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

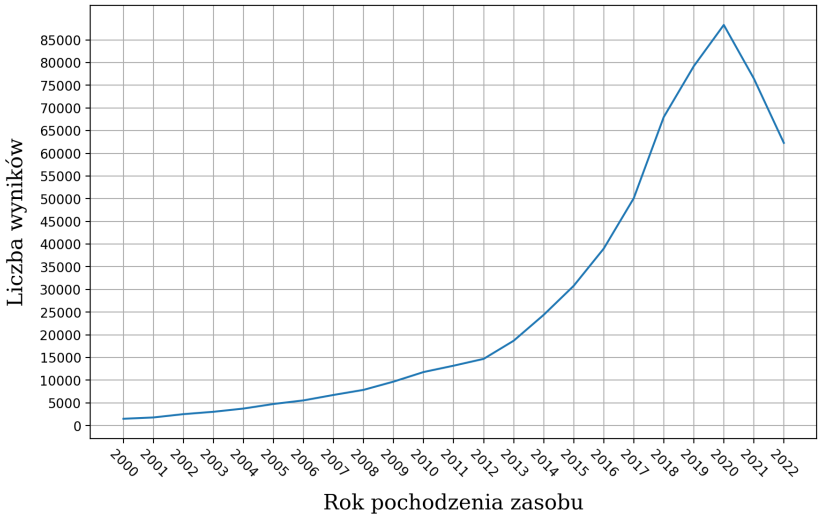
Spis treści

Streszczenie

W tej pracy przeprowadzam analizę możliwości optymalizacji pod kątem czasu wykonania oprogramowania napisanego w języku Python przez pryzmat 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[3][4], wersja wzbogacona o kompilację JIT przy pomocy biblioteki Numba[1][2], wersja skompilowana do kodu maszynowego przy pomocy biblioteki Cython i GCC oraz implementacja w języku Rust, również skompilowana do kodu maszynowego.

1 Wstęp

Język Python zachęca użytkowników prostotą składni, łatwością tworzenia kodu, ze względu na swoją dynamiczną naturę i automatyczne zarządzanie pamięcią, mnogością dostępnych bibliotek otwartoźródłowych, czy rozbudowaną społecznością programistów. Niestety, wygoda i elastyczność tego języka ma pewien ukryty koszt. Interpretowany kod, napisany w Pythonie, pod względem wydajności znacząco odstaje od kompilowanych języków programowania (C, Fortran, C++, Rust). Istnieją jednakże metody pozwalające na obejście tej niedogodności.



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

1.1 Cel pracy

Praca ma na celu rozpatrzenie efektów uzyskiwanych różnymi sposobami optymalizacji kodu w przypadku konkretnego oprogramowania. Analizy dotyczyć będą programu CSSFinder którego wielokrotnej re-implementacji, z wykorzystaniem różnych metod optymalizacji, dokonałem w ramach prac projektowych. Wraz z optymalizacją udoskonalony został interfejs użytkownika programu, a sam wynikowy kod dostępny jest na GitHub'ie[5][6][7].

1.2 Biblioteki współdzielone

Klasycznym rozwiązaniem jest ucieknięcie się do wykorzystania odpowiednio spreparowanych narzędzi napisanych w językach niższego poziomu (C, C++, Rust i potencjalnie inne), które interpreter jest w stanie zaimportować. Mamy więc w tym wypadku do czynienia z kompilacją przed czasem wykonywania (ahead-of-time compilation - AOT compilation). Pozwala to na utworzenie zbioru relatywnie prymitywnych narzędzi w językach wymagających więcej czasu i pracy, aby następnie komponować je w skompilowane oprogramowanie w języku wielokrotnie prostszym. Przykładami takich bibliotek są NumPy oraz CuPy. Pierwsza z nich koncentruje się na obliczeniach wykonywanych na CPU, druga jest bliźniaczo podobna, ale wykorzystuje GPU.

1.3 Kompilacja JIT

Możliwości przyspieszania Pythona nie kończą się na pisaniu kodu w innych językach. Dzięki odpowiednim narzędziom wykonalne jest podążenie w ślady języków takich jak JavaScrip, Lua czy Java i wykorzystanie kompilacji do kodu maszynowego w czasie wykonywania (just-in-time compilation - JIT compilation). W momencie pisania tej pracy istnieją dwa szeroko dostępne narzędzia oferujące kompilację JIT. Pierwszym jest biblioteka Numba, która pozwala na kompilację odpowiednio oznaczonych fragmentów kodu. Drugą jest pełna alternatywna implementacja interpretera Pythona, PyPy, która automatycznie decyduje które fragmenty kodu skompilować.

1.4 Cython

Rozwiązaniem pomiędzy uprzednio wymienionymi jest Cython. Jest to zarówno nazwa biblioteki jak i nadzbioru języka Python który pozwala na dodatkowe adnotowanie kodu informacjami o typach. Kod napisany w Cythonie można następnie transpilować do C lub C++ po czym skompilować do kodu

maszynowego. Cython nie wymaga aby w kompilowanym kodzie znajdowały się dodatkowe adnotacje, wzwiązku z czym możliwe jest skompilowanie czystego kodu Pythona do kodu maszynowego. Pozwala to na pozbycie się obciążenia ze strony procesu interpretacji i oraz skorzystać z optymalizacji które potrafią wykonywać współczesne kompilatory. Nie usuwa to jednak obciążenia ze strony dynamicznego systemu typów, czyniąc kompilację bez adnotacji mało efektywną.

1.5

Źródła

[1] 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.

[2] Inc. Anaconda i in. *Numba documentation*. 2020. URL: https://numba.readthedocs.io/en/stable/user/index.html.

[3] 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.

[4] NumPy Developers. *NumPy documentation*. 2022. URL: https://numpy.org/doc/stable/.

[5] Krzysztof Wiśniewski. *CSSFinder (Core)*. 2023. URL: https://github.com/Argmaster/CSSFinder.

[6] Krzysztof Wiśniewski. *CSSFinder Numpy Backend*. 2023. URL: https://github.com/Argmaster/cssfinder_backend_numpy.

[7] Krzysztof Wiśniewski. *CSSFinder Rust Backend*. 2023. URL: https://github.com/Argmaster/cssfinder_backend_rust.