UNIWERSYTET GDAŃSKI
WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI

Krzysztof Wiśniewski
numer albumu: 274276

Kierunek studiów: Bioinformatyka
Specjalność: Ogólna

Optymalizacja oprogramowania w języku Python do analizy stanów kwantowych.

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

Gdańsk 2023

Spis treści

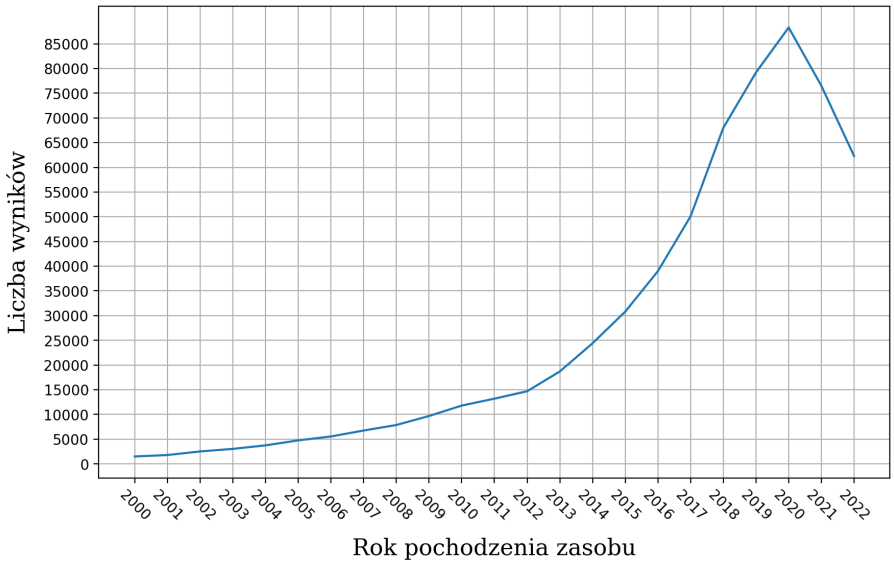
Streszczenie

W tej pracy przeprowadzam analizę efektywności metod optymalizacji, która koncentruje się na minimalizacji czasu wykonania, oprogramowania napisanego w języku Python[24][4], skupiającego się na arytmetyce macierzowej, na przypadku programu CSSFinder służącego do analizy stanów kwantowych pod kątem detekcji splątania kwantowego. Pośród rozważanych metod obecna będzie standardowa implementacja w języku Python z wykorzystaniem biblioteki NumPy[8][12], wersja wzbogacona o kompilację JIT przy pomocy biblioteki Numba[6][7], wersja skompilowana do kodu maszynowego przy pomocy biblioteki Cython[3][16] i kompilatora GCC[25] oraz implementacja w języku Rust[32], również skompilowana do kodu maszynowego.

1 Wstęp

1.1 Dlaczego Python?

Język Python zachęca użytkowników prostotą składni, łatwością tworzenia kodu, dynamicznym systemem typów, automatycznym zarządzanie pamięcią, mnogością dostępnych bibliotek otwartoźródłowych, oraz rozbudowaną społecznością programistów. Na przestrzeni ostatnich 20 lat język 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[34], uznające ten język za obecnie najbardziej rozpowszechniony pośród doświadczonych programistów (Maj 2023).



Rysunek 1: Ilość wyników zwróconych przez wyszukiwarkę Google Scholar dla zapytania 'python language' z podziałem na rok wydania.

Niestety, interpretowany kod, napisany w Pythonie, pomimo licznych zalet, posiada również dotkliwą wadę - pod względem wydajności znacząco odstaje od kompilowanych języków programowania (C[29], C++[30], Rust[31]). Natomiast, dzięki nakładowi pracy wielu zespołów programistów, obecnie istnieją metody pozwalające na obejście tej niedogodności.

1.2 Cel pracy

Praca ta ma na celu weryfikację efektywności wybranych metod poprawy czasu wykonania oprogramowania CSSFinder, zaimplementowanego w języku Python. W dalszej jej części opiszę specyfikę poszczególnych metod optymalizacji, w jaki sposób zmieniają wydajność programu oraz spróbuję wskazać prawdopodobne powody dla których niektóre z uzyskanych wyników konsekwentnie odstają od oczekiwań które można mieć wobec wykorzystanych narzędzi.

Do przeprowadzenia takich analiz konieczne było wielokrotne ponowne implementowanie algorytmu. Funkcjonalny kod opisywany w tej pracy dostępny jest w repozytoriach Gita[26] w serwisie GitHub [38][39][41]. W skutek prac projektowych utworzona została również grupa publicznych pakietów, które można pobrać z serwisu PyPI:

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

Zainstalowanie ich jest możliwe przy pomocy menadżera pakietów języka Python[23], 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 bazuje na algorytmie zaproponowanym przez E. Gilberta[2] pozwalającym na znalezienie odległości pomiędzy punktem, a zbiorem wypukłym. Korzysta z faktu że możliwe jest zastosowanie tego algorytmu do analizy stanów kwantowych pod kątem detekcji splątania kwantowego[9][10]. Algorytm ten wielokrotnie, z sukcesem, był wykorzystany do analizy problemów z dziedziny fizyki kwantowej[14][11].

2 Metody

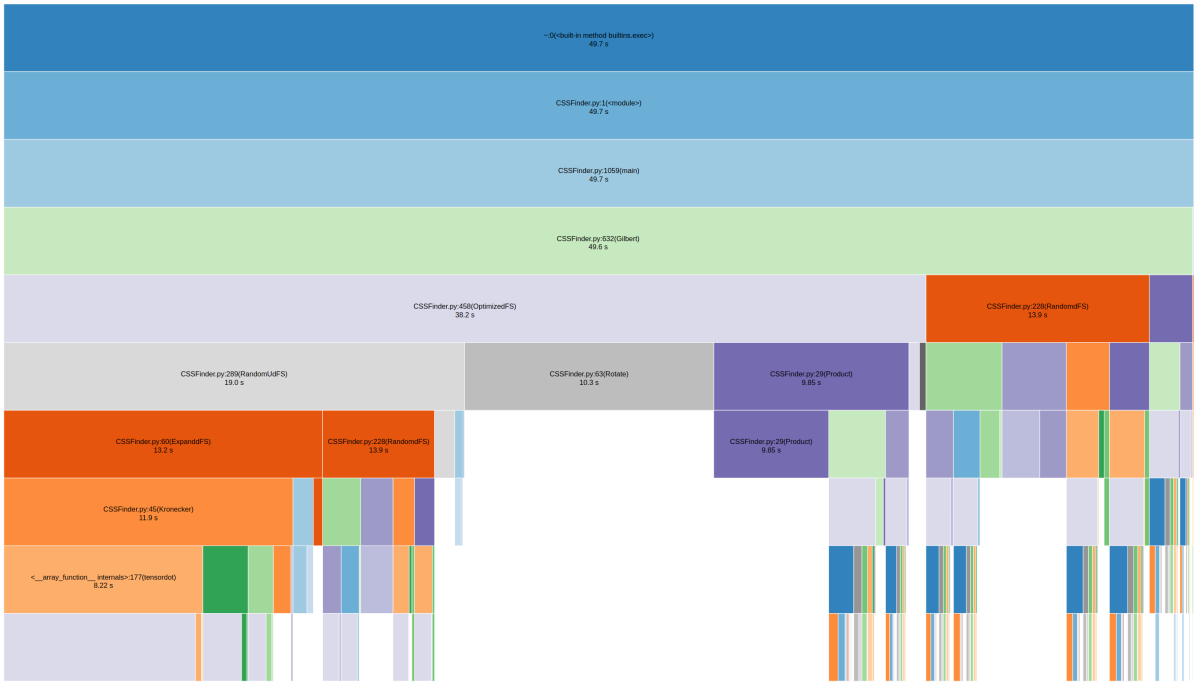
2.1 Środowisko testowe

Wszystkie wyniki wydajności uzyskane zostały przy użyciu następującej maszyny testowej i oprogramowania:

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) na układzie 5 qubitów (macierz 32×32 podwójnej precyzji zmiennoprzecinkowych liczb zespolonych) do uzyskania 1000 korekcji. Wykorzystałem do tego moduł z biblioteki standardowej języka Python, `cProfile`. Ze względu na silnie obciążającą procesor naturę programu i stosunkowo krótki czas pracy, istniała obawa że dodatkowe obciążenie ze strony modułu `profile` mogłoby zafałszować wyniki, dlatego preferowane było wykorzystanie `cProfile`. Przy analizie tak uzyskanych wyników posiłkowałem się wizualizacjami wykonanymi przy pomocy biblioteki `snakeviz`[17]. Podczas testów, program wykorzystywał domyślny globalny generator biblioteki NumPy (PCG64[18]) z ziarnem ustawionym na wartość 0.



Rysunek 2: Diagram przedstawiający udział całkowitego czasu wykonywania wywołań funkcji w całkowitym czasie 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. Program pracował w trybie 1 (full separability of an n-quDit state) na 5 qubitach (macierz 32×32 podwójnej precyzji zmiennoprzecinkowych liczb zespolonych)

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. Ujęte zostały tylko wywołania funkcji z kodu programu CSSFinder. Program pracował w trybie 1 (full separability of an n-quDit state) na 5 qubitach (macierz 32×32 podwójnej precyzji zmiennoprzecinkowych liczb zespolonych). 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%[1]) 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 dzia-

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

łania 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 wydajności jest tu maksymalizacja wydajności operacji macierzowych. W uzyskanych danych nie widać problemów z operacjami I/O[2].

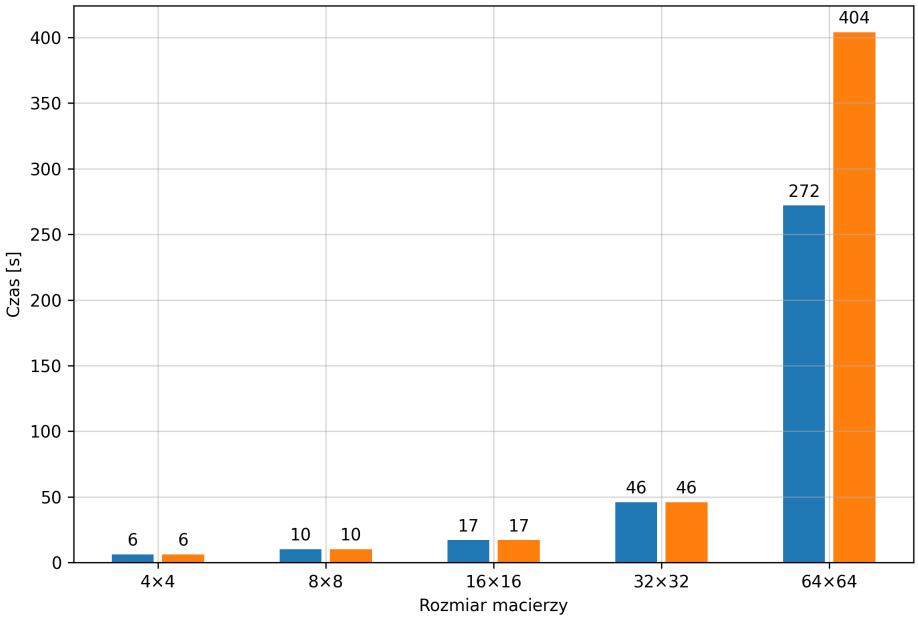
2.3 Wstępne pomiary wydajności

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[3].

Dane przekazywałem kolejno do programu z poleceniem działania w trybie 1 (full separability of an n-quDit state) do osiągnięcia 1000 korekcji lub do 2.000.000 iteracji algorytmu, w zależności od tego co nastąpi szybciej. Dla wszystkich macierzy algorytm uzyskał 1000 korekcji i w żadnym przypadku nie osiągnął maksymalnej ilości iteracji. Dla każdej macierzy pomiar był powtarzany pięciokrotnie. Podczas obliczeń ziarno globalnego generatora liczb losowych biblioteki NumPy było ustawione na 0. Pomiary czasu pracy dotyczyły wyłącznie samego algorytmu[4].



Rysunek 3: Wyniki wstępnych testów wydajności oryginalnego kodu z zablokowaną (niebieski) i odblokowaną (pomarańczowy) liczbą wątków obliczeniowych dla macierzy ρ_2 - ρ_6.

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

[3] 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.

[4] 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.

Podczas testów zaobserwowałem interesujące zjawisko dotyczące wydajności dla macierzy 64×64. W przypadku takich macierzy biblioteka NumPy automatycznie decyduje o wykorzystaniu wielowątkowej implementacji mnożenia macierzowego. Niestety, daje to efekt odwrotny do zamierzonego - obliczenia zamiast przyspieszać zwalniają. Na rysunku 3 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.

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 kodu. Tak też program został podzielony na dwie części: główną (core), z interfejsem użytkowników i narzędziami pomocniczymi oraz część implementującą algorytm (backend). Korpus jest w całości napisany w języku Python i wykorzystuje wbudowany w ten język mechanizm importowania bibliotek w celu wykrywania i ładowania implementacji algorytmu. Dane macierzowe w obrębie korpusu przechowywane są jako obiekty `ndarray` z biblioteki NumPy, ze względu na uniwersalność w świecie bibliotek do obliczeń tensorowych. Pozwala to na proste podmiany implementacji o dowolnie różnym pochodzeniu, w tym implementacje w językach kompilowanych. Uprościło to znacznie proces weryfikacji zmian w zachowaniu programu i przyspieszyło proces tworzenia kolejnych implementacji, jako że kod interfejsu programistycznego jest mniej pracochłonny niż kod pozwalający na interakcję z użytkownikiem.

2.5 Dostępne narzędzia

2.5.1 Kompilacja AOT

Obecnie najpowszechniej używana implementacja języka Python, CPython, posiada możliwość korzystania z bibliotek współdzielonych (`.so` - Linux, `.dll`/`.pyd` - Windows). 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 zawieraja transpilator, zdolny przetłumaczyć dedykowany język na C/C++, a następnie, wykorzystując osobno zainstalowany kompilator, skompilować do kodu maszynowego.

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

Ponieważ w każdym z wymienionych przypadków, kod niższego poziomu jest kompilowany przed dostarczeniem do użytkownika, pozwala to na wykorzystanie zaawansowanych możliwości automatycznej optymalizacji dostarczanych przez współczesne kompilatory, na przykład LLVM, które jest sercem

implementacji `clang` (język C++) oraz `rustc` (język Rust). Wiele bibliotek korzysta z mieszanek wymienionych powyżej metod, w tym cieszące się dużą popularnością NumPy, CuPy, Tensorflow czy PyTorch.

2.5.2 Kompilacja JIT

W momencie pisania tej pracy istnieją dwa szeroko dostępne i aktywnie utrzymywane narzędzia oferujące kompilację JIT dla języka Python. Jednym z nich jest pełna alternatywna implementacja języka Python - PyPy[36]. Wykonywana przez nią kompilacja JIT działa on na podobnej zasadzie do uprzednio wymienionych - śledzi cały kod który wykonuje i automatycznie decyduje które fragmenty skompilować do kodu maszynowego[1]. Drugim narzędziem jest biblioteka Numba[6][7]. Ona, w przeciwieństwie do PyPy, wymaga aby fragmenty kodu, które mają być skompilowane, miały postać funkcji oznaczonych dedykowanymi dekoratorami[5].

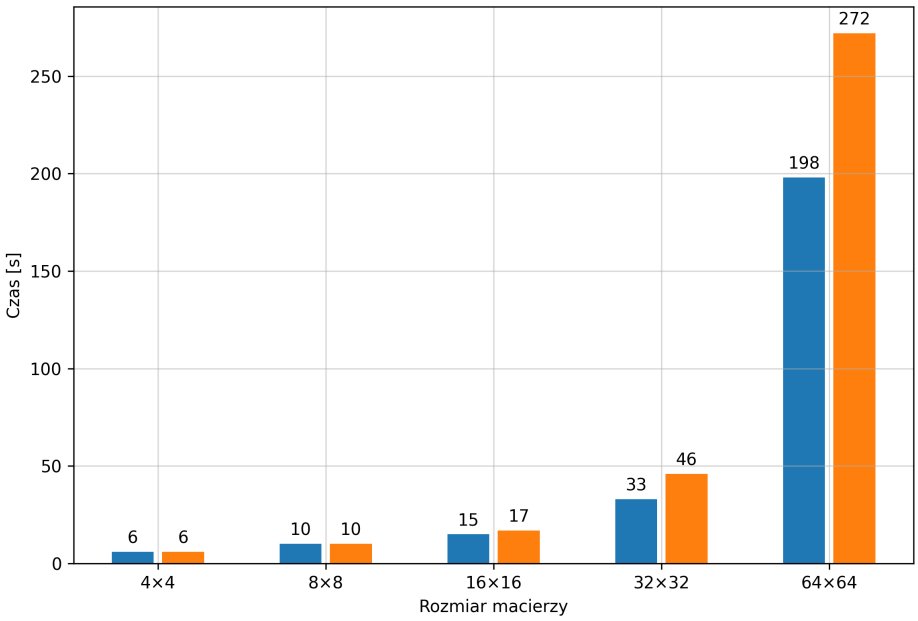
[5]Obecnie dostępny jest też dekorator pozwalający na kompilację klas, niestety jest on niestabilny i nie radzi sobie w wielu sytuacjach.

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 zmaksymalizować wykorzystanie funkcji zawartych w bibliotece NumPy. Ponieważ kluczowe dla wydajności fragmenty kodu 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 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[6] 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. Podczas obliczeń ziarno globalnego generatora liczb losowych biblioteki NumPy było ustawione na 0. Pomiary czasu pracy dotyczyły przede wszystkim samego algorytmu[7].



Rysunek 4: Wyniki testów wydajności alternatywnej implementacji Python i NumPy (niebieski) w porównaniu do implementacji oryginalnej z zablokowaną ilością wątków (pomarańczowy) dla macierzy ρ_2 - ρ_6.

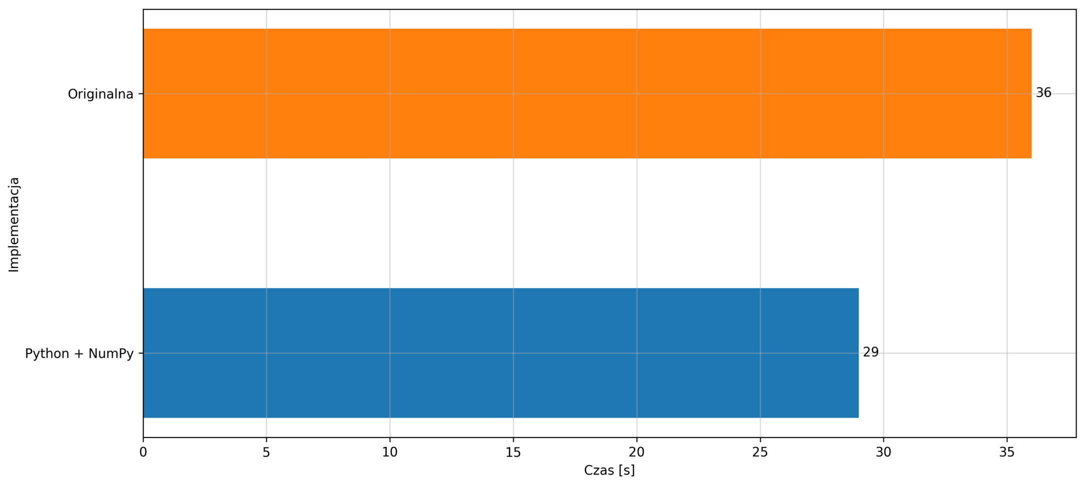
W testach uzyskałem blisko 30% redukcję długości czasu pracy. Na rysunku 4 kolorem pomarańczowym oznaczone zostały czasy pracy programu, w zależności od rozmiaru macierzy gęstości. Niebieskie kolumny to czasy pracy alternatywnej implementacji. Warto tutaj zauważyć że to, o ile wzrośnie wydajność, w dużej mierze zależy od danych wejściowych, ponieważ inny zestaw macierzy gęstości może spowodować, że inne fragmenty kodu będą podlegały szczególnemu obciążeniu.

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

[7] Nie biorą więc pod uwagę czasu pochłoniętego przez importowanie modułów itp., natomiast operacje wczytywania danych i pisania do plików które były wykonywane w obrębie tego fragmentu programu są wliczane w czas pracy.

$$\rho_7 = \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 5: Macierz ρ_7.



Rysunek 6: Wyniki testów wydajności alternatywnej implementacji Python i NumPy (niebieski) w porównaniu do implementacji oryginalnej z zablokowaną ilością wątków (pomarańczowy) dla macierzy ρ_7.

Tak też sięgając po macierz postaci ρ_7, przedstawioną na rysunku 5, uzyskałem wyniki różniące się od tych wykorzystujących macierze ρ_2 - ρ_6. Zostały one 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_7 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.

2.6.2 Python i NumPy z AOT

2.6.3 Python i NumPy z JIT

2.6.4 Rust

2.7 Precyzja obliczeń

Oryginalny program posługiwał się liczbami zespolonymi stworzonymi na bazie zmiennoprzecinkowych podwójnej precyzji. Jednak w przypadku analizowania niewielkich macierzy ponieważ analizowane 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 uzywają 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).

2.8 Pomiary czasu pracy

2.9 Ponowne profilowanie

2.10 Funkcja kronecker

2.11 Funkcja product

3 Wyniki

4 Dyskusja

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] Mark Lutz. *Learning python: Powerful object-oriented programming*. Ó'Reilly Media, Inc.", 2013.

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

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

[7] 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] Python Software Foundation. *Packaging PEPs | peps.python.org*. 2023. URL: `https://peps.python.org/topic/packaging/` (term. wiz. 14. 05. 2023).

[24] Python Software Foundation. *Welcome to Python.org*. 2023. URL: `https://www.python.org/` (term. wiz. 14. 05. 2023).

[25] Inc. Free Software Foundation. *GCC, the GNU Compiler Collection*. 2023. URL: https://gcc.gnu.org/ (term. wiz. 14. 05. 2023).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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