

Лекция 8: Оптимизация выполнения кода, векторизация, Numba

Автор: Сергей Вячеславович Макрушин, 2022 г.

е-mail: s-makrushin@yandex.ru (<mailto:s-makrushin@yandex.ru>)

При подготовке лекции использованы материалы:

- Документация к рассмотренным пакетам

V 0.6 30.10.2022

Разделы:

- [Профилирование](#)
 - [Numba](#)
 - [Векторизация в NumPy](#)
 - [Векторизация для скалярных функций NumPy](#)
 - [Векторизация для векторных функций в NumPy](#)
 - [Векторизация в Numba](#)
-
- [к оглавлению](#)

```
In [1]: # загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v2.css")
HTML(html.read().decode('utf-8'))
```

Out[1]:

Задача оптимизации

В процессе разработки кода и создания конвейеров обработки данных всегда присутствуют компромиссы между различными реализациями. В начале создания алгоритма забота о подобных вещах может оказаться контрпродуктивной. Согласно знаменитому афоризму Дональда Кнута: *«Лучше не держать в голове подобные “малые” вопросы производительности, скажем, 97 % времени: преждевременная оптимизация — корень всех зол»*.

Однако, как только ваш код начинает работать, часто бывает полезно заняться его производительностью. Иногда бывает удобно проверить время выполнения заданной команды или набора команд, а иногда — покопаться в процессе оптимизации и выяснить, где находится узкое место какого-либо сложного набора операций.

Профилирование

- [к оглавлению](#)

Профилирование — сбор характеристик работы программы, таких как:

- время выполнения отдельных фрагментов (например, функций)

- число верно предсказанных условных переходов
- число кэш-промахов
- объем используемой оперативной памяти
- и т. д.

Инструмент, используемый для анализа работы, называют **профайлером** (profiler). Обычно профилирование выполняется в процессе оптимизации программы.

Магические функции IPython для профилирования:

- `%time` - длительность выполнения отдельного оператора;
- `%timeit` - длительность выполнения отдельного оператора при неоднократном повторе (может использоваться для обеспечения большей точности оценки);
- `%prun` - выполнение кода с использованием профилировщика;
- `%lprun` - пошаговое выполнение кода с применением профилировщика;
- `%memit` - оценка использования оперативной памяти для отдельного оператора;
- `%mprun` - пошаговое выполнение кода с применением профилировщика памяти.

Для работы с одной строкой кода используются строчные магические команды (например, `%time`), для работы с целой ячейкой их блочные аналоги (например, `%%time`).

Магические функции в python

Магические функции (magic functions) — одно из важных улучшений, которые предлагает IPython по сравнению со стандартной оболочкой Python. Эти волшебные команды предназначены для решения распространенных задач при анализе данных с помощью Python. Фактически они контролируют поведение самого IPython.

Волшебные команды позволяют дают более удобный способ решения задач там, где использование синтаксиса Python было бы не очень эффективно / удобно.

Есть два типа магических команд:

- **Line magics** - аналог вызова команд в командной строке консоли. Эти команды начинаются с символа `%`, остальная часть строки — это аргумент, переданный без скобок или кавычек. Остальная часть строки — это аргумент line magic, переданный без скобок или кавычек. Магические линии могут использоваться как выражения, а их возвращаемое значение может быть присвоено переменной.
- **Cell magics** - вызов применяется на несколько строк всей ячейки, для которой применяется данная функция. Эти команды начинаются с префикса `%%`. Cell magic получает текст всей ячейки как переменную строку и может вносить произвольные изменения в получаемые входные данные, которые вообще не обязательно должны быть корректным кодом на Python.

При необходимости можно создавать собственные магические команды.

```
In [39]: %time sum(range(100))
```

```
CPU times: total: 0 ns
Wall time: 0 ns
```

```
Out[39]: 4950
```

```
In [40]: %timeit sum(range(100))
```

```
3.4 µs ± 422 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [41]: %%timeit

```
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

1.1 s ± 278 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Команда %timeit выполняет оценку времени многократного выполнения фрагментов кода и автоматически подстраивает колчество повторов выполнения под длительность работы функции.

Дольные приставки Си при измерении времени:

- 1 ms - 1 миллисекунда (мс): 1/1000 секунды
- 1 µs - 1 микросекунда (мкс): 1/1 000 000 секунды
- 1 ns - 1 наносекунда (нс): 1/1 000 000 000 секунды

In [5]: %%time

```
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

CPU times: total: 938 ms
Wall time: 936 ms

Команда %time выполняет однократный запуск кода. В отличие от %timeit %time не выполняет специальных действий, предотвращающих системные вызовы, поэтому *часто выполнение того же кода с замером %time происходит несколько медленнее чем с %timeit*.

Задача: за разумное время найти дубликаты в списке фильмов, содержащемся в файле tmdb_5000_credits.csv (размер: 4803 строки).

In [42]: `import csv`

```
def read_movies(src, skip_header=True, title_column_ind=1):
    """
    Parameters:
        src (String): имя файла с фильмами
        skip_header (bool, optional): Пропускать ли заголовок?
        title_column_ind (int, optional): Столбец с названиями фильмов
    Returns:
        list: Список названий фильмов из файла (столбец title_column_ind в CSV)
    """
    with open(src) as fd:
        csv_reader = csv.reader(fd, delimiter=',')
        movies = [row[title_column_ind] for row in csv_reader]
        if skip_header:
            movies = movies[1:]
        return movies

# with open('employee_birthday.txt') as csv_file:
#     csv_reader = csv.reader(csv_file, delimiter=',')
#     line_count = 0
#     for row in csv_reader:
#         if line_count == 0:
#             print(f'Column names are {", ".join(row)}')
#             line_count += 1
#         else:
#             print(f'\t{row[0]} works in the {row[1]} department, and was born in {row[2]}')
#             line_count += 1
#     print(f'Processed {line_count} lines.')
```

In [43]: `movies = read_movies('tmdb_5000_credits.csv')`
`movies[:5], type(movies), len(movies)`

Out[43]: (`['Avatar',`
`"Pirates of the Caribbean: At World's End",`
`'Spectre',`
`'The Dark Knight Rises',`
`'John Carter']`,
`list`,
`4803`)

In [44]: `# 1я попытка:`

```
def is_duplicate(needle, haystack):
    for movie in haystack:
        if needle.lower() == movie.lower():
            return True
    return False

def find_duplicate_movies(src='tmdb_5000_credits.csv'):
    movies = read_movies(src)
    duplicates = []
    while movies:
        movie = movies.pop()
        if is_duplicate(movie, movies):
            duplicates.append(movie)
    return duplicates
```

```
In [50]: %%time

duplicates = find_duplicate_movies()
print(duplicates)

['Batman', 'Out of the Blue', 'The Host']
CPU times: total: 11.1 s
Wall time: 11.5 s
```

```
In [46]: %%prun
# ищем "бутылочное горлышко":

duplicates = find_duplicate_movies()
```

Магическая функция `%prun` выдает результаты в формате, принятом у модулей профайлеров `profile` и `cProfile` (см. <https://docs.python.org/3/library/profile.html> (<https://docs.python.org/3/library/profile.html>)).

Столбцы содержат следующую информацию:

- **ncalls** - количество вызовов функций (если дано 2 значения, например 3/1, то это означает, что функция вызывалась рекурсивно (первое число - общее количество вызовов, второе - количество primitive call (вызовов которые не были порождены рекурсией))
- **tottime** - общее количество времени, проведенное в данной функции (*ИСКЛЮЧАЯ время проведенное в вызовах подфункций*)
- **percall** - tottime / ncalls
- **cumtime** - общее количество времени, проведенное в данной функции (*ВКЛЮЧАЯ время проведенное в вызовах подфункций*), это значение корректно рассчитывается и для рекурсивных вызовов функций
- **percall** - cumtime / primitive calls
- **filename:lineno(function)** - расположение функции

```
In [47]: # дополнительная информация по %prun :
%prun?
```

```
In [48]: # исправляем очевидное "слабое место" - огромное количество вызовов Lower()

def is_duplicate2(needle, haystack):
    for movie in haystack:
        if needle == movie:
            return True
    return False

def find_duplicate_movies2(src='tmdb_5000_credits.csv'):
    movies = [movie.lower() for movie in read_movies(src)]
    duplicates = []
    while movies:
        movie = movies.pop()
        if is_duplicate2(movie, movies):
            duplicates.append(movie)
    return duplicates
```

```
In [49]: %%time
```

```
duplicates = find_duplicate_movies2()  
print(duplicates)
```

```
['batman', 'out of the blue', 'the host']  
CPU times: total: 2.7 s  
Wall time: 2.69 s
```

Иногда больше пользы может принести построчный отчет профилировщика. Такая функциональность не встроена в язык Python или оболочку IPython, но можно установить пакет `line_profiler`, обладающий такой возможностью.

Установка пакета `line_profiler`:

- с помощью `pip`: `$ pip install line_profiler`
- с помощью `conda` (в Anaconda Prompt): `$ conda install -c anaconda line_profiler`

Документация: https://github.com/pyutils/line_profiler#id2 (https://github.com/pyutils/line_profiler#id2).

Другой аспект профилирования — количество используемой операциями памяти. Это количество можно оценить с помощью еще одного расширения оболочки IPython — `memory_profiler`.

Установки пакета `memory_profiler`:

- с помощью `pip`: `$ pip install memory_profiler`
- с помощью `conda` (в Anaconda Prompt): `conda install -c anaconda memory_profiler`

Документация:

```
In [18]: # !pip install line_profiler
```

```
In [21]: # !pip install memory_profiler
```

```
In [51]: # загружаем функционал line_profiler в Jupyter:  
%load_ext line_profiler
```

The `line_profiler` extension is already loaded. To reload it, use:
`%reload_ext line_profiler`

```
In [23]: %reload_ext line_profiler
```

```
In [24]: # загружаем функционал memory_profiler в Jupyter:  
%load_ext memory_profiler
```

```
In [52]: %lprun?
```

Сохраняем профилируемый код в файл `.py` :

```
In [53]: %%writefile mprun_demo.py
# v.3
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
    return total
```

Overwriting mprun_demo.py

```
In [54]: # импортируем интересующую функцию из файла:
from mprun_demo import sum_of_lists
```

```
In [55]: %memit sum_of_lists(5000)
```

peak memory: 68.51 MiB, increment: 0.00 MiB

```
In [56]: %lprun -f sum_of_lists sum_of_lists(5000)
```

Необходимо указать ей явным образом, какие функции мы хотели быть профилировать, например так:

```
In[10]: %lprun -f sum_of_lists sum_of_lists(5000)
```

Записываем интересующий код в файл и импортируем из него:

In [57]: %%writefile fdm_v2.py

```
#v.1

import csv

def read_movies(src, skip_header=True, title_column_ind=1):
    """
    Parameters:
        src (String): имя файла с фильмами
        skip_header (bool, optional): Пропускать ли заголовок?
        title_column_ind (int, optional): Столбец с названиями фильмов
    Returns:
        list: Список названий фильмов из файла (столбец title_column_ind в CSV)
    """
    with open(src) as fd:
        csv_reader = csv.reader(fd, delimiter=',')
        movies = [row[title_column_ind] for row in csv_reader]
        if skip_header:
            movies = movies[1:]
        return movies

# исправляем очевидное "слабое место" - огромное количество вызовов lower()

def is_duplicate2(needle, haystack):
    for movie in haystack:
        if needle == movie:
            return True
    return False

def find_duplicate_movies2(src='tmdb_5000_credits.csv'):
    movies = [movie.lower() for movie in read_movies(src)]
    duplicates = []
    while movies:
        movie = movies.pop()
        if is_duplicate2(movie, movies):
            duplicates.append(movie)
    return duplicates
```

Overwriting fdm_v2.py

In [58]: from fdm_v2 import find_duplicate_movies2

In [59]: %lprun -f find_duplicate_movies2 find_duplicate_movies2()

Более 80% времени выполняется проверка is_duplicate2(). Переработаем алгоритм, для оптимизации этой проверки:

In [32]: *# исправляем очередное "слабое место": неоптимальную проверку дубликатов:*

```
def find_duplicate_movies3(src='tmdb_5000_credits.csv'):
    duplicates = []
    unique = set()
    for movie in read_movies(src):
        movie = movie.lower()
        if movie in unique:
            duplicates.append(movie)
        else:
            unique.add(movie)
    return duplicates
```

In [60]: %%time

```
duplicates = find_duplicate_movies3()
print(duplicates)
```

```
['the host', 'out of the blue', 'batman']
CPU times: total: 2.3 s
Wall time: 2.3 s
```

In [34]: *# результат:*

```
duplicates
```

Out[34]: ['the host', 'out of the blue', 'batman']

Если планируется профилировать код модулей и скриптов на Python вне Jupyter то для профилирования может быть удобно использовать следующую технику:

- Добавить перед интересующими функциями декоратор @profile
- Запустить профилирование с помощью утилиты kernprof , пример прфилирования скрипта primes.py: kernprof -l -v primes.py
- Подробнее см.: <https://dwinston.github.io/python-second-language/extras/profiling.html>
(<https://dwinston.github.io/python-second-language/extras/profiling.html>)

На основе cProfile (лежащего в основе rgn) можно сделать декоратор:

```
In [61]: import cProfile, pstats, io
```

```
def profile(fnc):  
  
    """A decorator that uses cProfile to profile a function"""  
  
    def inner(*args, **kwargs):  
  
        pr = cProfile.Profile()  
        pr.enable()  
        retval = fnc(*args, **kwargs)  
        pr.disable()  
        s = io.StringIO()  
        sortby = 'cumulative'  
        ps = pstats.Stats(pr, stream=s).sort_stats(sortby)  
        ps.print_stats()  
        print(s.getvalue())  
        return retval  
  
    return inner
```

```
In [62]: @profile  
def find_duplicate_movies4(src='tmdb_5000_credits.csv'):  
    duplicates = []  
    unique = set()  
    for movie in read_movies(src):  
        movie = movie.lower()  
        if movie in unique:  
            duplicates.append(movie)  
        else:  
            unique.add(movie)  
    return duplicates
```

```
In [63]: find_duplicate_movies4()
```

19396 function calls in 1.915 seconds

Ordered by: cumulative time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.008	0.008	1.915	1.915	C:\Users\alpha\AppData\Local\Temp\ipykernel_7184\3142646199.py:1(find_duplicate_movies4)
1	0.000	0.000	1.903	1.903	C:\Users\alpha\AppData\Local\Temp\ipykernel_7184\507437900.py:3(read_movies)
1	1.704	1.704	1.902	1.902	C:\Users\alpha\AppData\Local\Temp\ipykernel_7184\507437900.py:14(<listcomp>)
4890	0.010	0.000	0.198	0.000	C:\Users\alpha\.conda\envs\teach_e2\lib\encodings\cp1251.py:22(decode)
4890	0.188	0.000	0.188	0.000	{built-in method _codecs.charmap_decode}
4803	0.002	0.000	0.002	0.000	{method 'lower' of 'str' objects}
4800	0.002	0.000	0.002	0.000	{method 'add' of 'set' objects}
1	0.000	0.000	0.000	0.000	{built-in method io.open}
1	0.000	0.000	0.000	0.000	{method '__exit__' of '_io._IOBase' objects}
1	0.000	0.000	0.000	0.000	C:\Users\alpha\.conda\envs\teach_e2\lib_bootlocale.py:11(getpreferredencoding)
1	0.000	0.000	0.000	0.000	{built-in method _locale._getdefaultlocale}
1	0.000	0.000	0.000	0.000	{built-in method _csv.reader}
3	0.000	0.000	0.000	0.000	{method 'append' of 'list' objects}
1	0.000	0.000	0.000	0.000	C:\Users\alpha\.conda\envs\teach_e2\lib\codecs.py:260(__init__)
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

```
Out[63]: ['the host', 'out of the blue', 'batman']
```

Numba

- [к оглавлению](#)

Numba - JIT компилятор с открытым исходным кодом, который компилирует подмножество кода Python и NumPy в быстрый машинный код.

- Официальная страница проекта: <https://numba.pydata.org/> (<https://numba.pydata.org/>)
- **JIT-компиляция** (Just-in-time compilation, компиляция «на лету»), динамическая компиляция (dynamic translation) — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы.

Преимущества и накладные расходы:

- достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом (сравнимая с компилируемыми языками)
- накладные расходы: увеличение потребления памяти (для хранения результатов компиляции) и дополнительные затраты времени на компиляцию на лету.

Ускорение функций на Python

- Numba компилирует функции Python в оптимизированный машинный код с использованием библиотеки для компиляции промышленного уровня **LLVM** (<https://ru.wikipedia.org/wiki/LLVM> (<https://ru.wikipedia.org/wiki/LLVM>)). Численные алгоритмы откомпилированные с помощью Numba могут достигать скорости сопоставимой с исполнением откомпилированного кода на C или FORTRAN.
- Numba обеспечивает удобство работы:
 - нет необходимости уходить от использования обычного интерпретатора Python
 - нет необходимости выполнять отдельную компиляцию кода
 - нет необходимости в установке компилятора C/C++
 - Достаточно использовать декораторы Numba для ваших функций, Numba выполнит все необходимые шаги автоматически.

Разработан для научных вычислений

- Numba спроектирована для работы с массивами и функциями NumPy.
- Numba генерирует специализированный код для различных типов массивов и их размещения для оптимизации производительности.
- Специализированные декораторы могут создавать `ufunc` которые могут использоваться для распространения по массивам NumPy, также как это делают `ufunc` NumPy.
- Numba хорошо интегрирована с работой Jupyter notebooks для обеспечения интерактивных вычислений и с распределенными вычислительными средами, такими как Dask и Spark.

Выполняет распараллеливание ваших алгоритмов

- Numba поддерживает Simplified Threading: может **автоматически параллельно выполнять выражения для NumPy на нескольких ядрах CPU**, что делает простым написание параллельных циклов.
- Numba **поддерживает SIMD Vectorization**: Numba может автоматически транслировать некоторые циклы в векторные инструкции для CPU, что может обеспечивать 2-4 кратный прирост производительности. Numba адаптируется к имеющимся возможностям CPU, определяя и используя поддержку таких SIMD возможностей CPU как SSE, AVX или AVX-512.
- Numba **поддерживает ускорение вычислений на GPU**: поддерживаются драйверы NVIDIA CUDA и AMD ROCm. Numba позволяет писать параллельные GPU алгоритмы полностью из Python.

Переносимые результаты компиляции

- Numba обеспечивает высокую производительность приложений на Python без сложностей бинарной компиляции и создания пакетов. Исходный код остается написан на чистом Python, а Numba обеспечивает его компиляцию на лету. Numba проходит тестирование на более чем 200 различных программно-аппаратных конфигурациях.
- Numba поддерживает:
 - разные CPU: Intel and AMD x86, POWER8/9, ARM.
 - разные GPU: NVIDIA и AMD.
 - разные версии Python: Python 2.7, Python 3.4-3.7
 - разные операционные системы: Windows, macOS, Linux
- Бинарные поставки Numba доступны для большинства систем в виде пакетов conda и wheel для установки с помощью pip.

Документация и учебные материалы:

<http://numba.pydata.org/numba-doc/latest/index.html> (<http://numba.pydata.org/numba-doc/latest/index.html>)

Основные возможности Numba:

- генерация кода "на лету" (во время импорта или во время исполнения, по выбору пользователя)
- генерация нативного кода для CPU (по умолчанию) или для GPU
- интеграция со стеком технологий Python для научных вычислений (на основе NumPy)

```
In [89]: import numpy as np
import numba
from numba import jit, njit
```

```
In [90]: # наивная реализация суммы квадратов элементов матрицы:
```

```
def sum_sq_2d(arr):
    m, n = arr.shape
    result = 0.0
    for i in range(m):
        for j in range(n):
            result += arr[i,j] ** 2
    return result
```

```
In [91]: np.full((10, 10), 42.0)
```

```
Out[91]: array([[42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
 [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
 [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
 [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
 [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
 [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
 [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
 [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
 [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.],
 [42., 42., 42., 42., 42., 42., 42., 42., 42., 42.]])
```

```
In [92]: arr = np.full((1000, 1000), 42.0)
arr[:3, :3], arr.shape
```

```
Out[92]: (array([[42., 42., 42.],
 [42., 42., 42.],
 [42., 42., 42.]]),
 (1000, 1000))
```

Время работы наивной реализации:

```
In [93]: %%timeit
sum_sq_2d(arr)
```

2.15 s ± 564 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Время работы реализации с использованием NumPy:

```
In [94]: %%timeit
np.sum(arr ** 2)
```

20.5 ms ± 4.05 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)

Как работает Numba:

1. Читает байткод Python для декорированной функции.
2. Собирает информацию о типах входных аргументов функций.
3. Анализирует и оптимизирует код.
4. Использует библиотеку для компиляции LLVM для генерации машинного кода функции для конкретного CPU.
5. Данный машинный код используется каждый раз при вызове данной функции (с аргументами того же типа).

Когда Numba даст хороший прирост производительности:

- Если код ориентирован на численные операции, в т.ч.:
 - активно использует NumPy
 - имеется много циклов (большое количество итераций)

Как получить байткод Python:

```
In [95]: import dis
```

```
In [96]: dis.dis(sum_sq_2d)
```

```

4          0 LOAD_FAST           0 (arr)
          2 LOAD_ATTR           0 (shape)
          4 UNPACK_SEQUENCE      2
          6 STORE_FAST          1 (m)
          8 STORE_FAST          2 (n)

5         10 LOAD_CONST         1 (0.0)
         12 STORE_FAST          3 (result)

6         14 LOAD_GLOBAL        1 (range)
         16 LOAD_FAST           1 (m)
         18 CALL_FUNCTION        1
         20 GET_ITER
      >>  22 FOR_ITER              38 (to 62)
         24 STORE_FAST          4 (i)

7         26 LOAD_GLOBAL        1 (range)
         28 LOAD_FAST           2 (n)
         30 CALL_FUNCTION        1
         32 GET_ITER
      >>  34 FOR_ITER              24 (to 60)
         36 STORE_FAST          5 (j)

8         38 LOAD_FAST           3 (result)
         40 LOAD_FAST           0 (arr)
         42 LOAD_FAST           4 (i)
         44 LOAD_FAST           5 (j)
         46 BUILD_TUPLE          2
         48 BINARY_SUBSCR
         50 LOAD_CONST         2 (2)
         52 BINARY_POWER
         54 INPLACE_ADD
         56 STORE_FAST          3 (result)
         58 JUMP_ABSOLUTE       34
      >>  60 JUMP_ABSOLUTE       22

9      >>  62 LOAD_FAST           3 (result)
         64 RETURN_VALUE
```

- Реализация **CPython** (не путайте с Cython !) интерпретирует не непосредственно исходный

код, а компилирует его в байт код и исполняет (интерпретирует) его с помощью виртуальной машины (см: <https://en.wikipedia.org/wiki/CPython> (<https://en.wikipedia.org/wiki/CPython>) , <https://ru.wikipedia.org/wiki/CPython> (<https://ru.wikipedia.org/wiki/CPython>)).

- Байткод Python хранится в автоматических создаваемых при компиляции файлах с расширением `.pyc` в папках `__pycache__` находящихся в папках рядом с файлами с расширением `.py` (кодом модулей и скриптов на Python).
- Байткод создается при компиляции "на лету" кода на Python исполняемого в первый раз.
- Среда Python автоматически отслеживает актуальность байткода в файлах с расширением `.pyc` и при необходимости выполняет их обновление.

Numba является хорошим выбором если ваш код численно ориентирован (выполняет много математических вычислений), много использует NumPy и/или имеет много циклов. В этом примере мы применим самый фундаментальный из JIT-декораторов Numba, `@jit`, чтобы попытаться ускорить некоторые функции.

Пример использования декоратора `@jit`.

Декоратор `@jit` имеет два режим работы:

- режим `numpython`
 - Устанавливается параметром `numpython=True` или использованием декоратора `@njit`
 - Это рекомендуемый для использования и наиболее быстрый режим.
 - Приводит к компиляции кода функции практически не использующего интерпретатор Python.
- режим `object`

In [97]: *# наивная реализация суммы квадратов элементов матрицы:*

```
@njit
def sum_sq_2d_jit__(arr):
    m, n = arr.shape
    result = 0.0
    for i in range(n):
        for j in range(n):
            result += arr[i,j] ** 2
    return result
```

In [98]: `%%time`
во время первого запуска с данным типом параметров производится компиляция функции:
`sum_sq_2d_jit__(arr)`

CPU times: total: 203 ms
Wall time: 205 ms

Out[98]: 1764000000.0

Время работы откомпилированной реализации:

In [99]: `%%timeit`
`sum_sq_2d_jit__(arr)`

4.47 ms ± 276 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

Возможно просмотреть код LLVM, который был сгенерирован при компиляции:

```
In [100]: sum_sq_2d_jit.inspect_types()

sum_sq_2d_jit (array(float64, 2d, C),)
-----
# File: C:\Users\alpha\AppData\Local\Temp\ipykernel_7184\399773312.py
# --- LINE 3 ---

@njit

# --- LINE 4 ---

def sum_sq_2d_jit(arr):

    # --- LINE 5 ---
    # label 0
    #   arr = arg(0, name=arr)  :: array(float64, 2d, C)
    #   $4load_attr.1 = getattr(value=arr, attr=shape)  :: UniTuple(int64 x 2)
    #   $6unpack_sequence.4 = exhaust_iter(value=$4load_attr.1, count=2)  :: UniTup
le(int64 x 2)
    #   del $4load_attr.1
    #   $6unpack_sequence.2 = static_getitem(value=$6unpack_sequence.4, index=0, in
```

Векторизация в NumPy

- [к оглавлению](#)

Векторизация для скалярных функций NumPy

- [к оглавлению](#)

Векторизация позволяет записывать применение функции для перобразования множества значений (вектора) за одну операцию.

Векторизация позволяет:

- писать более компактный и выразительный код
- оптимизировать выполнение векторных операций по сравнению с применением циклов за счет специальных оптимизаций, в т.ч. за счет использования специальных возможностей процессоров, многие из которых поддерживают векторные операции на аппаратном уровне.

В контексте высокоуровневых языков, таких как Python, термин векторизация означает использование оптимизированного заранее откомпилированного кода, написанного на низкоуровневом языке (например C) для выполнения математических операций над множеством значений (вектором, массивом (в т.ч. многомерным)). Это делается вместо явного итерирования по данным на исходном высокоуровневом языке (например с помощью циклов Python).

- Пример решения задачи на скалярном языке (C):

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i][j] += b[i][j];
```

Пример решения задачи на языке, поддерживающим векторные операции:

$a = a + b$

- Аналогичные примеры можно привести при переходе от кода на Python к использованию `ufunc` в NumPy.
- Современные языки, поддерживающие векторные операции: APL, J, Fortran 90, Mata, MATLAB, Analytica, TK Solver (as lists), Octave, R, Cilk Plus, Julia, Perl Data Language (PDL), Wolfram Language, **библиотека NumPy в Python**.
- Но: реализованное в NumPy *множество `ufunc` не обеспечивает решения всех возможных задач преобразования массивов*.

Пример для применения векторизации:

```
In [127]: # подсчитываем количество нулей:
def count_zeros(v):
    result = 0
    while v:
        v, digit = divmod(v, 10)
        if digit == 0:
            result += 1
    return result
```

```
In [128]: import numpy as np
import numpy.random
```

```
In [129]: numpy.random.randint(0, 10000, 100)
```

```
Out[129]: array([9676, 3944, 2869, 7123, 3192, 9918, 4089, 9426, 4486, 4305,  522,
        394, 2221, 6931,  926, 2032, 8022, 4850, 4459, 3796, 8545,  642,
        2552, 6435, 9517, 1239, 5958, 3400, 5267, 8400,  453,  346, 4320,
        5441, 2427, 5942, 4484, 7418, 3399, 5982, 5718, 9141, 2043, 7299,
        5466, 6541, 5550, 6876, 9055, 1920, 4457,  315, 5068, 9992, 8619,
        9038, 3341, 7278, 3180, 4122, 8494, 2410, 4681, 9963, 8403, 2924,
        9574, 4428, 1704, 7969, 5115, 1037, 1352, 8773, 5096, 9011, 9083,
        6601, 5801, 9093,  763, 7848, 6871, 3099, 8565, 9931, 5600, 8111,
        6550, 6402, 9263, 4193, 2105, 8565, 9127, 9425,  619, 4859, 2579,
        4323])
```

```
In [131]: vals = numpy.random.randint(0, 10_000, 1_000_000)
```

```
In [132]: %%time

z_count = 0
for v in vals:
    z_count += count_zeros(v)
print(z_count)
```

```
287641
CPU times: total: 10.1 s
Wall time: 11.5 s
```

```
In [134]: np.sum(np.sin(vals))
```

```
Out[134]: 1375.8971837527477
```

numpy.vectorize - это класс обобщенных функций, который позволяет создавать векторизованные функции в NumPy.

- `numpy.vectorize` позволяет определять векторизованные функции которые принимают массивы NumPy (или вложенные последовательности объектов) и возвращают массивы NumPy (единичные или кортежи).
- Конструктор класса выглядит следующим образом: `class numpy.vectorize(pyfunc, otypes=None, doc=None, excluded=None, cache=False, signature=None)`
- Ключевым аргументом является функция `pyfunc` - функция, которую требуется векторизовать.
- В результате применения конструктора `numpy.vectorize` появляется вызываемый (callable) объект типа `numpy.vectorize`, по сути это есть векторизованная функция.
- Векторизованная функция вызывает функцию `pyfunc` для элементов входных массивов аналогично функции `map` в Python, при этом применяются правила распространения (broadcasting) NumPy.

Подробнее о параметрах `numpy.vectorize` :

- `pyfunc` : callable - функция Python которую необходимо векторизовать
- `otypes` : str or list of dtypes, optional - тип выходных значений векторизованной функции. Может быть передан как строка с описанием кодов типов (typecode characters) или как список спецификаций типов данных. (*См примеры*)
- `doc` : str, optional - строка докумнетации функции, если передан None (значение по умолчанию) будет использована стрка документации функции `pyfunc` .
- `excluded` : set, optional - определение параметров по которым функция НЕ БУДЕТ векторизована, передается множество строк или чисел определяющих аргументы по именам параметров или по их позиции.
- `cache` : bool, optional - если True то при первом вызове кэшируется количество выходных значений, если параметр `otypes` не передан.
- `signature` : string, optional - обобщенная сигнатура функции, например `(m,n),(n)->(m)` для векторизованного матрично-векторного умножения. Если праметр передан `pyfunc` будет вызван для массивов с формой заданной размером соответствующих измерений. По умолчанию считается что `pyfunc` принимает на вход скаляры и возвращает скаляры.

```
In [135]: vcount_zeros = np.vectorize(count_zeros)
```

```
In [136]: type(count_zeros)
```

```
Out[136]: function
```

```
In [137]: type(vcount_zeros)
```

```
Out[137]: numpy.vectorize
```

```
In [138]: %%time
```

```
z_count = vcount_zeros(vals) # применение аналогично использованию ufunc
print(np.sum(z_count))
```

```
287641
```

```
CPU times: total: 2.61 s
```

```
Wall time: 2.61 s
```

```
In [139]: type(z_count)
```

```
Out[139]: numpy.ndarray
```

```
In [140]: # Тип возвращаемых значений определен автоматически:  
z_count[:3], type(z_count[0])
```

```
Out[140]: (array([1, 1, 0]), numpy.int32)
```

```
In [141]: vcount_zeros_f = np.vectorize(count_zeros, otypes=[float]) # явное задание возвращаемого
```

```
In [142]: z_count_f = vcount_zeros_f(vals[:100])  
z_count_f[:3], type(z_count_f[0])
```

```
Out[142]: (array([1., 1., 0.]), numpy.float64)
```

Пример использования параметра `excluded` :

```
In [143]: # Расчет значения полинома с коэффициентами p для значения x:  
def mypolyval(p, x):  
    _p = list(p)  
    res = _p.pop(0)  
    while _p:  
        res = res*x + _p.pop(0)  
    return res
```

```
In [144]: # При векторизации исключаем параметр p из параметров, по которым проводится векторизация  
vpolyval = np.vectorize(mypolyval, excluded=['p'])
```

```
In [145]: vpolyval(p=[1, 2, 3], x=[0, 1, 2, 3])
```

```
Out[145]: array([ 3,  6, 11, 18])
```

```
In [146]: vpolyval(p=[1, 2, 3], x=np.linspace(-1, 1, 100))
```

```
Out[146]: array([2.          , 2.00040812, 2.00163249, 2.00367309, 2.00652995,  
                2.01020304, 2.01469238, 2.01999796, 2.02611978, 2.03305785,  
                2.04081216, 2.04938272, 2.05876951, 2.06897255, 2.07999184,  
                2.09182736, 2.10447913, 2.11794715, 2.1322314 , 2.1473319 ,  
                2.16324865, 2.17998163, 2.19753086, 2.21589634, 2.23507805,  
                2.25507601, 2.27589022, 2.29752066, 2.31996735, 2.34323028,  
                2.36730946, 2.39220488, 2.41791654, 2.44444444, 2.47178859,  
                2.49994898, 2.52892562, 2.5587185 , 2.58932762, 2.62075298,  
                2.65299459, 2.68605244, 2.71992654, 2.75461688, 2.79012346,  
                2.82644628, 2.86358535, 2.90154066, 2.94031221, 2.97990001,  
                3.02030405, 3.06152433, 3.10356086, 3.14641363, 3.19008264,  
                3.2345679 , 3.2798694 , 3.32598714, 3.37292113, 3.42067136,  
                3.46923783, 3.51862055, 3.56881951, 3.61983471, 3.67166616,  
                3.72431385, 3.77777778, 3.83205795, 3.88715437, 3.94306703,  
                3.99979594, 4.05734109, 4.11570248, 4.17488011, 4.23487399,  
                4.29568411, 4.35731048, 4.41975309, 4.48301194, 4.54708703,  
                4.61197837, 4.67768595, 4.74420977, 4.81154984, 4.87970615,  
                4.94867871, 5.0184675 , 5.08907254, 5.16049383, 5.23273135,  
                5.30578512, 5.37965514, 5.45434139, 5.52984389, 5.60616264,  
                5.68329762, 5.76124885, 5.84001632, 5.91960004, 6.          ])
```

Векторизация для векторных функций в NumPy

- [к оглавлению](#)

Обобщенная сигнатура функции

Имеется потребность проводить векторизацию не только скалярных функций (принимающих в качестве аргументов один или несколько (фиксированное число!) скалярных аргументов и возвращающая одно значение), но и "векторных" (в нотации NumPy - работающих с массивами `ndarray` или аналогичными структурами) функций.

- В результате векторизации векторные функции могут эффективно (в смысле компактности записи и эффективности вычислений) применяться для массивов БОльших размерностей.
- Для реализации этого механизма конструктору `numpy.vectorize` необходимо передать информацию о том какая векторная структура у входных параметров и выходных значений. Это делается с помощью передачи *обобщенной сигнатуры функции* через параметр `signature`.

Обобщенная сигнатура функции (`generalized ufunc signature`) определяет как размерности каждого из входных/выходных массивов разбиваются на размерности относящиеся к ядру (т.е. становятся параметрами единичного вызова векторизуемой функции `ufunc`) и на размерности, использующиеся для векторизации.

Основные парвила:

- каждое измерение в сигнатуре соотносится с измерениями соответствующих передаваемых массивов (соответствие строится начиная с конца кортежа, определяющего форму (`shape`) передаваемого массива).
- Измерения ядра, которым присвоены одинаковые имена, должны точно совпадать по размерам, в этом случае распространение (`broadcasting`) не производится.
- При применении векторизации измерения ядра убираются из всех входов, а для остающиеся измерений выполняется бродкастинг для выполнения итераций по ним в рамках работы векторизации.

Примеры обобщенных сигнатур различных функций:

Имя функции	Сигнатуры	Описание
<code>add</code>	<code>() , () -> ()</code>	сложение, бинарная <code>ufunc</code>
<code>sum1d</code>	<code>(i) -> ()</code>	сумма элементов вектора (<code>reduction</code>)
<code>inner1d</code>	<code>(i) , (i) -> ()</code>	скалярное произведение двух векторов (<code>vector-vector multiplication</code>)
<code>matmat</code>	<code>(m,n) , (n,p) -> (m,p)</code>	матричное умножение
<code>vecmat</code>	<code>(n) , (n,p) -> (p)</code>	умножения одномерного вектора (рассматривается как вектор-строка) на матрицу (<code>vector-matrix multiplication</code>)
<code>matvec</code>	<code>(m,n) , (n) -> (m)</code>	умножение матрицы на одномерный вектора (рассматривается как вектор-столбец) (<code>matrix-vector multiplication</code>)
<code>matmul</code>	<code>(m?, n) , (n, p?) -> (m?, p?)</code>	функция, которая реализует все 4 варианта, рассмотренные выше
<code>outer_inner</code>	<code>(i, t) , (j, t) -> (i, j)</code>	произведение двух матриц не по правилу "строка на столбец", а по правилу "строка на строку", при этом индекс второй строки определяет индекс столбца в котором будет помещено произведение в итоговой матрице
<code>cross1d</code>	<code>(3) , (3) -> (3)</code>	Векторное произведение двух векторов размерности 3 (https://ru.wikipedia.org/wiki/Векторное_произведение)

Документация:

- `numpy.ufunc.signature`: - <https://numpy.org/doc/stable/reference/generated/numpy.ufunc.signature.html> (<https://numpy.org/doc/stable/reference/generated/numpy.ufunc.signature.html>)
- более подробно про обобщенную сигнатуру функции: <https://numpy.org/doc/1.17/reference/c-api.generalized-ufuncs.html> (<https://numpy.org/doc/1.17/reference/c-api.generalized-ufuncs.html>)

Для реализованных в NumPy ufunc можно посмотреть их сигнатуру:

```
In [147]: print(np.add.signature)
```

None

Отсутствие сигнатуры означает эквивалентно '(),()->()' (с поправкой на количество параметров функции).

```
In [148]: np.linalg._umath_linalg.det.signature
```

```
Out[148]: '(m,m)->()'
```

```
In [149]: def my_vecmat1(a, b):
           return np.sum(a * b)
```

```
In [150]: my_vecmat1(np.arange(1,4), np.ones(3))
```

```
Out[150]: 6.0
```

```
In [151]: a1 = np.arange(1,13).reshape(3,4)
           a1, a1.shape
```

```
Out[151]: (array([[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12]]),
           (3, 4))
```

```
In [152]: np.ones(4)
```

```
Out[152]: array([1., 1., 1., 1.])
```

```
In [153]: # (наверное) неожиданный результат:
           my_vecmat1(a1, np.ones(4))
```

```
Out[153]: 78.0
```

```
In [155]: def my_vecmat2(a, b):
           return sum(x*y for x, y in zip(a, b))
```

```
In [156]: my_vecmat2(np.arange(1,4), np.ones(3))
```

```
Out[156]: 6.0
```

```
In [270]: # (наверное) неожиданный результат:
           my_vecmat2(a1, np.ones(3))
```

```
Out[270]: array([15., 18., 21., 24.])
```

```
In [157]: # выполняем векторизацию векторной функции my_vecmat1 с описанием сигнатуры:
vmy_vecmat1 = np.vectorize(my_vecmat1, signature='(i),(i)->()')
```

```
In [158]: a1
```

```
Out[158]: array([[ 1,  2,  3,  4],
                 [ 5,  6,  7,  8],
                 [ 9, 10, 11, 12]])
```

```
In [159]: np.ones(4)
```

```
Out[159]: array([1., 1., 1., 1.])
```

```
In [160]: # применение векторизованной векторной функции:
vmy_vecmat1(a1, np.ones(4))
```

```
Out[160]: array([10., 26., 42.])
```

Что произошло:

1. На входе: (3, 4), (4)
2. Сигнатура ядра: (i),(i)->()
3. Векторизация: (A, i),(i)->(A), где A=3, i=4
4. Результат: (3)

```
In [161]: vmy_vecmat2= np.vectorize(my_vecmat2, signature='(i),(i)->()')
```

```
In [162]: # указание правильной сигнатуры при векторизации позволило получить ожидаемый результат:
vmy_vecmat2(a1, np.ones(4))
```

```
Out[162]: array([10., 26., 42.])
```

```
In [163]: b2 = np.vstack((np.ones((1,4)), np.full((1,4), 2), np.full((1,4), 3)))
b2, b2.shape
```

```
Out[163]: (array([[1., 1., 1., 1.],
                 [2., 2., 2., 2.],
                 [3., 3., 3., 3.]]),
          (3, 4))
```

```
In [164]: a1
```

```
Out[164]: array([[ 1,  2,  3,  4],
                 [ 5,  6,  7,  8],
                 [ 9, 10, 11, 12]])
```

```
In [165]: vmy_vecmat1(a1, b2)
```

```
Out[165]: array([ 10.,  52., 126.])
```

Что произошло:

1. На входе: (3, 4), (3, 4)
2. Сигнатура ядра: (i),(i)->()

3. Векторизация: $(A, i), (A, i) \rightarrow (A)$, где $A=3$, $i=4$

4. Результат: (3)

На самом деле это даже более простой случай, т.к. в первом случае для второго аргумента использовалось распространение (broadcasting)!

Кроме `numpy.vectorize` имеется еще функция `numpy.frompyfunc` которая позволяет преобразовывать скалярные функции Python в `ufunc` NumPy и использовать их с применением правил распространения.

```
In [166]: # Пример numpy.frompyfunc:

oct_array = np.frompyfunc(oct, 1, 1)
oct_array(np.array((10, 30, 100)))
```

```
Out[166]: array(['0o12', '0o36', '0o144'], dtype=object)
```

Векторизация в Numba

- [к оглавлению](#)

Применение векторизации в Numba

```
In [167]: # простой способ определить, какой тип Numba будет использовать для этих значений:
numba.typeof(vals[0])
```

```
Out[167]: int32
```

```
In [168]: # векторизация с помощью Numba (в явном виде передаем типы, компиляция происходит сразу)

numba_vcount_zeros = numba.vectorize(['int32(int32)'])(count_zeros)
```

```
In [169]: %%time

z_count = numba_vcount_zeros(vals)
print(np.sum(z_count))
```

```
287641
CPU times: total: 31.2 ms
Wall time: 24.9 ms
```

```
In [170]: # альтернативный способ:
from numba import vectorize

@vectorize(['int32(int32)'])
def numba2_vcount_zeros(v):
    result = 0
    while v:
        v, digit = divmod(v, 10)
        if digit == 0:
            result += 1
    return result
```

```
In [171]: %%time
```

```
z_count = numba2_vcount_zeros(vals)
print(np.sum(z_count))
```

287641

CPU times: total: 31.2 ms

Wall time: 30.9 ms

Декоратор @guvectorize

А что, если ускорить расчет перенеся все выполняемые вычисления в векторизованную функцию?

- декоратор `vectorize()` в Numba позволяет реализовывать скалярные `ufuncs`, которые обрабатывают один элемент за раз.
- декоратор `guvectorize()` идет на шаг вперед и позволяет векторизовать векторные `ufunc` которые обрабатывают массивы определенных размеров и возвращают массивы определенных размеров. Типовой пример, это расчет медианы или фильтры свертки (`convolution filter`).
- в отличие от функций, полученных с помощью `vectorize()`, функции, полученные с помощью `guvectorize()`, не возвращают своих значений, вместо этого они получают массив для возвращаемого значения как аргумент функции и заполняют его во время работы. Это происходит из-за того что в реальности массив формируется с помощью механизмов NumPy и потом для него вызывается код сгенерированный с помощью Numba.
- Обобщенные универсальные функции (`generalized universal functions`) требуют описания сигнатуры размерностей для которых реализована функция ядра. В Numba эта сигнатура определяется аналогично NumPy `generalized-ufunc signature`. (Не надо путать с сигнатурой типов, которую обычно требует Numba). Подробнее см.: <https://numpy.org/doc/1.17/reference/c-api.generalized-ufuncs.html> (<https://numpy.org/doc/1.17/reference/c-api.generalized-ufuncs.html>)

Рассмотрим, например сигнатуру матричного умножения `'(m,n), (n,p) -> (m,p)'`. Из нее видно, что:

- Первая с конца размерность первого аргумента и вторая с конца размерность второго аргумента должны совпадать (т.е. должно выполняться правило матричного умножения).
- Последние две размерности результата определяются соответствующими (по именам) размерностями первого и второго аргумента.
- Важно помнить: **соответствие** реальных размерностей передаваемых массивов именам **сигнатуры строится начиная с конца кортежа, определяющего форму (shape) передаваемого массива**.

каждое измерение в сигнатуре соотносится с измерениями соответствующих передаваемых массивов (

При написании функции ядра для `gufunc` необходимо:

- Продумать сигнатуру (`generalized-ufunc signature`) функции.
- Реализовывать функции соблюдая правила для размерностей вынесенные в сигнатуре
- Функция ядра для `gufunc` в Numba принимает в качестве параметров как сами аргументы функции так и переменную в которую будет помещаться результат работы функции
- Входной параметр для хранения результата является последним параметром функции.
- У функции не должно быть возвращаемых значений, все результаты должны сохраняться в последнем входном параметре функции.
- Последствия изменения значений других аргументов, кроме последнего, неопределены, поэтому полагаться на эти изменения нельзя.


```
In [172]: # реализация ядра матричного умножения с сигнатурой '(m,n),(n,p)->(n,p)
def matmulcore(A, B, C):
    m, n = A.shape
    n, p = B.shape
    for i in range(m):
        for j in range(p):
            C[i, j] = 0
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]
```

- Обратите внимание как размерности m , n и p извлекаются из входных аргументов.
- Размерность n извлекается дважды, для того чтобы подчеркнуть необходимость совпадения значений. На практике это действие не является необходимым.

Для построения `generalized-ufunc` из созданной функции ядра можно как явно вызывать функцию `numba.guvectorize` так и использовать декоратор `@guvectorize`. Интерфейс `numba.guvectorize` аналогичен функции `vectorize`, но дополнительно требует передачи сигнатуры.

```
In [173]: from numba import guvectorize
```

```
In [174]: gu_matmul = numba.guvectorize(['float32[:,:], float32[:,:], float32[:,:]', 'float64[:,:]'
                                         '(m,n),(n,p)->(n,p)')(matmulcore)
```

Результат: `gufunc`, которая может быть использована как любая `gufunc` в NumPy. Поддерживается распространение по правилам NumPy.

```
In [176]: matrix_ct = 10000
gu_test_A = np.arange(matrix_ct * 2 * 4, dtype=np.float32).reshape(matrix_ct, 2, 4)
gu_test_B = np.arange(matrix_ct * 4 * 5, dtype=np.float32).reshape(matrix_ct, 4, 5)
```

```
In [177]: %timeit gu_matmul(gu_test_A, gu_test_B)
```

3.72 ms \pm 538 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

Различия между функциями `vectorize` и `guvectorize`:

1. `vectorize` генерирует `ufuncs`, `guvectorize` генерирует `generalized-ufuncs`
2. В обоих случаях сигнатуры для типов входных аргументов и возвращаемых значений представлены в виде списка, но в функции `vectorize` для их определения используются сигнатуры, тогда как в `guvectorize` вместо этого используются списки типов, и последним специфицируется возвращаемое значение.
3. Для `guvectorize` необходимо передать сигнатуру NumPy `generalized-ufunc signature`. Эта сигнатура должна соответствовать переданной сигнатуре типов.
4. Помните, что в `guvectorize` результат передается через последний параметр функции, тогда как в `vectorize` результат возвращается функцией ядра.

```
In [178]: @guvectorize(['int32[:]', int32'], '(n)->()')
def numba_vcount_zeros_arr(arr, result):
    result = 0
    for i in range(arr.shape[0]):
        v = arr[i]
        while v:
            v, digit = divmod(v, 10)
            if digit == 0:
                result += 1
```

```
In [185]: vals.shape
```

```
Out[185]: (1000000,)
```

```
In [184]: %%timeit
```

```
z_count = numba_vcount_zeros_arr(vals)
# print(z_count)
```

5.71 μ s \pm 486 ns per loop (mean \pm std. dev. of 7 runs, 100,000 loops each)

- базовая реализация - 12 s
- реализация с векторизацией - 2.6 s 5x
- реализация с векторизацией на Numba - 31 ms 400x
- реализация с guvectorize на Numba - 6.2 mks 1 935 483x

```
In [183]: # Итоговый прирост производительности:
12/0.0000062
```

```
Out[183]: 1935483.870967742
```

```
In [ ]:
```