Использование библиотеки Numba для ускорения программ на Python

Севастопольский Артём

ВМК МГУ, кафедра ММП

27 октября, 2015

Почему Python такой медленный?

- Каждый объект (даже самый простой) хранится в сложной структуре данных
- Много косвенных обращений к памяти
- Проверки типов
- Python – не компилируемый язык. Компилятор мог бы заранее просмотреть всю программу.

Почему Python такой медленный?

```
1 a = 1
2 b = 2
3 c   = a + b
```

Почему Python такой медленный?

```
1 a = 1
2 b = 2
3 c   = a + b
```

1. `a = 1`
   - разместить в динамической памяти объект a
   - установить `a->PyObject_HEAD->typecode` в int
   - установить `a->val = 1`
2. `b = 2`
   - (аналогично)
3. `a + b`
   - посмотреть на `a->PyObject_HEAD->typecode`
   - a – целое, значение равно `a->val`
   - (аналогично с b)
   - вызвать оператор сложения, сохранить результат во временный объект `result`
4. Создать объект c и разместить в нем сумму
   - разместить в динамической памяти объект c
   - установить `c->PyObject_HEAD->typecode` в int
   - установить `c->val` в `result`

Задача сложения двух Numpy-массивов

Пусть требуется написать функцию, складывающую два Numpy-массива покомпонентно.

```
1 def naive_sum(x, y):
2     ans = np.empty_like(x)
3     for i in xrange(len(x)):
4         ans[i] = x[i] + y[i]
5     return ans
```

Задача сложения двух Numpy-массивов

Пусть требуется написать функцию, складывающую два Numpy-массива покомпонентно.

```
1 def naive_sum(x, y):
2     ans = np.empty_like(x)
3     for i in xrange(len(x)):
4         ans[i] = x[i] + y[i]
5     return ans
```

Возьмем 2 вектора длины 100000.
Время работы: 0.079495 (с)

Задача сложения двух Numpy-массивов

Пусть требуется написать функцию, складывающую два Numpy-массива покомпонентно.

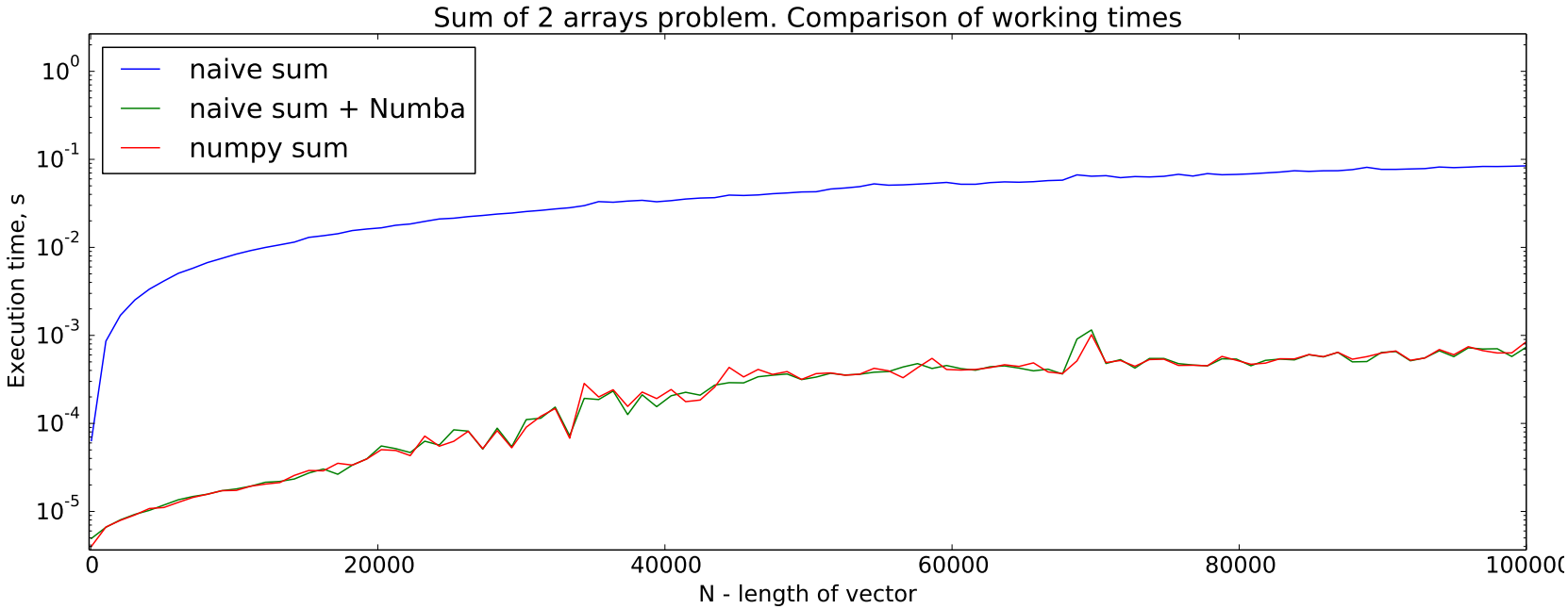
```
1  def naive_sum(x, y):
2      ans = np.empty_like(x)
3      for i in xrange(len(x)):
4          ans[i] = x[i] + y[i]
5      return ans
```

Возьмем 2 вектора длины 100000.
Время работы: 0.079495 (c)

```
1  from numba import jit
2
3  @jit
4  def naive_sum(x, y):
5      ans = np.empty_like(x)
6      for i in xrange(len(x)):
7          ans[i] = x[i] + y[i]
8      return ans
```

Время работы: 0.000662 (c)
Ускорение в **120 раз**.

Задача сложения двух Numpy-массивов



Производилось также сравнение с функцией, использующей сложение массивов в Numpy:

```
1 def numpy_sum(x, y):
2       return x + y
```

Принцип работы Numba

- JIT (just-in-time) компилятор Numba компилирует максимально возможную часть кода. Используется LLVM (Low Level Virtual Machine).
- Компиляция производится при первом запуске функции.
- Можно потребовать компиляции всего кода функции:

```
1 @jit(nopython=True)
```

Однако в таком режиме поддерживается мало возможностей.

Принцип работы Numba

- Большой выигрыш по времени осуществляется благодаря явной подстановке типов. Каждая компиляция производится для конкретного набора типов аргументов.

```
# let my_func be jit-compilable function
my_func(5, 3)       # my_func(int, int) gets compiled
my_func(5, 4.0)     # my_func(int, float) gets compiled
my_func('abc', {'a': 1, 'b': 3})
    # my_func(str, dict) gets compiled
```

- Можно ограничить возможные типы аргументов.

```
@jit('f8[:](f8[:],f8[:])')
def naive_sum(x, y):
    ans = np.empty_like(x)
    for i in xrange(len(x)):
        ans[i] = x[i] + y[i]
    return ans
```

Задача - вычисление матрицы попарных расстояний

Пусть $X, Y \in \mathbb{R}^{N \times D}$. Требуется построить матрицу $M = [m_{ij}] \in \mathbb{R}^{N \times N}$, такую что m_{ij} – расстояние между $X[i]$ и $Y[j]$.

Задача - вычисление матрицы попарных расстояний

Пусть $X, Y \in \mathbb{R}^{N \times D}$. Требуется построить матрицу $M = [m_{ij}] \in \mathbb{R}^{N \times N}$, такую что m_{ij} – расстояние между $X[i]$ и $Y[j]$.

1 решение. Только средствами Python.

```
1  def pdist_naive(X, Y):
2      ans = np.empty((X.shape[0], Y.shape[0]))
3      for i in xrange(X.shape[0]):
4          for j in xrange(Y.shape[0]):
5              dist = 0.0
6              for k in xrange(X.shape[1]):
7                  diff = X[i, k] - Y[i, k]
8                  dist += diff * diff
9              ans[i, j] = np.sqrt(dist)
10     return ans
```

```
1  @jit('f8[:,:](f8[:,:],f8[:,:])')
2  def pdist_naive_jit(X, Y):
3      # the same as pdist_naive
4      ...
```

Задача - вычисление матрицы попарных расстояний

2 решение. Двойной цикл for, вычисление очередного расстояния средствами Numpy.

```
1  def pdist_naive2(X, Y):
2      ans = np.empty((X.shape[0], Y.shape[0]))
3      for i in xrange(X.shape[0]):
4          for j in xrange(X.shape[1]):
5              ans[i, j] = sqrt(np.sum((X[i] - Y[i]) ** 2))
6      return ans
```

```
1  @jit('f8[:,:](f8[:,:],f8[:,:])')
2  def pdist_naive2_jit(X, Y):
3      # the same as pdist_naive2
4      ...
```

Задача - вычисление матрицы попарных расстояний

3 решение. Используем функцию cdist из Scipy.

```
1 from scipy.spatial.distance import cdist
2 def pdist_cdist(X, Y):
3       return cdist(X, Y)
```

Задача - вычисление матрицы попарных расстояний

Результаты:

	Python	Python, JIT	Python+NumPy	Python+NumPy, JIT	cdist
1000 x 3	12.394 s	0.0266 s	0.08208 s	0.00064 s	0.02770 s
100 x 100	1.41403 s	0.00384 s	0.28076 s	0.00475 s	0.00289 s
10 x 1000	0.14836 s	0.00036 s	0.00377 s	0.00028 s	0.00033 s

- Простейшие решения с JIT-компиляцией работают примерно так же быстро, как `cdist`, а иногда обгоняют его.
- Простейшее решение ускоряется до 460 раз.

Проблемы Numba

- Не любую функцию можно скомпилировать с помощью Numba. Например, создание функции внутри функции вызывает исключение NotImplementedError.
- Если в скомпилированной функции происходит выход за границы массива, возникает Ошибка сегментации.
- Оптимизация циклов работает не всегда. Например, если сделать return из цикла, оптимизация не включается.
- Оптимизации только для NumPy-массивов. Другие объекты не поддерживаются.

+ Numba активно развивается.

Выводы

- Numba дает огромное ускорение для функций, написанных на Python и Numpy.
- При этом больше всего оптимизируется Python-часть.
- Эффективна для оптимизации циклов.
- Функции компилируются при их первом запуске.
- Не любую функцию можно скомпилировать. Необходимо выносить критический код в отдельные функции и компилировать их.

Запуск на GPU

- Numba позволяет запускать функции на GPU.

```
1 from numba import cuda
2
3 @cuda.jit('(uint64[:],uint64)')
4 def my_cuda_function(A, b):
5     ...
```

- Однако на функцию накладываются большие ограничения.
- Необходимо составлять программу так, чтобы критические вычисления были в отдельных функциях.
- Статья NVIDIA про ускорение вычислений с помощью numba.cuda: `http://devblogs.nvidia.com/parallelforall/gpu-accelerated-graph-analytics-python-numba/`

Литература

- Документация по библиотеке Numba. http://numba.pydata.org/
- Статья NVIDIA CUDA Zone про ускорение вычислений с помощью numba.cuda. http://devblogs.nvidia.com/parallelforall/gpu-accelerated-graph-analytics-python-numba/
- Optimizing Python in the Real World: NumPy, Numba, and the NUFFT https://jakevdp.github.io/blog/2015/02/24/optimizing-python-with-numpy-and-numba/

Установка Numba

Пример комманд установки Numba для Ubuntu 14.04 и Python 2.7.

```
# apt-get install llvm-3.6 llvm-3.6-dev llvm-3.6-runtime
# pip install enum34 funcsigns
# LLVM_CONFIG=/usr/bin/llvm-config-3.6 pip install numba
```