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

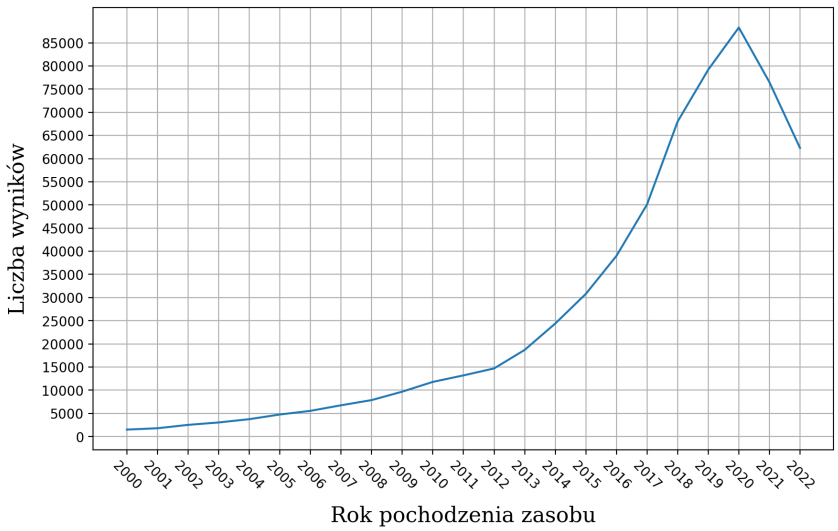
Streszczenie

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[8][12], Numba[5][7], Cython[3][16] oraz re-implementacja w języku Rust[29] 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[13], w którym Python w roku 2022 zajął 2 miejsce, czy też zestawienie TIOBE Index[31], 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[26], C++[27], Rust[28]). Jednak, podejmując odpowiednie wysiłki, możliwe jest aby programy, których kluczowa logika została napisana w języku Python, zbliżały się 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

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

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[23] w serwisie GitHub [35][36][38]. W skutek prac projektowych utworzona została również grupa publicznych pakietów, które można pobrać z serwisu PyPI:

- cssfinder[34]
- cssfinder_backend_numpy[37]
- cssfinder_backend_rust[39]

Zainstalowanie ich jest możliwe przy pomocy menadżera pakietów języka Python, np. `pip`[19]. 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 Pochodzenie programu

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[2] który służy do znajdowania odległości pomiędzy punktem, a zbiorem wypukłym. Oprogramowanie jest przeznaczone do detekcji splątania kwantowego[9][11][10] poprzez analizę macierzy gęstości opisujących układy kubitów[2]. Algorytm ten wielokrotnie, z sukcesem, był wykorzystany do analizy problemów z dziedziny fizyki kwantowej[9] przy okazji również pokonując rozwiązania bazujące na uczeniu maszynowym[14].

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. 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.

[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
[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 Środowisko testowe

Wszystkie wyniki wydajności uzyskane zostały przy użyciu następującego środowiska testowego:

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
Rust	1.68.2 64-bit
CPU	AMD Ryzen 9 7950X
RAM	64GB DDR5 5600MHz CL40
DRIVE	512GB SSD GOODRAM CX400 (SATA)

2.2 Wstępne profilowanie

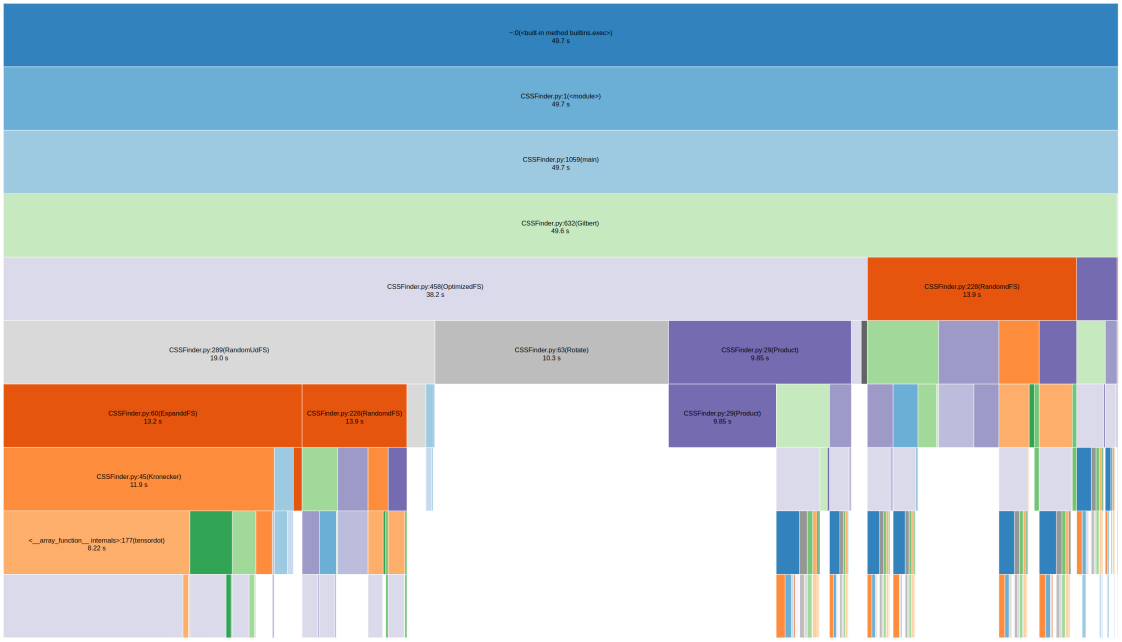
Prace nad optymalizacją kodu rozpocząłem od wstępnego profilowania pracy programu w trybie 1 (full separability of an n-quDit state) przekazując układ 5 kubitów opisany macierzą ρ_0 (Macierz 32×32 podwójnej precyzji zmiennoprzecinkowych liczb zespolonych).

$$
\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 2: Macierz ρ_0.

Program wykonywał proces analizy stanu aż do uzyskania 1000 korekcji. Do zbierania danych o pracy programy wykorzystałem moduł z biblioteki standardowej języka Python, `cProfile`. Przy analizie tak uzyskanych wyników posiłkowałem się wizualizacjami wykonanymi przy pomocy biblioteki snakeviz[17]. Podczas testów, program wykorzystywał domyślny globalny generator liczb losowych biblioteki NumPy (PCG64[18]) z ziarnem ustawionym na wartość 0.

Rysunek 3 przedstawia diagram udziału 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



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

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 1: 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` - `tottime` 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%[7]) 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[8], nie jest więc konieczne sięganie po rozwiązania takie jak asyncio czy wielowątkowość.

2.3 Wstępne pomiary wydajności

Aby uzyskać dobrą bazę porównawczą, wykonałem serię pomiarów czasu pracy programu na o różnych wymiarach macierzach gęstości. Reprezentowały one układy od 2 do 6 kubitów i przyjmowały rozmiary od 4×4 do 64×64. Przyjmowały one następującą postać:

$$\rho_n = \begin{bmatrix} 0.5 & 0 & \dots & 0 & 0.5 \\ 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \\ 0.5 & 0 & \dots & 0 & 0.5 \end{bmatrix}$$

W dalszej części pracy wielokrotnie będę wykorzystywał tę grupę macierzy do testów wydajności. W tekście macierze te będą oznaczane jako ρ_2 do ρ_6, w zależności od reprezentowanej ilości kubitów[9].

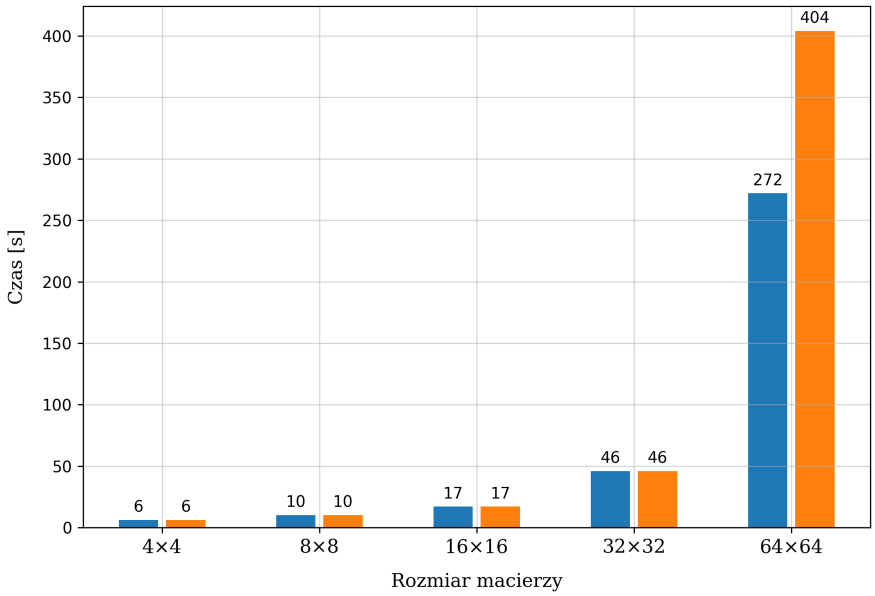
Dane przekazywałem kolejno do programu z poleceniem działania w trybie 1 (full separability of an n-quDit state) do osiągnięcia 1000 korekcji lub do 2.000.000 iteracji algorytmu, w zależności

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

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

[9]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.

od tego co nastąpi szybciej. Dla wszystkich macierzy algorytm uzyskał 1000 korekcji i w żadnym przypadku nie osiągnął maksymalnej ilości 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[10].



Rysunek 4: Wyniki wstępnych testów wydajności oryginalnego kodu z zablokowaną (niebieski) i odblokowaną (pomarańczowy) liczbą wątków obliczeniowych 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 4 czasy obliczeń dla różnych rozmiarów macierzy z domyślnym zachowaniem biblioteki przedstawiają kolumny pomarańczowe. Jeśli przy pomocy zmiennych środowiskowych ustawimy ilość wątków wykorzystywanych do obliczeń na 1, co przedstawiają kolumny niebieskie, uzyskujemy znaczące skrócenie czasu obliczeń dla macierzy 64×64. 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. W dalszej części pracy będę używał wariant z zablokowaną ilością wątków jako bazę porównawczą, ponieważ wyraźnie odstająca wartość czasu dla macierzy 64×64 powinna być postrzegana jako artefakt, który nie jest dobrym wyznacznikiem faktycznej wydajności.

2.4 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

[10]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.

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.

2.5 Dostępne narzędzia

2.5.1 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`[21]. 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 [22]. 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`[20] lub `GO` z użyciem biblioteki `gopy`[15].

3. Wykorzystując bibliotekę `Cython`[16][3]. 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`[32]. Ta, podobnie do biblioteki Cython, również zawiera transpilator, natomiast zamiast korzystać z dedykowanego języka , bazuje on na dodanych w Pythonie 3.5[4] (PEP 484[25] i PEP 483[24]), adnotacjach typów. Jest on rozwijany obok projektu mypy - pakietu do statycznej analizy typów dla języka Python, również opartej na adnotacjach typów[30].

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[11] biblioteka CuPy nie wzięta pod uwagę.

[11] CuPy jest odpowiednikiem NumPy który wykorzystuje do obliczeń GPU. Z tego względu radzi sobie wyśmienicie

2.5.2 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[33]. 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[1]. Niestety, posiada ona zasadniczą wadę - jej interfejs binarny[12] oraz programistyczny[13] 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[5][7]. Ona, w przeciwieństwie do PyPy, wymaga aby fragmenty kodu, które mają być skompilowane, miały postać funkcji oznaczonych dedykowanymi dekoratorami[14]. Została ona również zaprojektowana aby dobrze współgrać z biblioteką NumPy. Jej zastosowanie z założenia ma generować wzrost wydajności nawet w 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.

z operacjami na dużych macierzach, natomiast najprawdopodobniej macierze tutaj rozważane są zbyt małe aby uzyskać wzrost wydajności[6]. Jednocześnie pomimo podobieństwa do NumPy, biblioteka ta różni się i posiada problematyczne zależności (CUDA) co czyni adaptację kodu czasochłonną.

[12] ang. ABI - Application Binary Interface

[13] ang. API - Application Programming Interface.

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

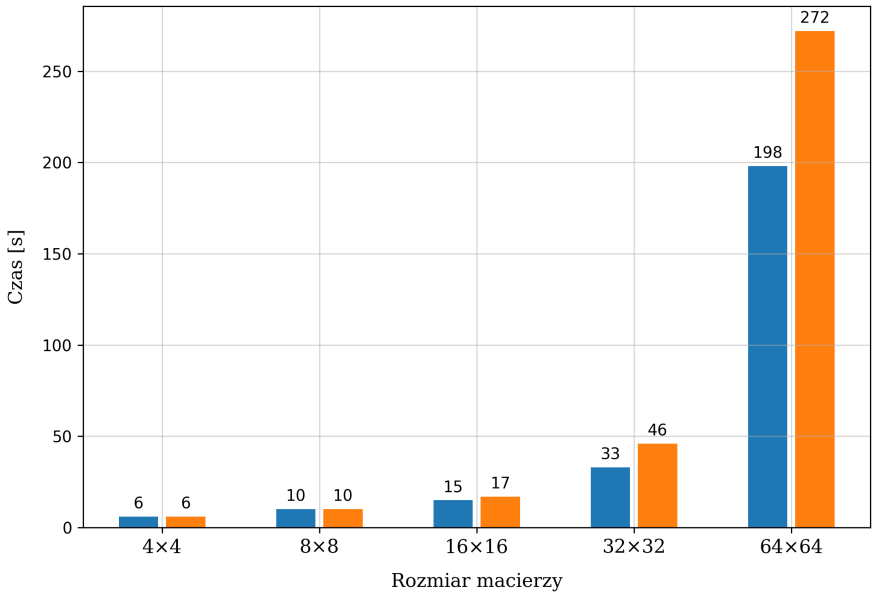
2.6 Re-implementacje

2.6.1 Python i NumPy

Pierwsza wykonana przeze mnie re-implementacja algorytmu wykorzystywała ten sam zestaw narzędzi co oryginalny kod. 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 łączenia części głównej programu z implementacją. Sam algorytm pozostał niezmieniony.

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 ilości 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 ilością wątków obliczeniowych. Pomiary czasu pracy dotyczyły przede wszystkim samego algorytmu[16].

Wyniki czasu pracy implementacji oryginalnej pochodzą z wcześniejszych pomiarów.

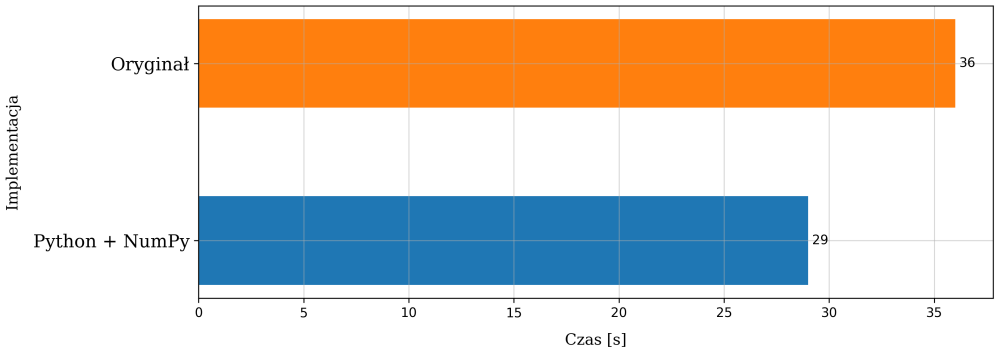


Rysunek 5: Wyniki testów wydajności alternatywnej implementacji Python i NumPy (niebieski) w porównaniu do implementacji oryginalnej (pomarańczowy) dla macierzy ρ_2 - ρ_6.

W testach uzyskałem blisko 30% redukcję długości czasu pracy względem oryginalnego kodu. Na rysunku 5 kolorem pomarańczowym oznaczone zostały czasy pracy oryginalnego programu, w zależności od rozmiaru macierzy gęstości. Niebieskie kolumny to czasy pracy alternatywnej implementacji. Warto tutaj zauważyć że to, o ile wzrośnie wydajność, w dużej mierze zależy od danych wejściowych, ponieważ inny zestaw macierzy gęstości może spowodować, że inne fragmenty kodu będą podlegały szczególnemu obciążeniu.

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

[16]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 6: Wyniki testów wydajności alternatywnej implementacji Python i NumPy w porównaniu do implementacji oryginalnej dla macierzy ρ_7.

Przekazując do programu macierz ρ_0, przedstawioną na rysunku 2, uzyskałem czasy pracy różniące się od tych wykorzystujących macierze ρ_2 - ρ_6. Wyniki te zostały przedstawione na rysunku 6, gdzie kolorem pomarańczowym oznaczone są wyniki oryginalnej implementacji, natomiast kolorem niebieskim, wyniki nowo utworzonego kodu. Na wykresie widoczna jest około 20% redukcja długości czasu pracy przy takich samych danych wejściowych.

Ponieważ macierz ρ_0 występuje tylko w jednym rozmiarze, nie jest możliwe aby uzyskać tak szerokie spektrum wyników jak w przypadku macierzy ρ_2 - ρ_6. Z tego względu macierze ρ_2 - ρ_6 stanowią wygodny zestaw danych do weryfikacji ogólnej charakterystyki zachowania alternatywnych implementacji algorytmu i będą dalej wykorzystywane w testach wydajności.

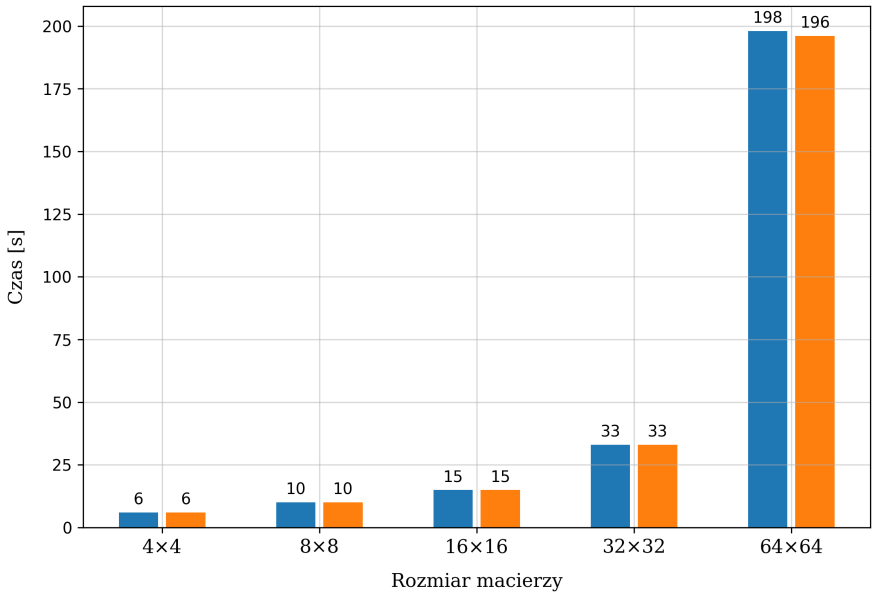
2.6.2 Python i NumPy z AOT

Następnym wykonanym przeze mnie krokiem było skompilowanie mojej implementacji bazującej na NumPy przy pomocy biblioteki Cython. Kod kompilowany w ten sposób nie musi być koniecznie 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 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 ilości 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 ilością wątków obliczeniowych. Pomiary czasu pracy dotyczyły przede wszystkim samego algorytmu[17].

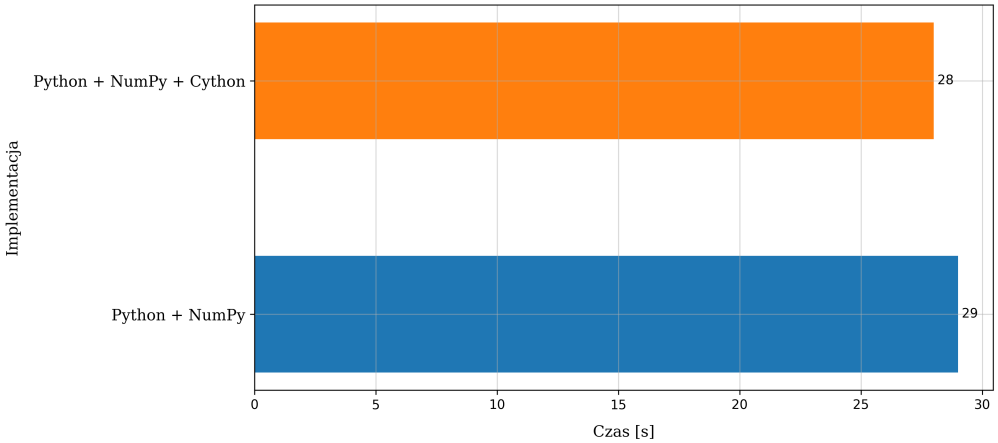
Wyniki czasu pracy implementacji Python i NumPy pochodzą z wcześniejszych pomiarów.

[17]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 7: Wyniki testów wydajności implementacji Python i NumPy w porównaniu do implementacji Python i NumPy z AOT dla macierzy ρ_2 - ρ_6.

Na rysunku 7 kolorem pomarańczowym oznaczone zostały czasy pracy implementacji z AOT, w zależności od rozmiaru macierzy gęstości. Niebieskie kolumny to czasy pracy alternatywnej implementacji bez AOT. Wszystkie kolumny reprezentują bardzo zbliżone wyniki czasu pracy, a ewentualne rozbieżności występują tylko w przypadku analizy największych macierzy. Dzieje się tak najprawdopodobniej dlatego że większość wykonywanych obliczeń odbywa się wewnątrz biblioteki NumPy, która nie uzyskuje żadnych znaczących korzyści z redukcji czasu pracy pozostałej części kodu.



Rysunek 8: Wyniki testów wydajności implementacji Python i NumPy Z AOT w porównaniu do implementacji bez AOT dla macierzy ρ_0.

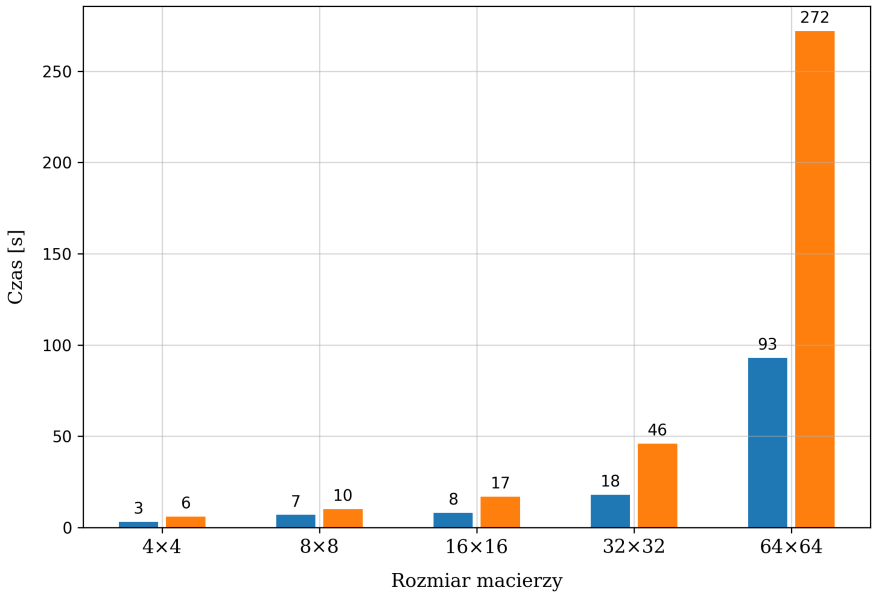
Wyniki dla testów wykonanych z użyciem macierzy ρ_0 widoczne na rysunku 8 prowadzą do analogicznych wniosków co w przypadku macierzy ρ_2 - ρ_6 - pre-kompilacja całości z wykorzystaniem biblioteki Cython nie poprawia istotnie wydajności rozważanego kodu.

2.6.3 Python i NumPy z JIT

Ostatnia re-implementacja bazująca na bibliotece NumPy dodatkowo korzysta z kompilacji JIT. Pakiet Numba, który to realizuje kompilację 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[18]. Drugi polega na śledzeniu typów wejściowych i wyjściowych funkcji i automatycznie kompiluje funkcję dla tych typów danych które są odpowiednio często używane używane[19].

Dodatkowo, Numba oferuje dwa tryby kompilacji, wybierane przy pomocy flagi `nopython` przekazywanej do dekoratora `@numba.jit`. 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 dalej odwołuje się do tej samej implementacji co przed skompilowaniem (mnożenie tensorowe w bibliotece NumPy jest z założenia skompilowane do kodu maszynowego). Niestety, brak możliwości skompilowania funkcji używającej `tensordot` powoduje również brak możliwości skompilowania funkcji wyżej w drzewie wywołań. Powoduje to że znacząca część implementacji używającej JIT musi używać trybu obiektowego.

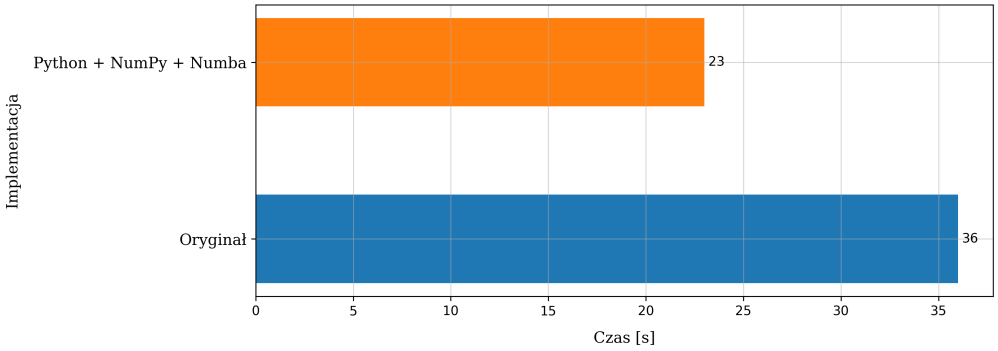
Pomiary czasu pracy były wykonywane przy użyciu macierzy ρ_2 - ρ_6. Dane przekazywałem kolejno do programu z poleceniem działania w trybie FSnQd do osiągnięcia co najmniej 1000 korekcji lub do 2.000.000 iteracji algorytmu, w zależności od tego co nastąpi szybciej. Dla wszystkich macierzy algorytm uzyskał co najmniej 1000 korekcji i w żadnym przypadku nie osiągnął maksymalnej ilości 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 ilością wątków obliczeniowych. Pomiary czasu pracy dotyczyły przede wszystkim samego algorytmu[20].



Rysunek 9: Wyniki testów wydajności implementacji Python i NumPy z Numba JIT w porównaniu do implementacji oryginalnej dla macierzy ρ_2 - ρ_6 .

[18] ang. eager (compilation) - niecierpliwa (kompilacja).
[19] ang. lazy (compilation) - leniwa (kompilacja).
[20] Nie biorą więc pod uwagę czasu pochłoniętego przez importowanie modułów itp., natomiast operacje wczytywania danych i pisania do plików są wliczane w czas pracy.

Na rysunku 7 kolorem niebieski oznaczone zostały czasy pracy implementacji z JIT, w zależności od rozmiaru macierzy gęstości. Pomarańczowe kolumny to czasy pracy oryginalnej implementacji. Kod który wykorzystywał kompilację JIT oferował podczas testów prawie trzykrotnie 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. Dodatkowo kompilator może brać poprawkę na inne zachowania charakterystyczne dla konkretnych architektur. Te dodatkowe informacje i możliwość dodatkowej specjalizacji kodu czynią kompilację JIT bardzo potężnym narzędziem



Rysunek 10: Wyniki testów wydajności implementacji Python i NumPy z Numba JIT w porównaniu do implementacji oryginalnej dla macierzy ρ_0.

W przypadku macierzy ρ_0 charakterystyka pracy algorytmu zmienia się, co skutkuje zmniejszeniem się różnicy pomiędzy czasem pracy wersji z JIT i bez JIT. Jednak dalej wariant z JIT oferuje około 36% krótszy czas pracy.

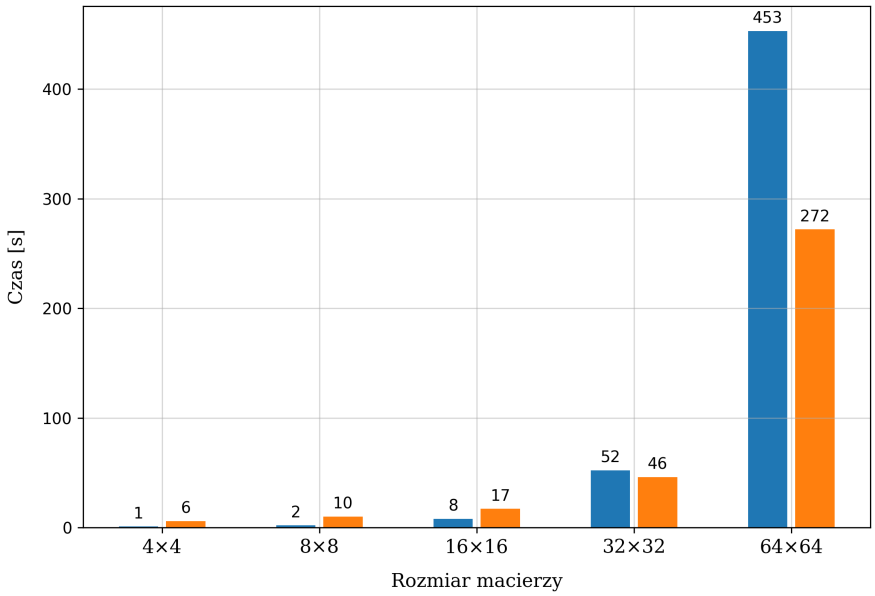
2.6.4 Rust + Ndaray

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 skompletowania systemu budowania, często opierającego się na rozwiązaniach podmiotów trzecich. Ponadto konkurencja nie posiada ujednoliconego standardu pozwalającego na łatwe uzyskanie odstępu do bibliotek otwartoźródłowych, Rust natomiast taki system posiada. Najbardziej dotkliwym problemem jest fakt, że inne języki niskopoziomowe (C/C++) wymagają manualnego zarządzania pamięcią, co nie jest konieczne w języku Rust. Dodatkowo do implementacji algorytmu wykorzystałem szereg otwartoźródłowych bibliotek, których wykaz dostępny jest w pliku `Cargo.toml` załączonym do pracy.

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 ilości iteracji. Dla każdej macierzy pomiar był powtarzany pięciokrotnie a wyniki zostały uśrednione. Pomiary czasu pracy dotyczyły przede wszystkim samego algorytmu[21].

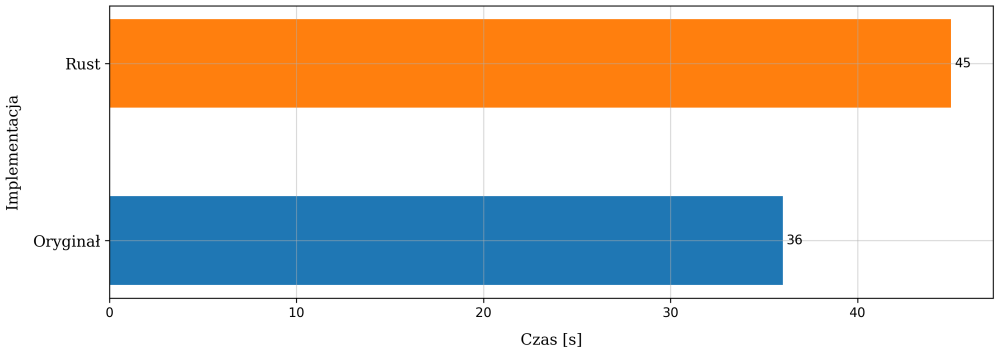
Wyniki czasu pracy wersji oryginalnej pochodzą z wcześniejszych pomiarów.

[21] 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 i implementacji oryginalnej dla macierzy ρ_2 - ρ_6.

Jak zostało zaprezentowane na rysunku 11, implementacja w języku Rust oferuje krótszy czas pracy dla mniejszych macierzy i dłuższy dla większych. Taka specyfika czyni tą implementację przydatną podczas obróbki małych macierzy, bo wzrost wydajności w takim wypadku jest kilkukrotny.



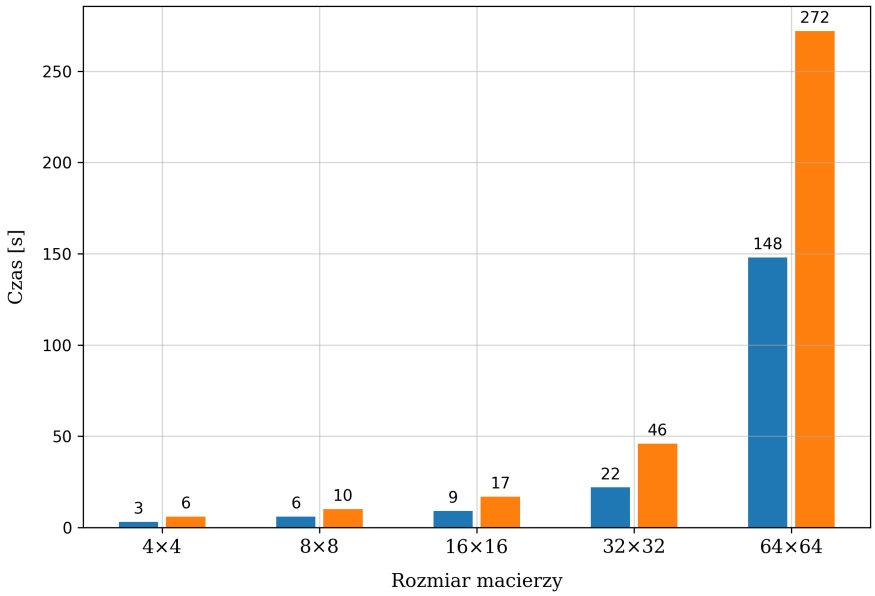
Rysunek 12: Wyniki testów wydajności implementacji w języku Rust i implementacji oryginalnej dla macierzy ρ_0.

Testy na macierzy ρ_0 których wyniki są widoczne na rysunku 12 podtrzymują uprzednio wyciągnięte wnioski.

2.6.5 Rust + Ndaray + OpenBLAS

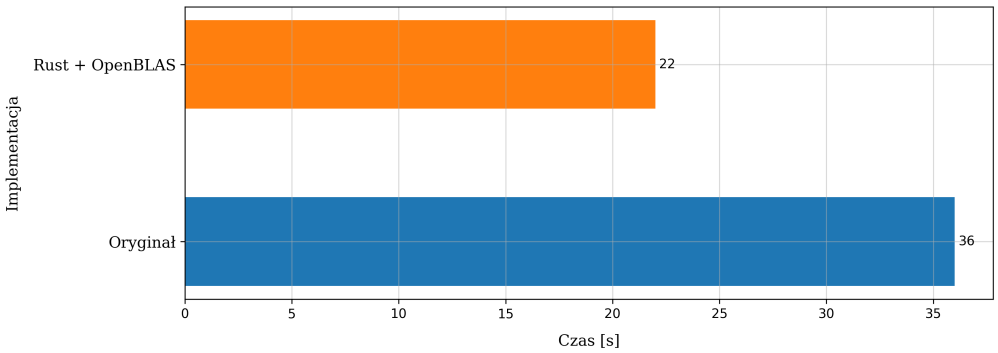
Biblioteka ndarray, która jest sercem implementacji w języku Rust, posiada przełącznik funkcjonalności[22] który pozwala wykorzystać funkcje zawarte w bibliotece OpenBLAS jako implementację mnożenia macierzowego. O ile kompilacja dla wszystkich platform które ma wspierać CSSFinder (Windows, Linux i MacOS) jest poza moim zasięgiem, to uznałem, że warto zweryfikować jakie efekty daje wykorzystanie tej możliwości.

[22] ang. feature switch



Rysunek 13: Wyniki testów wydajności implementacji w języku Rust z użyciem OpenBLAS i implementacji oryginalnej dla macierzy ρ_2 - ρ_6.

Wykorzystanie biblioteki OpenBLAS poskutkowało znaczącym wzrostem wydajności, przekraczającym możliwości oryginalnej implementacji. Wyniki te zostały przedstawione na rysunku 13, gdzie kolumny pomarańczowe to wyniki oryginalnego kodu, a niebieskie to wyniki implementacji w języku Rust, korzystającej z biblioteki OpenBLAS.



Rysunek 14: Wyniki testów wydajności implementacji w języku Rust z użyciem OpenBLAS i implementacji oryginalnej dla macierzy ρ_0.

2.7 Precyzja obliczeń

Oryginalny program posługiwał się liczbami zespolonymi stworzonymi na bazie zmiennoprzecinkowych podwójnej precyzji. Jednak w przypadku analizowania niewielkich macierzy, dane utrzymują zakres wartości blisko 0, do efektywnych obliczeń nie jest konieczna podwójna precyzja. Podstawową zaletą tej zmiany jest zmniejszenie rozmiaru macierzy w pamięci, a to w pozwala na umieszczenie większej części macierzy w pamięci podręcznej procesora. Dodatkowo zwiększa to przepustowość wektoryzowanego kodu (Zestawy instrukcji FMA używają rejestrów o rozmiarach 128 i 256 bit, można więc w jednym takim rejestrze umieścić dwukrotnie więcej 32 bitowych liczb pojedynczej precyzji niż 64 bitowych podwójnej).

Odwołania

[1] 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.

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

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

[4] 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).

[5] 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`.

[6] 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.

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

[8] 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`.

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

[10] 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.

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

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

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

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

[15] 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).

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

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

[18] 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).

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

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

[21] 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).

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

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

[24] 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).

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

[26] 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).

[27] 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).

[28] 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).

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

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

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

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

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

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

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

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

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

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

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