenia ze strony interpretera, które nie jest mierzalne podczas krótszych testów z mniejszymi macierzami. Możliwe jest również że ta różnica wynika z korzystniejszych warunków losowo zapewnionych przez system operacyjny.

[8]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.



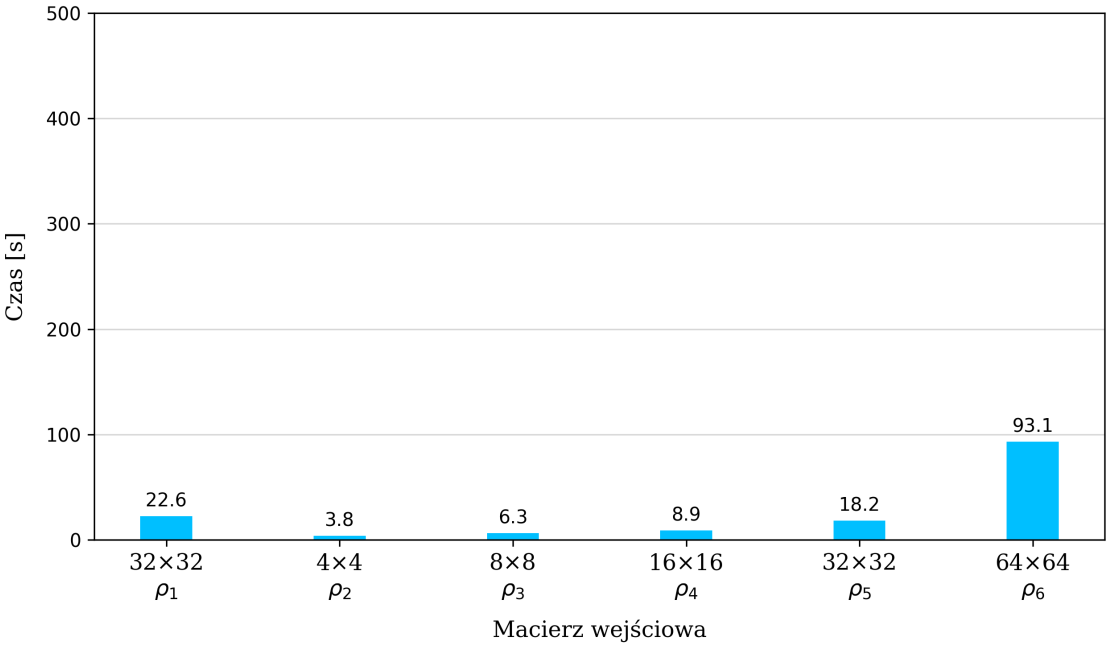
Rysunek 7: Wyniki testów wydajności implementacji Python z użyciem biblioteki NumPy oraz pakietu Cython do kompilacji AOT dla macierzy ρ_1 - ρ_6.

4.3.3 Python i NumPy z JIT

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

Na rysunku 8 przedstawione zostały wyniki uzyskane podczas pomiarów czasu pracy implementacji z kompilacją JIT wykonywaną przy pomocy biblioteki Numba, wykorzystując macierze ρ_1 - ρ_6. Implementacja ta oferowała podczas testów blisko dwukrotnie krótszy czas obliczeń, względem oryginału, w przypadku macierzy mniejszych niż 32×32. Dla macierzy większych uzysk wynosił odpowiednio $13.8s$ ($\approx 38\%$) dla ρ_1, $28s$ ($\approx 60\%$) dla ρ_5 i $310.9s$ ($\approx 77\%$) dla ρ_6.

Tak znaczącą poprawę implementacja zawdzięcza prawdopodobnie temu, że kompilator JIT całkowicie pozbywa się obciążenia ze strony dynamicznego systemu typów, generując wyspecjalizowany statycznie typowany kod maszynowy. Ponadto może on specjalizować kod dla dokładnie jednej platformy, korzystając z całego spektrum jej możliwości. Dotyczy to na przykład instrukcji SIMD, takich jak AVX2 i FMA, 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.



Rysunek 8: Wyniki testów wydajności implementacji w języku Python z użyciem biblioteki NumPy i pakietu Numba do kompilacji JIT dla macierzy ρ_1 - ρ_6.

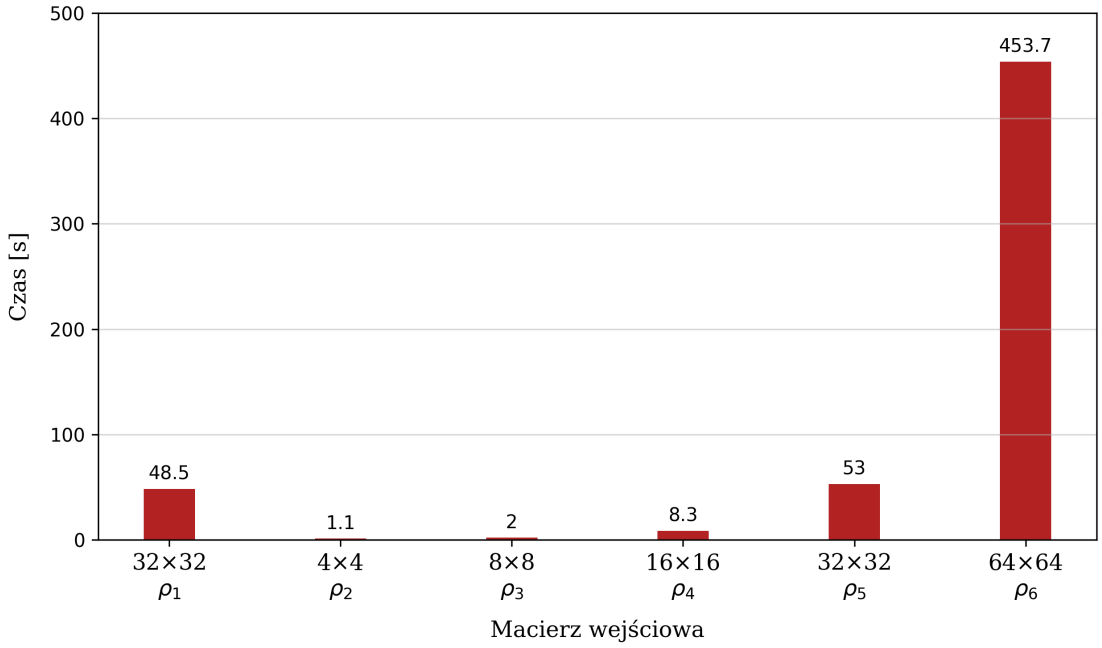
4.3.4 Rust i Ndaray

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[9].

Na rysunku 9 zaprezentowane zostały wyniki pomiarów czasu pracy implementacja w języku Rust. Względem oryginału czas pracy dla małych macierzy skrócił się około pięciokrotnie, dotyczy to rozmiarów 4×4 i 8×8. W przypadku macierzy 16×16 uzysk jest już tylko dwukrotny, natomiast dla większych macierzy uzyskuje ona wyniki gorsze niż oryginał.

Jest to prawdopodobnie spowodowane tym, że sama implementacja mnożenia macierzowego korzysta z ograniczonego zasobu instrukcji SIMD, aby zachować jak najszerszą kompatybilność w przeciwieństwie do biblioteki NumPy, która wewnętrznie wykorzystuje bibliotekę OpenBLAS[25].

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

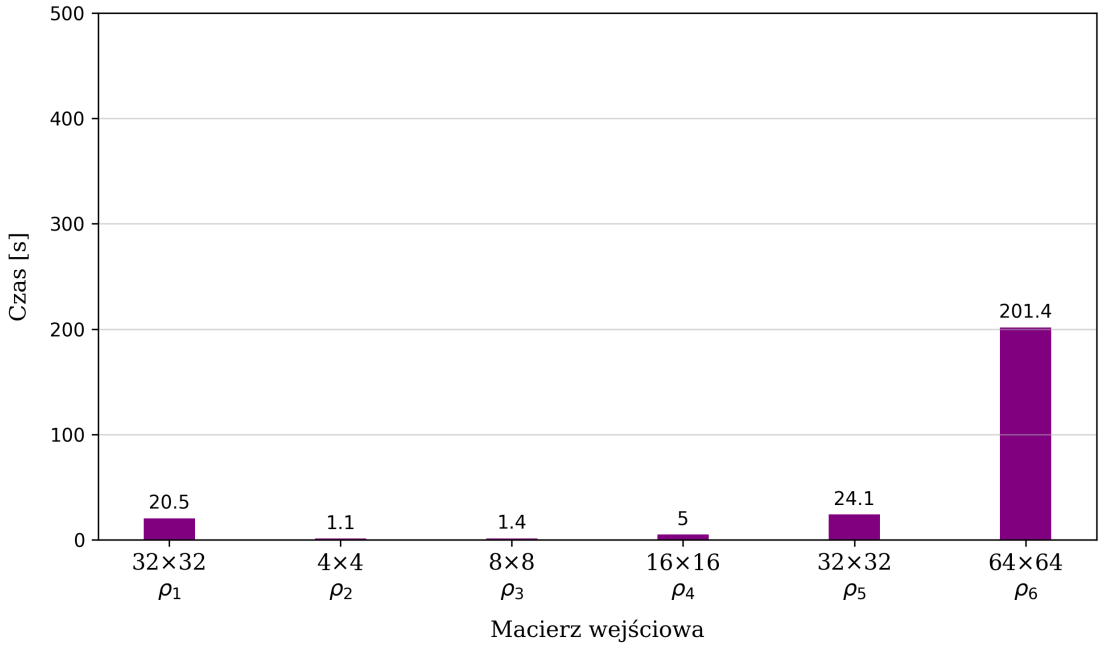


Rysunek 9: Wyniki testów wydajności implementacji w języku Rust dla macierzy ρ_1 - ρ_6.

4.3.5 Rust i Ndaray z OpenBLAS

Pomiary czasu pracy implementacji w języku Rust wykorzystującej OpenBLAS do wykonywania mnożenia macierzowego były wykonywane w taki sam sposób jak dla implementacji która nie korzystała z tej biblioteki.

Wyniki testów przeprowadzanych na implementacji korzystającej z biblioteki OpenBLAS poskutkowało znaczącym skróceniem czasu pracy względem oryginału. Wyniki te zaprezentowane zostały na rysunku 10. Największa różnica występuje dla małych macierzy, gdzie podobnie do wariantu nie korzystającego z OpenBLAS, przyspieszenie jest około pięciokrotne dla ρ_2, ρ_3 i ponad trzykrotne dla ρ_2. Dla macierzy 32×32 i 64×64 obliczenia zajęły około dwukrotnie mniej czasu. Pokazuje to jak znaczący wzrost wydajności można uzyskać przy pomocy wyspecjalizowanego kodu w języku Asemblera, który został wykorzystany w bibliotece OpenBLAS do stworzenia wyspecjalizowanych implementacji mnożenia macierzowego.



Rysunek 10: Wyniki testów wydajności implementacji w języku Rust z użyciem biblioteki OpenBLAS dla macierzy ρ_1 - ρ_6.

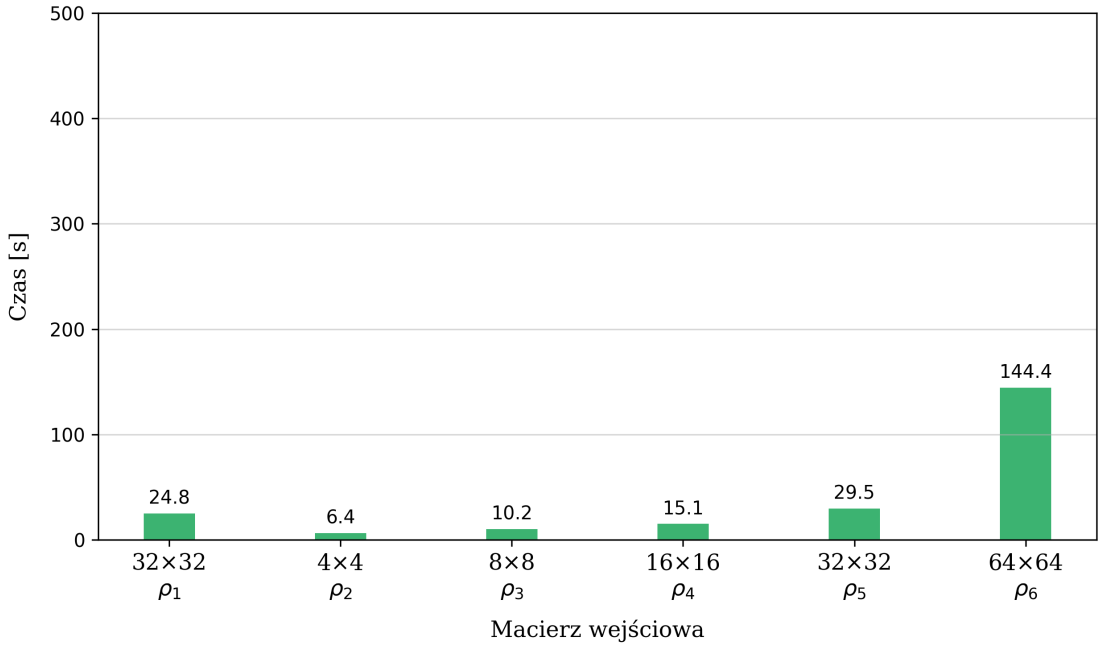
4.4 Pomiary z pojedynczą precyzją

Wszystkie testy wydajności z dla obliczeń wykorzystujących liczby zmiennoprzecinkowe pojedynczej precyzji były przeprowadzane w taki sam sposób jak odpowiadające testy z podwójną precyzją.

4.4.1 Python i NumPy

Na wykresie 11 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) 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[10], więc wymnażanie ich jest procesem bardzo szybkim. W momencie kiedy docieramy do macierzy 32×32 wzrost wydajności staje się zauważalny, 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

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

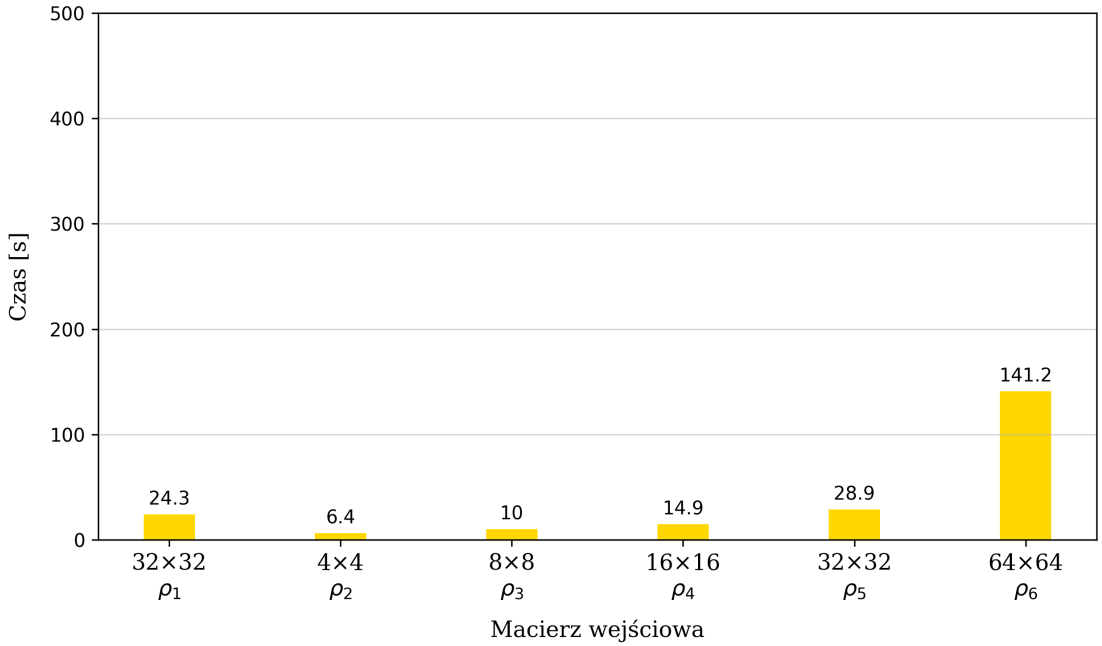


Rysunek 11: Wyniki testów wydajności implementacji w języku Python z użyciem biblioteki NumPy dla macierzy ρ_1 - ρ_6 i obliczeniami pojedynczej precyzji.

również bardzo istotną rolę, szczególnie w przypadku macierzy 64×64, dla których obliczenia przyspieszają znacznie bardziej niż w przypadku mniejszych macierzy.

4.4.2 Python i NumPy z AOT

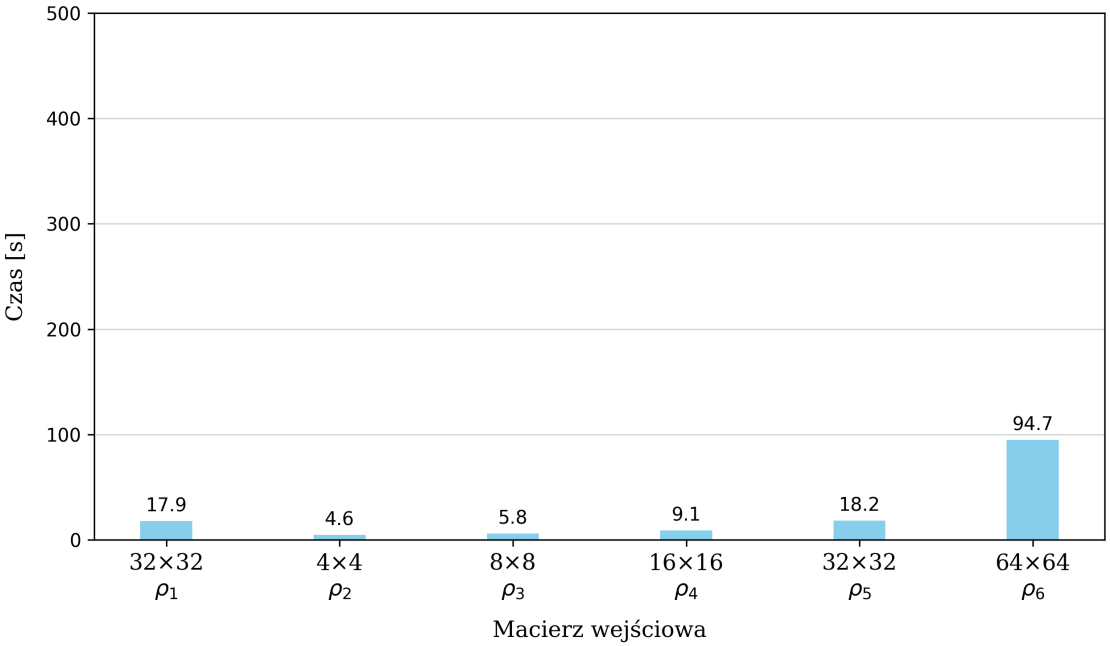
Wyniki dla wariantu pre-kompilowanego przy pomocy biblioteki Cython nie różnią się znacząco od wariantu nie pre-kompilowanego, podobnie jak w przypadku obliczeń podwójnej precyzji, zostały one zaprezentowane na rysunku 12. Podobnie jak w przypadku wyników dla obliczeń podwójnej precyzji, pre-kompilacja nie przynosi istotnych zysków wydajnościowych.



Rysunek 12: Wyniki testów wydajności implementacji w języku Python z użyciem biblioteki NumPy i pakietu Cython do kompilacji AOT dla macierzy ρ_1 - ρ_6 i obliczeniami pojedynczej precyzji.

4.4.3 Python i NumPy z JIT

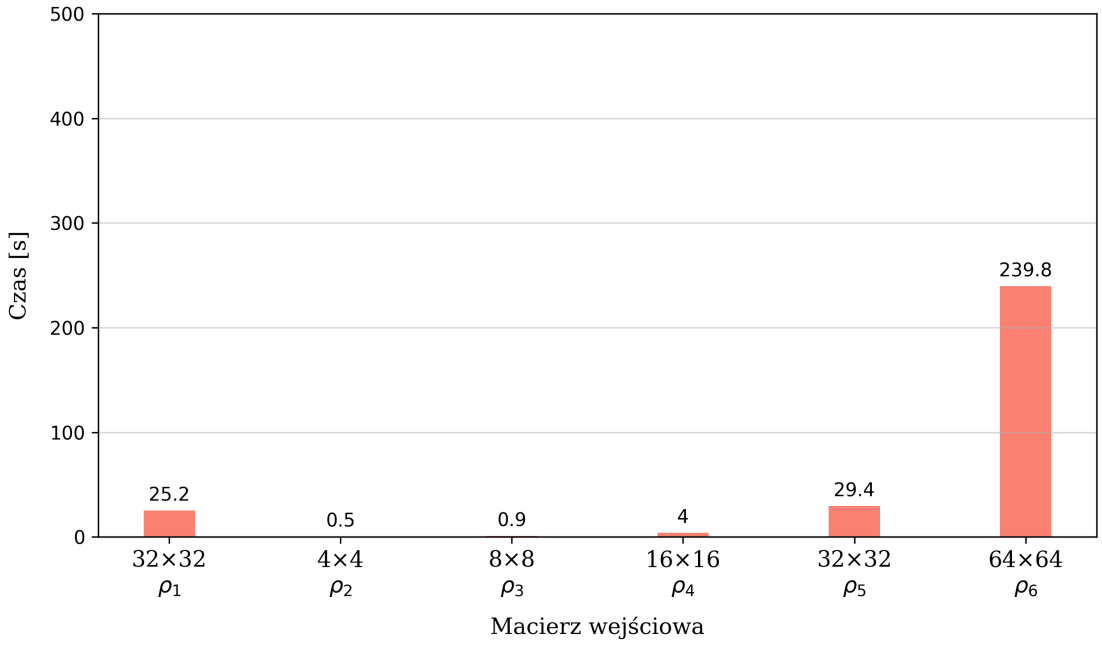
W przypadku wariantu wykorzystującego kompilację JIT, zysk czasowy wynikający z redukcji precyzji obliczeń jest minimalny lub wręcz nie występuje. Wyjątkiem są tutaj wyniki dla macierzy ρ_1 w przypadku której czas pracy skrócił się o $4.7s$ ($\approx 21\%$).



Rysunek 13: Wyniki testów wydajności implementacji w języku Python z użyciem biblioteki NumPy i pakietu Numba do kompilacji JIT dla macierzy ρ_1 - ρ_6 i obliczeniami pojedynczej precyzji.

4.4.4 Rust i Ndaray

Implementacja w języku Rust wykorzystująca bibliotekę Ndarray prezentuje znaczną poprawę wydajności dla wszystkich wymiarów macierzy. Podczas obliczeń na małych macierzach, do 16×16 włącznie, uzyskuje ona najlepsze wyniki w zestawieniu, natomiast dla większych macierzy poprawa występuje, ale ciągle implementacja w języku Python z JIT daje lepsze efekty.



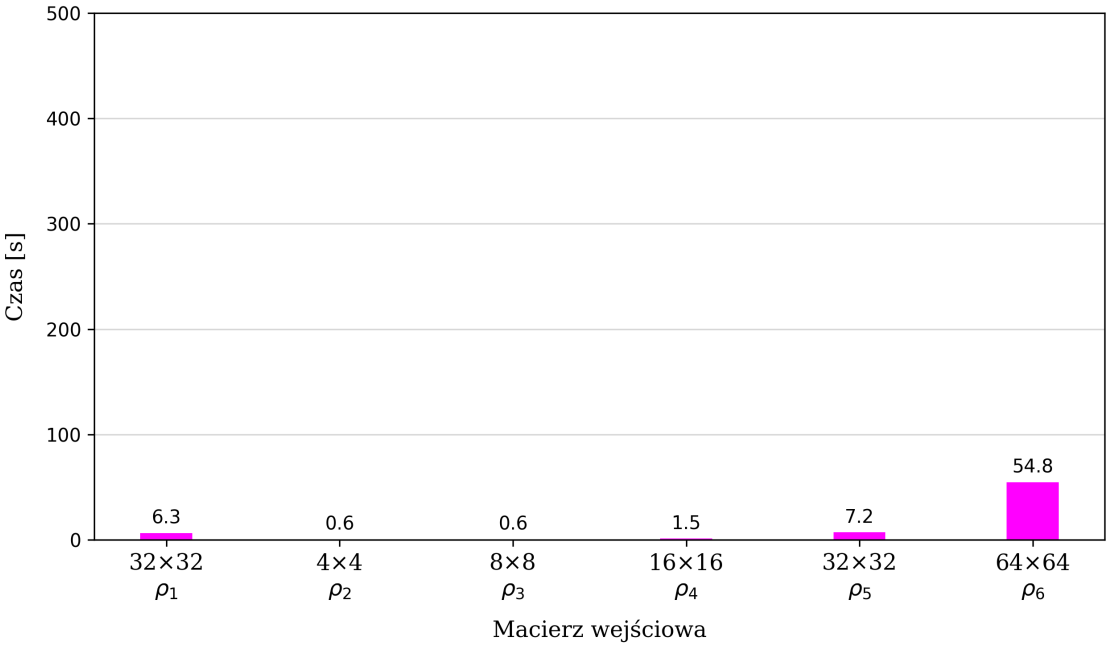
Rysunek 14: Wyniki testów wydajności implementacji w języku Rust z użyciem biblioteki Ndarray dla macierzy ρ_1 - ρ_6 i obliczeniami pojedynczej precyzji.

4.4.5 Rust i Ndaray z OpenBLAS

Zestawienie języka Rust i biblioteki Ndaray z pakietem OpenBLAS i liczbami zmiennoprzecinkowymi pojedynczej precyzji poskutkowało uzyskaniem znaczącej poprawy wyników wydajności, które zostały przedstawione na rysunku 15. W przypadku macierzy 4×4 i 8×8 wydajność jest bardzo zbliżona do wariantu nie korzystającego z OpenBLAS, natomiast wraz ze wzrostem rozmiaru macierzy, skrócenie czasu pracy staje się coraz bardziej widoczne. Względem oryginału, obliczenia dla macierzy:

- ρ_1 trwają $\approx 5.8\times$ krócej,
- ρ_5 trwają $\approx 6.4\times$ krócej,
- ρ_6 trwają $\approx 7.4\times$ krócej,

Biorąc pod zyski wydajności, wynikające z obniżenia precyzji obliczeń, dla pozostałych implementacji, tak istotne skrócenie czasu pracy jest zaskakujące. Z tego względu powtórnie upewniłem się, że praca programu kończy się uzyskaniem odpowiednich wyników liczbowych i nie wykryłem żadnych nieprawidłowości.



Rysunek 15: Wyniki testów wydajności implementacji w języku Rust z użyciem biblioteki Ndarray dla macierzy ρ_1 - ρ_6 i obliczeniami pojedynczej precyzji.

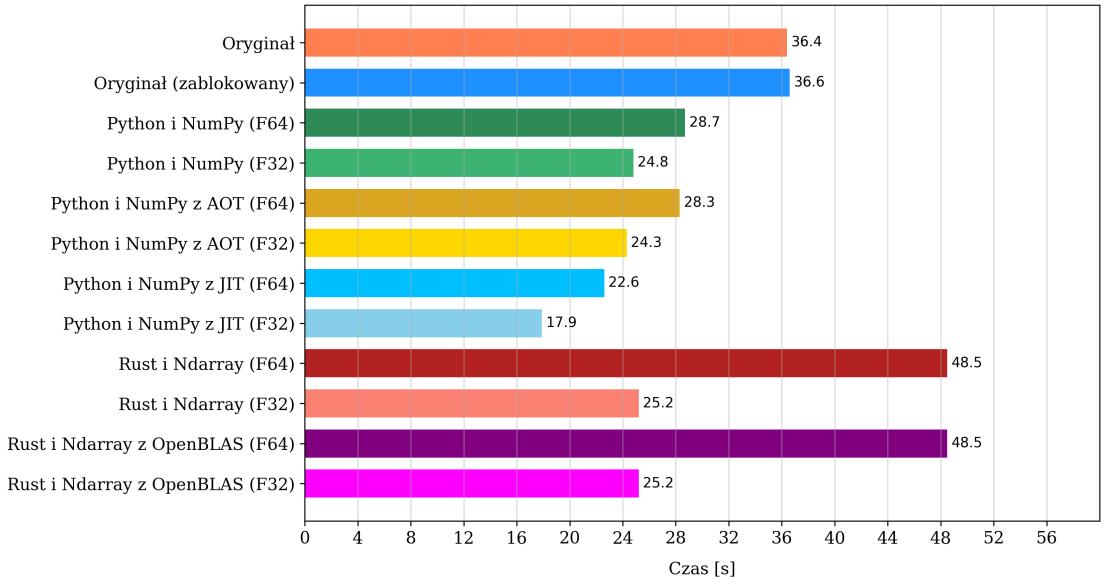
4.5 Zestawienia dla macierzy

W tej sekcji prezentuję wyniki tych samych pomiarów co w sekcjach 4.3 i 4.4 zestawiając je ze sobą względem macierzy wykorzystanej do testów.

Oznaczenie '(F64)' przy nazwie implementacji oznacza obliczenia z podwójną precyzją, analogicznie '(F32)' oznacza obliczenia z pojedynczą precyzją. 'Oryginał' to kod implementacji autorstwa dr hab. Marcin Wieśniak, prof. UG, natomiast pozycja podpisana 'Oryginał (zablokowany)' to ten sam kod ale z zablokowaną ilością wątków obliczeniowych.

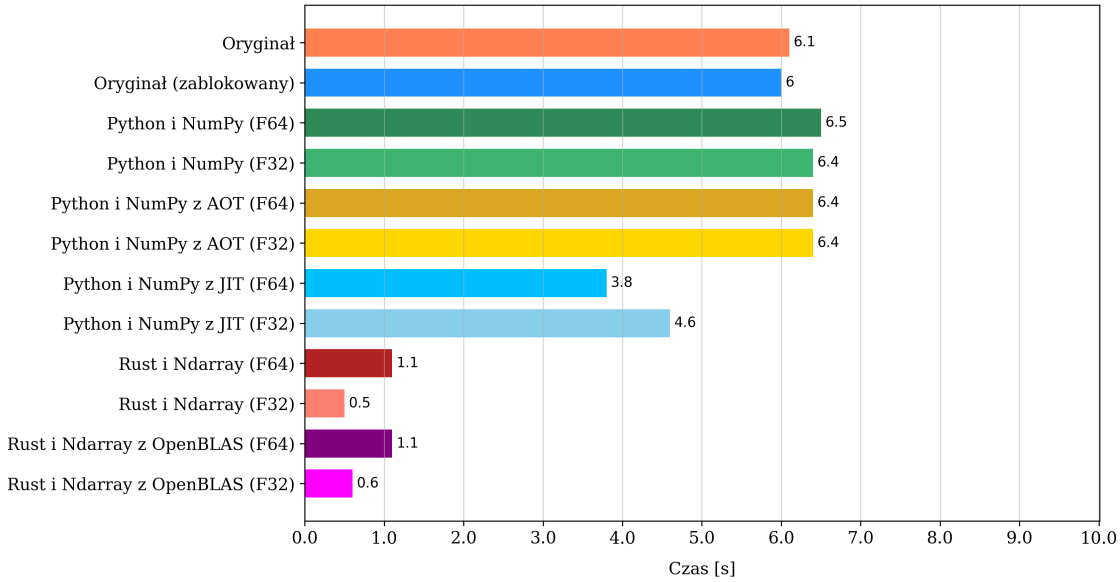
4.5.1 Macierz ρ_1 (32×32)

Na rysunku 16 przedstawione zostało zestawienie wyników wydajności dla testów przeprowadzanych przy pomocy macierzy ρ_1. Najkrótszy czas pracy w zestawieniu, $17.9s$, uzyskała implementacja napisana w języku Python z użyciem biblioteki NumPy z kompilacją JIT wykonująca obliczenia pojedynczej precyzji. Niewiele ustępuje jej wariant pracujący na liczbach podwójnej precyzji z wynikiem $22.6s$.



Rysunek 16: Zestawienie wyników testów wydajności wszystkich implementacji dla macierzy ρ_1.

4.5.2 Macierz ρ_2 (4×4)

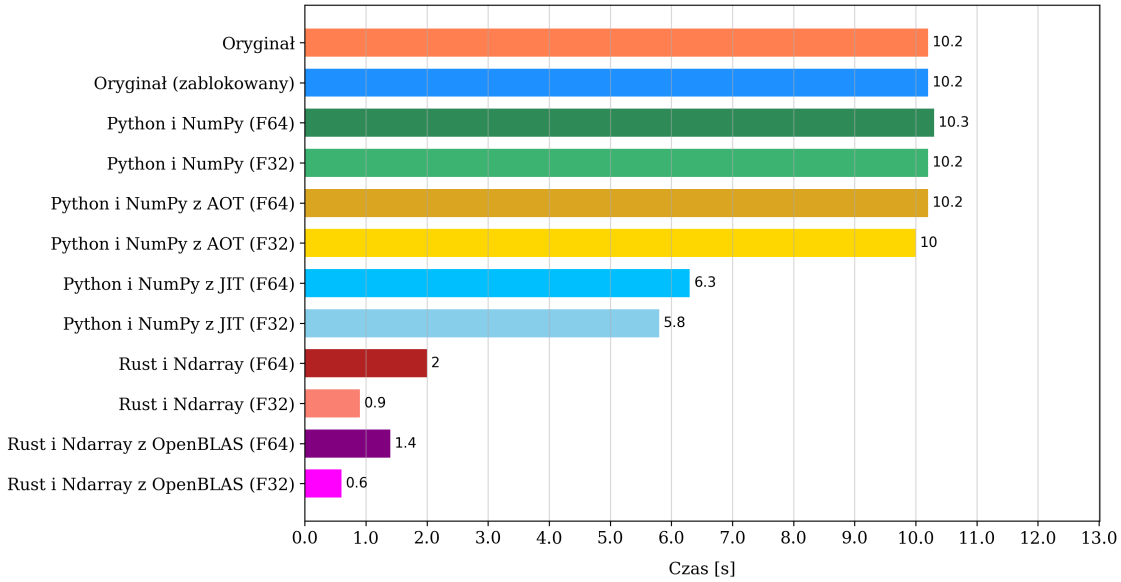


Rysunek 17: Zestawienie wyników testów wydajności wszystkich implementacji dla macierzy ρ_2.

Na rysunku 17 przedstawione zostało zestawienie wyników wydajności dla testów przeprowadzanych przy pomocy macierzy ρ_2. Najkrótszy czas pracy w zestawieniu, $0.5s$, uzyskała implementacja napisana w języku Rust wykonująca obliczenia pojedynczej precyzji. Bardzo zbliżony wynik uzyskał wariant korzystający z biblioteki OpenBLAS$0.6s$.

Z pośród implementacji w języku Python najlepiej sprawował się wariant z JIT (F64), natomiast jego czas pracy był około $7.6\times$ dłuższy.

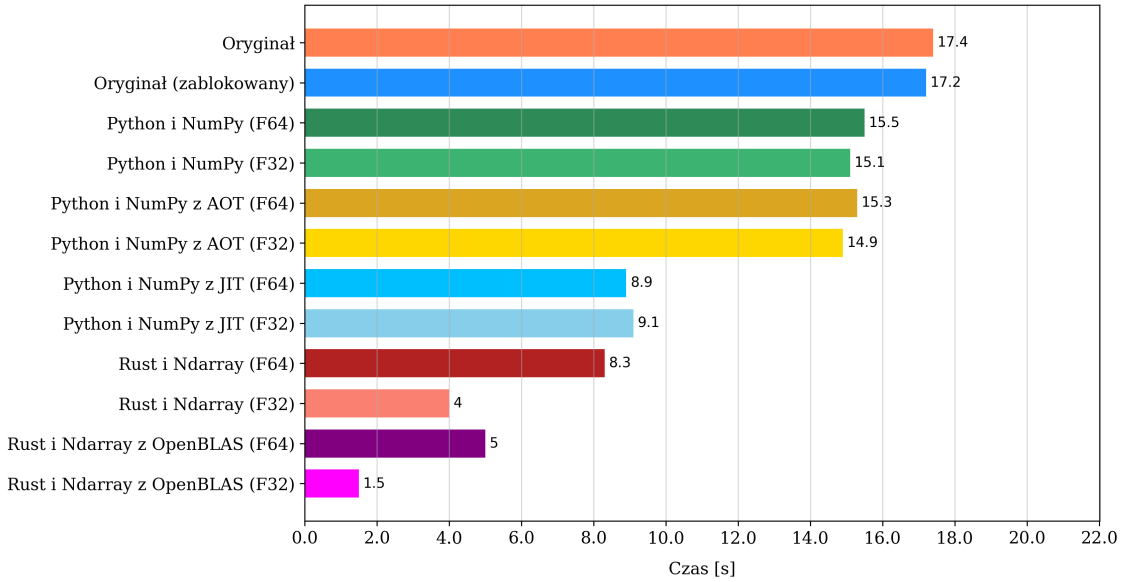
4.5.3 Macierz ρ_3 (8×8)



Rysunek 18: Zestawienie wyników testów wydajności wszystkich implementacji dla macierzy ρ_3.

Na rysunku 18 przedstawione zostało zestawienie wyników wydajności dla testów przeprowadzanych przy pomocy macierzy ρ_3. Najkrótszy czas pracy w zestawieniu, $0.6s$, uzyskała implementacja napisana w języku Rust z OpenBLAS wykonująca obliczenia pojedynczej precyzji. Bardzo zbliżony wynik uzyskał wariant który nie korzystał z OpenBLAS $(0.9s)$.

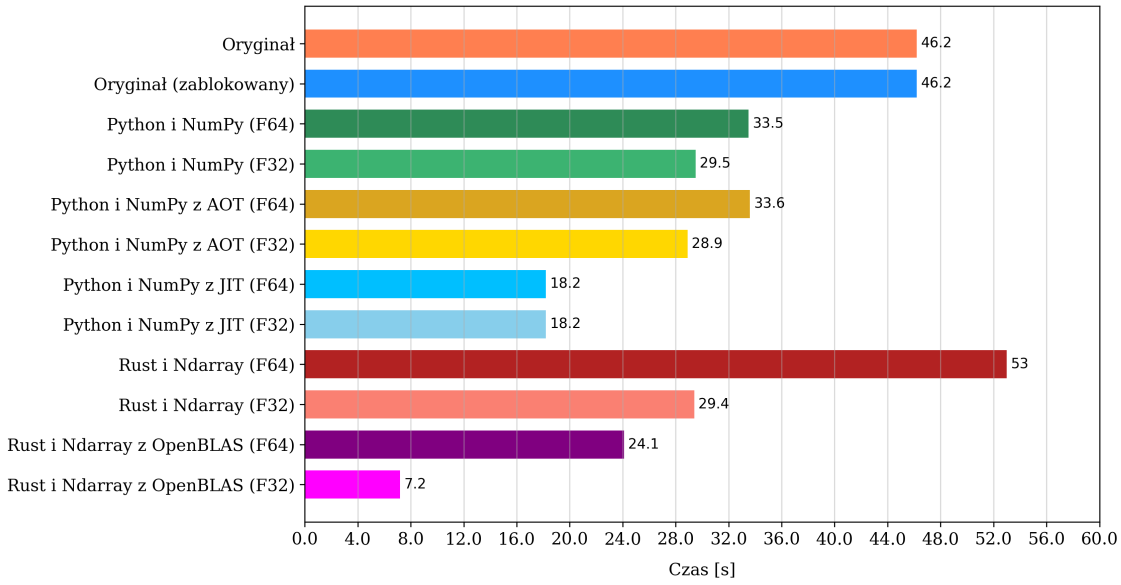
4.5.4 Macierz ρ_4 (16×16)



Rysunek 19: Zestawienie wyników testów wydajności wszystkich implementacji dla macierzy ρ_4.

Na rysunku 19 przedstawione zostało zestawienie wyników wydajności dla testów przeprowadzanych przy pomocy macierzy ρ_4. Najkrótszy czas pracy w zestawieniu, $1.5s$, uzyskała implementacja napisana w języku Rust z OpenBLAS wykonująca obliczenia pojedynczej precyzji. Bardzo zbliżony wynik uzyskał wariant który nie korzystał z OpenBLAS ($4s$).

4.5.5 Macierz ρ_5 (32×32)

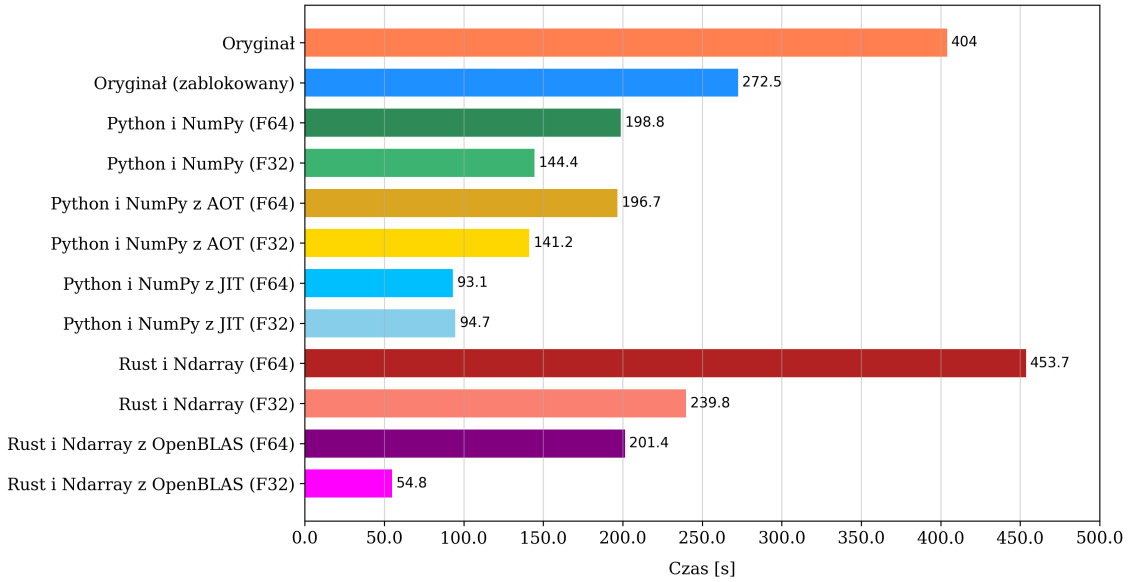


Rysunek 20: Zestawienie wyników testów wydajności wszystkich implementacji dla macierzy ρ_5.

Na rysunku 20 przedstawione zostało zestawienie wyników wydajności dla testów przeprowadzanych przy pomocy macierzy ρ_5. Najkrótszy czas pracy w zestawieniu, $7.2s$, uzyskała implementacja napisana w języku Rust z OpenBLAS wykonująca obliczenia pojedynczej precyzji. Następny najniższy wynik wynosił $18.2s$ i był osiągany przez wariant w języku Python korzystający z JIT, niezależnie od precyzji obliczeń.

4.5.6 Macierz ρ_6 (64×64)

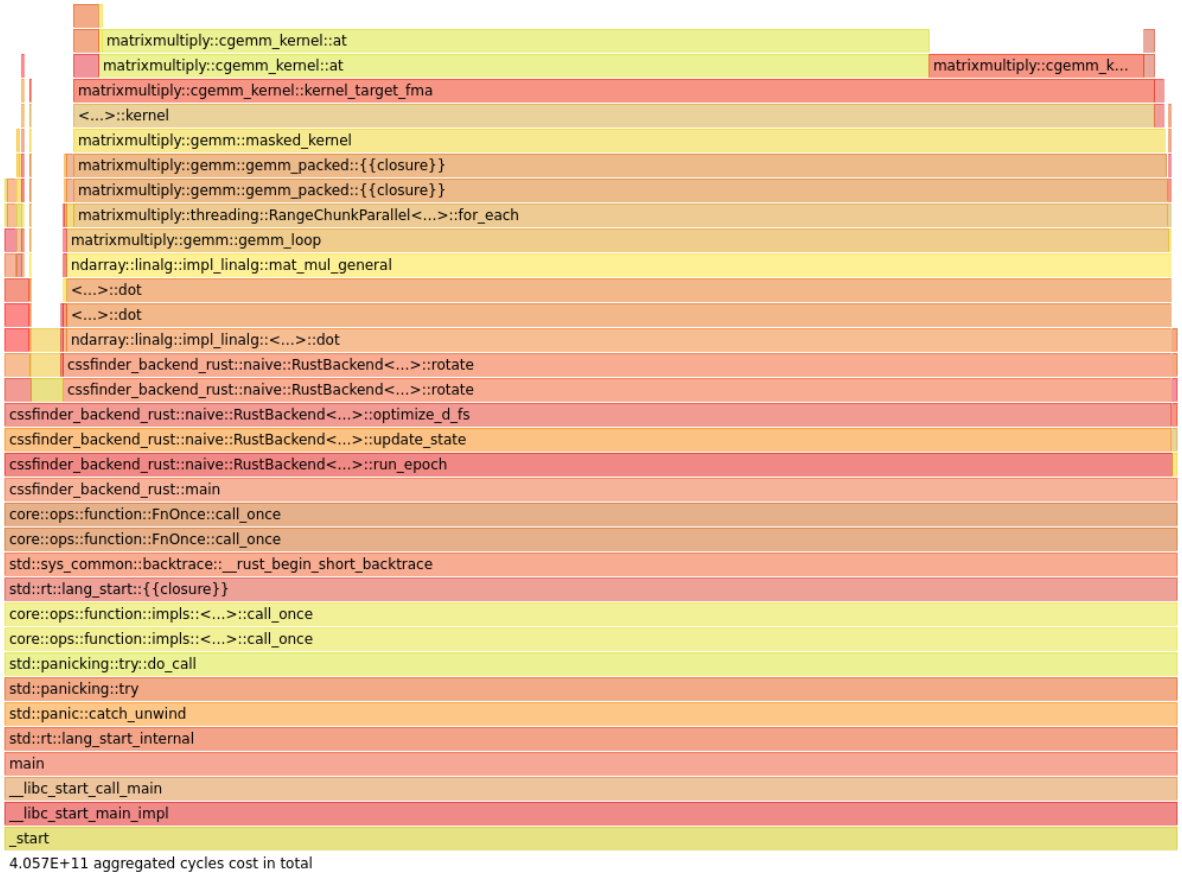
Na rysunku 21 przedstawione zostało zestawienie wyników wydajności dla testów przeprowadzanych przy pomocy macierzy ρ_6. Najkrótszy czas pracy w zestawieniu, $54.8s$, uzyskała implementacja napisana w języku Rust z OpenBLAS wykonująca obliczenia pojedynczej precyzji. Następny najniższy wynik wynosił $93.1s$ i był osiągany przez wariant w języku Python korzystający z JIT przy obliczeniach podwójnej precyzji. Kod wykonujący obliczenia pojedynczej precyzji wypadł nieznacznie gorzej.



Rysunek 21: Zestawienie wyników testów wydajności wszystkich implementacji dla macierzy ρ_6.

4.6 Profilowanie Rust z OpenBLAS

Po przeprowadzeniu testów wydajności uznałem że konieczne jest wykonanie profilowania na najwydajniejszej implementacji, aby sprawdzić czy i gdzie istnieje jeszcze szansa na poprawę. Dlatego też przygotowałem odpowiedni program pomocniczy który wywoływał implementację w języku Rust z OpenBLAS i pojedynczą precyzją bez konieczności angażowania interpretera. Było to rozwiązanie które zarówno pozwalało usunąć duże ilości zbędnych informacji z danych profilowania jak również obejść błędy które otrzymywałem gdy próbowałem profilować zachowanie interpretera.



Rysunek 22: Wizualizacja charakterystyki zachowania implementacji Rust i Ndaray z OpenBLAS podczas obliczeń pojedynczej precyzji.

Wizualizacja typu flame graph wyników tego profilowania została umieszczona na rysunku 22. Obraz został stworzony przy pomocy programu Hotspot[22] rozwijanego przez firmę KDAB.

4.7 Pomiary dla wielu zadań

5 Dyskusja

5.1 Skuteczność

Praca ta miała na celu eksplorację dostępnych metod maksymalizacji wydajności algorytmu QGA. Udało mi się w jej ramach zbadać skuteczność wykorzystania pięciu wariantów programu, w tym implementacji w dwóch różnych językach programowania. Algorytm QGA został z powodzeniem wielokrotnie zaimplementowany, a wszystkie stworzone implementacje pozwalały uzyskać oczekiwane wyniki liczbowe.

Przeprowadziłem również liczne testy wydajności dla różnych danych wejściowych dla wszystkich wariantów programu, co pozwoliło mi ustalić które metody okazały się najbardziej skuteczne. Ponadto każdy z wariantów był testowany zarówno podczas obliczeń pojedynczej jak i podwójnej precyzji, co pozwoliło zbadać wpływ precyzji obliczeń na wydajność kodu. W

przypadku dwóch wariantów programu, udało się się uzyskać znaczącą poprawę wydajności. Są nimi:

1. Python i NumPy z JIT
2. Rust i Ndarray z OpenBLAS

Z zastrzeżeniem, że w przypadku wariantu 1. dotyczy to zarówno obliczeń podwójnej jak i pojedynczej precyzji, natomiast w przypadku 2. znacząca poprawa występowała dla obliczeń pojedynczej precyzji.

Warto podkreślić, że wydajność obu wariantów jest zbliżona, natomiast nakład pracy konieczny do stworzenia implementacji w języku Rust był nieporównywalnie większy niż ten potrzebny do dodania kompilacji JIT do kodu w języku Python. Podobnych wyników można spodziewać sie w przypadku wielu innych programów skupiających się na dużej ilości obliczeń macierzowych - kompilacja JIT jest w stane dość skutecznie usunąć wady języka Python. Wymaga to poświęcenia pewnej części swobody którą ten język daje, ale ciągle pozostawia jej na tyle dużo, że proces pisania kodu jest mniej szybszy niż w przypadku języków niskopoziomowych. Oczywiście, języki te dają większą kontrolę nad komputerem i pozwalają na wiele błyskotliwych manualnych optymalizacji, tak jak ma się to w przypadku biblioteki OpenBLAS.

Kolejnym ważnym spostrzeżeniem jest, że 4 z 5 zaprezentowanych implementacji wykorzystywały do wykonywania mnożenia macierzowego bibliotekę OpenBLAS. Pomimo tego różnice w wydajności pomiędzy nimi są bardzo znaczące, ponieważ znaczącą część czasu wykonywania kodu zajmowały inne operacje.

5.2 Opublikowany kod

Stworzony przeze mnie kod został zamieszczony w trzech repozytoriach w serwisie GitHub:

1. cssfinder[26],
2. cssfinder_backend_numpy[27],
3. cssfinder_backend_rust[28],

Dla każdego z tych repozytorium istnieje odpowiadający pakiet menadżera pakietów pip zamieszczony na serwerze PyPI. Zostały one utworzone w sposób zgodny z ekosystemem języka Python. Pozwala to w bardzo prosty sposób rozpocząć korzystanie z programu poprzez instalację następujących pakietów:

1. cssfinder[29],
2. cssfinder_backend_numpy[30],
3. cssfinder_backend_rust[31],

Pakiety są kompatybilne z systemami Linux, MacOS i Windows. Wymagają interpretera języka Python w wersji 3.8 lub nowszej.

Odwołania

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

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

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

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

[5] KR Srinath. „Python–the fastest growing programming language". W: *International Research Journal of Engineering and Technology* 4.12 (2017), s. 354–357.

[6] 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).

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

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

[9] 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).

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

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

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

[13] 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).

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

[15] 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).

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

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

[18] 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.

[19] 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.

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

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

[22] 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).

[23] 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.

[24] 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).

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

[26] Krzysztof Wiśniewski. *CSSFinder (Core)*. 2023. URL: https://github.com/Argmaster/CSSFinder (term. wiz. 09. 06. 2023).

[27] Krzysztof Wiśniewski. *CSSFinder Numpy Backend*. 2023. URL: https://github.com/Argmaster/cssfinder_backend_numpy (term. wiz. 09. 06. 2023).

[28] Krzysztof Wiśniewski. *CSSFinder Rust Backend*. 2023. URL: https://github.com/Argmaster/cssfinder_backend_rust (term. wiz. 09. 06. 2023).

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

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

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