O'REILLY®

Python

Programmuj szybko i wydajnie

WYDAJNE APLIKACJE W JĘZYKU PYTHON!

Micha Gorelick, Ian Ozsvald

Tytuł oryginału: High Performance Python: Practical Performant Programming for Humans

Tłumaczenie: Piotr Pilch

ISBN: 978-83-283-0466-6

© 2015 Helion S.A.

Authorized Polish translation of the English edition of High Performance Python, ISBN 9781449361594 © 2014 Micha Gorelick and Ian Ozsvald.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pytpsw
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa 9

1. Wydajny kod Python 15

Podstawowy system komputerowy 15
Jednostki obliczeniowe 16
Jednostki pamięci 19
Warstwy komunikacji 21
Łączenie ze sobą podstawowych elementów 22
Porównanie wyidealizowanego przetwarzania z maszyną wirtualną języka Python 23
Dlaczego warto używać języka Python? 26

2. Użycie profilowania do znajdowania wąskich gardeł 29

Efektywne profilowanie 30
Wprowadzenie do zbioru Julii 31
Obliczanie pełnego zbioru Julii 34
Proste metody pomiaru czasu — instrukcja print i dekorator 37
Prosty pomiar czasu za pomocą polecenia time systemu Unix 40
Użycie modułu cProfile 41
Użycie narzędzia runsnake do wizualizacji danych wyjściowych modułu cProfile 46
Użycie narzędzia line_profiler do pomiarów dotyczących kolejnych wierszy kodu 46
Użycie narzędzia memory_profiler do diagnozowania wykorzystania pamięci 51
Inspekcja obiektów w stercie za pomocą narzędzia heapy 56
Użycie narzędzia dowser do generowania aktywnego wykresu dla zmiennych z utworzonymi instancjami 58
Użycie modułu dis do sprawdzania kodu bajtowego narzędzia CPython 60
Różne metody, różna złożoność 62
Testowanie jednostkowe podczas optymalizacji w celu zachowania poprawności 64
Dekorator @profile bez operacji 64
Strategie udanego profilowania kodu 66
Podsumowanie 67

Kup książkę   Poleć książkę

3. **Listy i krotki** 69
Bardziej efektywne wyszukiwanie 71
Porównanie list i krotek 73
Listy jako tablice dynamiczne 74
Krotki w roli tablic statycznych 77
Podsumowanie 78

4. **Słowniki i zbiory** 79
Jak działają słowniki i zbiory? 82
  Wstawianie i pobieranie 82
  Usuwanie 85
  Zmiana wielkości 85
  Funkcje mieszania i entropia 86
Słowniki i przestrzenie nazw 89
Podsumowanie 92

5. **Iteratory i generatory** 93
Iteratory dla szeregów nieskończonych 96
Wartościowanie leniwe generatora 97
Podsumowanie 101

6. **Obliczenia macierzowe i wektorowe** 103
Wprowadzenie do problemu 104
Czy listy języka Python są wystarczająco dobre? 107
  Problemy z przesadną alokacją 109
Fragmentacja pamięci 111
  Narzędzie perf 113
  Podejmowanie decyzji z wykorzystaniem danych wyjściowych narzędzia perf 115
  Wprowadzenie do narzędzia numpy 116
Zastosowanie narzędzia numpy w przypadku problemu dotyczącego dyfuzji 119
  Przydziały pamięci i operacje wewnętrzne 121
  Optymalizacje selektywne: znajdowanie tego, co wymaga poprawienia 124
Moduł numexpr: przyspieszanie i upraszczanie operacji wewnętrznych 127
Przestroga: weryfikowanie „optymalizacji” (biblioteka scipy) 129
Podsumowanie 131

7. **Kompilowanie do postaci kodu C** 133
Jakie wzrosty szybkości są możliwe? 134
Porównanie kompilatorów JIT i AOT 136
Dlaczego informacje o typie ułatwiają przyspieszenie działania kodu? 136
Użycie kompilatora kodu C 137
Analiza przykładu zbioru Julii 138
Cython 139
  Kompilowanie czystego kodu Python za pomocą narzędzia Cython 139
  Użycie adnotacji kompilatora Cython do analizowania bloku kodu 141
  Dodawanie adnotacji typu 143

Kup ksi k   Pole ksi k

Shed Skin ... 147
Tworzenie modułu rozszerzenia ... 148
Koszt związany z kopiami pamięci ... 150
Cython i numpy ... 151
Przetwarzanie równoległe rozwiązania na jednym komputerze z wykorzystaniem interfejsu OpenMP ... 152
Numba ... 154
Pythran ... 155
PyPy ... 157
Różnice związane z czyszczeniem pamięci ... 158
Uruchamianie interpretera PyPy i instalowanie modułów ... 159
Kiedy stosować poszczególne technologie? ... 160
Inne przyszłe projekty ... 162
Uwaga dotycząca układów GPU ... 162
Oczekiwania dotyczące przyszłego projektu kompilatora ... 163
Interfejsy funkcji zewnętrznych ... 163
ctypes ... 164
cffi ... 166
f2py ... 169
Moduł narzędzia CPython ... 171
Podsumowanie ... 174

8. Współbieżność ... 175
Wprowadzenie do programowania asynchronicznego ... 176
Przeszukiwacz szeregowy ... 179
gevent ... 181
tornado ... 185
AsyncIO ... 188
Przykład z bazą danych ... 190
Podsumowanie ... 193

9. Moduł multiprocessing ... 195
Moduł multiprocessing ... 198
Przybliżenie liczby pi przy użyciu metody Monte Carlo ... 200
Przybliżanie liczby pi za pomocą procesów i wątków ... 201
Zastosowanie obiektów języka Python ... 201
Liczby losowe w systemach przetwarzania równoległego ... 208
Zastosowanie narzędzia numpy ... 209
Znajdowanie liczb pierwszych ... 211
Kolejki zadań roboczych ... 217
Weryfikowanie liczb pierwszych za pomocą komunikacji międzyprocesowej ... 221
Rozwiązanie z przetwarzaniem szeregowym ... 225
Rozwiązanie z prostym obiektem Pool ... 225
Rozwiązanie z bardzo prostym obiektem Pool dla mniejszych liczb ... 227
Użycie obiektu Manager.Value jako flagi ... 228

Kup ksi k Pole ksi k

Użycie systemu Redis jako flagi 229
Użycie obiektu RawValue jako flagi 232
Użycie modułu mmap jako flagi 232
Użycie modułu mmap do odtworzenia flagi 233
Współużytkowanie danych narzędzia numpy za pomocą modułu multiprocessing 236
Synchronizowanie dostępu do zmiennych i plików 242
Blokowanie plików 242
Blokowanie obiektu Value 245
Podsumowanie 248

10. Klastry i kolejki zadań 249
Zalety klastrowania 250
Wady klastrowania 251
Strata o wartości 462 milionów dolarów na giełdzie Wall Street z powodu kiepskiej strategii aktualizacji klastra 252
24-godzinny przestój usługi Skype w skali globalnej 253
Typowe projekty klastrowe 254
Metoda rozpoczęcia tworzenia rozwiązania klastrowego 254
Sposoby na uniknięcie kłopotów podczas korzystania z klastrów 255
Trzy rozwiązania klastrowe 257
Użycie modułu Parallel Python dla prostych klastrów lokalnych 257
Użycie modułu IPython Parallel do obsługi badań 259
Użycie systemu NSQ dla niezawodnych klastrów produkcyjnych 262
Kolejki 263
Publikator/subskrybent 264
Rozproszone obliczenia liczb pierwszych 266
Inne warte uwagi narzędzia klastrowania 268
Podsumowanie 269

11. Mniejsze wykorzystanie pamięci RAM 271
Obiekty typów podstawowych są kosztowne 272
Moduł array zużywa mniej pamięci do przechowywania wielu obiektów typu podstawowego 273
Analiza wykorzystania pamięci RAM w kolekcji 276
Bajty i obiekty Unicode 277
Efektywne przechowywanie zbiorów tekstowych w pamięci RAM 279
Zastosowanie metod dla 8 milionów tokenów 280
Wskazówki dotyczące mniejszego wykorzystania pamięci RAM 288
Probabilistyczne struktury danych 289
Obliczenia o bardzo dużym stopniu przybliżenia z wykorzystaniem jednobajtowego licznika Morrisa 290
Wartości k-minimum 291
Filtry Blooma 295
Licznik LogLog 299
Praktyczny przykład 303

Kup ksi k Pole ksi k

12. Rady specjalistów z branży 307
Narzędzie Social Media Analytics (SoMA) firmy Adaptive Lab 307
Język Python w firmie Adaptive Lab 308
Projekt narzędzia SoMA 308
Zastosowana metodologia projektowa 309
Serwisowanie systemu SoMA 309
Rada dla inżynierów z branży 310
Technika głębokiego uczenia prezentowana przez firmę RadimRehurek.com 310
Strzał w dziesiątkę 311
Rady dotyczące optymalizacji 313
Podsumowanie 315
Uczenie maszynowe o dużej skali gotowe do zastosowań produkcyjnych w firmie Lyst.com 315
Rola języka Python w witrynie Lyst 316
Projekt klastra 316
Ewolucja kodu w szybko rozwijającej się nowej firmie 316
Budowanie mechanizmu rekomendacji 316
Raportowanie i monitorowanie 317
Rada 317
Analiza serwisu społecznościowego o dużej skali w firmie Smesh 318
Rola języka Python w firmie Smesh 318
Platforma 318
Dopasowywanie łańcuchów w czasie rzeczywistym z dużą wydajnością 319
Raportowanie, monitorowanie, debugowanie i wdrażanie 320
Interpreter PyPy zapewniający powodzenie systemów przetwarzania danych i systemów internetowych 322
Wymagania wstępne 322
Baza danych 323
Aplikacja internetowa 323
Mechanizm OCR i tłumaczenie 324
Dystrybucja zadań i procesy robocze 324
Podsumowanie 325
Kolejki zadań w serwisie internetowym Lanyrd.com 325
Rola języka Python w serwisie Lanyrd 325
Zapewnianie odpowiedniej wydajności kolejki zadań 326
Raportowanie, monitorowanie, debugowanie i wdrażanie 326
Rada dla programistów z branży 326

Skorowidz 329

Kup ksi k Pole ksi k

Kup ksi k Pole ksi k

ROZDZIAŁ 7.

Kompilowanie do postaci kodu C

Pytania, na jakie będziesz w stanie udzielić odpowiedzi po przeczytaniu rozdziału

- Jak możesz sprawić, że kod Python będzie działać jako kod niższego poziomu?
- Jaka jest różnica między kompilatorem JIT i kompilatorem AOT?
- Jakie zadania mogą być wykonywane przez skompilowany kod Python szybciej niż w przypadku zwykłego kodu Python?
- Dlaczego adnotacje typu zwiększają szybkość skompilowanego kodu Python?
- Jak możesz utworzyć moduły dla kodu Python za pomocą języka C lub Fortran?
- Jak możesz użyć w kodzie Python bibliotek języka C lub Fortran?

Najprostszym sposobem przyspieszenia kodu jest ograniczenie liczby operacji, jakie będzie wykonywać. Zakładając, że zostały już wybrane dobre algorytmy i zmniejszono ilość przetwarzanych danych, najprostsza metoda, by wykonywać mniejszą liczbę instrukcji, polega na skompilowaniu kodu do postaci kodu maszynowego.

W tym zakresie język Python oferuje kilka opcji obejmujących narzędzia do kompilowania oparte na czystym kodzie C, takie jak Cython, Shed Skin i Pythran, kompilowanie bazujące na kompilatorze LLVM za pośrednictwem narzędzia Numba oraz zastępczą maszynę wirtualną PyPy, która zawiera wbudowany kompilator JIT (*Just in Time*). Przy podejmowaniu decyzji dotyczącej ścieżki, jaka zostanie obrana, konieczne jest zrównoważenie wymagań związanych z łatwością dostosowania kodu i impetu zespołu.

Każde z wymienionych narzędzi dodaje nową zależność do szeregu używanych narzędzi. Dodatkowo narzędzie Cython wymaga pisania w języku nowego typu (hybrydzie języków Python i C), co wiąże się z koniecznością zdobycia nowej umiejętności. Wykorzystanie nowego języka narzędzia Cython może mieć negatywny wpływ na impet zespołu, ponieważ jego członkowie bez znajomości języka C mogą mieć problem z obsługą takiego kodu. Jednak w praktyce jest to raczej niewielki kłopot, gdyż kod narzędzia Cython będzie używany tylko w dobrze wybranych, niewielkich obszarach kodu.

Kup ksi k Pole ksi k

Godne uwagi jest to, że przeprowadzanie profilowania kodu w odniesieniu do pamięci i procesora prawdopodobnie spowoduje, że zaczniesz się zastanawiać nad optymalizacjami algorytmicznymi wyższego poziomu, które mogą zostać zastosowane. Takie zmiany algorytmiczne (czyli użycie dodatkowej logiki w celu uniknięcia obliczeń lub buforowanie eliminujące ponowne obliczenia) mogą pomóc Ci zapobiegać wykonywaniu w kodzie zbędnych działań. Ekspresywność kodu Python ułatwia zauważenie takich możliwości algorytmicznych. W podrozdziale „Technika głębokiego uczenia prezentowana przez firmę RadimRehurek.com" z rozdziału 12. Radim Řehůřek wyjaśnia, jak implementacja kodu Python może wygrać z czystą implementacją stworzoną w języku C.

W rozdziale dokonamy przeglądu następujących narzędzi:

- Cython — jest to najczęściej używane narzędzie do kompilowania do postaci kodu C, które uwzględnia zarówno narzędzie `numpy`, jak i zwykły kod Python (wymagana jest znajomość języka C).
- Shed Skin — zautomatyzowany konwerter Python-C przeznaczony dla kodu, który nie bazuje na narzędziu `numpy`.
- Numba — nowy kompilator stworzony z myślą o kodzie bazującym na narzędziu `numpy`.
- Pythran — nowy kompilator przeznaczony zarówno dla kodu bazującego na narzędziu `numpy`, jak i innego kodu.
- PyPy — stabilny kompilator JIT dla kodu, który nie bazuje na narzędziu `numpy`. Kod taki zastępuje zwykły plik wykonywalny Python.

W dalszej części rozdziału przyjrzymy się interfejsom funkcji zewnętrznych, które umożliwiają skompilowanie kodu C do postaci modułów rozszerzeń dla języka Python. Wbudowany interfejs API tego języka jest używany razem z narzędziami `ctypes` i `cffi` (twórców kompilatora PyPy) oraz z konwerterem Fortran-Python `f2py`.

Jakie wzrosty szybkości są możliwe?

Jeśli problem związany jest z metodą kompilowania, całkiem prawdopodobne są wzrosty szybkości wynoszące rząd wielkości lub więcej. Przyjrzymy się tutaj różnym metodom osiągania przyspieszeń wynoszących jeden lub dwa rzędy wielkości dla pojedynczego rdzenia, a także w przypadku zastosowania wielu rdzeni za pośrednictwem interfejsu OpenMP.

Kod Python, który zwykle będzie działać szybciej po skompilowaniu, prawdopodobnie służy do zastosowań matematycznych, a ponadto zawiera raczej wiele pętli wielokrotnie powtarzających te same operacje. W obrębie tych pętli możliwe jest tworzenie wielu obiektów tymczasowych.

Mało prawdopodobne jest to, że kod wywołujący biblioteki zewnętrzne (np. wyrażenia regularne, operacje na łańcuchach, wywołania bibliotek bazy danych) wykaże jakiekolwiek przyspieszenie po skompilowaniu. Programy powiązane z operacjami wejścia-wyjścia również raczej nie pozwolą osiągnąć znacznych przyspieszeń.

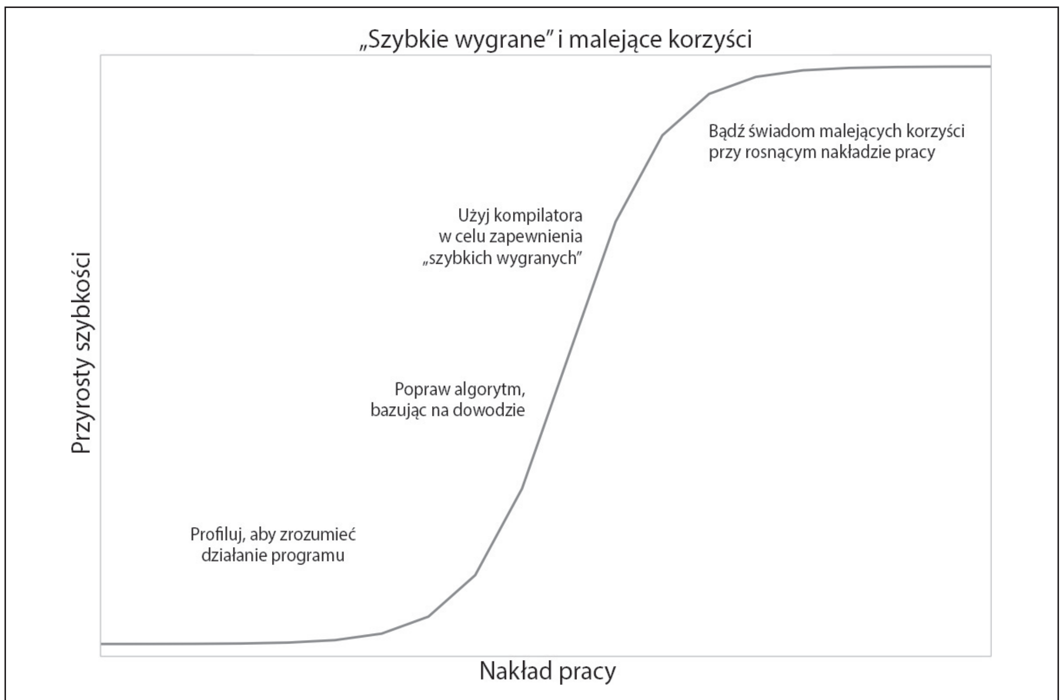
Jeśli kod Python koncentruje się na wywoływaniu wektoryzowanych funkcji narzędzia `numpy`, wcale może nie działać szybciej po skompilowaniu. Inaczej będzie tylko wtedy, gdy kompilowany kod to głównie kod Python (a ponadto prawdopodobnie w sytuacji, kiedy w kodzie jest wykonywana pętla). W rozdziale 6. omówiono operacje narzędzia `numpy`. Okazuje się, że w ich przypadku kompilowanie nie będzie pomocne, ponieważ nie występuje wiele obiektów pośrednich.

Kup ksi k Pole ksi k

Ogólnie rzecz biorąc, bardzo mało prawdopodobne jest to, że skompilowany kod będzie w ogóle szybszy od funkcji napisanej w języku C. Niemniej jednak taki kod nie będzie też działać znacznie wolniej. Całkiem możliwe jest to, że kod C wygenerowany z kodu Python będzie działać tak samo szybko jak ręcznie napisana funkcja C, chyba że programista używający języka C dysponuje szczególnie dużą wiedzą na temat metod dostrajania kodu C do architektury docelowej platformy sprzętowej.

W przypadku kodu stworzonego z myślą o operacjach matematycznych możliwe jest, że ręcznie stworzona funkcja języka Fortran przewyższy odpowiadającą jej funkcję C. Jednakże i tym razem będzie to raczej wymagać wiedzy eksperckiej. Generalnie rzecz biorąc, wynik kompilacji (uzyskany prawdopodobnie z wykorzystaniem narzędzia Cython, Pythran lub Shed Skin) będzie zbliżony do wyniku dla ręcznie napisanego kodu C w stopniu wymaganym przez większość programistów.

Podczas profilowania algorytmu i korzystania z niego trzeba pamiętać o diagramie z rysunku 7.1. Trochę czasu poświęconego na zrozumienie kodu poprzez jego profilowanie powinno dać możliwość podjęcia lepszych decyzji na poziomie algorytmicznym. Skoncentrowanie się w dalszej kolejności na kompilatorze powinno zaowocować dodatkowym przyspieszeniem. Prawdopodobnie możliwe będzie dalsze dostrajanie algorytmu, ale nie należy być zaskoczonym coraz mniejszymi przyspieszeniami wynikającymi z coraz większej ilości włożonej pracy. Trzeba samemu stwierdzić, kiedy dodatkowe starania przestaną być opłacalne.



Rysunek 7.1. Trochę czasu poświęconego na profilowanie i kompilowanie zapewnia spore korzyści, ale dalsze działania zwykle są coraz mniej opłacalne

Kup ksi k Pole ksi k

Jeśli używasz kodu Python i całej grupy dołączonych bibliotek bez narzędzia `numpy`, podstawowymi opcjami wyboru będą narzędzia Cython, Shed Skin i PyPy. Jeśli używasz narzędzia `numpy`, odpowiednimi propozycjami są narzędzia Cython, Numba i Pythran. Obsługują one język Python 2.7, a część z nich jest też zgodna z językiem Python w wersji 3.2 lub nowszej.

Niektóre z przedstawionych dalej przykładów wymagają ogólnej znajomości kompilatorów kodu C oraz samego kodu C. W przypadku braku takiej wiedzy przed zagłębieniem się w tę tematykę powinieneś w podstawowym zakresie poznać język C i skompilować działający program napisany w tym języku.

Porównanie kompilatorów JIT i AOT

Omawiane narzędzia można podzielić na dwie grupy: służące do wcześniejszego kompilowania (kompilatory AOT: Cython, Shed Skin, Pythran) oraz do kompilowania przy pierwszej próbie użycia kodu (kompilatory JIT: Numba, PyPy).

Kompilowanie za pomocą kompilatora AOT (*Ahead of Time*) powoduje utworzenie biblioteki statycznej przeznaczonej dla używanej platformy sprzętowej. Po pobraniu narzędzia `numpy`, `scipy` lub `scikit-learn` nastąpi skompilowanie przez jedno z nich części biblioteki z wykorzystaniem kompilatora Cython na danej platformie sprzętowej (ewentualnie w przypadku użycia dystrybucji takiej jak Continuum Anaconda, zostanie zastosowana wcześniej zbudowana biblioteka kompilowana). Dzięki kompilacji kodu przed jego użyciem uzyskuje się bibliotekę, która może od razu zostać zastosowana przy rozwiązywaniu problemu.

Kompilowanie za pomocą kompilatora JIT (*Just in Time*) w dużym stopniu (lub całkowicie) eliminuje początkowe działania. Kompilator może rozpocząć kompilację tylko odpowiednich części kodu w momencie ich użycia. Oznacza to wystąpienie problemu zimnego startu. Polega on na tym, że może wystąpić sytuacja, gdy większość kodu programu została już skompilowana, a aktualnie używana porcja kodu jeszcze nie, więc w momencie rozpoczynania uruchamiania kodu w czasie trwania kompilacji będzie on działać bardzo wolno. Jeśli ma to miejsce każdorazowo przy uruchamianiu skryptu, który jest uaktywniany wielokrotnie, związany z tym spadek wydajności może stać się znaczny. Ponieważ problem ten dotyczy kompilatora PyPy, korzystanie z niego w przypadku krótkich, lecz często wykonywanych skryptów może okazać się niepożądane.

W tym miejscu omówienia widać, że wcześniejsze kompilowanie daje korzyść w postaci najlepszych przyspieszeń, ale często wymaga też największego nakładu pracy. Kompilatory JIT oferują duże przyspieszenia przy bardzo małej liczbie ręcznie wprowadzanych zmian, ale mogą powodować opisany problem. Przy wyborze właściwej technologii na potrzeby konkretnego zastosowania konieczne będzie rozważenie tych kwestii.

Dlaczego informacje o typie ułatwiają przyspieszenie działania kodu?

W języku Python typy są dynamicznie określane. Zmienna może odwoływać się do obiektu dowolnego typu, a dowolny wiersz kodu może zmienić typ przywoływanego obiektu. Utrudnia to maszynie wirtualnej optymalizację metody wykonywania kodu na poziomie kodu maszynowego,

Kup ksi k Pole ksi k

ponieważ nie dysponuje ona informacją o tym, jaki podstawowy typ danych będzie używany dla przyszłych operacji. Utrzymywanie kodu w uogólnionej postaci powoduje, że będzie on dłużej wykonywany.

W poniższym przykładzie `v` identyfikuje liczbę zmiennoprzecinkową lub parę takich liczb, które reprezentują liczbę zespoloną `complex`. Oba warunki mogą wystąpić w tej samej pętli w różnym czasie lub w powiązanych kolejnych sekcjach kodu:

```
v = -1.0
print type(v), abs(v)
<type 'float'> 1.0
v = 1-1j
print type(v), abs(v)
<type 'complex'> 1.41421356237
```

Funkcja `abs` działa różnie w zależności od bazowego typu danych. Funkcja ta użyta dla liczby całkowitej lub zmiennoprzecinkowej po prostu powoduje przekształcenie wartości ujemnej w wartość dodatnią. W przypadku liczby zespolonej funkcja `abs` pobiera pierwiastek kwadratowy sumy elementów podniesionych do kwadratu:

$$abs(c) = \sqrt{c.real^2 + c.imag^2}$$

Kod maszynowy dla przykładu liczby zespolonej `complex` uwzględnia więcej instrukcji i do wykonania wymaga więcej czasu. Przed wywołaniem funkcji `abs` dla zmiennej interpreter języka Python musi najpierw poszukać typu zmiennej, a następnie zdecydować, jaką wersję funkcji wywołać. Związane z tym obciążenie zwiększa się w przypadku wykonywania wielu powtarzanych wywołań.

W obrębie kodu Python każdy podstawowy obiekt, taki jak liczba całkowita, zostanie opakowany za pomocą obiektu języka Python wyższego poziomu (np. za pomocą obiektu `int` w przypadku liczby całkowitej). Tego rodzaju obiekt oferuje dodatkowe funkcje, takie jak `__hash__` (ułatwia przechowywanie) i `__str__` (obsługuje wyświetlanie łańcuchów).

Wewnątrz sekcji kodu powiązanego z procesorem częstą sytuacją jest to, że typy zmiennych nie zmieniają się. Daje to możliwość zastosowania kompilacji statycznej i szybszego wykonywania kodu.

Jeśli wymaganych jest jedynie wiele pośrednich operacji matematycznych, nie są potrzebne funkcje wyższego poziomu, a ponadto mogą być zbędne mechanizmy służące do zliczania odwołań. W tym przypadku można po prostu przejść do poziomu kodu maszynowego i przeprowadzić szybko obliczenia przy użyciu kodu maszynowego i bajtów, a nie poprzez modyfikowanie obiektów wyższego poziomu języka Python, z czym wiąże się większe obciążenie. W tym celu wcześniej określane są typy obiektów, aby możliwe było wygenerowanie poprawnego kodu C.

Użycie kompilatora kodu C

W dalszych przykładach zostaną zastosowane kompilatory `gcc` i `g++` z zestawu narzędziowego GNU C Compiler. Jeśli poprawnie skonfigurujesz środowisko, możesz skorzystać z alternatywnego kompilatora (np. `icc` Intela lub `cl` Microsoftu). Narzędzie Cython korzysta z kompilatora `gcc`, a narzędzie Shed Skin używa kompilatora `g++`.

Kup ksi k Pole ksi k

Kompilator `gcc` stanowi znakomity wybór w przypadku większości platform, ponieważ jest dobrze obsługiwany i dość zaawansowany. Często możliwe jest uzyskanie większej wydajności za pomocą dostrojonego kompilatora (np. w przypadku urządzeń Intela kompilator `icc` tej firmy może wygenerować szybszy kod niż kompilator `gcc`), ale wiąże się to z koniecznością poszerzenia wiedzy specjalistycznej i uzyskania informacji o sposobie dostosowywania flag dla alternatywnego kompilatora.

Języki C i C++ często są używane do kompilacji statycznej w miejsce innych języków, takich jak Fortran, ze względu na ich wszechobecność i bogatą gamę bibliotek pomocniczych. Kompilator i konwerter (w tym przypadku konwerterem jest narzędzie Cython i inne podobne) mają możliwość analizowania kodu z adnotacją w celu określenia, czy mogą zostać zastosowane kroki optymalizacji statycznej (np. wstawianie funkcji i rozwijanie pętli). Agresywna analiza pośredniego drzewa składni abstrakcyjnej (przeprowadzana przez narzędzia Pythran, Numba i PyPy) zapewnia możliwości łączenia wiedzy o tym, jak w języku Python wyrażane są informacje o najlepszej metodzie wykorzystania napotkanych wzorców w celu przekazania ich bazowemu kompilatorowi.

Analiza przykładu zbioru Julii

W rozdziale 2. dokonano profilowania generatora zbioru Julii. Użyty kod generuje obraz wyjściowy z wykorzystaniem liczb całkowitych i liczb zespolonych. Obliczenia obrazu są powiązane z procesorem.

Główne obciążenie związane z wykonywaniem kodu miało postać powiązanej z procesorem pętli wewnętrznej, która oblicza listę `output`. Lista może mieć postać kwadratowej tablicy pikseli, w której każda wartość reprezentuje koszt wygenerowania piksela.

Kod funkcji wewnętrznej został zaprezentowany w przykładzie 7.1.

Przykład 7.1. Analiza powiązanego z procesorem kodu funkcji zbioru Julii

```
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

W przypadku laptopa jednego z autorów obliczenie oryginalnego zbioru Julii dla siatki 1000×1000 przy wartości `maxit` równej `300` zajęło w przybliżeniu 11 sekund (użyto implementacji czystego kodu Python wykonywanego za pomocą narzędzia CPython 2.7).

Kup ksi k Pole ksi k

Cython

Cython (*http://cython.org/*) to kompilator, który przekształca kod Python z adnotacją typu w skompilowany moduł rozszerzenia. Adnotacje typu przypominają te stosowane w języku C. Takie rozszerzenie może być importowane za pomocą narzędzia `import` jako zwykły moduł języka Python. Choć rozpoczęcie działań nie przysparza trudności, wiąże się z tym konieczność poszerzania wiedzy w coraz większym stopniu wraz z każdym dodatkowym poziomem złożoności i optymalizacji. Przez jednego z autorów narzędzie to jest wykorzystywane do przekształcania funkcji wymagających wielu obliczeń w szybszy kod. Wybrał to narzędzie z powodu jego powszechnego użycia, dojrzałości i obsługi interfejsu OpenMP.

W przypadku standardu OpenMP możliwe jest radzenie sobie z problemami dotyczącymi przetwarzania równoległego poprzez zastosowanie modułów obsługujących wieloprocesorowość, które są uruchamiane w wielu procesorach jednego komputera. Wątki są ukrywane przed kodem Python. Działają za pośrednictwem wygenerowanego kodu C.

Kompilator Cython (opublikowany w 2007 r.) wywodzi się z kompilatora Pyrex (wprowadzonego w 2002 r.). Cython rozszerza możliwości pierwotnych zastosowań kompilatora Pyrex. Biblioteki, które używają kompilatora Cython, to: `scipy`, `scikit-learn`, `lxml` i `zmq`.

Kompilator Cython może być stosowany za pośrednictwem skryptu *setup.py* do kompilacji modułu. Może też zostać użyty interaktywnie w powłoce IPython, co umożliwia „magiczne" polecenie. Adnotacja typów jest zwykle przeprowadzana przez programistę, choć możliwa jest pewna forma zautomatyzowanego tworzenia adnotacji.

Kompilowanie czystego kodu Python za pomocą narzędzia Cython

Prosta metoda rozpoczęcia tworzenia kompilowanego modułu rozszerzenia uwzględnia trzy pliki. W przypadku użycia zbioru Julii jako przykładu są to następujące pliki:

- Plik wywołującego kodu Python (spora część wcześniej przedstawionego kodu zbioru Julii).
- Nowy plik *.pyx* z funkcją do skompilowania.
- Plik *setup.py*, który zawiera instrukcje wywołujące kompilator Cython do utworzenia modułu rozszerzenia.

Przy użyciu tej metody wywoływany jest skrypt *setup.py* w celu wykorzystania kompilatora Cython do skompilowania pliku *.pyx* do postaci skompilowanego modułu. W systemach uniksowych skompilowany moduł będzie prawdopodobnie plikiem *.so*. W systemie Windows powinien to być plik *.pyd* (biblioteka języka Python przypominająca bibliotekę DLL).

W przypadku przykładu zbioru Julii zostaną zastosowane następujące pliki:

- *julia1.py*. Służy do zbudowania list wejściowych i wywołania funkcji obliczeniowej.
- *cythonfn.pyx*. Zawiera funkcję powiązaną z procesorem, dla której można utworzyć adnotacje.
- *setup.py*. Zawiera instrukcje procesu budowania.

Wynikiem uruchomienia skryptu *setup.py* jest możliwy do zaimportowania moduł. W skrypcie *julia1.py* z przykładu 7.2 wymagane jest jedynie wprowadzenie kilku drobnych zmian w celu zaimportowania nowego modułu za pomocą instrukcji `import` i wywołania funkcji.

Kup ksi k Pole ksi k

Przykład 7.2. Importowanie nowo skompilowanego modułu do głównego kodu

```
...
import calculate  # zgodnie z definicją w skrypcie setup.py
...
def calc_pure_python(desired_width, max_iterations):
    # ...
    start_time = time.time()
    output = calculate.calculate_z(max_iterations, zs, cs)
    end_time = time.time()
secs = end_time - start_time
    print "Czas trwania:", secs, "s"
...
```

W przykładzie 7.3 zaczniemy od czystego kodu Python bez adnotacji typu.

Przykład 7.3. Niezmieniony czysty kod Python z pliku cythonfn.pyx (ze zmienionym rozszerzeniem na .py) dla skryptu setup.py kompilatora Cython

```
# cythonfn.pyx
def calculate_z(maxiter, zs, cs):
    """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

Skrypt *setup.py* z przykładu 7.4 jest krótki. Zdefiniowano w nim sposób przekształcenia pliku *cythonfn.pyx* w plik *calculate.so*.

Przykład 7.4. Skrypt setup.py przekształca plik cythonfn.pyx w kod C, który ma zostać skompilowany przez kompilator Cython

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
setup(
        cmdclass = {'build_ext': build_ext},
        ext_modules = [Extension("calculate", ["cythonfn.pyx"])]
        )
```

Po uruchomieniu skryptu *setup.py* z przykładu 7.5 z argumentem `build_ext` kompilator Cython poszuka pliku *cythonfn.pyx* i utworzy plik *calculate.so*.

Przykład 7.5. Uruchamianie skryptu setup.py w celu zbudowania nowo skompilowanego modułu

```
$ python setup.py build_ext --inplace
running build_ext
cythoning cythonfn.pyx to cythonfn.c
building 'calculate' extension
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall
    -Wstrict-prototypes -fPIC -I/usr/include/python2.7 -c cythonfn.c
    -o build/temp.linux-x86_64-2.7/cythonfn.o
gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl,
    -Bsymbolic-functions -Wl,-z,
    relro build/temp.linux-x86_64-2.7/cythonfn.o -o calculate.so
```


Kup ksi k Pole ksi k

Pamiętaj o tym, że jest to krok wykonywany ręcznie. Gdy zaktualizujesz plik *.pyx* lub *setup.py* i zapomnisz ponownie uruchomić polecenie do budowania, nie będzie dostępny zaktualizowany moduł *.so* do zaimportowania. Jeśli nie masz pewności, czy kod został skompilowany, sprawdź znacznik czasu pliku *.so*. W razie wątpliwości usuń wygenerowane pliki kodu C oraz plik *.so*, a następnie zbuduj je ponownie.

Argument `--inplace` nakazuje kompilatorowi Cython zbudowanie skompilowanego modułu w bieżącym katalogu, a nie w osobnym katalogu *build*. Po zakończeniu procesu budowania dostępny będzie plik *cythonfn.c*, który jest raczej mało czytelny, a także plik *calculate.so*.

Po uruchomieniu kodu z pliku *julia1.py* importowany jest skompilowany moduł. Na laptopie jednego z autorów zbiór Julii został obliczony w czasie wynoszącym 8,9 sekundy, a nie w bardziej typowym czasie równym 11 sekund. Jest to niewielki wzrost wydajności kosztem znikomego nakładu pracy.

Użycie adnotacji kompilatora Cython do analizowania bloku kodu

W poprzednim przykładzie pokazano, że możliwe jest szybkie zbudowanie skompilowanego modułu. W przypadku intensywnych pętli i operacji matematycznych już samo to często prowadzi do wzrostu szybkości. Oczywiście nie należy po omacku przeprowadzać optymalizacji. Konieczne jest stwierdzenie, jaka część kodu jest wolna, aby możliwe było zdecydowanie o tym, co wymaga większego nakładu pracy.

Kompilator Cython oferuje opcję tworzenia adnotacji, która zapewnia plik wyjściowy HTML możliwy do wyświetlenia w przeglądarce. Do wygenerowania adnotacji używane jest polecenie `cython -a cythonfn.pyx`, które generuje plik wyjściowy *cythonfn.html*. Po wyświetleniu w przeglądarce zawartość pliku przypomina widoczną na rysunku 7.2. Podobny rysunek jest dostępny w *dokumentacji kompilatora Cython* (*http://docs.cython.org/src/quickstart/cythonize.html*).

```
Generated by Cython 0.21.1

Raw output: cythonfn.c

+01: def calculate_z(maxiter, zs, cs):
 02:     """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
+03:     output = [0] * len(zs)
+04:     for i in range(len(zs)):
+05:         n = 0
+06:         z = zs[i]
+07:         c = cs[i]
+08:         while n < maxiter and abs(z) < 2:
+09:             z = z * z + c
+10:             n += 1
+11:         output[i] = n
+12:     return output
```

Rysunek 7.2. Kolorem wyróżniono dane wyjściowe funkcji bez adnotacji uzyskane za pomocą kompilatora Cython

Dwukrotne kliknięcie każdego wiersza powoduje jego rozwinięcie i wyświetlenie wygenerowanego kodu C. Intensywniejszy żółty kolor oznacza więcej wywołań w obrębie maszyny wirtualnej języka Python, bardziej białe wiersze natomiast wskazują na kod C, który w mniejszym stopniu przypomina kod Python. Celem jest usunięcie jak największej liczby żółtych wierszy i zakończenie działań z jak najmniejszą liczbą białych wierszy.

Kup ksi k Pole ksi k

Choć bardziej żółte wiersze oznaczają więcej wywołań w obrębie maszyny wirtualnej, niekoniecznie spowoduje to wolniejsze działanie kodu. Każde wywołanie w maszynie wirtualnej wiąże się z obciążeniem, ale dla wszystkich takich wywołań będzie ono znaczne tylko w przypadku wywołań występujących wewnątrz dużych pętli. Wywołania poza obrębem dużych pętli (np. wiersz kodu używany do utworzenia listy `output` na początku funkcji) nie są kosztowne w porównaniu z kosztem obliczeń w pętli wewnętrznej. Nie marnuj czasu na wiersze, które nie powodują spowolnienia kodu.

W przykładzie wiersze z największą liczbą wywołań w obrębie maszyny wirtualnej języka Python (najbardziej żółte) mają numery 4 i 8. Na podstawie wyników dotychczasowych operacji profilowania można stwierdzić, że wiersz 8. zostanie prawdopodobnie wywołany ponad 30 milionów razy, dlatego jest znakomitym kandydatem do tego, by się na nim skoncentrować.

Wiersze 9., 10. i 11. są prawie żółte. Ponadto wiadomo, że znajdują się w środku intensywnej pętli wewnętrznej. Ogólnie rzecz biorąc, odpowiadają za sporą część czasu wykonywania funkcji. Z tego powodu w pierwszej kolejności trzeba się nimi zająć. Jeśli musisz przypomnieć sobie, ile czasu trwało wykonywanie tej sekcji kodu, zajrzyj do podrozdziału „Użycie narzędzia line_profiler do pomiarów dotyczących kolejnych wierszy kodu” z rozdziału 2.

Wiersze 6. i 7. są mniej żółte. Ponieważ są wywoływane tylko milion razy, mają znacznie mniejszy wpływ na końcową szybkość. Oznacza to, że później można skupić uwagę na nich. Okazuje się, że ponieważ są one obiektami `list`, właściwie nic nie można zrobić, aby skrócić czas dostępu do nich. Jak wspomniano w podrozdziale „Cython i numpy”, wyjątkiem jest operacja polegająca na zastąpieniu obiektów `list` tablicami narzędzia `numpy`, które zapewnią niewielki przyrost szybkości.

Aby lepiej zrozumieć żółte obszary, możesz rozwinąć każdy wiersz przez dwukrotne kliknięcie. Na rysunku 7.3 widać, że do utworzenia listy `output` iterowana jest długość elementu `zs`. Powoduje to utworzenie obiektów języka Python, w przypadku których maszyna wirtualna tego języka zlicza odwołania. Choć te wywołania są kosztowne, tak naprawdę nie mają wpływu na czas wykonywania tej funkcji.

```
Generated by Cython 0.21.1

Raw output: cythonfn.c

+01: def calculate_z(maxiter, zs, cs):
 02:     """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
+03:     output = [0] * len(zs)
    __pyx_t_1 = PyObject_Length(__pyx_v_zs); if (unlikely(__pyx_t_1 == -1)) {__pyx_filename = __pyx_f[0]; __pyx_l
    __pyx_t_2 = PyList_New(1 * ((__pyx_t_1<0) ? 0:__pyx_t_1)); if (unlikely(!__pyx_t_2)) {__pyx_filename = __pyx_
    __Pyx_GOTREF(__pyx_t_2);
    { Py_ssize_t __pyx_temp;
      for (__pyx_temp=0; __pyx_temp < __pyx_t_1; __pyx_temp++) {
        __Pyx_INCREF(__pyx_int_0);
        PyList_SET_ITEM(__pyx_t_2, __pyx_temp, __pyx_int_0);
        __Pyx_GIVEREF(__pyx_int_0);
      }
    }
    __pyx_v_output = ((PyObject*)__pyx_t_2);
    __pyx_t_2 = 0;
+04:     for i in range(len(zs)):
+05:         n = 0
+06:         z = zs[i]
+07:         c = cs[i]
+08:         while n < maxiter and abs(z) < 2:
+09:             z = z * z + c
+10:             n += 1
+11:         output[i] = n
+12:     return output
```

Rysunek 7.3. Kod C ukryty w wierszu kodu Python

Kup książkę Poleć książkę

Aby poprawić czas wykonywania funkcji, konieczne jest rozpoczęcie deklarowania typów obiektów, które są uwzględniane w pętlach wewnętrznych generujących duże obciążenie. Dzięki temu pętle te mogą tworzyć mniej dość kosztownych wywołań kierowanych do maszyny wirtualnej języka Python. W ten sposób oszczędza się czas.

Ogólnie rzecz biorąc, do wierszy kodu, które prawdopodobnie zajmują najwięcej czasu procesora, zaliczają się następujące:

- wiersze znajdujące się w intensywnych pętlach wewnętrznych,
- wiersze usuwające odwołania do elementów obiektów `list`, `array` lub `np.array`,
- wiersze wykonujące operacje matematyczne.

Jeśli nie wiesz, jakie wiersze są najczęściej wykonywane, wykorzystaj narzędzie do profilowania `line_profiler`, które omówiono w podrozdziale „Użycie narzędzia line_profiler do pomiarów dotyczących kolejnych wierszy kodu” z rozdziału 2. Dowiesz się, jakie wiersze są najczęściej wykonywane, a także które z nich powodują największe obciążenie wewnątrz maszyny wirtualnej języka Python. Dzięki temu uzyskasz wyraźny dowód na to, jakie wiersze wymagają uwagi w celu osiągnięcia najlepszego przyrostu szybkości.

Dodawanie adnotacji typu

Na rysunku 7.2 pokazano, że prawie każdy wiersz funkcji jest wywoływany w maszynie wirtualnej języka Python. Wszystkie obliczenia numeryczne również są wywoływane w tej maszynie, ponieważ używane są obiekty języka Python wyższego poziomu. Konieczne jest przekształcenie tych obiektów w lokalne obiekty języka C, a następnie, po przeprowadzeniu kodowania numerycznego, przekształcenie wyniku z powrotem w obiekt języka Python.

W przykładzie 7.6 widoczny jest sposób dodawania typów podstawowych za pomocą składni słowa kluczowego `cdef`.

Przykład 7.6. Dodawanie typów podstawowych języka C w celu rozpoczęcia przyspieszania działania skompilowanej funkcji. W tym celu używany jest w większym stopniu język C, a w mniejszym kod wykorzystujący maszynę wirtualną języka Python

```
def calculate_z(int maxiter, zs, cs):
    """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
    cdef unsigned int i, n
    cdef double complex z, c
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

Godne uwagi jest to, że takie typy będą zrozumiałe dla kompilatora Cython, lecz *nie* dla interpretera języka Python. Kompilator Cython używa tych typów do przekształcania kodu Python w obiekty języka C, które nie muszą być wywoływane w stosie języka Python. Oznacza to, że operacje są szybsze, ale towarzyszy temu utrata elastyczności i szybkości tworzenia kodu.

Kup książkę Poleć książkę

Dodawane są następujące typy:

- `int` dla liczby całkowitej ze znakiem,
- `unsigned int` dla liczby całkowitej, która może być tylko dodatnia,
- `double complex` dla liczb zespolonych podwójnej precyzji.

Słowo kluczowe `cdef` umożliwia zadeklarowanie zmiennych wewnątrz zawartości funkcji. Musi ono być deklarowane na początku funkcji, ponieważ jest to wymóg specyfikacji języka C.

Podczas dodawania adnotacji kompilatora Cython dodajesz kod inny niż kod Python do pliku *.pyx*. Oznacza to, że rezygnujesz z interaktywności tworzenia kodu Python w interpreterze. Z myślą o osobach zaznajomionych z pisaniem kodu w języku C powracamy do cyklu tworzenie kodu – kompilowanie – uruchamianie – debugowanie.

Możesz zastanawiać się, czy możliwe jest dodanie adnotacji typu do przekazywanych list. Choć używane jest słowo kluczowe `list`, w omawianym przykładzie nie ma to praktycznie żadnego znaczenia. Obiekty `list` nadal muszą być sprawdzane na poziomie interpretera języka Python w celu wyodrębnienia ich zawartości. Jest to bardzo wolna operacja.

Przypisywanie typów niektórym podstawowym obiektom odzwierciedlane jest w danych wyjściowych widocznych na rysunku 7.4. Co ważne, wiersze 11. i 12., czyli dwa z najczęściej wywoływanych wierszy kodu, zmieniły teraz kolor z żółtego na biały. Wskazuje to, że nie są one już wywoływane w maszynie wirtualnej języka Python. W porównaniu z poprzednim przykładem można spodziewać się znacznego wzrostu szybkości. Wiersz 10. jest wywoływany ponad 30 milionów razy, dlatego nadal warto się na nim koncentrować.

```
Generated by Cython 0.21.1

Raw output: cythonfn.c

+01: def calculate_z(int maxiter, zs, cs):
 02:     """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
 03:     cdef unsigned int i, n
 04:     cdef double complex z, c
+05:     output = [0] * len(zs)
+06:     for i in range(len(zs)):
+07:         n = 0
+08:         z = zs[i]
+09:         c = cs[i]
+10:         while n < maxiter and abs(z) < 2:
+11:             z = z * z + c
+12:             n += 1
+13:         output[i] = n
+14:     return output
```

Rysunek 7.4. Pierwsze adnotacje typu

Po skompilowaniu zakończenie wykonywania tej wersji kodu zajmuje 4,3 sekundy. Po wprowadzeniu zaledwie kilku zmian w funkcji uzyskujemy szybkość dwukrotnie większą niż w przypadku oryginalnego kodu Python.

Godne uwagi jest to, że wzrost szybkości wynika z tego, że więcej często wykonywanych operacji kierowanych jest do poziomu kodu C (w tym przypadku są to aktualizacje do wartości zmiennych `z` i `n`). Oznacza to, że kompilator kodu C może optymalizować sposób przetwarzania przez funkcje niskiego poziomu bajtów, które reprezentują te zmienne, bez wywoływania funkcji w stosunkowo wolnej maszynie wirtualnej języka Python.

Kup ksi k Pole ksi k

Na rysunku 7.4 widać, że pętla `while` nadal w pewnym stopniu generuje koszty (ma kolor żółty). Kosztowne wywołanie w maszynie wirtualnej języka Python dotyczy funkcji `abs` na potrzeby liczby zespolonej `z`. Kompilator Cython nie zapewnia wbudowanej funkcji `abs` dla liczb zespolonych. Zamiast niej można udostępnić własne lokalne rozszerzenie.

Jak wspomniano wcześniej w rozdziale, użycie funkcji `abs` dla liczby zespolonej uwzględnia obliczenie pierwiastka kwadratowego sumy kwadratów składowych rzeczywistych i urojonych. W teście pożądane jest sprawdzenie, czy pierwiastek kwadratowy wyniku jest mniejszy niż 2. Zamiast wyznaczania pierwiastka kwadratowego można obliczyć kwadrat drugiej strony porównania. Oznacza to, że `< 2` zostanie przekształcone w `< 4`. Dzięki temu eliminuje się konieczność obliczania pierwiastka kwadratowego jako ostatniej części funkcji `abs`.

Rozpoczęto od postaci:

$$\sqrt{c.real^2 + c.imag^2} < \sqrt{4}$$

Operację uproszczono do następującej postaci:

$$c.real^2 + c.imag^2 < 4$$

Jeśli w poniższym kodzie zostałaby zachowana operacja `sqrt`, w dalszym ciągu byłby zauważalny wzrost szybkości wykonywania. Jednym z sekretów optymalizowania kodu jest sprawienie, aby realizował jak najmniej działań. Dzięki usunięciu stosunkowo kosztownej operacji po zastanowieniu się nad ostatecznym celem funkcji kompilator kodu C będzie mógł wykonać to, z czym sobie dobrze radzi, zamiast próbować „odgadnąć”, jakiego efektu końcowego oczekuje programista.

Tworzenie równoważnego, lecz bardziej wyspecjalizowanego kodu do rozwiązania tego samego problemu, jest określane mianem *redukowania mocy* (ang. *strength reduction*). Kosztem mniejszej elastyczności (i być może czytelności) zyskuje się krótszy czas wykonywania.

To matematyczne rozwinięcie prowadzi do przykładu 7.7, w którym dość kosztowna funkcja `abs` została zastąpiona uproszczonym wierszem rozszerzonych działań matematycznych.

Przykład 7.7. Rozwijanie funkcji abs za pomocą kompilatora Cython

```
def calculate_z(int maxiter, zs, cs):
    """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
    cdef unsigned int i, n
    cdef double complex z, c
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

Tworzenie adnotacji dla kodu pozwala nieznacznie poprawić wydajność instrukcji `while` w wierszu 10. (rysunek 7.5). Obecnie instrukcja obejmuje mniej wywołań w wirtualnej maszynie języka Python. Choć skala wzrostu szybkości, jaki zostanie uzyskany, nie jest od razu oczywista, wiadomo, że wiersz ten jest wywoływany ponad 30 milionów razy. Oznacza to, że przewidywane jest odpowiednie zwiększenie wydajności.

Kup ksi k   Pole ksi k

```
Generated by Cython 0.21.1

Raw output: cythonfn.c

+01: def calculate_z(int maxiter, zs, cs):
 02:     """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
 03:     cdef unsigned int i, n
 04:     cdef double complex z, c
+05:     output = [0] * len(zs)
+06:     for i in range(len(zs)):
+07:         n = 0
+08:         z = zs[i]
+09:         c = cs[i]
+10:         while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
+11:             z = z * z + c
+12:             n += 1
+13:         output[i] = n
+14:     return output
```

Rysunek 7.5. Rozszerzone działania matematyczne pozwalające na ostateczną wygraną w procesie optymalizacji

Ta zmiana ma diametralne znaczenie. Przez zmniejszenie liczby wywołań w najbardziej wewnętrznej pętli znacząco skracany jest czas obliczeniowy funkcji. Czas wykonania nowej wersji kodu wynosi zaledwie 0,25 sekundy, co oznacza niesamowite 40-krotne przyspieszenie w porównaniu z oryginalną wersją kodu.

Kompilator Cython obsługuje kilka metod kompilowania do postaci kodu C. Niektóre z nich są prostsze od opisanej tutaj metody tworzenia pełnej adnotacji typu. Aby ułatwić sobie rozpoczęcie korzystania z kompilatora Cython, należy zaznajomić się z trybem czystego kodu Python, a ponadto przyjrzeć się narzędziu `pyximport`, które ułatwia zaprezentowanie tego kompilatora współpracownikom.

Aby dla omawianej porcji kodu uzyskać dodatkowy możliwy wzrost wydajności, możesz wyłączyć sprawdzanie ograniczeń dla każdego zastąpienia odwołania na liście. Celem sprawdzania ograniczeń jest zapewnienie, że program nie będzie korzystał z danych poza obrębem przydzielonej tablicy. W przypadku kodu C z łatwością można przypadkowo uzyskać dostęp do pamięci poza granicami tablicy, co spowoduje nieoczekiwane wyniki (i prawdopodobnie błąd segmentacji!).

Domyślnie kompilator Cython chroni programistę przed przypadkowym adresowaniem poza granicami listy. Choć taka ochrona wiąże się z niewielkim wykorzystaniem czasu procesorowego, występuje w zewnętrznej pętli funkcji. Z tego powodu sumarycznie nie powoduje znacznego wydłużenia czasu wykonywania. Zwykle bezpieczne jest wyłączenie sprawdzania ograniczeń, o ile nie przeprowadzasz własnych obliczeń związanych z adresowaniem tablicy. W tym przypadku konieczne będzie zadbanie o to, aby nie zostały przekroczone granice listy.

Kompilator Cython oferuje zestaw flag, które mogą być określane na różne sposoby. Najprostszy polega na dodaniu ich jako jednowierszowych komentarzy na początku pliku *.pyx*. Do zmiany tych ustawień możliwe jest też użycie dekoratora lub flagi czasu kompilowania. W celu wyłączenia sprawdzania granic dodajemy dyrektywę kompilatora Cython w obrębie komentarza na początku pliku *.pyx*.

```
#cython: boundscheck=False
def calculate_z(int maxiter, zs, cs):
```


Kup ksi k Pole ksi k

Jak widać, wyłączenie sprawdzania ograniczeń spowoduje tylko nieznaczne skrócenie czasu, ponieważ ma to miejsce w pętli zewnętrznej, a nie wewnętrznej, co jest kosztowniejsze. W przypadku omawianego przykładu nie zapewni to żadnego skrócenia czasu.

Spróbuj wyłączyć sprawdzanie ograniczeń i przepełnienia, jeśli kod powiązany z procesorem znajduje się w pętli, która często zastępuje odwołania dla elementów.

Shed Skin

Shed Skin (*http://code.google.com/p/shedskin/*) to eksperymentalny kompilator Python-C++, który współdziała z językiem Python w wersjach 2.4 – 2.7. Kompilator używa inferencji typów do *automatycznego* sprawdzenia programu Python w celu tworzenia adnotacji typów stosowanych dla każdej zmiennej. Taki kod z adnotacjami jest następnie przekształcany w kod C, aby można go było skompilować za pomocą standardowego kompilatora (np. `g++`). Automatyczna introspekcja to bardzo interesująca funkcja kompilatora Shed Skin. Użytkownik musi jedynie zapewnić przykład prezentujący sposób wywołania funkcji z wykorzystaniem właściwego rodzaju danych, a kompilator sam określi resztę.

Zaletą inferencji typów jest to, że programista nie musi jawnie określać typów. Aby tak było, analizator musi mieć możliwość zidentyfikowania typów dla każdej zmiennej w programie. W bieżącej wersji kompilatora tysiące wierszy kodu Python mogą być automatycznie przekształcane do postaci kodu C. Kompilator korzysta z narzędzia Boehma oczyszczającego pamięć, które umożliwia dynamiczne zarządzanie pamięcią. Narzędzie to jest używane także w przypadku kompilatorów Mono i GNU Compiler for Java. Wadą kompilatora Shed Skin jest to, że dla standardowych bibliotek stosuje zewnętrzne implementacje. Wszystko, co nie zostało zaimplementowane (dotyczy to również narzędzia `numpy`), nie będzie obsługiwane.

Projekt kompilatora Shed Skin zawiera ponad 75 przykładów, w tym wiele modułów matematycznych utworzonych w czystym kodzie Python, a nawet w pełni działający emulator Commodore 64. Każdy z przykładowych kodów działa znacznie szybciej po skompilowaniu za pomocą kompilatora Shed Skin (nawet w porównaniu z uruchomieniem w obrębie narzędzia CPython).

Kompilator Shed Skin może tworzyć odrębne programy wykonywalne, które nie zależą od używanej instalacji interpretera języka Python lub modułów rozszerzeń wykorzystywanych wraz z instrukcją `import` w zwykłym kodzie Python.

Skompilowane moduły zarządzają własną pamięcią. Oznacza to, że pamięć z procesu kodu Python jest kopiowana, a wyniki są z powrotem kopiowane — nie występuje żadne jawne współużytkowanie pamięci. W przypadku dużych bloków pamięci (np. dużej macierzy) koszt wykonywania operacji kopiowania może być znaczny. Przyjrzymy się temu na końcu tego podrozdziału.

Kompilator Shed Skin zapewnia podobny zestaw korzyści co kompilator PyPy (więcej informacji zamieszczono w podrozdziale „PyPy"). Oznacza to, że kompilator PyPy może być łatwiejszy w użyciu, ponieważ nie wymaga żadnych kroków kompilacji. Sposób automatycznego dodawania adnotacji typu przez kompilator Shed Skin może być interesujący dla niektórych użytkowników. Jeśli ponadto zamierzasz modyfikować wynikowy kod C, wygenerowany

Kup ksi k Pole ksi k

kod C może być bardziej czytelny niż kod C utworzony przez kompilator Cython. Podejrzewamy, że kod z automatyczną inferencją typów będzie szczególnie interesujący dla innych twórców kompilatorów w społeczności.

Tworzenie modułu rozszerzenia

W przedstawionym tutaj przykładzie zostanie zbudowany moduł rozszerzenia. Za pomocą instrukcji `import` można zaimportować wygenerowany moduł tak, jak to miało miejsce w przypadku przykładów dotyczących kompilatora Cython. Moduł ten może też zostać skompilowany w postaci odrębnego programu wykonywalnego.

W przykładzie 7.8 zamieszczono kod w osobnym module. Zawiera on zwykły kod Python, dla którego w żaden sposób nie są tworzone adnotacje typu. Zauważ również, że dodano test `__main__`, który powoduje, że moduł ten ma niezależną postać na potrzeby analizy typów. Kompilator Shed Skin może użyć tego bloku `__main__`, który zapewnia przykładowe argumenty, do identyfikacji typów przekazywanych do funkcji `calculate_z`, a także do określenia typów wykorzystywanych wewnątrz funkcji powiązanej z procesorem.

Przykład 7.8. Przenoszenie funkcji powiązanej z procesorem do osobnego modułu (jak w przypadku kompilatora Cython) w celu umożliwienia działania systemu automatycznej inferencji typów kompilatora Shed Skin

```
# shedskinfn.py
def calculate_z(maxiter, zs, cs):
    """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
if __name__ == "__main__":
    # Tworzenie trywialnego przykładu za pomocą poprawnych typów w celu umożliwienia
    # wywołania funkcji przez inferencję typów, aby kompilator Shed Skin mógł analizować typy
    output = calculate_z(1, [0j], [0j])
```

Moduł ten można zaimportować w zwykły sposób (przykład 7.9) zarówno przed skompilowaniem go, jak i po kompilacji. Ponieważ kod nie jest modyfikowany (inaczej niż w przypadku kompilatora Cython), przed kompilacją możliwe jest wywołanie oryginalnego modułu Python. Jeśli kod nie zostanie skompilowany, nie uzyska się przyrostu szybkości, ale możliwe będzie przeprowadzenie debugowania w uproszczony sposób za pomocą zwykłych narzędzi powiązanych z językiem Python.

Przykład 7.9. Importowanie modułu zewnętrznego w celu umożliwienia kompilatorowi Shed Skin skompilowania tylko tego modułu

```
...
import shedskinfn
...
def calc_pure_python(desired_width, max_iterations):
    #...
    start_time = time.time()
```


Kup ksi k   Pole ksi k

```
    output = shedskinfn.calculate_z(max_iterations, zs, cs)
        end_time = time.time()
        secs = end_time - start_time
        print "Czas trwania:", secs, "s"
...
```

Jak zaprezentowano w przykładzie 7.10, możliwe jest sprawienie, by kompilator Shed Skin udostępnił dane wyjściowe z adnotacją związane z jego analizą. Umożliwia to polecenie `shedskin -ann shedskinfn.py`, które generuje plik *shedskinfn.ss.py*. W przypadku kompilowania modułu rozszerzenia konieczne jest jedynie zainicjowanie analizy za pomocą fikcyjnej funkcji `__main__`.

Przykład 7.10. Sprawdzanie danych wyjściowych z adnotacją kompilatora Shed Skin w celu stwierdzenia, jakie typy zostały przez niego zidentyfikowane

```
# shedskinfn.ss.py
def calculate_z(maxiter, zs, cs):        # maxiter: [int],
                                         # zs: [list(complex)],
                                         # cs: [list(complex)]
    """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
    output = [0] * len(zs)               # [list(int)]
    for i in range(len(zs)):             # [__iter(int)]
        n = 0                            # [int]
        z = zs[i]                        # [complex]
        c = cs[i]                        # [complex]
        while n < maxiter and abs(z) < 2: # [complex]
            z = z * z + c                # [complex]
            n += 1                       # [int]
        output[i] = n                    # [int]
    return output                        # [list(int)]
if __name__ == "__main__":               # []
    # Tworzenie trywialnego przykładu za pomocą poprawnych typów w celu umożliwienia
    # wywołania funkcji przez inferencję typów, aby kompilator Shed Skin mógł analizować typy
    output = calculate_z(1, [0j], [0j])  # [list(int)]
```

Po przeanalizowaniu typów `__main__` wewnątrz funkcji `calculate_z` mogą być identyfikowane zmienne, takie jak `z` i `c`, na podstawie obiektów, z którymi prowadzą one interakcję.

Moduł jest kompilowany przy użyciu polecenia `shedskin --extmod shedskinfn.py`. Generowane są następujące pliki:

- *shedskinfn.hpp* (plik nagłówkowy C++),
- *shedskinfn.cpp* (plik źródłowy C++),
- *Makefile*.

Uruchomienie programu `make` powoduje wygenerowanie pliku *shedskinfn.so*. Instrukcja `import shedskinfn` pozwala użyć tego pliku w kodzie Python. Czas wykonywania skryptu `julia1.py` z wykorzystaniem pliku *shedskinfn.so* wynosi 0,4 sekundy. Jest to ogromna poprawa wydajności w porównaniu z wersją bez kompilacji, która wymagała bardzo niewielkiego nakładu pracy.

Tak jak w przypadku kompilatora Cython w przykładzie 7.7, możliwe jest również rozwinięcie funkcji `abs`. Po uruchomieniu tej wersji kodu (ze zmodyfikowanym tylko jednym wierszem funkcji `abs`) i użyciu kilku dodatkowych flag (`--nobounds --nowrap`) ostatecznie uzyskujemy czas wykonywania wynoszący 0,3 sekundy. Choć jest to czas trochę dłuższy (o 0,05 sekundy) niż w przypadku wersji z kompilatorem Cython, *nie było konieczne podawanie wszystkich informacji o typach*. Oznacza to, że eksperymentowanie z wykorzystaniem kompilatora Shed Skin jest bardzo łatwe. Kompilator PyPy uruchamia tę samą wersję kodu z podobną szybkością.

Kup ksi k   Pole ksi k

To, że w przypadku omawianego przykładu kompilatory Cython, PyPy i Shed Skin korzystają z podobnych środowisk wykonawczych, nie oznacza, że uzyskany wynik może zostać uogólniony. Aby w realizowanym projekcie osiągnąć najlepsze czasy wykonywania, trzeba sprawdzić różne narzędzia i przeprowadzić własne eksperymenty.

Kompilator Shed Skin umożliwia określenie dodatkowych flag dotyczących kompilacji, takich jak `-ffast-math` lub `-O3`. W dwóch krokach (w pierwszym gromadzone są statystyki dotyczące wykonywania, a w drugim wygenerowany kod jest optymalizowany na podstawie uzyskanych statystyk) można dodać optymalizację PGO (*Profile-Guided Optimization*) w celu podjęcia próby osiągnięcia dodatkowego wzrostu szybkości. Optymalizacja PGO nie spowodowała jednak przyspieszenia wykonywania kodu dla przykładu zbioru Julii. W praktyce optymalizacja ta często zapewnia niewielki rzeczywisty wzrost wydajności lub żaden.

Należy zauważyć, że domyślnie liczby całkowite są 32-bitowe. Jeśli wymagane są większe zakresy z 64-bitowymi liczbami całkowitymi, podaj flagę `--long`. Należy też unikać przydzielania małych obiektów (np. nowych krotek) w obrębie pętli wewnętrznych, ponieważ proces czyszczenia pamięci nie obsługuje ich tak efektywnie, jak można byłoby oczekiwać.

Koszt związany z kopiami pamięci

W przykładzie kompilator Shed Skin kopiuje do swojego środowiska obiekty `list` języka Python, upraszczając dane do postaci podstawowych typów języka C. Kompilator przekształca następnie wynik funkcji języka C na końcu jej wykonywania z powrotem w obiekt `list` języka Python. Takie przekształcenia i kopiowania zajmują czas. Czy może to oznaczać brakujący czas wynoszący 0,05 sekundy, o którym wspomniano przy okazji poprzedniego wyniku?

Aby określić jedynie koszt związany z kopiowaniem danych do/z funkcji za pośrednictwem kompilatora Shed Skin, można zmodyfikować plik *shedskinfn.py* w celu usunięcia kodu odpowiedzialnego za realizowanie rzeczywistych operacji. Następujący wariant funkcji `calculate_z` to właśnie to, co jest potrzebne:

```
def calculate_z(maxiter, zs, cs):
    """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
    output = [0] * len(zs)
    return output
```

W przypadku wykonywania skryptu `julia1.py` za pomocą tej funkcji szkieletowej czas wynosi w przybliżeniu 0,05 sekundy (oczywiście skrypt nie oblicza poprawnego wyniku!). Czas ten stanowi koszt kopiowania 2 milionów liczb zespolonych do funkcji `calculate_z` oraz ponownego kopiowania z niej miliona liczb całkowitych. Zasadniczo kompilatory Shed Skin i Cython generują ten sam kod maszynowy. Różnica w szybkości wykonywania wynika z tego, że kompilator Shed Skin działa w niezależnym obszarze pamięci, oraz z obciążenia związanego z koniecznością kopiowania danych. Z drugiej strony w przypadku kompilatora Shed Skin nie ma potrzeby tworzenia na początku adnotacji, co zapewnia dość znaczne oszczędności czasu.

Kup ksi k Pole ksi k

Cython i numpy

Obiekty listy (więcej informacji zamieszczono w rozdziale 3.) powodują obciążenie w przypadku każdej operacji zastępowania odwołania, ponieważ przywoływane przez nie obiekty mogą znajdować się w dowolnym miejscu w pamięci. Dla porównania, obiekty tablicy przechowują typy podstawowe w ciągłych blokach pamięci RAM, co pozwala na szybsze adresowanie.

Język Python oferuje moduł `array`, który zapewnia jednowymiarowe przechowywanie typów podstawowych (w tym liczb całkowitych, liczb zmiennoprzecinkowych i łańcuchów Unicode). Moduł `numpy.array` narzędzia `numpy` umożliwia wielowymiarowe przechowywanie oraz oferuje szerszą gamę typów podstawowych, w tym liczby zespolone.

W przypadku iterowania obiektu `array` w sposób możliwy do przewidzenia kompilator może zostać poinstruowany w celu uniknięcia żądania od interpretera języka Python, by obliczył odpowiedni adres. Zamiast tego interpreter może zająć się następnym elementem podstawowym w sekwencji, co polega na bezpośrednim przejściu do jego adresu pamięci. Ponieważ dane są rozmieszczone w ciągłym bloku, trywialnym zadaniem jest obliczenie za pomocą przesunięcia adresu następnego elementu kodu C. Dzięki temu nie ma potrzeby instruowania narzędzia CPython, aby obliczyło taki sam wynik, co wiązałoby się z użyciem wolnego wywołania w obrębie maszyny wirtualnej.

Należy zauważyć, że jeśli zostanie uruchomiona wersja kodu narzędzia `numpy` *bez* żadnych adnotacji kompilatora Cython (czyli kod po prostu zostanie wykonany jako zwykły skrypt Python), zajmie to około 71 sekund. Jest to zdecydowanie gorszy wynik niż dla wersji kodu ze zwykłym obiektem `list` języka Python, którego wykonanie zajęło około 11 sekund. Spowolnienie jest spowodowane obciążeniem wynikającym z zastępowania odwołań dla poszczególnych elementów list narzędzia `numpy`. Nie zostało przewidziane używanie tych list w ten sposób, nawet mimo tego, że dla początkujących programistów może się to wydać intuicyjną metodą obsługi operacji. Kompilowanie kodu eliminuje to obciążenie.

W odniesieniu do tego kompilator Cython oferuje dwie specjalne postaci składni. Starsze wersje kompilatora udostępniają specjalny typ dostępu dla tablic narzędzia `numpy`, a później za pośrednictwem interfejsu `memoryview` wprowadzono ogólny protokół interfejsu bufora. Zapewnia on ten sam niskopoziomowy dostęp do dowolnego obiektu, który implementuje interfejs bufora, uwzględniając tablice narzędzia `numpy` i języka Python.

Dodatkową korzyścią oferowaną przez interfejs bufora jest to, że umożliwia łatwe współużytkowanie bloków pamięci z innymi bibliotekami języka C bez potrzeby przekształcania ich z obiektów języka Python w inną postać.

Blok kodu z przykładu 7.11 przypomina trochę oryginalną implementację, z tym wyjątkiem, że zostały dodane adnotacje interfejsu `memoryview`. Drugi argument funkcji to `double complex[:] zs`. Oznacza to, że używany jest obiekt liczb zespolonych o podwójnej precyzji, który korzysta z protokołu bufora (określony za pomocą znaków `[]`) zawierającego jednowymiarowy blok danych (określony przy użyciu dwukropka `:`).

Kup ksi k   Pole ksi k

Przykład 7.11. Wersja kodu z adnotacjami narzędzia numpy dla funkcji obliczającej zbiór Julii

```
# cython_np.pyx
import numpy as np
cimport numpy as np
def calculate_z(int maxiter, double complex[:] zs, double complex[:] cs):
    """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
    cdef unsigned int i, n
    cdef double complex z, c
    cdef int[:] output = np.empty(len(zs), dtype=np.int32)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

Oprócz podawania argumentów wejściowych przy użyciu składni adnotacji bufora tworzone są też adnotacje dla zmiennej `output` przez przypisanie jej obiektu tablicy jednowymiarowej `array` narzędzia `numpy` za pośrednictwem funkcji `empty`. Wywołanie tej funkcji spowoduje przydzielenie bloku pamięci, ale nie zainicjuje pamięci przy użyciu rozsądnych wartości, dlatego może ona zawierać cokolwiek. Ponieważ zawartość takiej tablicy zostanie nadpisana w pętli wewnętrznej, nie będzie konieczne ponowne przypisywanie tablicy przy użyciu wartości domyślnej. Jest to trochę szybsze niż przydzielanie i ustawianie zawartości tablicy za pomocą wartości domyślnej.

Używając szybszej i bardziej jawnej wersji matematycznej, rozwinięto również wywołanie funkcji `abs`. Czas działania tej wersji wynosi 0,23 sekundy, czyli jest to wynik nieznacznie lepszy niż w przypadku oryginalnej wersji kodu używającego kompilatora Cython, która bazuje na czystym kodzie Python z przykładu 7.7 zbioru Julii. Czysta wersja kodu powoduje obciążenie każdorazowo przy zastępowaniu odwołania dla obiektu `complex` kodu Python, ale operacje te występują w pętli zewnętrznej, dlatego nie mają dużego udziału w czasie wykonywania. Po pętli zewnętrznej tworzone są macierzyste wersje zmiennych, które działają z „szybkością kodu C". Pętla wewnętrzna, zarówno w przypadku przykładowego kodu narzędzia `numpy`, jak i wcześniejszego przykładu czystego kodu Python, realizuje te same działania dla tych samych danych. Oznacza to, że różnica w czasie wykonywania wynika z operacji zastępowania odwołań w pętli zewnętrznej oraz tworzenia tablic `output`.

Przetwarzanie równoległe rozwiązania na jednym komputerze z wykorzystaniem interfejsu OpenMP

W ramach ostatniego kroku rozwijania omawianej wersji kodu przyjrzyjmy się użyciu rozszerzeń języka C++ interfejsu OpenMP do zastosowania przetwarzania równoległego dla trudnego, ale umożliwiającego takie rozwiązanie problemu. Jeśli problem, który rozpatrujesz, jest podobny, możesz szybko skorzystać z wielu rdzeni obecnych w komputerze.

OpenMP (*Open Multi-Processing*) to dobrze zdefiniowany interfejs API dla wielu platform, który obsługuje wykonywanie równoległe i współużytkowanie pamięci dla kodu utworzonego przy użyciu języków C, C++ i Fortran. Interfejs ten wbudowany jest w większość nowoczesnych kompilatorów kodu C. Jeśli kod C zostanie właściwe napisany, przetwarzanie równoległe występuje na poziomie kompilatora. Oznacza to stosunkowo niewielki nakład pracy dla programisty, który korzysta z kompilatora Cython.

Kup ksi k Pole ksi k

W przypadku tego kompilatora interfejs OpenMP może zostać dodany za pomocą operatora `prange` (*parallel range*) i przez dodanie do skryptu *setup.py* dyrektywy kompilatora `-fopenmp`. Działania w obrębie pętli tego operatora mogą być wykonywane równolegle, ponieważ wyłączana jest blokada GIL (*Global Interpreter Lock*).

Przykład 7.12 prezentuje zmodyfikowaną wersję kodu z obsługą operatora `prange`. Instrukcja `with nogil:` określa blok, w którym wyłączana jest blokada GIL. Wewnątrz tego bloku operator `prange` umożliwia pętli `for` przetwarzania równoległego interfejsu OpenMP niezależne obliczenie każdej wartości zmiennej `i`.

Przykład 7.12. Dodawanie operatora prange w celu zastosowania przetwarzania równoległego za pomocą interfejsu OpenMP

```
# cython_np.pyx
from cython.parallel import prange
import numpy as np
cimport numpy as np
def calculate_z(int maxiter, double complex[:] zs, double complex[:] cs):
    """Obliczanie listy output za pomocą reguły aktualizacji zbioru Julii"""
    cdef unsigned int i, length
    cdef double complex z, c
    cdef int[:] output = np.empty(len(zs), dtype=np.int32)
    length = len(zs)
    with nogil:
        for i in prange(length, schedule="guided"):
            z = zs[i]
            c = cs[i]
            output[i] = 0
            while output[i] < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
                z = z * z + c
                output[i] += 1
    return output
```

Podczas wyłączania blokady GIL *nie* można przetwarzać zwykłych obiektów języka Python (np. list). Konieczne jest przetwarzanie wyłącznie obiektów podstawowych i obiektów, które obsługują interfejs `memoryview`. W przypadku przetwarzania równoległego zwykłych obiektów języka Python wymagane byłoby rozwiązanie problemów towarzyszących zarządzaniu pamięcią, których blokada GIL celowo unika. Kompilator Cython nie zapobiega modyfikowaniu obiektów języka Python. Jeśli sam to zrobisz, spowoduje to tylko problemy i zamieszanie!

Aby skompilować plik *cython_np.pyx*, konieczne jest zmodyfikowanie skryptu *setup.py* w sposób pokazany w przykładzie 7.13. Po modyfikacji skrypt instruuje kompilator kodu C o użyciu flagi `-fopenmp` jako argumentu podczas kompilacji w celu włączenia interfejsu OpenMP i połączenia z jego bibliotekami.

Przykład 7.13. Dodawanie do skryptu setup.py flag kompilatora i programu konsolidującego interfejsu OpenMP dla kompilatora Cython

```
#setup.py
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
setup(
        cmdclass = {'build_ext': build_ext},
        ext_modules = [Extension("calculate",
                                 ["cython_np.pyx"],
                                  extra_compile_args=['-fopenmp'],
                                  extra_link_args=['-fopenmp'])]
        )
```


Kup ksi k   Pole ksi k

Operator `prange` kompilatora Cython umożliwia wybranie różnych metod szeregowania. W przypadku opcji `static` obciążenie jest równomiernie rozkładane między dostępne procesory. Część obliczeń wymaga więcej czasu, a część nie. Jeśli kompilator Cython zostanie poinstruowany, aby równomiernie szeregować porcje zadań między procesorami przy użyciu opcji `static`, wyniki dla części obliczeń zostaną uzyskane szybciej niż dla innych. Szybsze wątki przejdą następnie w stan bezczynności.

Dzięki opcjom szeregowania `dynamic` i `guided` można zmniejszyć skalę tego problemu przez dynamiczne przydzielanie zadań w mniejszych porcjach podczas wykonywania kodu. Dzięki temu w przypadku zmiennego czasu obliczeń obciążenie jest równomiernie rozkładane między procesorami. Właściwy wybór dla utworzonego kodu będzie zmieniał się w zależności od natury obciążenia.

Zastosowanie interfejsu OpenMP i opcji `schedule="guided"` pozwala skrócić czas wykonywania w przybliżeniu do 0,07 sekundy. Szeregowanie `guided` spowoduje dynamiczne przydzielanie zadań, dzięki czemu mniej wątków będzie oczekiwać na nowe zadania.

Używając instrukcji `#cython: boundscheck=False`, można też wyłączyć dla omawianego przykładu sprawdzanie ograniczeń, ale nie spowodowałoby to skrócenia czasu wykonywania.

Numba

Narzędzie *Numba* (*http://numba.pydata.org/*) firmy Continuum Analytics to kompilator JIT specjalizujący się w kodzie narzędzia `numpy`, który dokonuje kompilacji tego kodu w czasie wykonywania za pośrednictwem kompilatora LLVM (a *nie*, tak jak we wcześniej prezentowanych przykładach, za pomocą kompilatora `g++` lub `gcc`). Numba nie wymaga kroku prekompilacji, dlatego po uruchomieniu dla nowego kodu kompiluje każdą funkcję z adnotacją, która jest wymagana przez używany sprzęt. Zaletą jest to, że kompilatorowi udostępniany jest dekorator, który informuje go o tym, jakimi funkcjami ma się zająć, po czym Numba zaczyna realizować swoje zadania. Kompilator Numba przeznaczony jest do stosowania dla każdego standardowego kodu narzędzia `numpy`.

Numba to krócej istniejący projekt (w książce użyto wersji 0.13), a związany z nim interfejs API może się nieznacznie zmieniać z każdą wersją. Z tego powodu na chwilę obecną należy traktować go jako bardziej przydatny w środowisku badawczym. Jeśli korzystasz z tablic narzędzia `numpy` i kodu bez wektoryzacji, który dokonuje iteracji dla wielu elementów, kompilator Numba powinien umożliwić Ci uzyskanie szybkiego efektu optymalizacji bez większego nakładu pracy.

Mankamentem związanym z użyciem kompilatora Numba jest łańcuch narzędzi. Korzysta on z kompilatora LLVM, a ponadto ma wiele zależności. Zalecamy zastosowanie dystrybucji Continuum Anaconda, ponieważ zapewnia wszystkie składniki. W przeciwnym razie instalowanie kompilatora Numba w nowym środowisku może być bardzo czasochłonnym zadaniem.

Przykład 7.14 prezentuje dodanie dekoratora `@jit` do podstawowej funkcji zbioru Julii. Nie jest wymagane nic więcej. To, że kompilator `numba` został zaimportowany, oznacza, że mechanizmy kompilatora LLVM zostaną uruchomione w czasie wykonywania w celu skompilowania tej funkcji w tle.

Kup ksi k Pole ksi k

Przykład 7.14. Zastosowanie dekoratora @jit dla funkcji

```
from numba import jit
...
@jit()
def calculate_z_serial_purepython(maxiter, zs, cs, output):
```

Po usunięciu dekoratora `@jit` będzie to jedynie wersja kodu narzędzia `numpy` z demonstracją zbioru Julii obsługiwaną przez interpreter języka Python 2.7. Wykonanie takiego kodu zajmie 71 sekund. Dodanie tego dekoratora powoduje skrócenie czasu wykonywania do 0,3 sekundy. Jest to czas bardzo zbliżony do wyniku osiągniętego w przypadku kompilatora Cython, lecz bez całego nakładu pracy związanego z tworzeniem adnotacji.

Jeśli ta sama funkcja zostanie uruchomiona drugi raz w tej samej sesji interpretera języka Python, zadziała jeszcze szybciej. Nie ma potrzeby kompilowania funkcji docelowej w drugim przejściu, jeśli jednakowe są typy argumentów. W efekcie ogólny czas wykonywania jest krótszy. W przypadku drugiego uruchomienia wynik kompilatora Numba odpowiada wcześniej uzyskanemu wynikowi zastosowania kompilatora Cython z narzędziem `numpy` (a zatem przy znikomym nakładzie pracy kompilator Numba okazał się równie szybki jak Cython!). Kompilator PyPy ma takie same wymagania związane z uruchamianiem.

W przypadku debugowania za pomocą kompilatora Numba warto zauważyć, że można go poinstruować w celu pokazania typu zmiennej, którą się zajmuje w obrębie skompilowanej funkcji. W przykładzie 7.15 widać, że zmienna `zs` jest rozpoznawana przez kompilator JIT jako tablica liczb zespolonych.

Przykład 7.15. Debugowanie identyfikowanych typów

```
print("Zmienna zs ma typ:", numba.typeof(zs))
    array(complex128, 1d, C))
```

Kompilator Numba obsługuje też inne formy introspekcji, takie jak `inspect_types`, która umożliwia przegląd skompilowanego kodu w celu stwierdzenia, gdzie zostały zidentyfikowane informacje o typach. W przypadku braku typów możliwe jest doprecyzowanie, jak wyrażono funkcję, aby ułatwić kompilatorowi Numba określenie większej liczby możliwości inferencji typów.

Płatna wersja kompilatora Numba, czyli *NumbaPro* (*http://docs.continuum.io/numbapro/*), oferuje eksperymentalną obsługę operatora przetwarzania równoległego `prange` z wykorzystaniem interfejsu OpenMP. Dostępna jest również eksperymentalna obsługa układów GPU. Projekt ten ma na celu uproszczenie przekształcania wolniejszego kodu Python z pętlami bazującego na narzędziu `numpy` w bardzo szybki kod, który może być wykonywany w procesorze lub układzie GPU. Kompilator NumbaPro jest wart uwagi.

Pythran

Pythran (*http://pythonhosted.org/pythran/*) to kompilator Python-C++ przeznaczony dla podzbioru instrukcji języka Python, który oferuje częściową obsługę narzędzia `numpy`. Kompilator ten działa trochę podobnie do kompilatorów Numba i Cython. Po utworzeniu przez programistę adnotacji argumentów funkcji kompilator Pythran zajmuje się dalej dodatkowymi adnotacjami typu i specjalizacją kodu. Wykorzystuje możliwości związane z wektoryzacją i przetwarzaniem równoległym opartym na interfejsie OpenMP. Działa wyłącznie w przypadku języka Python 2.7.

Kup ksi k Pole ksi k

Bardzo interesującą funkcją kompilatora Pythran jest to, że próbuje automatycznie wykryć możliwości przetwarzania równoległego (np. w sytuacji, gdy używasz instrukcji `map`) i przekształcić kod w kod przetwarzania równoległego bez konieczności wykonywania przez programistę dodatkowych działań. Możliwe jest też określenie przy użyciu dyrektyw `pragma omp` sekcji przetwarzania równoległego. Pod tym względem sposób działania kompilatora Pythran bardzo przypomina obsługę interfejsu OpenMP przez kompilator Cython.

W tle kompilator Pythran pobiera zarówno zwykły kod Python, jak i kod narzędzia `numpy`, a następnie próbuje agresywnie kompilować je do postaci bardzo szybkiego kodu C++, który zapewnia wyniki jeszcze lepsze od uzyskanych dla kompilatora Cython. Należy zauważyć, że projekt ten jest stosunkowo nowy i mogą w nim występować błędy. Godne uwagi jest także to, że zespół programistów jest bardzo przyjaźnie nastawiony i zwykle usuwa zgłoszone błędy w ciągu kilku godzin.

Ponownie przyjrzyj się równaniu dyfuzji z przykładu 6.9. Część obliczeniową funkcji wyodrębniono do osobnego modułu, aby mogła zostać skompilowana do postaci biblioteki binarnej. Przydatną funkcją kompilatora Pythran jest to, że przy użyciu tego kompilatora *nie jest tworzony kod niezgodny z językiem Python*. Przypomnij sobie, że w przypadku kompilatora Cython konieczne było tworzenie plików *.pyx* z dołączonym kodem Python, który nie mógł być bezpośrednio uruchomiony przez interpreter języka Python. W przypadku kompilatora Pythran dodawane są jednowierszowe komentarze, które mogą zostać przez niego wykryte. Oznacza to, że jeśli zostanie usunięty generowany moduł kompilowany *.so*, można po prostu uruchomić kod przy użyciu interpretera języka Python. Jest to znakomita możliwość w odniesieniu do debugowania.

W przykładzie 7.16 zaprezentowano równanie przewodnictwa cieplnego. Funkcja `evolve` zawiera jednowierszowy komentarz, który dołącza dla niej informacje o typach (ponieważ jest to komentarz, uruchomienie kodu bez kompilatora Pythran sprawi, że interpreter języka Python po prostu go zignoruje). Po załadowaniu kompilator Pythran wykryje ten komentarz i dokona propagacji informacji o typach (bardzo podobnie jak w przypadku narzędzia Shed Skin) w każdej powiązanej funkcji.

Przykład 7.16. Dodawanie jednowierszowego komentarza w celu dołączenia punktu wejścia do funkcji evolve()

```
import numpy as np
def laplacian(grid):
    return np.roll(grid, +1, 0) +
                np.roll(grid, -1, 0) +
                np.roll(grid, +1, 1) +
                np.roll(grid, -1, 1) - 4 * grid
#pythran export evolve(float64[][], float)
def evolve(grid, dt, D=1):
    return grid + dt * D * laplacian(grid)
```

Moduł ten można skompilować za pomocą polecenia `pythran diffusion_numpy.py`, które zwróci plik *diffusion_numpy.so*. Z poziomu funkcji testowej można zaimportować ten nowy moduł i wywołać funkcję `evolve`. Na laptopie jednego z autorów *bez* zainstalowanego kompilatora Pythran czas wykonywania tej funkcji dla siatki 8192×8192 wyniósł 3,8 sekundy. W przypadku kompilatora Pythran czas zmniejszył się do 1,5 sekundy. Oczywiście jeśli kompilator Pythran obsługuje wymagane funkcje, może zapewnić naprawdę imponujące wzrosty wydajności przy bardzo niewielkim nakładzie pracy.

Kup ksi k Pole ksi k

Przyczyną przyspieszenia jest to, że kompilator Pythran używa własnej wersji funkcji `roll`, która cechuje się mniejszą funkcjonalnością. Oznacza to, że przeprowadza kompilację do postaci mniej złożonego kodu, który może działać szybciej. Ponadto kod ten jest mniej elastyczny niż kod narzędzia `numpy` (twórcy kompilatora Pythran zwracają uwagę na to, że implementuje on tylko niektóre elementy narzędzia `numpy`). Pod względem wyników Pythran może jednak prześcignąć inne wcześniej omówione narzędzia.

Zastosujmy teraz tę samą metodę dla przykładu rozszerzonych operacji matematycznych w przypadku zbioru Julii. Samo dodanie jednowierszowej adnotacji do funkcji `calculate_z` powoduje skrócenie czasu wykonywania do 0,29 sekundy, czyli wyniku trochę gorszego niż w przypadku wyniku kompilatora Cython. Dodanie jednowierszowej deklaracji interfejsu OpenMP na początku pętli zewnętrznej powoduje skrócenie czasu wykonywania do 0,1 sekundy. Czas ten nie odbiega zbytnio od najlepszego wyniku uzyskanego dla interfejsu OpenMP kompilatora Cython. Kod z adnotacją został zaprezentowany w przykładzie 7.17.

Przykład 7.17. Dodawanie adnotacji do funkcji calculate_z dla kompilatora Pythran z obsługą interfejsu OpenMP

```
#pythran export calculate_z(int, complex[], complex[], int[])
def calculate_z(maxiter, zs, cs, output):
    #omp parallel for schedule(guided)
    for i in range(len(zs)):
```

Przedstawione dotychczas technologie uwzględniają użycie kompilatora, który towarzyszy zwykłemu interpreterowi CPython. Przyjrzymy się teraz narzędziu PyPy, które oferuje całkowicie nowy interpreter.

PyPy

PyPy (*http://pypy.org/*) to alternatywna implementacja języka Python, która obejmuje śledzący kompilator JIT. Implementacja jest zgodna z językiem Python 2.7. Dostępna jest też eksperymentalna wersja dla języka Python 3.2.

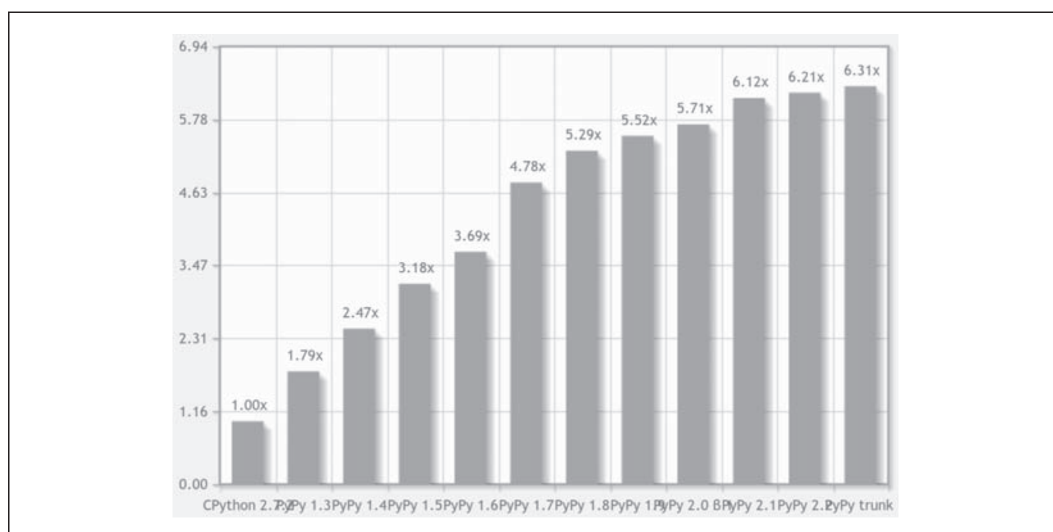
Docelowo narzędzie PyPy zastępuje narzędzie CPython, oferując wszystkie wbudowane moduły. Projekt składa się z łańcucha narzędziowego RPython Translation Toolchain, który służy do budowania interpretera PyPy (i może zostać wykorzystany do tworzenia innych interpreterów). Kompilator JIT w interpreterze PyPy jest bardzo wydajny. Odpowiednie przyspieszenia można zauważyć przy niewielkim lub żadnym nakładzie pracy programisty. W podrozdziale „Interpreter PyPy zapewniający powodzenie systemów przetwarzania danych i systemów internetowych” z rozdziału 12. opisano historię dużego wdrożenia zakończonego sukcesem, które bazowało na interpreterze PyPy.

Interpreter PyPy uruchamia bez żadnych modyfikacji prezentację zbioru Julii bazującą na czystym kodzie Python. W przypadku narzędzi CPython i PyPy czas wykonywania tego kodu wynosi odpowiednio 11 sekund i 0,3 sekundy. Oznacza to, że interpreter PyPy osiąga wynik bardzo zbliżony do wyniku uzyskanego dla przykładowego kodu, dla którego zastosowano kompilator Cython (przykład 7.7). Nie jest z tym związany *zupełnie żaden nakład pracy*, co naprawdę robi wrażenie! Jak zaobserwowano przy okazji omawiania kompilatora Numba, jeśli obliczenia są przeprowadzane ponownie *w tej samej sesji*, drugie i kolejne uruchomienia są szybsze od pierwszego, ponieważ w ich przypadku kompilacja została już wykonana.

Kup ksi k Pole ksi k

Interesujące jest to, że interpreter PyPy obsługuje wszystkie wbudowane moduły. Oznacza to, że moduł `multiprocessing` działa tak jak w przypadku narzędzia CPython. Jeśli zajmujesz się problemem wykorzystującym moduły z dołączonymi bibliotekami, który może być przetwarzany równolegle za pomocą modułu `multiprocessing`, oczekuj, że dostępne będą wszystkie przyrosty szybkości, jakich możesz się spodziewać.

Szybkość interpretera PyPy zwiększała się z czasem. Wykres na rysunku 7.6 uzyskany z serwisu *speed.pypy.org* pozwala zorientować się w dojrzałości interpretera. Pokazane testy szybkości reprezentują szeroki zestaw przypadków użycia, a nie tylko operacje matematyczne. Oczywiste jest to, że interpreter PyPy oferuje większą wydajność niż narzędzie CPython.



Rysunek 7.6. Każda nowa wersja interpretera PyPy oferuje zwiększenie szybkości

Różnice związane z czyszczeniem pamięci

Interpreter PyPy korzysta z procesu czyszczenia pamięci innego typu niż narzędzie CPython. Może to spowodować w kodzie pewne nieoczywiste zmiany w działaniu. Narzędzie CPython używa zliczenia odwołań, interpreter PyPy natomiast korzysta ze zmodyfikowanej metody zaznaczania i usuwania, która może znacznie później oczyścić nieużywany obiekt. Oba warianty są poprawnymi implementacjami specyfikacji języka Python. Trzeba tylko być świadomym tego, że niektóre modyfikacje kodu mogą być niezbędne podczas czyszczenia.

Niektóre metody tworzenia kodu omawiane w odniesieniu do narzędzia CPython zależą do działania licznika odwołań. Dotyczy to zwłaszcza opróżniania plików bez ich jawnego zamknięcia, gdy wykonywane są dla nich operacje otwierania i zapisywania. W przypadku interpretera PyPy ten sam kod zostanie uruchomiony, ale aktualizacje pliku mogą zostać w późniejszym czasie umieszczone na dysku przy następnym uruchomieniu procesu czyszczenia pamięci. Alternatywną formą, która sprawdza się zarówno dla interpretera PyPy, jak i interpretera języka Python, jest użycie menedżera kontekstu, korzystającego z instrukcji `with` do otwierania i automatycznego zamykania plików. Na stronie *Differences between PyPy and CPython* (różnice między interpreterem PyPy i narzędziem CPython) witryny internetowej interpretera PyPy podano odpowiednie szczegóły (*http://pypy.readthedocs.org/en/latest/cpython_differences.html*).

Kup ksi k Pole ksi k

Uruchamianie interpretera PyPy i instalowanie modułów

Jeśli nigdy nie uruchamiałeś alternatywnego interpretera języka Python, powinno Ci pomóc zaznajomienie się z krótkim przykładem. Zakładając, że pobrałeś i rozpakowałeś interpreter PyPy, zobaczysz strukturę folderów zawierającą katalog *bin*. W celu uruchomienia interpretera PyPy użyj polecenia z przykładu 7.18.

Przykład 7.18. Uruchamianie interpretera PyPy w celu stwierdzenia, że implementuje język Python 2.7.3

```
$ ./bin/pypy
Python 2.7.3 (84efb3ba05f1, Feb 18 2014, 23:00:21)
[PyPy 2.3.0-alpha0 with GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
And now for something completely different: ``<arigato> (not thread-safe, but
well, nothing is)''
```

Zauważ, że interpreter PyPy 2.3 działa jako interpreter języka Python 2.7.3. Konieczne jest teraz skonfigurowanie narzędzia `pip`. Pożądane będzie zainstalowanie narzędzia `ipython` (zwróć uwagę, że narzędzie IPython jest uruchamiane z tą samą instalacją interpretera języka Python 2.7.3, o której wcześniej wspomniano). Kroki przedstawione w przykładzie 7.19 są takie same jak te, które zostałyby wykonane w przypadku narzędzia CPython, gdyby zainstalowano narzędzie `pip` bez korzystania z istniejącej dystrybucji lub menedżera pakietów.

Przykład 7.19. Instalowanie narzędzia pip dla interpretera PyPy w celu zainstalowania modułów zewnętrznych, takich jak IPython

```
$ mkdir sources  # tworzenie lokalnego katalogu pobierania
   $ cd sources
# pobieranie dystrybucji i narzędzia pip
$ curl -O http://python-distribute.org/distribute_setup.py
$ curl -O https://raw.github.com/pypa/pip/master/contrib/get-pip.py
# uruchamianie za pomocą polecenia pypy plików instalacyjnych dla pobranych plików
$ ../bin/pypy ./distribute_setup.py
...
$ ../bin/pypy get-pip.py
...
$ ../bin/pip install ipython
...
$ ../bin/ipython
Python 2.7.3 (84efb3ba05f1, Feb 18 2014, 23:00:21)
Type "copyright", "credits" or "license" for more information.
IPython 2.0.0—An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
```

Zauważ, że interpreter PyPy *nie* obsługuje projektów takich jak narzędzie `numpy` w przydatny sposób (istnieje warstwa pomostowa udostępniana przez narzędzie *cpyext* (*https://bitbucket.org/pypy/compatibility/wiki/c-api*), ale jest ono zbyt wolne, aby mogło okazać się wartościowe w przypadku narzędzia `numpy`). Z tego powodu nie należy oczekiwać dużego wsparcia narzędzia `numpy` ze strony interpretera PyPy. Oferuje on eksperymentalny port narzędzia `numpy` o nazwie *numpypy* (instrukcje instalacji są dostępne na *blogu jednego z autorów książki http://ianozsvald.com/2014/01/14/installing-the-numpy-module-in-pypy/*), który jednak nie zapewnia na razie żadnych godnych uwagi wzrostów szybkości[1].[BS1]

[1] Może się to zmienić do końca roku 2014 (więcej informacji pod adresem *http://bit.ly/numpypy*).

Kup ksi k Pole ksi k

Jeśli wymagasz innych pakietów, wszystko, co ma postać czystego kodu Python, *prawdopodobnie* zostanie zainstalowane, a to, co bazuje na bibliotekach rozszerzeń języka C, *raczej* nie będzie działać w przydatny sposób. Interpreter PyPy nie zawiera procesu czyszczącego pamięć ze zliczaniem odwołań. Wszystko, co zostanie skompilowane dla narzędzia CPython, będzie korzystać z wywołań biblioteki, które obsługują proces czyszczący pamięć narzędzia CPython. Interpreter PyPy zapewnia obejście tego problemu, ale wiąże się ono ze znacznym dodatkowym obciążeniem. W praktyce podejmowanie próby wymuszania współpracy starszych bibliotek rozszerzeń bezpośrednio z interpreterem PyPy nie ma żadnej wartości. W przypadku tego interpretera należy w miarę możliwości spróbować usunąć wszelki kod rozszerzenia C (taki kod może istnieć tylko w celu przyspieszenia kodu Python, za co obecnie ma być odpowiedzialny interpreter PyPy). Na stronie wiki dla interpretera PyPy utrzymywana jest *lista zgodnych modułów* (*https://bitbucket.org/pypy/compatibility/wiki/Home*).

Inną wadą interpretera PyPy jest to, że może zużywać wiele pamięci RAM. W tym zakresie każda kolejna wersja interpretera jest lepsza, ale w praktyce może on używać więcej pamięci RAM niż narzędzie CPython. Ponieważ jednak pamięć RAM jest dosyć tania, sensowne jest podjęcie próby poświęcenia jej dla zwiększonej wydajności. Niektórzy użytkownicy zgłosili również *mniejsze* zużycie pamięci RAM podczas korzystania z interpretera PyPy. Jak zawsze, jeśli jest to ważna kwestia, przeprowadź eksperyment przy użyciu reprezentatywnych danych.

Choć interpreter PyPy jest powiązany z blokadą GIL (*Global Interpreter Lock*), zespół programistów realizuje projekt o nazwie STM (*Software Transactional Memory*), który ma na celu podjęcie próby usunięcia wymogu stosowania blokady GIL. Projekt STM przypomina trochę transakcje bazy danych. Jest to mechanizm kontroli współbieżności stosowany dla operacji dostępu do pamięci. Mechanizm ten może wycofać zmiany, jeśli w tym samym obszarze pamięci wystąpią operacje powodujące konflikt. Celem integracji projektu STM jest umożliwienie systemom o wysokim stopniu współbieżności dysponowania pewną formą kontroli współbieżności. Będzie się to wiązać z utratą części efektywności w przypadku operacji, ale w zamian poprawi się wydajność programistów, którzy nie będą zmuszeni do zajmowania się wszystkimi aspektami kontroli jednoczesnego dostępu.

Na potrzeby profilowania polecane narzędzia to *jitviewer* (*https://bitbucket.org/pypy/jitviewer*) i *logparser* (*http://morepypy.blogspot.co.uk/2009/11/hi-all-this-week-i-worked-on-improving.html*).

Kiedy stosować poszczególne technologie?

Jeśli realizujesz projekt o charakterze numerycznym, użycie każdej z opisanych technologii może okazać się przydatne. W tabeli 7.1 podsumowano główne opcje.

Tabela 7.1. Podsumowanie opcji kompilatorów

	Cython	Shed Skin	Numba	Pythran	PyPy
Dojrzałość	T	T	-	-	T
Rozpowszechnienie	T	-	-	-	-
Obsługa narzędzia numpy	T	-	T	T	-
Zmiany w kodzie niepowodujące rozdzielania	-	T	T	T	T
Wymóg znajomości języka C	T	-	-	-	-
Obsługa interfejsu OpenMP	T	-	T	T	T

Kup ksi k Pole ksi k

Jeśli problem mieści się w ograniczonym zakresie obsługiwanych funkcji, kompilator Pythran oferuje prawdopodobnie największe przyrosty szybkości w przypadku problemów rozwiązywanych za pomocą narzędzia `numpy` przy najmniejszym nakładzie pracy. Kompilator ten zapewnia też kilka prostych w użyciu opcji przetwarzania równoległego interfejsu OpenMP. Ponadto jest stosunkowo nowym projektem.

Kompilator Numba może oferować szybkie przyrosty szybkości przy niewielkim nakładzie pracy, ale ma zbyt wiele ograniczeń, które mogą sprawić, że nie będzie dobrze działać dla napisanego kodu. Kompilator ten także jest stosunkowo nowym projektem.

Kompilator Cython oferuje prawdopodobnie najlepsze wyniki w przypadku najszerszej grupy problemów, ale wymaga większego nakładu pracy, a ponadto ponieważ korzysta z kombinacji kodu Python i adnotacji kodu C, jego obsługa jest utrudniona.

Kompilator PyPy stanowi mocną propozycję, jeśli ma zostać przeprowadzona kompilacja do kodu C, a ponadto nie jest używane narzędzie `numpy` ani żadna inna biblioteka zewnętrzna.

Shed Skin może okazać się przydatny, gdy kod ma zostać skompilowany do postaci kodu C, a ponadto nie używasz narzędzia `numpy` lub innych bibliotek zewnętrznych.

Jeśli wdrażasz narzędzie produkcyjne, prawdopodobnie pożądane będzie pozostanie przy dobrze znanych narzędziach. Kompilator Cython powinien być podstawową opcją wyboru. Możesz przeczytać treść podrozdziału „Technika głębokiego uczenia prezentowana przez firmę RadimRehurek.com" z rozdziału 12. W konfiguracjach produkcyjnych używa się też kompilatora PyPy (więcej informacji zawiera podrozdział „Interpreter PyPy zapewniający powodzenie systemów przetwarzania danych i systemów internetowych" z rozdziału 12.).

Jeśli masz do czynienia z niewielkimi wymaganiami numerycznymi, zauważ, że interfejs buforu kompilatora Cython akceptuje macierze `array.array`. Jest to prosta metoda przekazywania bloku danych kompilatorowi Cython w celu przeprowadzenia szybkiego przetwarzania numerycznego bez konieczności dodawania narzędzia `numpy` jako zależności projektu.

Generalnie rzecz biorąc, kompilatory Pythran i Numba to dość nowe projekty, ale bardzo obiecujące. Z kolei kompilator Cython jest bardzo dojrzały. Kompilator PyPy jest uważany za dość dojrzały i zdecydowanie powinien być wykorzystywany w przypadku długotrwałych procesów.

Na zajęciach prowadzonych w 2014 r. przez jednego z autorów zdolny student zaimplementował wersję kodu C algorytmu zbioru Julii. Był rozczarowany, gdy stwierdził, że kod wykonywany był wolniej niż wersja skompilowana za pomocą kompilatora Cython. Okazało się, że student użył 32-bitowych liczb zmiennoprzecinkowych dla komputera 64-bitowego (takie liczby 32-bitowe są wolniej przetwarzane na komputerze 64-bitowym niż 64-bitowe liczby o podwójnej precyzji). Pomimo tego, że student był dobrym programistą używającym języka C, nie wiedział, że coś takiego mogło spowodować zmniejszenie szybkości. Po zmodyfikowaniu kodu okazało się, że wersja kodu bazująca na kompilatorze C, choć znacznie krótsza od wersji automatycznie wygenerowanej za pomocą kompilatora Cython, działała w przybliżeniu z tą samą szybkością. Pisanie czystego kodu C, porównywanie jego szybkości i określanie sposobu jego modyfikacji zajęło więcej czasu niż użycie od razu kompilatora Cython.

Jest to jedynie anegdota. Nie sugerujemy, że kompilator Cython wygeneruje najlepszy kod. Kompetentni programiści tworzący kod w języku C mogą prawdopodobnie stwierdzić, jak sprawić, że *ich* kod będzie działać szybciej od wersji wygenerowanej przez kompilator Cython.

Kup ksi k Pole ksi k

Godne uwagi jest jednak to, że nie będzie bezpieczne przyjęcie, że ręcznie napisany kod C będzie szybszy od przekształconego kodu Python. Zawsze konieczne jest wykonywanie testów porównawczych i podejmowanie decyzji na podstawie uzyskanego dowodu. Kompilatory języka C naprawdę dobrze radzą sobie z przekształcaniem kodu w dość wydajny kod maszynowy. Z kolei język Python sprawdza się naprawdę nieźle w roli języka pozwalającego na opisanie problemu w zrozumiały sposób. Z głową połącz ze sobą te dwie mocne strony.

Inne przyszłe projekty

Na stronie kompilatorów *PyData* (*http://compilers.pydata.org/*) znajduje się lista kompilatorów i narzędzi o dużej wydajności. *Theano* (*http://deeplearning.net/software/theano/*) to język wysokiego poziomu, który umożliwia wyrażanie operatorów matematycznych w tablicach wielowymiarowych. Język ten jest silnie zintegrowany z narzędziem `numpy`, a ponadto może eksportować kod skompilowany dla procesorów i układów GPU. Co interesujące, okazał się przydatny członkom społeczności zajmującej się sztuczną inteligencją z wykorzystaniem techniki głębokiego uczenia. Kompilator *Parakeet* (*http://www.parakeetpython.com/*) koncentruje się na kompilowaniu operacji obejmujących tablice narzędzia `numpy` o dużej gęstości, które korzystają z podzbioru instrukcji języka Python. Obsługuje też układy GPU.

PyViennaCL (*http://viennacl.sourceforge.net/pyviennacl.html*) to powiązanie języka Python z biblioteką algebry liniowej i obliczeń numerycznych ViennaCL. Obsługuje procesory i układy GPU z wykorzystaniem narzędzia `numpy`. Biblioteka ViennaCL napisana w języku C++ generuje kod dla interfejsów CUDA, OpenCL i OpenMP. Obsługuje ona operacje gęstej i rzadkiej algebry liniowej, bibliotekę BLAS i moduły rozwiązujące.

Nuitka (*http://nuitka.net/pages/overview.html*) to kompilator kodu Python, który ma być alternatywą dla zwykłego interpretera CPython, oferując opcję tworzenia skompilowanych plików wykonywalnych. W pełni obsługuje język Python 2.7, choć w naszych testach nie zapewnił żadnych zauważalnych przyrostów szybkości w przypadku prostych testów numerycznych kodu Python.

Pyston (*https://github.com/dropbox/pyston*) to najnowsza branżowa technologia. Korzysta z kompilatora LLVM i jest obsługiwana przez interfejs Dropbox. Z powodu braku obsługi modułów rozszerzenia kompilatora Pyston może dotyczyć ten sam problem co kompilatora PyPy, ale w ramach projektu planowane jest podjęcie próby rozwiązania go. W przeciwnym razie mało prawdopodobne jest, że obsługa narzędzia `numpy` będzie praktycznym rozwiązaniem.

Społeczność programistów nie może raczej narzekać na niedobór opcji kompilacji. Choć wszystkie wymagają kompromisów, oferują też mnóstwo możliwości, dzięki czemu w złożonych projektach może być wykorzystana pełna moc procesorów i architektur wielordzeniowych.

Uwaga dotycząca układów GPU

Układy graficzne GPU (*Graphics Processing Unit*) to obecnie modna technologia. Zdecydowaliśmy się jednak nie omawiać jej co najmniej do następnego wydania książki. Wynika to stąd, że w branży zachodzą szybkie zmiany, a ponadto całkiem prawdopodobne jest to, że wszystko, co zawarliśmy w tej książce, ulegnie zmianie, gdy będziesz w trakcie jej czytania. A na poważnie, nie chodzi o to, że zmiany mogą wymagać wiersze utworzonego kodu, ale o to, że wraz z rozwojem architektur konieczna może okazać się znacząca zmiana sposobu, w jaki będziesz rozwiązywać problemy.

Kup ksi k Pole ksi k

Jeden z autorów zajmował się problemem z fizyki, korzystając przez rok z układu GPU NVIDIA GTX 480 oraz języka Python i środowiska PyCUDA. Po upływie roku została wykorzystana pełna moc układu GPU i system działał 25 razy szybciej niż ta sama funkcja na komputerze z procesorem 4-rdzeniowym. Wariant kodu dla tego procesora został napisany w języku C przy użyciu biblioteki przetwarzania równoległego. Z kolei wariant kodu dla układu GPU był tworzony głównie w języku C architektury CUDA opakowanym w środowisku PyCUDA w celu obsługi danych. Niedługo później pojawiły się w sprzedaży układy GPU z serii GTX 5xx. W efekcie zmianie uległo wiele optymalizacji dotyczących serii 4xx. Efekty prawie rocznej pracy ostatecznie zostały zaprzepaszczone na rzecz łatwiejszego do utrzymania rozwiązania w postaci kodu C, który był wykonywany z wykorzystaniem procesorów.

Choć jest to pojedynczy przykład, zwraca uwagę na zagrożenie wynikające z tworzenia niskopoziomowego kodu dla architektury CUDA (lub OpenCL). Biblioteki bazujące na układach GPU i oferujące funkcje wyższego poziomu ze znacznie większym prawdopodobieństwem będą mogły nadawać się do ogólnego zastosowania (np. biblioteki, które zapewniają interfejsy do analizy obrazu lub transkodowania wideo). Zachęcamy do rozważenia tych kwestii przed sprawdzeniem opcji tworzenia kodu bezpośrednio dla układów GPU.

Projekty, których celem jest automatyczne zarządzanie układami GPU, obejmują narzędzia Numba, Parakeet i Theano.

Oczekiwania dotyczące przyszłego projektu kompilatora

Wśród obecnych opcji kompilatorów dostępnych jest kilka komponentów technologii o dużych możliwościach. Osobiście życzyłbym sobie uogólnienia mechanizmu tworzenia adnotacji kompilatora Shed Skin, aby mógł współpracować z innymi narzędziami (na przykład generując dane wyjściowe zgodne z kompilatorem Cython w celu wygładzenia krzywej uczenia podczas rozpoczynania korzystania z tego kompilatora, a zwłaszcza w przypadku używania narzędzia `numpy`). Kompilator Cython jest dojrzały i integruje się silnie z językiem Python i narzędziem `numpy`. Jeśli krzywa uczenia oraz wymagania dotyczące obsługi nie byłyby tak bardzo zniechęcające, więcej osób zastosowałoby ten kompilator.

W dłuższej perspektywie czasowej życzeniem byłoby pojawienie się rozwiązania przypominającego kompilatory Numba i PyPy, które oferuje działanie w stylu kompilatora JIT zarówno w przypadku zwykłego kodu Python, jak i kodu narzędzia `numpy`. Obecnie nie zapewnia tego żadne narzędzie. Narzędzie rozwiązujące ten problem byłoby mocnym kandydatem do zastąpienia zwykłego interpretera CPython, który aktualnie jest używany przez wszystkich, bez konieczności modyfikowania kodu przez projektantów.

Przyjazna rywalizacja i duży rynek otwarty na nowe pomysły sprawiają, że nasz ekosystem staje się naprawdę wartościowym miejscem.

Interfejsy funkcji zewnętrznych

Czasem zautomatyzowane rozwiązania po prostu nie są odpowiednie, dlatego sam musisz napisać niestandardowy kod w języku C lub Fortran. Może to wynikać z tego, że metody kompilacji nie znajdują pewnych potencjalnych optymalizacji lub wymagane jest wykorzystanie funkcji bibliotek albo języka, które są niedostępne w języku Python. We wszystkich takich przypadkach niezbędne będzie zastosowanie interfejsów funkcji zewnętrznych, które zapewniają dostęp do kodu pisanego i kompilowanego przy użyciu innego języka.

Kup ksi k Pole ksi k

W pozostałej części rozdziału podejmiemy próbę użycia zewnętrznej biblioteki do rozwiązania równania dyfuzji dwuwymiarowej w taki sam sposób jak w rozdziale 6[2]. Przykład 7.20 prezentuje kod tej biblioteki, który może reprezentować zainstalowaną bibliotekę lub być kodem utworzonym własnoręcznie. Metody, którym się przyjrzymy, znakomicie nadają się do pobrania niewielkich części kodu i przemieszczenia ich do innego języka w celu przeprowadzenia specjalistycznych optymalizacji bazujących na języku.

Przykład 7.20. Przykładowy kod C służący do rozwiązywania problemu dyfuzji dwuwymiarowej

```
void evolve(double in[][512], double out[][512], double D, double dt) {
    int i, j;
    double laplacian;
    for (i=1; i<511; i++) {
        for (j=1; j<511; j++) {
            laplacian = in[i+1][j] + in[i-1][j] + in[i][j+1] + in[i][j-1]\
                - 4 * in[i][j];
            out[i][j] = in[i][j] + D * dt * laplacian;
        }
    }
}
```

Aby użyć tego kodu, konieczne jest skompilowanie go do postaci współużytkowanego modułu, który tworzy plik *.so*. W tym celu zostanie zastosowany kompilator `gcc` (lub dowolny inny kompilator języka C) przez wykonanie następujących poleceń:

```
$ gcc -O3 -std=gnu99 -c diffusion.c
$ gcc -shared -o diffusion.so diffusion.o
```

Ostatni plik biblioteki współużytkowanej można umieścić w dowolnym miejscu dostępnym dla kodu Python, ale w przypadku standardowej organizacji plików w systemach uniksowych takie biblioteki są przechowywane w katalogach */usr/lib* i */usr/local/lib*.

ctypes

W przypadku narzędzia CPython[3] najbardziej podstawowy interfejs funkcji zewnętrznej jest dostępny za pośrednictwem modułu `ctypes`. Generalnie rzecz biorąc, moduł ten może być czasami dość ograniczający. Odpowiadasz za zrealizowanie każdego działania. Trochę czasu może zająć Ci upewnienie się, że wszystko jest w porządku. Taki dodatkowy poziom złożoności jest widoczny w kodzie dyfuzji z modułem `ctypes` (przykład 7.21).

Przykład 7.21. Kod dyfuzji dwuwymiarowej z modułem ctypes

```
import ctypes
grid_shape = (512, 512)
_diffusion = ctypes.CDLL("../diffusion.so") # ❶
# Tworzenie odwołań do typów języka C, które będą wymagane do uproszczenia przyszłego kodu
TYPE_INT = ctypes.c_int
TYPE_DOUBLE = ctypes.c_double
TYPE_DOUBLE_SS = ctypes.POINTER(ctypes.POINTER(ctypes.c_double))
# Inicjowanie sygnatury funkcji evolve do postaci:
# void evolve(int, int, double**, double**, double, double)
_diffusion.evolve.argtypes = [
```

2 Dla uproszczenia nie będą implementowane warunki brzegowe.

3 W bardzo dużym stopniu jest to zależne od narzędzia CPython. Inna wersja języka Python może zawierać własne wersje modułu `ctypes`, które mogą działać bardzo różnie.

Kup ksi k Pole ksi k

```
        TYPE_INT,
        TYPE_INT,
        TYPE_DOUBLE_SS,
        TYPE_DOUBLE_SS,
        TYPE_DOUBLE,
        TYPE_DOUBLE,
    ]
    _diffusion.evolve.restype = None
    def evolve(grid, out, dt, D=1.0):
        # Najpierw typy języka Python są przekształcane w odpowiednie typy języka C
        cX = TYPE_INT(grid_shape[0])
        cY = TYPE_INT(grid_shape[1])
        cdt = TYPE_DOUBLE(dt)
        cD = TYPE_DOUBLE(D)
        pointer_grid = grid.ctypes.data_as(TYPE_DOUBLE_SS) # ❷
        pointer_out = out.ctypes.data_as(TYPE_DOUBLE_SS)
        # W tym miejscu możliwe jest wywołanie funkcji
        _diffusion.evolve(cX, cY, pointer_grid, pointer_out, cD, cdt) # ❸
```

❶ Przypomina to importowanie biblioteki `diffusion.so`.

❷ `grid` i `out` to tablice narzędzia `numpy`.

❸ Po całkowitym zakończeniu niezbędnej konfiguracji możliwe jest bezpośrednie wywołanie funkcji języka C.

Pierwszą realizowaną operacją jest „importowanie” biblioteki współużytkowanej. W tym celu używane jest wywołanie `ctypes.CDLL`. W tym wierszu może zostać określona dowolna biblioteka współużytkowana dostępna dla interpretera języka Python (na przykład moduł `ctypes-opencv` ładuje bibliotekę `libcv.so`). W wyniku tego można uzyskać obiekt `_diffusion` zawierający wszystkie elementy składowe znajdujące się w bibliotece współużytkowanej. W przykładzie biblioteka `diffusion.so` zawiera tylko jedną funkcję `evolve`, która nie jest właściwością obiektu. Jeśli biblioteka ta zawierałaby wiele funkcji i właściwości, do wszystkich dostęp byłby możliwy za pośrednictwem obiektu `_diffusion`.

Jednakże nawet pomimo tego, że obiekt `_diffusion` zawiera funkcję `evolve` dostępną w jego obrębie, nie ma informacji o tym, jak z niej skorzystać. W języku C typy są określane statycznie, a funkcja ma bardzo specyficzną sygnaturę. Aby mieć możliwość poprawnego użycia funkcji `evolve`, konieczne jest jawne ustawienie typów argumentów wejściowych i typu zwracanej wartości. Może to okazać się dość żmudne podczas projektowania bibliotek łącznie z interfejsem języka Python lub w przypadku używania szybko zmieniającej się biblioteki. Co więcej, ponieważ moduł `ctypes` nie może sprawdzić, czy zostały mu podane poprawne typy, w przypadku popełnienia błędu bez wyświetlenia żadnych informacji kod może przestać działać lub spowodować błąd segmentacji!

Oprócz ustawienia argumentów i typu zwracanej wartości obiektu funkcji konieczne jest też przekształcenie wszelkich danych, które mają zostać użyte z obiektem (jest to nazywane „rzutowaniem”). Każdy argument wysyłany do funkcji musi zostać poddany uważnemu rzutowaniu do wbudowanego typu języka C. Czasem może to okazać się dość trudne, ponieważ interpreter języka Python bardzo swobodnie traktuje swoje typy zmiennych. Na przykład w przypadku `num1 = 1e5` konieczne byłoby stwierdzenie, czy jest to typ `float` języka Python, co oznaczałoby, że należy użyć typu `ctype.c_float`. Z kolei dla `num2 = 1e30` niezbędne byłoby zastosowanie typu `ctype.c_double`, ponieważ w przeciwnym razie wystąpiłby błąd przepełnienia standardowego typu `float` języka C.

Kup ksi k   Pole ksi k

Narzędzie `numpy` oferuje swoim tablicom właściwość `.ctypes`, która ułatwia zapewnienie zgodności z modułem `ctypes`. Jeśli narzędzie `numpy` nie udostępniałoby takiej funkcjonalności, konieczne byłoby zainicjowanie tablicy poprawnego typu modułu `ctypes`, a następnie znalezienie położenia oryginalnych danych i wskazanie na nie przez nowy obiekt modułu `ctypes`.

Jeśli obiekt, który przekształcasz w obiekt modułu `ctypes`, nie implementuje bufora (podobnie do modułu `array`, tablic narzędzia `numpy`, modułu `cStringIO` itp.), dane będą kopiowane do nowego obiektu. W przypadku rzutowania typu `int` do typu `float` nie ma to dużego wpływu na wydajność kodu. Jeśli jednak rzutowana jest bardzo długa lista w kodzie Python, może się to wiązać ze znacznym obciążeniem! W takich przypadkach pomocne może być użycie modułu `array` lub tablicy narzędzia `numpy`, a nawet zbudowanie własnego obiektu buforowanego za pomocą modułu `struct`. Spowoduje to jednak zmniejszenie czytelności kodu, ponieważ takie obiekty są zwykle mniej elastyczne od ich odpowiedników wbudowanych w język Python.

Może się to skomplikować jeszcze bardziej w przypadku konieczności wysłania bibliotece złożonej struktury danych. Jeśli na przykład biblioteka oczekuje typu złożonego `struct` języka C, który reprezentuje punkt w przestrzeni z właściwościami `x` i `y`, niezbędne będzie zdefiniowanie następującego kodu:

```
from ctypes import Structure
class cPoint(Structure):
    _fields_ = ("x", c_int), ("y", c_int)
```

Dla tego punktu można rozpocząć tworzenie obiektów zgodnych z językiem C, inicjując obiekt cPoint (czyli `point = cPoint(10, 5)`). Nie wiąże się z tym ogromna ilość pracy, ale praca ta może stać się żmudna i powodować uzyskanie niezbyt trwałego kodu. Co się stanie, gdy pojawi się nowa wersja biblioteki, która nieznacznie zmieni strukturę? Spowoduje to, że kod będzie bardzo trudny do utrzymania, a ponadto przestanie być rozwijany. W przypadku takiego kodu programiści po prostu zdecydują się zrezygnować z aktualizowania bazowych bibliotek używanych przez kod.

Z tych powodów użycie modułu `ctypes` jest znakomitą opcją, gdy dysponujesz już dobrą znajomością języka C, a ponadto wymagasz możliwości dostrojenia każdego aspektu interfejsu. Moduł zapewnia znakomite możliwości przenoszenia, ponieważ stanowi część standardowej biblioteki. Jeśli realizowane zadanie jest proste, oferuje proste rozwiązania. Trzeba tylko zachować ostrożność, gdyż złożoność rozwiązań opartych na module `ctypes` (i podobnych niskopoziomowych) może szybko sprawić, że staną się niemożliwe do zarządzania.

cffi

Uświadomienie sobie przez twórców narzędzia `cffi`, że czasami użycie modułu `ctypes` może być dość nieporęczne, skutkuje tym, że narzędzie to podejmuje próbę uproszczenia wielu standardowych operacji wykonywanych przez programistów. W tym celu korzysta z wewnętrznego analizatora składni języka C, który rozpoznaje definicje funkcji i struktur.

W rezultacie możliwe jest po prostu utworzenie kodu C definiującego strukturę biblioteki, która ma zostać użyta. W dalszej kolejności narzędzie `cffi` zajmie się całością żmudnych zadań, czyli importowaniem modułu i upewnieniem się, że dla funkcji wynikowych określono poprawne typy. Okazuje się, że zadania te mogą być niemal trywialne, jeśli dostępny jest kod źródłowy biblioteki. Wynika to stąd, że pliki nagłówkowe (zakończone rozszerzeniem *.h*) będą zawierać wszystkie wymagane, odpowiednie definicje. Przykład 7.22 prezentuje wersję kodu dyfuzji dwuwymiarowej bazującego na narzędziu `cffi`.

Kup książkę Poleć książkę

Przykład 7.22. Kod dyfuzji dwuwymiarowej bazujący na narzędziu cffi

```
from cffi import FFI
ffi = FFI()
ffi.cdef(r'''
   void evolve(
     int Nx, int Ny,
     double **in, double **out,
     double D, double dt
   ); # ❶
''')
lib = ffi.dlopen("../diffusion.so")
def evolve(grid, dt, out, D=1.0):
    X, Y = grid_shape
    pointer_grid = ffi.cast('double**', grid.ctypes.data) # ❷
    pointer_out = ffi.cast('double**', out.ctypes.data)
    lib.evolve(X, Y, pointer_grid, pointer_out, D, dt)
```

❶ Zawartość tej definicji może być standardowo uzyskana z dokumentacji używanej biblioteki lub po sprawdzeniu jej plików nagłówkowych.

❷ Choć wymagane jest jeszcze rzutowanie obiektów, które nie są wbudowane w język Python, w celu użycia ich z modułem kodu C, składnia będzie wyglądać znajomo dla osób mających doświadczenie z zakresu języka C.

Inicjalizacja narzędzia `cffi` w poprzednim kodzie może być traktowana jako proces dwukrokowy. Najpierw tworzony jest obiekt `FFI`, dla którego są podawane wszystkie niezbędne globalne deklaracje języka C. Może to obejmować typy danych, a także sygnatury funkcji. Dalej możliwe jest zaimportowanie za pomocą funkcji `dlopen` biblioteki współużytkowanej do jej własnej przestrzeni nazw, która jest podrzędną przestrzenią obiektu `FFI`. Oznacza to, że możliwe byłoby załadowanie dwóch bibliotek za pomocą tej samej funkcji `evolve` do zmiennych `lib1` i `lib2`, a następnie użycie ich niezależnie (jest to znakomita opcja w przypadku debugowania i profilowania!).

Oprócz zwykłego importowania biblioteki współużytkowanej języka C narzędzie `cffi` umożliwia po prostu napisanie kodu C, który zostanie skompilowany za pomocą kompilatora JIT i funkcji `verify`. Coś takiego zapewnia wiele natychmiastowych korzyści. Z łatwością możesz ponownie utworzyć niewielkie porcje kodu C bez wywoływania pokaźnych mechanizmów osobnej biblioteki języka C. Jeśli istnieje biblioteka, której chcesz użyć, ale do idealnego działania interfejsu niezbędny jest kod łączący napisany w języku C, alternatywnie możesz po prostu wstawić go do kodu narzędzia `cffi` (przykład 7.23). Dzięki temu wszystko będzie znajdować się w centralnym miejscu. Ponieważ kod jest kompilowany za pomocą kompilatora JIT, możesz określić instrukcje kompilowania dla każdej porcji kodu, która wymaga skompilowania. Zauważ jednak, że z taką kompilacją związane jest jednorazowe obciążenie, które występuje za każdym razem, gdy uruchamiana jest funkcja `verify` w celu faktycznego przeprowadzenia kompilacji.

Przykład 7.23. Kod narzędzia cffi z wstawionym kodem dyfuzji dwuwymiarowej

```
ffi = FFI()
ffi.cdef(r'''
    void evolve(
        int Nx, int Ny,
        double **in, double **out,
        double D, double dt
    );
''')
```


Kup ksi k   Pole ksi k

```
lib = ffi.verify(r'''
void evolve(int Nx, int Ny,
        double in[][Ny], double out[][Ny],
        double D, double dt) {
    int i, j;
    double laplacian;
    for (i=1; i<Nx-1; i++) {
        for (j=1; j<Ny-1; j++) {
            laplacian = in[i+1][j] + in[i-1][j] + in[i][j+1] + in[i][j-1]\
                - 4 * in[i][j];
            out[i][j] = in[i][j] + D * dt * laplacian;
        }
    }
}
''', extra_compile_args=["-O3",]) # ❶
```

❶ Ponieważ kod jest kompilowany za pomocą kompilatora JIT, możliwe jest też podanie odpowiednich flag kompilacji.

Inną korzyścią z funkcjonalności funkcji `verify` jest to, że świetnie radzi sobie ona ze złożonymi instrukcjami `cdef`. Jeśli na przykład zostałaby użyta biblioteka z bardzo skomplikowaną strukturą, ale pożądane byłoby zastosowanie tylko jej części, można byłoby skorzystać z częściowej definicji typu `struct`. W tym celu w definicji typu `struct` dodajemy łańcuch `...` w bloku `ffi.cdef` oraz instrukcję `#include` dla odpowiedniego pliku nagłówkowego w zamieszczonej dalej w kodzie funkcji `verify`.

Dla przykładu załóżmy, że korzystamy z biblioteki z plikiem nagłówkowym `complicated.h` zawierającym strukturę o następującej postaci:

```
struct Point {
    double x;
    double y;
    bool isActive;
    char *id;
    int num_times_visited;
}
```

Jeśli interesowałyby nas tylko właściwości `x` i `y`, moglibyśmy utworzyć prosty kod narzędzia `cffi`, który zajmie się wyłącznie wartościami tych właściwości:

```
from cffi import FFI
ffi = FFI()
ffi.cdef(r"""
    struct Point {
        double x;
        double y;
        ...;
    };
    struct Point do_calculation();
""")
lib = ffi.verify(r"""
    #include <complicated.h>
""")
```

W dalszej kolejności można uruchomić funkcję `do_calculation` z biblioteki pliku nagłówkowego `complicated.h`, która zwróci obiekt `Point` z jego dostępnymi właściwościami `x` i `y`. Pod kątem możliwości przenoszenia jest to znakomite rozwiązanie, gdyż taki kod będzie działać bez zarzutu w systemach z różną implementacją obiektu Point lub po pojawieniu się nowych wersji pliku `complicated.h`, pod warunkiem że wszystkie będą mieć właściwości `x` i `y`.

Kup ksi k   Pole ksi k

Wszystko to sprawia, że `cffi` to naprawdę znakomite narzędzie, gdy używasz kodu C w kodzie Python. Jest znacznie prostsze niż moduł `ctypes`, a jednocześnie oferuje taki sam poziom szczegółowej kontroli, która może być pożądana podczas pracy bezpośrednio z interfejsem funkcji zewnętrznej.

f2py

W przypadku wielu zastosowań naukowych język Fortran w dalszym ciągu ma status złotego standardu. Choć minęły już czasy, gdy pełnił rolę języka do ogólnych zastosowań, nadal oferuje wiele pożytecznych funkcji, które ułatwiają dość szybkie tworzenie operacji wektorowych. Ponadto w języku Fortran napisano wiele wydajnych bibliotek matematycznych (*LAPACK* (*http://www.netlib.org/lapack/*), *BLAS* (*http://www.netlib.org/blas/*) itp.). Kluczowe znaczenie może mieć możliwość użycia ich w tworzonym wydajnym kodzie Python.

W takich sytuacjach moduł `f2py` zapewnia wyjątkowo prostą metodę importowania kodu Fortran do kodu Python. Moduł ten może być tak łatwy w obsłudze z powodu przejrzystości typów w języku Fortran. Ponieważ typy mogą być bez problemu analizowane i rozpoznawane, moduł `f2py` bez trudu może sprawić, że moduł CPython, który korzysta z wbudowanej w język C obsługi funkcji zewnętrznych, użyje kodu Fortran. Oznacza to, że podczas stosowania modułu `f2py` w rzeczywistości ma miejsce automatyczne generowanie modułu C, który potrafi użyć kodu Fortran! W efekcie wiele niejasności związanych z rozwiązaniami bazującymi na narzędziach `ctypes` i `cffi` po prostu nie istnieje.

Przykład 7.24 prezentuje prosty kod zgodny z modułem `f2py`, który służy do rozwiązywania równania dyfuzji. Okazuje się, że cały wbudowany kod Fortran jest zgodny z tym modułem. Jednakże adnotacje argumentów funkcji (instrukcje poprzedzone przez `!f2py`) upraszczają wynikowy moduł Python i przyczyniają się do uzyskania interfejsu prostszego w użyciu. Adnotacje niejawnie informują moduł `f2py` o tym, czy argumentem mają być jedynie dane wyjściowe, czy tylko dane wejściowe, czy też ma nim być coś, co ma zostać zmodyfikowane lokalnie lub całkowicie ukryte. Ukryty typ jest szczególnie przydatny w przypadku wielkości wektorów: w kodzie Fortran może być wymagane jawne określenie tych wartości, w kodzie Python natomiast takie informacje są od razu dostępne. Po ustawieniu typu jako ukrytego moduł `f2py` może automatycznie określić te wartości, co zasadniczo oznacza ukrywanie ich w ostatecznym interfejsie Python.

Przykład 7.24. Kod Fortran dyfuzji dwuwymiarowej z adnotacjami modułu f2py

```
SUBROUTINE evolve(grid, next_grid, D, dt, N, M)
    !f2py threadsafe
    !f2py intent(in) grid
    !f2py intent(inplace) next_grid
    !f2py intent(in) D
    !f2py intent(in) dt
    !f2py intent(hide) N
    !f2py intent(hide) M
    INTEGER :: N, M
    DOUBLE PRECISION, DIMENSION(N,M) :: grid, next_grid
    DOUBLE PRECISION, DIMENSION(N-2, M-2) :: laplacian
    DOUBLE PRECISION :: D, dt
    laplacian = grid(3:N, 2:M-1) + grid(1:N-2, 2:M-1) + &
                grid(2:N-1, 3:M) + grid(2:N-1, 1:M-2) - 4 * grid(2:N-1, 2:M-1)
    next_grid(2:N-1, 2:M-1) = grid(2:N-1, 2:M-1) + D * dt * laplacian
END SUBROUTINE evolve
```


Kup ksi k   Pole ksi k

W celu przekształcenia kodu w moduł Python zostanie uruchomione następujące polecenie:

```
$ f2py -c -m diffusion --fcompiler=gfortran --opt='-O3' diffusion.f90
```

Spowoduje to utworzenie pliku *diffusion.so*, który może zostać zaimportowany bezpośrednio do kodu Python.

Poeksperymentowanie z wynikowym modułem w trybie interaktywnym pozwala zauważyć korzyści, jakie zapewnił moduł `f2py` dzięki użyciu adnotacji, a także możliwość analizowania kodu Fortran:

```
In [1]: import diffusion
In [2]: diffusion?
Type:        module
String form: <module 'diffusion' from 'diffusion.so'>
File:        .../examples/compilation/f2py/diffusion.so
Docstring:
This module 'diffusion' is auto-generated with f2py (version:2).
Functions:
  evolve(grid,next_grid,d,dt)
.
In [3]: diffusion.evolve?
Type:        fortran
String form: <fortran object>
Docstring:
evolve(grid,next_grid,d,dt)
Wrapper for ``evolve``.
Parameters
grid : input rank-2 array('d') with bounds (n,m)
next_grid :  rank-2 array('d') with bounds (n,m)
d : input float
dt : input float
```

Powyższe dane pokazują, że wynik generowania przeprowadzonego przez moduł `f2py` jest automatycznie dokumentowany, a interfejs jest dość uproszczony. Na przykład zamiast zmuszać nas do wyodrębniania wielkości wektorów, moduł `f2py` sprawdza, jak automatycznie znaleźć te informacje, i po prostu ukrywa je w wynikowym interfejsie. Okazuje się, że wynikowa funkcja `evolve` w swojej sygnaturze wygląda tak samo jak wersja czystego kodu Python utworzonego w przykładzie 6.14.

Jedyną rzeczą, o którą trzeba zadbać, jest uporządkowanie tablic narzędzia `numpy` w pamięci. Ponieważ większość zadań realizowanych z wykorzystaniem narzędzia `numpy` i języka Python skupia się na kodzie uzyskanym z kodu C, zawsze stosuj konwencję języka C dotyczącą uporządkowania danych w pamięci (określanego mianem *głównego uporządkowania wierszowego*). W języku Fortran wykorzystywana jest inna konwencja (*główne uporządkowanie kolumnowe*), która musi być przestrzegana przez używane wektory. Uporządkowania te po prostu określają, czy dla tablicy dwuwymiarowej kolumny lub wiersze są ciągłe w pamięci[4]. Na szczęście sprowadza się to jedynie do określenia parametru `order='F'` dla narzędzia `numpy` podczas deklarowania wektorów.

---

[4] Więcej informacji zawiera strona serwisu Wikipedia pod następującym adresem: *http://en.wikipedia.org/wiki/Row-major_order*.

Kup ksi k   Pole ksi k

Różnica między głównym uporządkowaniem wierszowym i kolumnowym oznacza, że macierz [[1, 2], [3, 4]] zostanie zapisana w pamięci w postaci [1, 2, 3, 4] (główne uporządkowanie wierszowe) oraz jako [1, 3, 2, 4] w przypadku głównego uporządkowania kolumnowego. Różni się to jedynie konwencją. Jeśli jest właściwie stosowana, w rzeczywistości nie powoduje żadnych negatywnych konsekwencji dotyczących wydajności.

W rezultacie uzyskujemy poniższy kod, który umożliwia użycie podprocedury kodu Fortran. Kod jest taki sam jak użyty w przykładzie 6.14. Różni się jedynie importowaniem z biblioteki bazującej na module `f2py` i jawnym uporządkowaniem danych zgodnie z wymaganiami języka Fortran:

```
from diffusion import evolve
def run_experiment(num_iterations):
    next_grid = np.zeros(grid_shape, dtype=np.double, order='F') # ❶
    grid = np.zeros(grid_shape, dtype=np.double, order='F')
    # ...standardowa inicjalizacja...
    for i in range(num_iterations):
        evolve(grid, next_grid, 1.0, 0.1)
        grid, next_grid = next_grid, grid
```

❶ W przypadku języka Fortran liczby są porządkowane w pamięci w odmienny sposób, dlatego trzeba pamiętać o ustawieniu tablic narzędzia `numpy` pod kątem korzystania z tego standardu.

Moduł narzędzia CPython

Zawsze możliwe jest przejście bezpośrednio do poziomu interfejsu API narzędzia CPython i utworzenie jego modułu. Wymaga to napisania kodu w ten sam sposób, w jaki zaprojektowano narzędzie CPython, a także zadbania o wszystkie interakcje między kodem i implementacją narzędzia CPython.

Zaletą tego rozwiązania są jego niebywałe możliwości dotyczące przenoszenia (zależnie od wersji języka Python). Nie są konieczne żadne moduły lub biblioteki zewnętrzne, ale jedynie kompilator kodu C i język Python! Niemniej jednak rozwiązanie to niekoniecznie zapewnia odpowiednie skalowanie w przypadku nowych wersji języka Python. Na przykład moduły narzędzia CPython utworzone dla języka Python 2.7 współpracują z językiem Python 3.

Z możliwościami przenoszenia wiąże się jednak duży koszt. Odpowiadasz za każdy aspekt interfejsu między kodem Python i modułem. Może to oznaczać, że nawet najprostsze zadania będą liczyć dziesiątki wierszy kodu. Aby na przykład uzyskać interfejs z biblioteką procesu dyfuzji z przykładu 7.20, konieczne jest napisanie 28 wierszy kodu tylko w celu wczytania argumentów do funkcji i poddania ich analizie (przykład 7.25). Oczywiście oznacza to, że masz możliwość niezwykle dokładnego kontrolowania tego, co ma miejsce. Sprowadza się to nawet do możliwości ręcznej zmiany liczników odwołań dla procesu czyszczenia pamięci kodu Python (może to być przyczyną wielu utrudnień podczas tworzenia modułów narzędzia CPython, które korzystają z wbudowanych typów języka Python). Z tego powodu wynikowy kod będzie zwykle nieznacznie szybszy niż w przypadku innych metod interfejsu.

Kup ksi k Pole ksi k

Przykład 7.25. Moduł narzędzia CPython zapewniający interfejs dla biblioteki dyfuzji dwuwymiarowej

```
// python_interface.c
// – interfejs modułu narzędzia CPython dla pliku diffusion.c
#define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION
#include <Python.h>
#include <numpy/arrayobject.h>
#include "diffusion.h"
/* Literały docstring */
static char module_docstring[] =
    "Zapewnia zoptymalizowaną metodę rozwiązywania równania dyfuzji";
static char cdiffusion_evolve_docstring[] =
    "Rozwijanie siatki dwuwymiarowej za pomocą równania dyfuzji";
PyArrayObject* py_evolve(PyObject* self, PyObject* args) {
    PyArrayObject* data;
    PyArrayObject* next_grid;
    double dt, D=1.0;
    /* Funkcja „rozwijania” będzie mieć sygnaturę:
     *      evolve(data, next_grid, dt, D=1)
     */
    if (!PyArg_ParseTuple(args, "OOd|d", &data, &next_grid, &dt, &D)) {
        PyErr_SetString(PyExc_RuntimeError, "Niepoprawne argumenty");
        return NULL;
    }
    /* Sprawdzenie, czy tablice narzędzia numpy są ciągłe w pamięci */
    if (!PyArray_Check(data) || !PyArray_ISCONTIGUOUS(data)) {
        PyErr_SetString(PyExc_RuntimeError,"Tablica data nie jest ciągłą tablicą.");
        return NULL;
    }
    if (!PyArray_Check(next_grid) || !PyArray_ISCONTIGUOUS(next_grid)) {
        PyErr_SetString(PyExc_RuntimeError,"Tablica next_grid nie jest ciągłą tablicą.");
        return NULL;
    }
    /* Sprawdzenie, czy tablice siatki data i next_grid są takiego samego typu i mają identyczne wymiary */
    if (PyArray_TYPE(data) != PyArray_TYPE(next_grid)) {
        PyErr_SetString(PyExc_RuntimeError,
                        "Tablice next_grid i data powinny być tego samego typu.");
        return NULL;
    }
    if (PyArray_NDIM(data) != 2) {
        PyErr_SetString(PyExc_RuntimeError,"Tablica data powinna być dwuwymiarowa.");
        return NULL;
    }
    if (PyArray_NDIM(next_grid) != 2) {
        PyErr_SetString(PyExc_RuntimeError,"Tablica next_grid powinna być dwuwymiarowa.");
        return NULL;
    }
    if ((PyArray_DIM(data,0) != PyArrayDim(next_grid,0)) ||
        (PyArray_DIM(data,1) != PyArrayDim(next_grid,1))) {
        PyErr_SetString(PyExc_RuntimeError,
                        "Tablice data i next_grid muszą mieć takie same wymiary.");
        return NULL;
    }
    /* Pobranie wielkości przetwarzanej siatki */
    const int N = (int) PyArray_DIM(data, 0);
    const int M = (int) PyArray_DIM(data, 1);
    evolve(
        N,
        M,
        PyArray_DATA(data),
        PyArray_DATA(next_grid),
        D,
```


Kup ksi k   Pole ksi k

```
        dt
    );
    Py_XINCREF(next_grid);
    return next_grid;
}
/* Specyfikacja modułu */
static PyMethodDef module_methods[] = {
/* { nazwa metody, funkcja języka C, typy argumentów, docstring              } */
   { "evolve"     , py_evolve    , METH_VARARGS    , cdiffusion_evolve_docstring }    ,
   { NULL         , NULL         , 0               , NULL                        }
};
/* Inicjowanie modułu */
PyMODINIT_FUNC initcdiffusion(void)
{
    PyObject *m = Py_InitModule3("cdiffusion", module_methods, module_docstring);
    if (m == NULL)
        return;
/* Ładowanie funkcji narzędzia numpy */
    import_array();
}
```

Podsumowując, z tej metody należy korzystać tylko w ostateczności. Choć zapewnia sporo informacji podczas tworzenia modułu narzędzia CPython, wynikowy kod nie oferuje takich możliwości ponownego wykorzystania lub utrzymywania jak inne potencjalne metody. Wprowadzanie minimalnych zmian w module wymaga często jego całkowitej przebudowy. Tak naprawdę kod modułu i plik *setup.py* wymagany do jego kompilacji (przykład 7.26) dołączamy ku przestrodze.

Aby utworzyć powyższy kod, konieczne jest napisanie skryptu *setup.py*, który używa modułu `distutils` do określenia sposobu budowania kodu zgodnego z językiem Python (przykład 7.26). Oprócz standardowego modułu `distutils` narzędzie `numpy` zapewnia własny moduł, aby ułatwić dodawanie integracji narzędzia `numpy` do modułów narzędzia CPython.

Przykład 7.26. Plik konfiguracyjny dla interfejsu dyfuzji modułu narzędzia CPython

```
"""
Plik setup.py dla modułu dyfuzji narzędzia CPython. Rozszerzenie może zostać utworzone za pomocą polecenia:
  $ python setup.py build_ext --inplace
Utworzy ono plik __cdiffusion.so__, który może być importowany bezpośrednio do
kodu Python
"""
from distutils.core import setup, Extension
import numpy.distutils.misc_util
__version__ = "0.1"
cdiffusion = Extension(
    'cdiffusion',
    sources = ['cdiffusion/cdiffusion.c', 'cdiffusion/python_interface.c'],
    extra_compile_args = ["-O3", "-std=c99", "-Wall", "-p", "-pg", ],
    extra_link_args = ["-lc"],
)
setup (
    name = 'diffusion',
    version = __version__,
    ext_modules = [cdiffusion,],
    packages = ["diffusion", ],
    include_dirs = numpy.distutils.misc_util.get_numpy_include_dirs(),
)
```


Kup ksi k Pole ksi k

Wynikiem wykonania tego kodu jest plik `cdiffusion.so`, który może być importowany bezpośrednio z kodu Python i całkiem łatwo używany. Ponieważ uzyskano pełną kontrolę sygnatury funkcji wynikowej, a także tego, jak dokładnie kod C prowadził interakcję z biblioteką, możliwe było (po wykonaniu sporej ilości pracy) stworzenie prostego do zastosowania modułu:

```
from cdiffusion import evolve
def run_experiment(num_iterations):
    next_grid = np.zeros(grid_shape, dtype=np.double)
    grid = np.zeros(grid_shape, dtype=np.double)
# ...standardowa inicjalizacja...
    for i in range(num_iterations):
        evolve(grid, next_grid, 1.0, 0.1)
        grid, next_grid = next_grid, grid
```

Podsumowanie

Różne strategie zaprezentowane w tym rozdziale umożliwiają dostosowanie kodu w różnym stopniu w celu zmniejszenia liczby instrukcji, jakie procesor musi wykonać, a także zwiększenia efektywności programów. Choć czasem można to uzyskać w sposób algorytmiczny, często trzeba zrobić to ręcznie (zajrzyj do podrozdziału „Porównanie kompilatorów JIT i AOT"). Ponadto metody te należy sporadycznie zastosować po prostu w celu użycia bibliotek, które zostały już utworzone w innych językach. Niezależnie od motywacji język Python pozwala wykorzystać wzrost wydajności możliwy do uzyskania przy użyciu innych języków w przypadku niektórych problemów przy jednoczesnym zachowaniu szczegółowości i elastyczności, jeśli jest to wymagane.

Godne uwagi jest jednak to, że opisane optymalizacje są przeprowadzane w celu poprawienia wydajności wyłącznie instrukcji procesora. W przypadku operacji wejścia-wyjścia powiązanych z problemem bazującym na użyciu procesora skompilowanie kodu może nie zapewnić rozsądnych przyspieszeń. W odniesieniu do takich problemów konieczne będzie ponowne przeanalizowanie dostępnych rozwiązań i ewentualne użycie przetwarzania równoległego, które pozwala na jednoczesne uruchamianie różnych zadań.

Kup ksi k Pole ksi k

Skorowidz

A

adnotacje typu, 144
aktualizowanie
  klastra, 252
  siatki, 109
algorytm
  HyperLogLog, 302
  HyperLogLog++, 290
  LogLog, 301
  odchylenia standardowego, 98
  online, 98
  sortowania, 72
  techniki głębokiego uczenia, 313
  word2vec, 313
  wyszukiwania, 72
alokacja, 109
  pamięci, 110, 111, 113
  pamięci statycznej, 315
analizowanie
  bloku kodu, 138, 141
  kodu Fortran, 170
  serwisu społecznościowego, 318
  tekstu, 313
  wykorzystania pamięci, 276
  zbioru danych, 97
AOT, Ahead of Time, 136
aplikacje internetowe, 27, 323
aproksymacja różnic skończonych, 105
asynchroniczna kolejka zadań, 269
asynchroniczne dodawanie zadań, 220
atrybucja, 13
AWS, Amazon Web Services, 249

B

baza danych, 27, 190, 323
biblioteka
  array, 27
  asyncio, 178, 188, 193
  biopython, 27
  BLAS, 27, 162, 169
  bytes, 27
  collections, 27
  diffusion.so, 165
  gevent, 181–183, 187, 193
  grequests, 183
  itertools, 97
  LAPACK, 169
  LIBLINEAR, 26
  LIBSVM, 26
  math, 27
  numpy, 27
  OpenCV, 28
  pandas, 27, 28
  PrettyTable, 238
  scikit-learn, 27
  scipy, 27, 129, 130
  sqlite3, 27
  tornado, 27
  unicode, 27
  ViennaCL, 162
biblioteki asynchroniczne, 181
blokada GIL, 153, 160, 197, 206
blokowanie
  interpretera, 26
  lokalne, 247
  obiektu Value, 245
  plików, 242
błąd, 243, 290, 296, 302
błędy zaokrąglania, 67
BPython, 28
BSB, Backside Bus, 21

C

centralna jednostka obliczeniowa, 16
chronologia żądań, 180, 184, 187, 190
ciąg Fibonacciego, 96
CPU, Central Processing Unit, 16

czas
działania funkcji, 118
działania pętli, 81
działania schematów, 129
tworzenia instancji, 78
wykonywania programu, 51
wykonywania wyrażeń, 49
wyszukiwania, 71, 281, 282
częstość występowania dzielników, 222
czyszczenie
flagi, 228
pamięci, 53, 158

D

dane
losowe, 19
sieciowe, 179, 183–185, 188
wyjściowe, 57, 65, 141, 237
debugowanie, 320
debugowanie identyfikowanych typów, 155
definicja entropii, 87
definiowanie
dekoratora, 38
klasy, 86
deklarowanie wektorów, 170
dekorator, 37
@jit, 155
@profile, 64
@require, 262
@timefn, 39
czasu, 30
dekoratory fikcyjne, 47
dezasemblacja, 90
diagnozowanie wykorzystania pamięci, 51
dodawanie
adnotacji, 157
adnotacji typu, 143
etykiet, 55
flag kompilatora, 153
operatora prange, 153
typów podstawowych, 143
dokumentacja, 39
kompilatora Cython, 141
modułu bisect, 72
modułu lockfile, 244
modułu multiprocessing, 244
dokumenty PEP, 189
dostarczanie komunikatów, 263
drzewa trie
drzewo, 271
datrie, 285, 304
trie, 280, 283
trie HAT, 286
trie Marisa, 285
duże operacje macierzowe, 127
dyfuzja, 104, 119
dwuwymiarowa, 107–110, 164, 167
jednowymiarowa, 106, 107
dysk twardy SSD, 20
dystrybucja zadań, 324
działanie filtrów Blooma, 295

E

efektywne profilowanie, 30
entropia, 86, 87
etykieta calculate_output, 55

F

fałszywa skala szarości, 32, 36
filtr laplace, 130
filtry Blooma, 295, 297
firma
Adaptive Lab, 308
Lyst.com, 315
RadimRehurek.com, 310
Smesh, 318
flaga, 228–233
CHECK_EVERY, 234
FLAG_CLEAR, 228
FLAG_SET, 228, 235
przenośności, 40
format JSON, 256
fragmentacja, 115
fragmentacja pamięci, 111
FSB, Frontside Bus, 21
funkcja
%timeit, 62
__builtin__, 65
__hash__, 86
__main__, 149, 241
abs, 33, 137
anomalous_dates.next, 100
apply_sync, 259
asizeof, 277
assert, 241
calc_pure_python, 37, 43, 57
calculate_z, 150, 157
calculate_z_serial_purepython, 42, 51, 54
chain, 97
check_anomaly, 99, 100
check_prime, 213, 226
check_prime_in_range, 230
cycle, 97

Kup ksi k   Pole ksi k

day_grouper, 100
definite_primes_queue, 217
evolve, 108, 110, 156, 165
feed_new_jobs, 220
fibonacci_transform, 96
fn_expressive, 63
fn_terse, 63
getsizeof, 276
gevent.iwait, 182
hash, 83
hpy.setrelheap, 58
id, 86
islice, 97
iwait, 185
lambda, 98
laplace, 130
laplacian, 124–130
main, 36, 203
memit, 277
norm_square_numpy, 117
ord, 84
range, 36, 55, 95
roll, 119
roll_add, 125
set, 282
takewhile, 97
time.time(), 30
timefn, 38
timeit, 281
total.__add__, 63
twoletter_hash, 89
wait, 181
work, 243, 244
worker_fn, 238, 240
xrange, 55, 93, 95
yield, 94
funkcje
magiczne, 39
mieszania, 84, 86, 88
powiązane z procesorem, 36
wbudowane, 62, 97

G

generator, 93, 95
aktywnego wykresu, 58
kongruencji liniowej, 83
liczb losowych, 210
text_example.readers, 281
zbioru Julii, 34
GIL, Global Interpreter Lock, 153
globalna blokada interpretera GIL, 19
głębokie uczenie, 310
główne uporządkowanie
kolumnowe, 170
wierszowe, 170
GPU, Graphics Processing Unit, 16, 162
graf, 271
graf słów DAWG, 279, 283, 284
graficzne jednostki obliczeniowe, 16
greenlet, 181
grupowanie danych, 98

H

hiperwątki, 216
hipoteza, 50

I

IDE, Integrated Development Environment, 28
idealna funkcja mieszania, 88
idempotentność, 293
identyfikator
id, 122
PID, 237
identyfikowanie wąskich gardeł, 46
iloczyn skalarny, 118
implementacja
algorytmu HyperLogLog, 302
algorytmu LogLog, 301
filtru Blooma, 296
licznika LogLog, 300
licznika Morrisa, 291
struktury KMinValues, 294
importowanie
kodu Fortran, 169
modułów, 260
modułu, 140, 148
modułu zewnętrznego, 148
inferencja typów, 148
informacje o
elemencie wywołującym, 44
profilowaniu, 38
typie, 70, 136, 155
inicjalizacja
dyfuzji dwuwymiarowej, 109
narzędzia cffi, 167
inspekcja
obiektów, 56
sterty, 56
instalowanie
modułów, 159
modułu Parallel Python, 257
narzędzia line_profiler, 47
narzędzia pip, 159
narzędzia runsnake, 46

Kup ksi k   Pole ksi k

instrukcja
  import, 148
  print, 37
  SIMD, 17
  yield, 178
interfejs
  API, 28, 134, 152
  API HTTP, 190
  CUDA, 162
  FileLock, 245
  IPython, 259
  memoryview, 151
  MPI, 222, 259
  OpenCL, 162
  OpenMP, 152–154, 162, 196
  powłoki, 28
  StrictRedis, 230
interfejsy funkcji zewnętrznych, 163
interpreter, 25
  CPython, 10
  GIL, 26
  PyPy, 157–160, 322
IPC, Instructions Per Cycle, 16
IPC, Interprocess Communication, 199
IPython, 28
IPython Notebook, 28
iterator, 93, 96
iterator ifilter, 100
iterowanie, 81

J

jednostka MiB, 53
jednostka obliczeniowa, 16
  CPU, 16
  GPU, 16
  IPC, 16
jednostki pamięci, 19
język
  C, 133
  C++, 138
  Fortran, 133, 169
  GO, 263
  Java, 199
  Python 2.7, 11
  Python 3, 12
  Theano, 162
JIT, Just in Time, 10, 133, 136

K

katalog bin, 159
klasa
  AsyncBatcher, 191, 192
  CompositeError, 262
  Point, 87
klaster, 249, 250
  Beowulfa, 249
  IPython, 259
klastrowanie
  wady, 251
  zalety, 250
klastry
  lokalne, 257, 262
  modułu IPython Parallel, 257
  produkcyjne, 262
klucz, 81, 83, 85, 98
kod
  asynchroniczny, 178
  bajtowy, 62
  C, 142, 164
  deterministyczny, 36
  dyfuzji, 119
  Fortran, 169
kodowanie UTF-8, 278
kolejka, 221, 263
  Celery, 269
  FIFO, 198, 218
  PyRes, 221
  Queue, 220
kolejki
  zadań, 249, 325
  zadań roboczych, 217
kolekcja, 276
kolekcja unikalnych kluczy, 81
kolizje wartości mieszania, 85
kompilator
  AOT, 133, 136
  Cython, 139, 146, 151, 161
  g++, 137
  gcc, 137
  JIT, 133, 136, 154
  kodu C, 137
  LLVM, 133, 154
  Nuitka, 162
  Numba, 134, 154, 161
  Parakeet, 162
  PyPy, 10, 149, 161
  Pyston, 162
  Python-C++, 147, 155
  Pythran, 134, 155
  Shed Skin, 147, 150, 161

Kup ksi k   Pole ksi k

kompilatory języka C, 162
kompilowanie, 133, 139
komponent
  ctypes, 198
  Manager, 198
  Pipe, 198
  Pool, 198, 199
  Process, 198
  Queue, 198
kompresja, 271
komputer zdalny, 261
komunikacja międzyprocesowa, 199, 217, 221, 248
komunikaty, 263
komunikaty MPI, 222
konfigurowanie procesów, 244
kongruencja liniowa, 83
konstrukcja future, 177, 178, 181
konsument danych, 264
kontrola poprawności, 36
kontroler, 259
konwerter
  Fortran-Python, 134
  Python-C, 134
kopie pamięci, 150
kopiowanie profilu, 261
koszt funkcji, 44
koszt kopiowania danych, 150
krotki, 69, 73, 77

L

leniwe wczytywanie danych, 98
licencja Creative Commons, 13
liczba
  instrukcji, 120
  przydziałów pamięci, 122, 123
  transferów, 115
  współbieżnych żądań, 181
  zespolona, 33, 137
  zmiennoprzecinkowa, 137
  żądań współbieżnych, 182
liczby
  Fibonacciego, 96
  losowe, 208
  pierwsze, 211, 227
licznik
  branches, 115
  branch-miss, 115
  context-switches, 113
  CPU-migrations, 113
  instructions, 114, 126
  LogLog, 299, 300
  Morrisa, 290, 291
  page-faults, 126
  stalled-cycles, 115
  stalled-cycles-backend, 114
  stalled-cycles-frontend, 114
liczniki wydajności, 113, 120–130
lista, 69, 73, 107
  kompilatorów, 162
  new_grid, 110
losowość sekwencji zadań, 215

M

macierze, 103
magazyn danych NAS, 21
magistrala
  BSB, 16, 21
  FSB, 16, 21
mapowanie obiektowo-relacyjne, 323
maska, 82, 88
maszyna wirtualna, 23, 25
maszyna wirtualna PyPy, 133
Matlab, 28
mebibajt, 53
mechanizm
  OCR, 324
  rekomendacji, 316
  sondowania, 83
menedżer
  kontekstu, 55, 245–247
  zadań PyRes, 269
metoda
  Eulera, 105
  Monte Carlo, 199, 200
metodologia projektowa, 309
metody
  pomiaru czasu, 37
  synchronizacji, 242
  typu LogLog, 300
mieszanie, hashable, 79
moduł
  array, 116, 151, 274
  asyncio, 189
  bisect, 72, 282
  concurrent.futures, 199
  cProfile, 41
  ctypes, 164, 165
  dis, 60
  distutils, 173
  f2py, 169, 170
  IPython Parallel, 259, 262
  itertools, 98
  lockfile, 244, 245
  memory_profiler, 52, 53

Kup ksi k   Pole ksi k

moduł
mmap, 232, 233
multiprocessing, 192, 195–248
komponenty, 198
ograniczenia, 200
strategie użycia, 215
współużytkowanie danych, 236
narzędzia CPython, 171, 172
numexpr, 127, 128
numpy.array, 151
Parallel Python, 257, 258
pickle, 219
pp, 259
requests, 179
struct, 166
timeit, 38, 39
modyfikowanie kodu źródłowego, 53
monitorowanie, 317, 320
monitorowanie pamięci, 313
MPI, Message Passing Interface, 222, 259

N

narzędzia
klastrowania, 268
profilujące, 42
narzędzie
Circus, 255
cffi, 166, 167, 168
cProfile, 42
cpyext, 159
CPython, 31, 60, 151, 164
cron, 255
Cython, 28, 133, 137, 139
diff, 67
Docker, 321
dowser, 58, 67
dozer, 67
Ganglia, 256
Graphite, 317
heapy, 30, 56
hotshot, 42
htop, 237
IPython, 159, 257
jitviewer, 160
joblib, 216
line_profiler, 30, 46, 64, 109
lsprof, 44
make, 149
memory_profiler, 31, 51, 64, 65
mprof, 54, 55
nosetests, 64, 65
Numba, 133, 154
numpy, 74, 103, 116–128, 151, 170, 199, 209, 236, 241, 248, 275
perf, 112, 113, 115
pip, 159
profile, 42
PyPy, 134, 157
Pythran, 134, 155
runsnake, 46
Shed Skin, 134, 137, 147
SoMA, 307
System Monitor, 40
Upstart, 255
word2vec, 313, 314
NAS, Network Attached Storage, 21
nawias kwadratowy, 54
niejawna pętla, 118
niskopoziomowe typy numeryczne, 118
notka dokumentacyjna, 38

O

obiekt
_diffusion, 165
BLOB, 266
deque, 100
FileLock, 245, 247
HubFactory, 260
list, 281
Lock, 246, 247
Manager.Value, 228
multiprocessing.Value, 245
multprocessing.Array, 239
Pool, 203, 216, 225, 227
Process, 244
Queue, 197, 216, 219
RawValue, 232, 247
Unicode, 278
Value, 245
vector, 117
obiekty
języka Python, 201
listy, 151
Python, 208
typów podstawowych, 272
Unicode, 277
obietnica wyniku, 178
obliczanie
dyfuzji dwuwymiarowej, 107
liczby zespolonej, 33
listy output, 150
pochodnych, 119
zbioru Julii, 34

Kup ksi k Pole ksi k

obliczenia
  macierzowe, 103
  probabilistyczne, 290
  rozproszone, 266
  wektorowe, 103
obraz systemu, 256
obrót, 125
odchylenie standardowe, 98
odczyt sekwencyjny, 19
odnośniki TRACE, 60
odtwarzanie flagi, 233
ograniczanie liczby operacji, 133
opcje kompilatorów, 160
OpenMP, Open Multi-Processing, 152
operacja
  FMA, 16
  numpy.dot, 117
  resize, 75
  sqrt, 145
operacje
  dołączania, 76
  macierzowe, 127
  wejścia-wyjścia, 175, 188
  wewnętrzne, 121–124
operator
  __cmp__, 86
  append, 77
  prange, 153, 154
opóźnienie, 19
oprogramowanie Matlab, 28
optymalizacja, 64, 132, 146, 313
  operacji macierzowych, 127
  operacji wektorowych, 127
  PGO, 150
  selektywna, 124
  wyszukiwarek, 313
  obciążającej logiki, 234
optymalna funkcja mieszania, 88
ORM, Object Relational Mapper, 323
oszczędzenie pamięci, 271

P

pakiet
  countmemaybe, 294
  distarray, 269
  guppy, 56
  java.util.concurrent, 199
  tornado, 185–187
  wxPython, 46
pamięć
  podręczna, 16, 20
  RAM, 16, 20, 53, 203, 271
parametr chunksize, 213–216
PEP, Python Enhancement Proposals, 189
pętla
  for, 63, 93, 94
  while, 49, 50, 145
  zdarzeń, 176, 186
  zwrotna, 231
pętle
  niejawne, 118
  wewnętrzne, 81
PGO, Profile-Guided Optimization, 150
pierwiastek kwadratowy, 137, 145
plik
  cdiffusion.so, 174
  complicated.h, 168
  cythonfn.html, 141
  cythonfn.pyx, 139, 140
  diffusion.so, 170
  diffusion_numpy.so, 156
  ipcontroller-engine.json, 261
  julia1.py, 139
  Makefile, 149
  setup.py, 139, 173
  shedskinfn.cpp, 149
  shedskinfn.hpp, 149
  shedskinfn.so, 149
  shedskinfn.ss.py, 149
pliki
  .deb, 256
  .lprof, 47
  .pyx, 146
  .rpm, 256
  .so, 139
  statystyk, 46, 54
pobieranie, 82
podejmowanie decyzji, 115
polecenie
  grep, 239
  ipcluster, 259
  memit, 273
  pmap, 239
  print, 42
  ps, 239
  push, 260
  redis-cli, 231
  shedskin, 149
  polecenie time, 40
połączenie generatorów, 99
pomiar
  czasu, 37, 40, 43, 88
  czasu sortowania, 282
  wierszy kodu, 46
  wykorzystania pamięci, 272, 273

Kup ksi k Pole ksi k

porównanie
grafu i drzew, 280
kompilatorów JIT i AOT, 136
list i krotek, 73
probabilistycznych struktur danych, 303, 304
programów, 176
port numpypy, 159
potokowanie, 112, 266
powłoka
interaktywna, 28
IPython, 260
prawo Amdahla, 18, 196
probabilistyczne struktury danych, 289, 303–305
problem
Cauchy'ego, 105
z alokacją, 109
procedura
BLAS, 314
obsługi, 266
proces, 202, 205, 206
nadrzędny, 239
nsdq, 267
procesor CPU, 16
procesy
robocze, 215, 324
szeregujące, 259
współbieżne, 243
profilowanie, 29, 42, 66
profilowanie dyfuzji, 109
wierszy, 111, 124
program, *Patrz* narzędzie
programowanie asynchroniczne, 176
programy
szeregowe, 176
współbieżne, 176
projekt
klastra, 254, 316
Micro Python, 288
MPI4PY, 222
narzędzia SoMA, 308
protokół TCP, 231
przechowywanie
typów, 275
typów podstawowych, 151
zbiorów tekstowych, 279
przegląd skompilowanego kodu, 155
przekazywanie
obiektu, 228, 229
współużytkowanych danych, 260
przełączanie kontekstu, 176
przenoszenie kodu, 35
przepustowość interfejsów, 23
przestój usługi, 253
przestrzenie nazw, 89
przeszukiwacz, 186
przeszukiwacz szeregowy, 179
przetwarzanie
klastrowe, 249
równoległe, 152, 196, 208, 314
szeregowe, 225
wyidealizowane, 23, 24
przewidywanie rozgałęzień, 112
przybliżanie liczby pi, 200–210, 262
przydziały pamięci, 121
przydzielanie dla list, 75
przyspieszenie
działania kodu, 132, 135, 196, 210
kodu asynchronicznego, 192
pseudokod, 106
publikator, 264
punkty skoku, 61

R

RAID, 254
RAM, Random Access Memory, 16
raport narzędzia memory_profiler, 54–56
raportowanie, 317, 320
redukowanie mocy, 145
redundancja, 265
rekomendacje, 316
rozbicie
pętli while, 48
testów jednostkowych, 53
rozkład Gaussa, 290
rozproszone obliczenia, 266
rozwiązania klastrowe, 257
rozwijanie funkcji, 34
rozwijanie funkcji abs, 145
równanie dyfuzji, 104–107
równoważenie obciążenia, 211

S

sekwencyjne przechowywanie danych, 116
semafory synchronizujące, 198
serializacja, 219
serwis Lanyrd.com, 325
serwisowanie systemu SoMA, 309
skalowalny filtr Blooma, 297
skalowanie dynamiczne, 250
skrypt, *Patrz także* plik
coverage.py, 64
kernprof.py, 47–50, 109
setup.py, 140, 153
słownik, 79, 89

Kup ksi k   Pole ksi k

słownik globals(), 89
słowo kluczowe cdef, 143, 144
sondowanie, 83
sortowanie, 43, 72, 282
sprawdzanie
  danych wyjściowych, 57, 149
  kodu bajtowego, 60
  współużytkowanych danych, 222
stacja robocza, 16
stała sys.maxint, 276
sterta, 56
strategie profilowania, 66
struktura
  drzewa trie, 283
  grafu DAWG, 283
  HyperLogLog++, 289
  K-Minimum Values, 292, 294
  Point, 168
strumieniowanie danych, 313
subskrybent, 264
symbol >>, 61
synchroniczny interpreter, 259
synchronizacja, 198, 219
  zapisów, 246
  danych, 222
  dostępu, 242
system
  Gearman, 269
  komputerowy, 15
  NSQ, 262–267
  przetwarzający zadania, 269
  przetwarzania danych, 322
  Redis, 224, 229
  SaltStack, 309
  SoMA, 309
  SQS, 269
  wdrażania
    Chef, 256
    Fabric, 256
    Puppet, 256
    Salt, 256
szereg nieskończony, 96
szybkości połączeń interfejsów, 23
szybkość zegara, 17

Ś

średnia online, 98
środowisko
  Canopy, 28
  EPD, 28
  Sage, 28
  systemu NSQ, 267

T

tabela mieszająca, 82, 84
  usuwanie wartości, 85
  zmiana wielkości, 85
tablica, 69
tablica locals(), 89
tablice
  dynamiczne, 73, 74
  statyczne, 73, 77
  współużytkowane, 238
technika głębokiego uczenia, 310
testowanie szybkości, 62
testy jednostkowe, 64–67
tłumaczenie, 324
tokeny, 280, 281
topologia
  połączeń, 265
  systemu NSQ, 264
TTL, Time to Live, 321
twierdzenie Pitagorasa, 201
tworzenie
  adnotacji, 145
  dwóch kolejek, 218
  funkcji obrotu, 125
  hipotezy, 42
  lokalnego profilu, 261
  modułu rozszerzenia, 148
  norm wektora, 117
  operacji wektorowych, 169
  procesu szeregowego, 196
  systemu klastrowego, 254
  tabeli mieszającej, 82
  tablicy, 70
typ
  array, 116
  double complex, 144
  Future, 178
  int, 144
  struct, 166, 168
  unsigned int, 144
typy podstawowe, 274

U

uczenie maszynowe, 315
układy graficzne GPU, 162
uruchamianie
  interpretera PyPy, 159
  modułu timeit, 39
  narzędzia dowser, 59
  serwera WWW, 59
  skryptu, 140, 283

Kup ksi k
Pole ksi k

urządzenie typu SS, 19
usługa
  AWS, 249
  EC2, 254
  Skype, 253
usuwanie elementu, 85
utrata danych, 256
użycie
  adnotacji, 170
  adnotacji kompilatora, 141
  dekoratora, 38
  dekoratora @profile, 64
  drzew trie, 287
  dwóch kolejek, 217, 219
  filtru Blooma, 298
  filtru laplace, 130
  flagi przenośności, 40
  funkcji %timeit, 62
  funkcji set, 282
  funkcji wbudowanych, 62
  generatorów, 97
  grafu DAWG, 284
  iloczynu skalarnego, 118
  interaktywnego debugera, 60
  kodu Fortran, 169
  kompilatora kodu C, 137
  menedżera kontekstów, 55
  menedżera kontekstu, 245
  modułu cProfile, 41
  modułu ctypes, 164
  modułu dis, 60, 63
  modułu IPython Parallel, 259
  modułu lockfile, 244, 245
  modułu mmap, 232, 233
  modułu numexpr, 127
  modułu Parallel Python, 257
  narzędzia cffi, 167
  narzędzia dowser, 58
  narzędzia heapy, 57
  narzędzia line_profiler, 46
  narzędzia memory_profiler, 51
  narzędzia numpy, 119, 209
  narzędzia runsnake, 46
  obiektu Lock, 246
  obiektu Manager.Value, 228
  obiektu RawValue, 232
  procesów, 210
  profilowania, 29
  serwera Redis, 230
  skryptu kernprof.py, 48
  słownika, 80
  struktury datrie, 286
  systemu NSQ, 262
  systemu Redis, 229
  wątku, 220
  wykonywania szeregowego, 202

W

warstwy komunikacji, 21
wartości k-minimum, 291
wartościowanie leniwe generatora, 97
warunki brzegowe, 106
wąskie gardło, 29
wątki, 197, 202, 206–208
wdrażanie, 320
wdrażanie aktualizacji, 256
wektory, 103
wektoryzacja, 17, 119
weryfikowanie
  liczb pierwszych, 221
  optymalizacji, 129
  wyniku, 238
wielowątkowość współbieżna, 18
wiersz poleceń, 39
wirtualny procesor, 18
wizualizacja
  danych wyjściowych, 46
  pliku profilowania, 47
  w czasie rzeczywistym, 67
właściwość
  length, 95
  next(), 98
wskaźnik, 112
wskaźnik Major page faults, 41
współbieżne procesy, 243
współbieżność, 175
współczynnik błędu, 301
współprogramy, coroutine, 178
współprogramy greenlet, 181
współrzędne zespolone, 35
współużytkowanie
  danych, 236, 260
  tablicy, 239, 240
  zadań, 197
wstawianie, 82
wstawianie z kolizjami, 84
wybór struktury danych, 73
wydajność
  algorytmu, 72
  instrukcji while, 145
  kolejki zadań, 326
  wyszukiwań, 87
wyjątek NameError, 64
wykonywanie szeregowe, 209
wykorzystanie pamięci, 237, 272, 276, 288

Kup książkę Poleć książkę

wykres zbioru Julii, 32, 37
wykrywanie nieprawidłowości, 99
wyłączanie wektoryzacji, 121
wyodrębnianie danych sieciowych, 179, 183–185, 188
wyszukiwanie, 80
  binarne, 72
  efektywne, 71
  liniowe, 80
  liniowe listy, 71
  powolne, 91
  w przestrzeniach nazw, 90
  w słowniku, 83, 87
wyświetlanie liczby instrukcji, 63
wywołania zwrotne, 177, 186
wywoływanie narzędzia dowser, 59
wzrosty szybkości, 134

Z

zastosowanie, *Patrz* użycie
zbiory, 79
zbiór Julii, 31, 34, 138, 152
zespolony warunek początkowy, 35
zestawienie przyspieszeń, 132
zewnętrzny system kolejek, 221
zintegrowane środowisko programistyczne, IDE, 28
złożoność, 62, 72, 80
zmiana
  funkcji, 56
  szybkości zegara, 17
  wielkości listy, 76
zmienne globalne, 108
zmniejszanie
  liczby alokacji pamięci, 110–113
  liczby przydziałów pamięci, 123
znajdowanie
  elementu, 85
  liczb pierwszych, 211, 212
  nieprawidłowości, 101
  unikalnych imion, 81
  wartości, 72
zrównoważone obciążenie, 259
zwiększanie wydajności, 145

Ż

żądania współbieżne, 182
żądany współczynnik błędu, 296

Kup ksi k   Pole ksi k

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

O'REILLY®

Python
Programuj szybko i wydajnie

Python to skryptowy język programowania istniejący na rynku od wielu lat — jego pierwsza wersja pojawiła się w 1991 roku. Przejrzystość kodu źródłowego była jednym z głównych celów Guida van Rossuma, twórcy tego języka. Dziś Python cieszy się dużą popularnością, co z jednej strony świadczy o jego przydatności, a z drugiej gwarantuje użytkownikom szerokie wsparcie społeczności programistów języka. Python jest elastyczny, dopuszcza różne style programowania, a dzięki temu znajduje zastosowanie w wielu miejscach świata IT.

Jeżeli chcesz w pełni wykorzystać możliwości Pythona i tworzyć wydajne rozwiązania, koniecznie zaopatrz się w tę książkę! Dzięki niej dowiesz się, jak użyć profilowania do lokalizowania „wąskich gardeł", oraz poznasz efektywne techniki wyszukiwania danych na listach, w słownikach i zbiorach. Ponadto zdobędziesz wiedzę na temat obliczeń macierzowych i wektorowych oraz zobaczysz, jak kompilacja do postaci kodu C wpływa na wydajność Twojego rozwiązania. Osobne rozdziały zostały poświęcone współbieżności oraz modułowi multiprocessing. Opanowanie tych zagadnień pozwoli Ci ogromnie przyspieszyć działanie Twojej aplikacji. Na sam koniec nauczysz się tworzyć klastry i kolejki zadań oraz optymalizować zużycie pamięci RAM. Rozdział dwunasty to gratka dla wszystkich — zawiera najlepsze porady specjalistów z branży! Książka ta jest obowiązkową lekturą dla wszystkich programistów chcących tworzyć wydajne rozwiązania w języku Python.

Dzięki tej książce:

- poznasz sposoby profilowania aplikacji
- zrozumiesz, jak działają listy, słowniki i zbiory
- zoptymalizujesz zużycie pamięci RAM
- poznasz porady ekspertów z branży IT

Wyciśnij z Pythona siódme poty!

Micha Gorelick — zajmuje się problematyką uczenia się maszyn i wydajnymi strumieniami danych. Współzałożyciel firmy Fast Forward Labs.

Ian Ozsvald — analityk danych i nauczyciel w firmie ModelInsight, z 10-letnim doświadczeniem w programowaniu w języku Python. Prelegent na konferencjach PyCon oraz PyData.

Helion

32028 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-283-0466-6

9 788328 304666

Informatyka w najlepszym wydaniu

cena 59,00 zł