Лекция 3: Оптимизация выполнения кода, векторизация, Numba

Автор: Сергей Вячеславович Макрушин e-mail: SVMakrushin@fa.ru

Финансовый универсиет, 2020 г.

При подготовке лекции использованы материалы:

- Документация к рассмотренным пакетам

V 0.3 17.09.2020

Разделы:

-

- к оглавлению

In [1]:
```
# загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v1.css")
HTML(html.read().decode('utf-8'))
```

Out[1]: b.n { font-weight: normal; } b.grbg { background-color: #a0a0a0; } b.r { color: #ff0000; } b.b { color: #0000ff; } b.g { color: #00ff00; } ul.s { // list-style-type: none; list-style: none; // background-color: #ff0000; // color: #ffff00; // padding-left: 1.2em; // text-indent: -1.2em; } li.t { list-style: none; // padding-left: 1.2em; // text-indent: -1.2em; } *.r { color: #ff0000; } li.t:before { content: "\21D2"; // content: "►"; // padding-left: -1.2em; text-indent: -1.2em; display: block; float: left; // width: 1.2em; // color: #ff0000; } i.m:before { font-style: normal; content: "\21D2"; } i.m { font-style: normal; } /*--------------------*/ /* em { font-style: normal; } */ em.bl { font-style: normal; font-weight: bold; } /* em.grbg { font-style: normal; background-color: #a0a0a0; } */ em.cr { font-style: normal; color: #ff0000; } em.cb { font-style: normal; color: #0000ff; } em.cg { font-style: normal; color: #00ff00; } /*--------------------*/ em.qs { font-style: normal; } em.qs::before { font-weight: bold; color: #ff0000; content: "Q:"; } em.an { font-style: normal; } em.an:before { font-weight: bold; color: #0000ff; content: "A:"; } em.nt { font-style: normal; } em.nt:before { font-weight: bold; color: #0000ff; content: "Note:"; } em.ex { font-style: normal; } em.ex:before { font-weight: bold; color: #00ff00; content: "Ex:"; } em.df { font-style: normal; } em.df:before { font-weight: bold; color: #0000ff; content: "Def:"; } em.pl { font-style: normal; } em.pl:before { font-weight: bold; color: #0000ff; content: "+"; } em.mn { font-style: normal; } em.mn:before { font-weight: bold; color: #0000ff; content: "-"; } em.plmn { font-style: normal; } em.plmn:before { font-weight: bold; color: #0000ff; content: "\00B1;\\±"; } em.hn { font-style: normal; } em.hn:before { font-weight: bold; color: #0000ff; content: "\21D2;\\⇒"; }

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 состоящем из множества строк процессе 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 обсепечения 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`).

In [18]:
```
%time sum(range(100))
```

```
Wall time: 0 ns
```

Out[18]:
```
4950
```

In [19]:
```
%timeit sum(range(100))
```

```
1.14 µs ± 121 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [20]:
```
%%timeit

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

```
329 ms ± 94 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 выполненеия под длительность работы функции.

Дольные приставки Си при измерении времени:

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

In [21]:
```
%%time

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

```
Wall time: 298 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 [22]:
```
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 [23]:
```
movies = read_movies('tmdb_5000_credits.csv')
movies[:5], type(movies), len(movies)
```

Out[23]:
```
(['Avatar',
  "Pirates of the Caribbean: At World's End",
  'Spectre',
  'The Dark Knight Rises',
  'John Carter'],
 list,
 4803)
```

In [24]:
```
# 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 [25]:
```
%%time

duplicates = find_duplicate_movies()
print(duplicates)
```

```
['Batman', 'Out of the Blue', 'The Host']
Wall time: 1.89 s
```

In [26]:
```
%%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).

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, провдеенное в данной функции (*ИСКЛЮЧАЯ время проведенное в вызовах подфункций*)
- **percall** - tottime / ncalls
- **cumtime** - общее количество времени, провдеенное в данной функции (*ВКЛЮЧАЯ время проведенное в вызовах подфункций*), это значение корректно расчитывается и для рекурсивных вызовов функций
- **percall** - cumtime / primitive calls
- **filename:lineno(function)** - расположение функции

In [27]:
```
# дополнительная информация по %prun :
```

```
%prun?
```

In [28]:
```
# 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 [29]:
```
%%time

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

```
['batman', 'out of the blue', 'the host']
Wall time: 787 ms
```

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

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 [30]:
```
# загружаем функционал line_profiler в Jupyther:
%load_ext line_profiler
```

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

In [31]:
```
%reload_ext line_profiler
```

In [32]:
```
# загружаем функционал memory_profiler в Jupyther:
```

```
%load_ext memory_profiler
```

In [33]:
```
%lprun?
```

Сохраняем профилируемый код в файл .py :

In [34]:
```
%%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 [35]:
```
# импортируем интересующую функцию из файла:
from mprun_demo import sum_of_lists
```

In [36]:
```
%memit sum_of_lists(5000)
```

```
peak memory: 67.11 MiB, increment: 0.71 MiB
```

In [37]:
```
%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 [38]:
```
%%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 [39]:
```
from fdm_v2 import find_duplicate_movies2
```

In [40]:
```
%lprun -f find_duplicate_movies2 find_duplicate_movies2()
```

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

In [41]:
```
# 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 [42]:
```
%%time

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

```
['the host', 'out of the blue', 'batman']
Wall time: 556 ms
```

In [43]:
```
# результат:
duplicates
```

Out[43]:
```
['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`
- Запуститить 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

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

In [44]:
```
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 [45]:
```
@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 [46]:
```
find_duplicate_movies4()
```

```
         19396 function calls in 0.571 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    0.571    0.571 C:\Users\super\AppData\Local\Temp/ipykernel_
14880/4011568183.py:1(find_duplicate_movies4)
        1    0.000    0.000    0.568    0.568 C:\Users\super\AppData\Local\Temp/ipykernel_
14880/2195998717.py:3(read_movies)
        1    0.509    0.509    0.568    0.568 C:\Users\super\AppData\Local\Temp/ipykernel_
14880/2195998717.py:14(<listcomp>)
     4890    0.002    0.000    0.059    0.000 C:\Python39\lib\encodings\cp1251.py:22(decod
e)
     4890    0.057    0.000    0.057    0.000 {built-in method _codecs.charmap_decode}
     4803    0.000    0.000    0.000    0.000 {method 'lower' of 'str' objects}
     4800    0.000    0.000    0.000    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:\Python39\lib\_bootlocale.py:11(getpreferr
edencoding)
        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}
        1    0.000    0.000    0.000    0.000 C:\Python39\lib\codecs.py:260(__init__)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' obje
cts}
        3    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
```

Out[46]: ['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/
- **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 оптимзированный машинный код с использованием библиотеки для компиляции промышленного уровня **LLVM** (https://ru.wikipedia.org/wiki/LLVM). Численные алгоритмы откомпелированные 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++
    - Дотстаточно использовать декораторы Numba для ваших функций, Numba выполнит все необходимы 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 созвать 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 распеределнными вычислительными средами, такими как Dask и Spark.

Выполняет распараллеливание ваших алгоритмов

- Numba поддерживает Simplified Threading: может автоматически 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 таки 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-апаратных 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 паетов conda и wheel для инсталляции с помощью pip.

Документация и учебные материалы:

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 [47]:
```
import numpy as np
import numba
from numba import jit, njit
```

In [48]:
```
# 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 [49]:
```
np.full((10, 10), 42.0)
```

Out[49]:
```
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 [50]:
```
arr = np.full((1000, 1000), 42.0)
arr[:3, :3], arr.shape
```

Out[50]:
```
(array([[42., 42., 42.],
        [42., 42., 42.],
        [42., 42., 42.]]),
 (1000, 1000))
```

Время работы наивной реализации:

In [51]:
```
%%timeit
sum_sq_2d(arr)
```

```
419 ms ± 20.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Время работы реализации с использованием NumPy:

In [52]:
```
%%timeit
np.sum(arr ** 2)
```

```
3.42 ms ± 206 µs 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 фонкции для конкретного CPU.
5. Данный машинный код используется каждый раз при вызове данной функции (с аргументами тогоже 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 [53]:
```
import dis
```

In [54]:
```
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://ru.wikipedia.org/wiki/CPython).
- Байткод Python хранится в автоматических создаваемых при компиляци 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 комплияции "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 `nopython`
  - Устанавливается параметром `nopython=True` или использованием декоратора `@njit`
  - Это рекомендуемый для использования и наиболее быстрый режим.
  - Приводит к компиляции кода функции практически не используещего интерпретатор Python.
- режим `object`

In [55]:

```
# Пример использования декоратора @jit с параметром nopython=True

@jit(nopython=True) # Set "nopython" mode for best performance, equivalent to @njit
```

```
def go_fast(a): # Function is compiled to machine code when called the first time
    trace = 0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting
```

In [56]:
```
# 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 [57]:
```
%%time
# во время первого запуска с данным типом параметров производистя компиляция функции:
sum_sq_2d_jit(arr)
```

```
Wall time: 413 ms
1764000000.0
```

Out[57]:

Время работы откомпилированной реализации:

In [58]:
```
%%timeit
sum_sq_2d_jit(arr)
```

```
1.16 ms ± 74.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Возможно просмотреть код LLVM, который был сгенирирован при компиляции:

In [59]:
```
sum_sq_2d_jit.inspect_types()
```

```
sum_sq_2d_jit (array(float64, 2d, C),)
--------------------------------------------------------------------------------
# File: C:\Users\super\AppData\Local\Temp/ipykernel_14880/2903878094.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)  :: UniTuple(int6
4 x 2)
    #   del $4load_attr.1
    #   $6unpack_sequence.2 = static_getitem(value=$6unpack_sequence.4, index=0, index_var
=None, fn=<built-in function getitem>)  :: int64
    #   $6unpack_sequence.3 = static_getitem(value=$6unpack_sequence.4, index=1, index_var
=None, fn=<built-in function getitem>)  :: int64
    #   del $6unpack_sequence.4
    #   m = $6unpack_sequence.2  :: int64
    #   del m
    #   del $6unpack_sequence.2
```

```
    #   n = $6unpack_sequence.3  :: int64
    #   del $6unpack_sequence.3

    m, n = arr.shape

    # --- LINE 6 ---
    #   result = const(float, 0.0)  :: float64

    result = 0.0

    # --- LINE 7 ---
    #   $16load_global.6 = global(range: <class 'range'>)  :: Function(<class 'range'>)
    #   $20call_function.8 = call $16load_global.6(n, func=$16load_global.6, args=[Var(n,
2903878094.py:5)], kws=(), vararg=None, target=None)  :: (int64,) -> range_state_int64
    #   del $16load_global.6
    #   $22get_iter.9 = getiter(value=$20call_function.8)  :: range_iter_int64
    #   del $20call_function.8
    #   $phi24.0 = $22get_iter.9  :: range_iter_int64
    #   del $22get_iter.9
    #   jump 24
    # label 24
    #   $24for_iter.1 = iternext(value=$phi24.0)  :: pair<int64, bool>
    #   $24for_iter.2 = pair_first(value=$24for_iter.1)  :: int64
    #   $24for_iter.3 = pair_second(value=$24for_iter.1)  :: bool
    #   del $24for_iter.1
    #   $phi26.1 = $24for_iter.2  :: int64
    #   del $24for_iter.2
    #   branch $24for_iter.3, 26, 64
    # label 26
    #   del $24for_iter.3
    #   i = $phi26.1  :: int64
    #   del $phi26.1

    for i in range(n):

        # --- LINE 8 ---
        #   result.3 = phi(incoming_values=[Var(result, 2903878094.py:6), Var(result.2, 29
03878094.py:8)], incoming_blocks=[0, 62])  :: float64
        #   del result.2
        #   $28load_global.2 = global(range: <class 'range'>)  :: Function(<class 'rang
e'>)
        #   $32call_function.4 = call $28load_global.2(n, func=$28load_global.2, args=[Var
(n, 2903878094.py:5)], kws=(), vararg=None, target=None)  :: (int64,) -> range_state_int64
        #   del $28load_global.2
        #   $34get_iter.5 = getiter(value=$32call_function.4)  :: range_iter_int64
        #   del $32call_function.4
        #   $phi36.1 = $34get_iter.5  :: range_iter_int64
        #   del $34get_iter.5
        #   jump 36
        # label 36
        #   result.2 = phi(incoming_values=[Var(result.3, 2903878094.py:8), Var(result.1,
2903878094.py:9)], incoming_blocks=[26, 38])  :: float64
        #   $36for_iter.2 = iternext(value=$phi36.1)  :: pair<int64, bool>
        #   $36for_iter.3 = pair_first(value=$36for_iter.2)  :: int64
        #   $36for_iter.4 = pair_second(value=$36for_iter.2)  :: bool
        #   del $36for_iter.2
        #   $phi38.2 = $36for_iter.3  :: int64
        #   del $36for_iter.3
        #   branch $36for_iter.4, 38, 62
        # label 38
        #   del $36for_iter.4
        #   j = $phi38.2  :: int64
        #   del $phi38.2

        for j in range(n):
```

```
            # --- LINE 9 ---
            #   $48build_tuple.7 = build_tuple(items=[Var(i, 2903878094.py:7), Var(j, 2903
878094.py:8)])  :: UniTuple(int64 x 2)
            #   del j
            #   $50binary_subscr.8 = getitem(value=arr, index=$48build_tuple.7, fn=<built-
in function getitem>)  :: float64
            #   del $48build_tuple.7
            #   $const52.9 = const(int, 2)  :: Literal[int](2)
            #   $54binary_power.10 = $50binary_subscr.8 ** $const52.9  :: float64
            #   del $const52.9
            #   del $50binary_subscr.8
            #   $56inplace_add.11 = inplace_binop(fn=<built-in function iadd>, immutable_f
n=<built-in function add>, lhs=result.2, rhs=$54binary_power.10, static_lhs=Undefined, sta
tic_rhs=Undefined)  :: float64
            #   del result.2
            #   del $54binary_power.10
            #   result.1 = $56inplace_add.11  :: float64
            #   del $56inplace_add.11
            #   jump 36
            # label 62
            #   del result.3
            #   del i
            #   del $phi38.2
            #   del $phi36.1
            #   del $36for_iter.4
            #   jump 24

            result += arr[i,j] ** 2

    # --- LINE 10 ---
    # label 64
    #   del result.1
    #   del result
    #   del n
    #   del arr
    #   del $phi26.1
    #   del $phi24.0
    #   del $24for_iter.3
    #   $66return_value.1 = cast(value=result.3)  :: float64
    #   del result.3
    #   return $66return_value.1

    return result


================================================================================
```

Векторизация

- к оглавлению

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

Векторизация позвоялет:

- писать более компактный и выразительный код
- оптимизировать выполенние 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, теримн векторизация означает использование оптимизированного заранее откомпилированного кода, написанного на низкоуровневом языке (например C) для выполнения математических операций над множестовм 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 *множестов ufunc не обеспечивает решения всех возможных задач преобразования массивов*.

Пример для применения векторизации:

In [60]:
```
# подсчитываем количество нулей:
def count_zeros(v):
    result = 0
    while v:
        v, digit = divmod(v, 10)
        if digit == 0:
            result += 1
    return result
```

In [61]:
```
import numpy as np
import numpy.random
```

In [62]:
```
numpy.random.randint(0, 10000, 100)
```

Out[62]:
```
array([5881, 3591,  166, 1638, 2783, 8275, 4246, 4434, 8289, 3469, 7775,
        895, 2268, 2352, 6928, 6975, 6325, 6015, 7325, 8755, 7331, 9553,
       9803, 6038, 5670, 2131, 9324, 2090, 8288, 8967, 4946, 8760, 9550,
       1370, 4359,  403, 6418, 7625, 3744, 4248, 1698, 4811, 5519, 5574,
       1840, 2550, 5506, 8791, 7943, 2240, 6035, 1462, 3662, 6176, 4825,
       5834, 9845, 7172, 4421, 4322, 8090, 2385, 9891, 7222, 6920, 8579,
       4651, 7958, 3058, 8507, 4497, 8463, 6291, 9762,  448, 2874, 3544,
       9581, 9005, 6371, 8998, 1939, 4729, 2476, 8768,  501, 2706,  858,
       1712, 4805, 1580, 8004, 8452, 2030, 6226, 1220, 1033, 7846, 1723,
       4584])
```

In [63]:
```
vals = numpy.random.randint(0, 10000, 10000000)
```

In [64]:
```
%%time
```

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

```
2885863
Wall time: 13.7 s
```

Векторизация в NumPy

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 массиов 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 соответсвующих измерений. По умолчанию считается что `pyfunc` принимает на вход скаляры и возвращает скаляры.

In [65]:
```
vcount_zeros = np.vectorize(count_zeros)
```

In [66]:
```
type(vcount_zeros)
```

Out[66]:
```
numpy.vectorize
```

In [67]:
```
%%time
```

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

```
2885863
Wall time: 4.11 s
```

In [68]:
```
# Тип возвращаемых значений определен автоматически:
z_count[:3], type(z_count[0])
```

Out[68]:
```
(array([0, 1, 0]), numpy.int32)
```

In [69]:
```
vcount_zeros_f = np.vectorize(count_zeros, otypes=[float]) # явное задание возвращаемого т
```

In [70]:
```
z_count_f = vcount_zeros_f(vals[:100])
z_count_f[:3], type(z_count_f[0])
```

Out[70]:
```
(array([0., 1., 0.]), numpy.float64)
```

Пример использования параметра `excluded` :

In [71]:
```
# 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 [72]:
```
# При векторизации исключаем параметр p из параметров, по которым проводится векторизация.
vpolyval = np.vectorize(mypolyval, excluded=['p'])
```

In [73]:
```
vpolyval(p=[1, 2, 3], x=[0, 1])
```

Out[73]:
```
array([3, 6])
```

In [74]:
```
vpolyval(p=[1, 2, 3], x=np.linspace(-1, 1, 100))
```

Out[74]:
```
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.        ])
```

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 анлогоичными структурами) функций.

- В результате векторизации векторные функции могут эффективно (в смысле компактности записи и эффективности вычислений) применяться для массивов бОльших разменостей.
- 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 `pyfunc`) 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) предаваемого массива).
- Измерения ядра, которым присвоены одинаковые имена, должны точно совпадать по размерам, в этом случае распространение (bradcasting) 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
add	`(),()->()`	сложение, бинарная ufunc
sum1d	`(i)->()`	сумма элементов вектора (reduction)
inner1d	`(i),(i)->()`	скалярное произведение двух векторов (vector-vector multiplication)
matmat	`(m,n),(n,p)->(m,p)`	матричное умножение
vecmat	`(n),(n,p)->(p)`	умножения одномерного вектора (рассматривается как вектор-строка) на матрицу (vector-matrix multiplication)
matvec	`(m,n),(n)->(m)`	умножение матрицы на одномерный вектора (рассматривается как вектор-столбец) (matrix-vector multiplication)
matmul	`(m?,n),(n,p?)->(m?,p?)`	функция, которая реализует все 4 варианта, рассмотренные выше

Имя функции	Сигнатуры	Описание
outer_inner	`(i,t),(j,t)->(i,j)`	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
cross1d	`(3),(3)->(3)`	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
- более подробно про обобщенную сигнатуру функции: https://numpy.org/doc/1.17/reference/c-api.generalized-ufuncs.html

Для реализованных в NumPy ufunc можно просмотреть их сигнатуру:

In [75]:
```
print(np.add.signature)
```

```
None
```

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

In [76]:
```
np.linalg._umath_linalg.det.signature
```

Out[76]:
```
'(m,m)->()'
```

In [77]:
```
def my_vecmat1(a, b):
    return np.sum(a * b)
```

In [78]:
```
my_vecmat1(np.arange(1,4),np.ones(3))
```

Out[78]:
```
6.0
```

In [79]:
```
a1 = np.arange(1,13).reshape(3,4)
a1, a1.shape
```

Out[79]:
```
(array([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]]),
 (3, 4))
```

In [80]:
```
np.ones(4)
```

Out[80]:
```
array([1., 1., 1., 1.])
```

In [81]:
```
# (наверное) неожиданный результат:
my_vecmat1(a1, np.ones(4))
```

Out[81]:
```
78.0
```

In [82]:
```
def my_vecmat2(a, b):
```

```
    return sum(x*y for x, y in zip(a, b))
```

In [83]:
```
my_vecmat2(np.arange(1,4),np.ones(3))
```

Out[83]:
```
6.0
```

In [84]:
```
# (наверное) неожиданный результат:
my_vecmat2(a1, np.ones(3))
```

Out[84]:
```
array([15., 18., 21., 24.])
```

In [85]:
```
# выполняем векторизацию векторной функции my_vecmat1 с описанием сигнатуры:
vmy_vecmat1 = np.vectorize(my_vecmat1, signature='(i),(i)->()')
```

In [86]:
```
a1
```

Out[86]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [87]:
```
np.ones(4)
```

Out[87]:
```
array([1., 1., 1., 1.])
```

In [88]:
```
# применение векторизованной векторной функции:
vmy_vecmat1(a1, np.ones(4))
```

Out[88]:
```
array([10., 26., 42.])
```

Что произошло:

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

In [89]:
```
vmy_vecmat2= np.vectorize(my_vecmat2, signature='(i),(i)->()')
```

In [90]:
```
# указание правильной сигнатуры при векторизации позволило получить ожидаемый результат:
vmy_vecmat2(a1, np.ones(4))
```

Out[90]:
```
array([10., 26., 42.])
```

In [91]:
```
b2 = np.vstack((np.ones((1,4)), np.full((1,4), 2), np.full((1,4), 3)))
b2, b2.shape
```

Out[91]:
```
(array([[1., 1., 1., 1.],
        [2., 2., 2., 2.],
        [3., 3., 3., 3.]]),
 (3, 4))
```

In [92]:
```
a1
```

Out[92]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [93]:
```
vmy_vecmat1(a1, b2)
```

Out[93]:
```
array([ 10.,  52., 126.])
```

Что произошло:

1. На входе: `(3, 4),(3, 4)`
2. Сигнатура ядра: `(i),(i)->()`
3. Векторизаця: `(A, i),(A, i)->(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` имется еще функция `numpy.frompyfunc` которая позволяет проеобразовывать скалярные функции Python в ufunc NumPy и использовать их с применением правил распространения.

In [94]:
```
# Пример numpy.frompyfunc:

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

Out[94]:
```
array(['0o12', '0o36', '0o144'], dtype=object)
```

Применение векторизации в Numba

In [95]:
```
# простой способ определить, какой тип Numba будет использовать для этих значений:
numba.typeof(vals[0])
```

Out[95]:
```
int32
```

In [96]:
```
# векторизация с помощью Numba (в явном виде передаем типы, компиляция происходит сразу):

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

In [97]:
```
%%time

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

```
2885863
Wall time: 102 ms
```

In [98]:
```
# 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 [99]:

```
%%time

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

```
2885863
Wall time: 125 ms
```

Декоратор @guvectorize

А что, если ускорить рассчет 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 массиывы 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 значенйи, 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

Рассмтрим, намриер сигнатуру матичного умножения `'(m,n), (n,p) -> (m,p)'`. Из нее видно, что:

- Ппервая 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) предаваемого массива**.

каждое измерение в сигнатуре соотносится с измерениями соответствующих передаваемых массивов (

При написании функции ядра для gufunc необходимо:

- Продумать сигнатуру (generalized-ufunc signature) функции.
- Рреализовывать 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 [100…
```
# 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 [101…
```
from numba import guvectorize
```

In [102…
```
gu_matmul = numba.guvectorize(['float32[:,:], float32[:,:], float32[:,:]', 'float64[:,:],
                               '(m,n),(n,p)->(n,p)')(matmulcore)
```

The result is a gufunc, that can be used as any othe gufunc in NumPy. Broadcasting and type promotion rules are those on NumPy.

In [103…
```
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 [104…
```
%timeit gu_matmul(gu_test_A, gu_test_B)
```

```
654 µs ± 3.97 µs per loop (mean ± std. dev. of 7 runs, 1000 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 дложна соответствовать переданной сигнатуре типов.
4. Помните, что в guvectorize результат передается через последний параметр функции, тогда как в vectorize результат возвращается функцией ядра.

In [105…
```
@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 [106…

```
%%time

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

```
2885863
Wall time: 3.04 ms
```

- базовая реализаия - 21.1 s
- реализация с векторизацией - 6.61 s
- реализация с векторизацией на Numba - 105 ms
- реализация с guvectorize на Numba - 69.3 ms

In [107…

```
# Итоговый прирост производительности:
21.1/0.0693
```

Out[107…

```
304.4733044733045
```