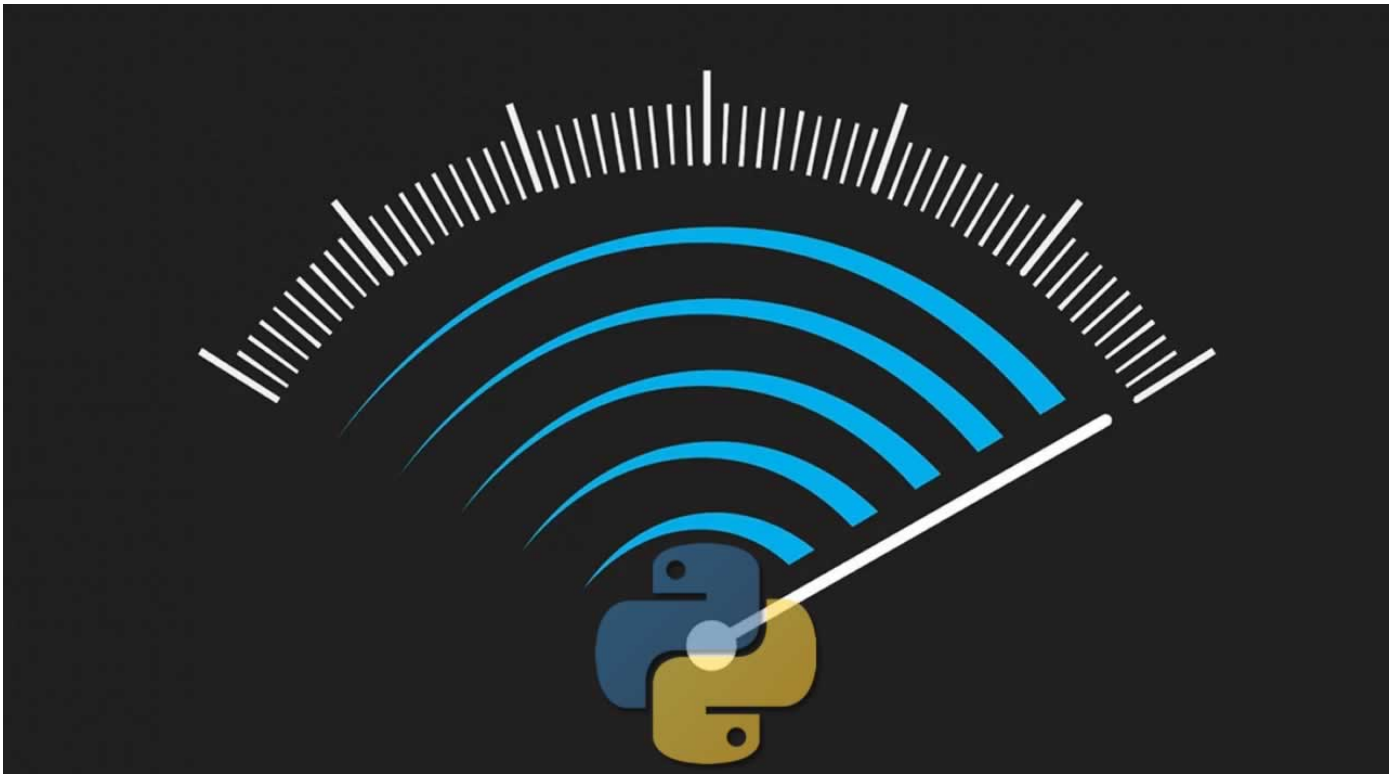


Содержание

Введение

В науке, аналитике, машинном обучении код на `Python` преимущественно используют как "клей", позволяющий собрать воедино библиотеки, написанные на других языках программирования (часто C/C++, Fortran). Это определяет специфику разработки: вся вычислительная сложность переносится на библиотеки, к которым нужно правильно обращаться. В данном руководстве описаны техники оптимизации такого рода программ: распараллеливание вызовов функций и `for`-циклов (зачаствую в парадигме `map-reduce`), перенос вычислений на распределённый кластер или GPU, мемоизация. Также на примере рассматриваются многопоточные альтернативы популярных библиотек в духе `pandas`.

В первой части руководства рассматривается `numba` — `jit`-компилятор, переводящий функции на `Python` и `numpy` в байт-код. `numba` позволяет использовать в коде `for`-циклы, т.к. после компиляции они не приводят к падению производительности. Кроме того, декораторы `numba` позволяют легко распараллеливать вызовы функций и `for`-циклы, автоматически генерировать универсальные (в смысле `numpy`, т.е. принимающие на вход произвольный тензор) функции из обычных, а также переносить код на `GPU` в одну строчку кода. Преимущество `numba` в том, что его использование почти

не требует усилий разработчика. Она идеально подходит для быстрого прототипирования, когда оптимизировать код вручную нерационально, а без этого он не отработает за разумное время. При этом `numba` нередко даёт результат сравнимый с оптимизацией того же кода вручную с переписыванием на чистом `numpy` векторизацией всех вычислений, распараллеливанием и т.д., что экономит время и силы программиста без потери качества.

Во второй части руководства идёт речь о `joblib` — библиотеке для максимально удобного написания `map-reduce` кода и мемоизации.

При написании руководства акцент был сделан на простоту в использовании и выразительность. Низкоуровневые детали происходящего, по возможности, опущены: заинтересованный читатель может ознакомиться с ними самостоятельно. Многие советы приобретают актуальность только при работе в многопроцессорной среде, но именно в таких условиях чаще всего происходит работа аналитика. Более продвинутые методы оптимизации — в духе `CPython`, — а также альтернативные интерпретаторы — в духе `PyPy`, `Jython` и `Iron Python` — не рассмотрены сознательно, т.к. чаще всего не поддерживают основные библиотеки для `data science`.

Numba: `jit`-компиляция

При написании этого раздела были использованы материалы `Stan Seibert`, директора по инновациям в `Anaconda`.
С низкоуровневыми деталями реализации библиотеки можно ознакомиться в его [презентации (https://indico.cern.ch/event/709711/contributions/2915722/attachments/1638199/2614603/2018_04_23_Numba_](https://indico.cern.ch/event/709711/contributions/2915722/attachments/1638199/2614603/2018_04_23_Numba_)

`numba` [(http://numba.pydata.org/)](http://numba.pydata.org/) это `jit` [-компилятор (https://en.wikipedia.org/wiki/Just-in-time_compilation)](https://en.wikipedia.org/wiki/Just-in-time_compilation) `Python`, т.е. динамический компилятор, который по запросу транслирует функции в байт-код, к которому потом обращаются последующие вызовы функции. Прирост производительности при этом обеспечивается за счёт того, что интерпретатору больше не нужно выполнять тело функции построчно, проверять, на что указывает каждое имя в коде и т.д. `Python` — динамически типизированный интерпретируемый язык, потому далеко не весь его функционал можно `jit`-компилировать, но то подмножество, которое используется для решения вычислительных задачи, чаще всего компилировать можно.

Особенности `numba` можно резюмировать следующим списком:

- Работает со стандартным интерпретатором `Python`.
- Несмотря на то, что это компилятор, явным образом указывать типы ненужно: они выводятся автоматически.
- Тесная интеграция с `numpy`: поддерживается [почти весь функционал (http://numba.pydata.org/numba-doc/latest/reference/numpysupported.html)](http://numba.pydata.org/numba-doc/latest/reference/numpysupported.html).
- Не поддерживается `scipy`. В частности, это ограничивает использование `numba` в статистике.
- Работает в многопоточной, многопроцессорной и распределённой среде, а также на `GPU`.

- Позволяет сочетать удобство программирования на `Python` с производительностью `C` и `Fortran`.

При этом `numba` не является:

- Заменой стандартного интерпретатора `Python` в отличие от `PyPy`, `Jython` и т.д.
- Транслятором кода на `Python` в `C/C++`

Функционал `numba` реализован в виде декораторов. Ниже рассмотрены основные из них

@numba.jit(nopython = True)

`jit`-компиляция работает эффективнее всего, если функция, кроме вызова библиотек, использует только базовые конструкции языка в духе циклов и простейших элементов стандартной библиотеки. В противном случае `numba` генерирует код, который обращается к `Python C API` (https://docs.python.org/2/c-api/index.html), чтобы обрабатывать все переменные как универсальные питоновские объекты — такой режим компиляции называется **объектным** (object mode).

Декоратор `@numba.jit` по умолчанию сам решает, переходить ли в объектный режим или нет. Директива `nopython=True` заставляет его выбрасывать исключение, если перехода в объектный режим нельзя избежать. **Если не указать `nopython=True`, то прирост производительности гарантировать нельзя.**

Работу `numba` наглядно иллюстрирует пример вычисления медианы средних Уолша:

1. Наивная реализация на чистом Python

In [1]:

```
import numba
import numpy as np

def walsh_median_pure_python(sample):
    walsh_averages = []
    n = len(sample)
    for i in range(n):
        for j in range(i + 1, n):
            walsh_averages.append((sample[i] + sample[j]) / 2)
    walsh_averages = sorted(walsh_averages)
    if n % 2 == 0:
        return 0.5 * (walsh_averages[(n - 1) // 2] + walsh_averages[n // 2])
    return walsh_averages[n // 2]
```

2. Тот же код, но с декоратором `@numba.jit`

In [2]:

```python
@numba.jit(nopython=True)
def walsh_median_jit(sample):
    walsh_averages = []
    n = len(sample)
    for i in range(n):
        for j in range(i + 1, n):
            walsh_averages.append((sample[i] + sample[j]) / 2)
    walsh_averages = sorted(walsh_averages)
    if n % 2 == 0:
        return 0.5 * (walsh_averages[(n - 1) // 2] + walsh_averages[n // 2])
    return walsh_averages[n // 2]
```

3. Полностью векторизованная реализация на чистом `numpy`

In [3]:

```python
def walsh_median_numpy(sample):
    sample_vec = sample.reshape(-1, 1)
    pairwise_sums = (sample_vec + sample_vec.T) / 2
    indices = np.triu_indices_from(pairwise_sums) # https://vk.cc/8D03ZI
    pairwise_sums = np.asarray( pairwise_sums[indices] )
    return np.median(pairwise_sums)
```

Сравнение времени работы в каждом из трёх подходов

In [11]:

```python
sample = np.random.rand(1000)оправдывает

print("Pure Python:")
%timeit walsh_median_pure_python(sample)
print("@numba.jit(nopython=True):")
%timeit walsh_median_jit(sample)
print("Pure numpy")
%timeit walsh_median_numpy(sample)
```

```
Pure Python:
1.46 s ± 85.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
@numba.jit(nopython=True):
141 ms ± 6.13 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
Pure numpy
58.8 ms ± 5.46 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Всего одна строчка кода даёт почти десятикратный прирост производительности по сравнению с чистым `Python`.
Полностью векторизованная реализация на `numpy`, использующая нетривиальные конструкции, всего в два раза быстрее.

Директива `parallel` и `numba.prange`

В силу [особенностей языка (https://en.wikipedia.org/wiki/Global_interpreter_lock)](), программы на `Python` могут работать только в однопоточном режиме. Тем не менее, скомпилированный код можно выполнять на нескольких потоках, как в `C++`. Директива `parallel=True` указывает `numba` использовать все

свободные ядра для вычислений.

Кроме того, генератор `numba.prange` позволяет распараллеливать циклы. В качестве примера ситуации, когда это полезно, можно привести множественную проверку гипотез, где гипотезы проверяются в цикле, одна за другой. Поскольку эти тесты независимы, их можно проводить параллельно, а МПГ-коррекцию полученных `p` -значений провести в самом конце. Это типичная `map-reduce` задача.

Nак можно организовать проверку большого числа нормальных выборок с известной дисперсией σ на несмещённость с помощью критерий Вальда уровня доверия α.

$$S_\alpha := \left| \frac{\sqrt{n}\overline{X}}{\sigma} \right| \geq z_{1-\alpha/2}$$

Для удобства, $\sigma = 1,\ \alpha = 0.05$, а МПГ-коррекция проводится по методу Бонферрони.

In [5]:

```
import math

@numba.jit(nopython=True)
def wald_test_p_value(sample):
    return 1 - math.erf(np.abs(np.mean(sample)))
```

Реализация на чистом `Python` . Гипотезы проверяются последовательно.

In [6]:

```
def multiple_testing_pure_python(samples):
    n_samples = samples.shape[0]
    p_values = np.zeros(n_samples)
    for i in range(n_samples):
        p_values[i] = wald_test_p_value(samples[i])
    corrected_p_values = p_values * n_samples
    overflow_mask = corrected_p_values > 1
    corrected_p_values[overflow_mask] = 1
    return corrected_p_values
```

Тот же код, но с `jit` -компиляцией и параллельным выполнением итераций цикла.

In [7]:

```
@numba.jit(nopython=True, parallel=True)
def multiple_testing_numba_jit(samples):
    n_samples = samples.shape[0]
    p_values = np.zeros(n_samples)
    for i in numba.prange(n_samples):
        p_values[i] = wald_test_p_value(samples[i])
    corrected_p_values = p_values * n_samples
    overflow_mask = corrected_p_values > 1
    corrected_p_values[overflow_mask] = 1
    return corrected_p_values
```

Сравнение: 10000 выборок по 100 элементов.

In [10]:

```
import multiprocessing as mp
print(f"Число ядер в системе: {mp.cpu_count()}")
```

```
Число ядер в системе: 64
```

In [8]:

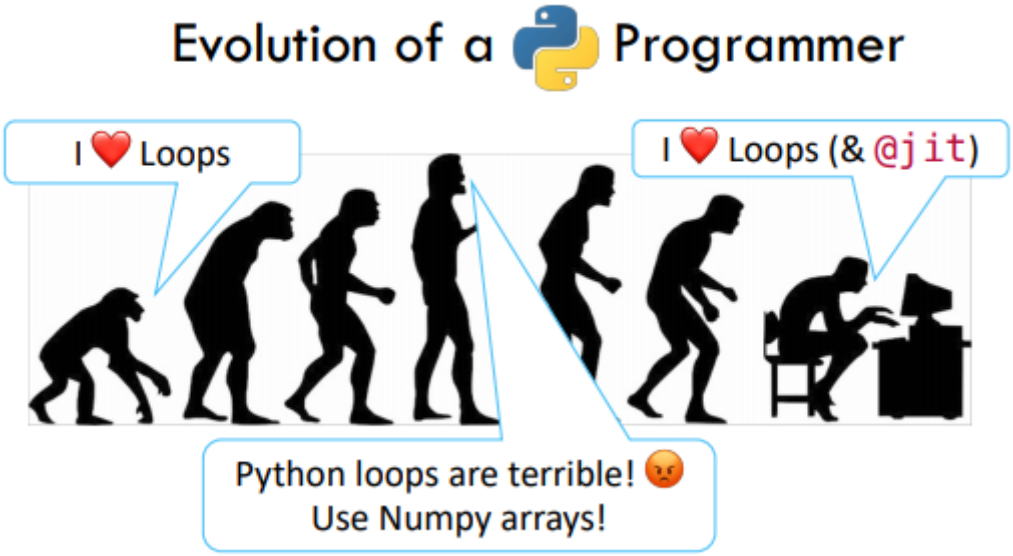
```
import scipy as sp
import scipy.stats as sps

n_samples = 10000
sample_size = 100
samples = sps.norm(loc=0.1, scale=1).rvs(size=(n_samples, sample_size))

%timeit multiple_testing_pure_python(samples)
%timeit multiple_testing_numba_jit(samples)
```

```
22.2 ms ± 347 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.15 ms ± 643 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Многопоточная версия более чем в **20 раз** быстрее. Для более сложных критериев разница была бы пропорциональна числу ядер в системе, т.е. было бы примерно в 64 раза быстрее.



Внимательный читатель заметит, что функцию `wald_test_p_value` пришлось компилировать отдельно. Это одна из неприятных особенностей `numba` : рекурсивная компиляция не поддерживается, нужно вручную прописывать декораторы каждый раз.

Компиляция функции может занимать продолжительное время. Чтобы это не происходило при каждом запуске программы, можно дополнительно указать директиву `cache=True` , которая сохранит закэширует код на диске. Это уместно, когда при запуске программы `jit` -компилируется не одна простая функция, а целая библиотека.

`@numba.vectorize`

По функционалу этот декоратор аналогичен функции `numpy.vectorize`, но результат оказывается лучше, т.к. он скомпилирован, а обработка входных данных автоматически распараллеливается. `@numba.vectorize` можно применять только к функциям, которые принимают и возвращают скаляры. Разве что сигнатуру функции при этом нужно прописывать явно (список поддерживаемых типов (https://numba.pydata.org/numba-doc/dev/reference/types.html)).

Его работу иллюстрирует параллельное вычисление плотности нормального распределения в `n` точках выборки.
Такая задача возникает при обучении метода опорных векторов (`SVM`) на больших выборках при использовании радиального ядра

$$K(x, x') := \exp\left(-\frac{\|x - x'\|}{2\sigma^2}\right)$$

In [83]:

```
@numba.vectorize("float64(float64, float64, float64)",
                 nopython=True, target="parallel")
def normal_pdf_numba_vectorize(x, mu, sigma):
    normalizing_constant = 1 / (sigma * math.sqrt(2 * math.pi))
    power = (x - mu) ** 2 / (2 * sigma**2)
    return normalizing_constant * math.exp(power)
```

Для наглядности, векторизованный метод сравнивается с функцией `scipy.norm.pdf`.

In [84]:

```
normal_rv = sps.norm()
sample_size = int(1e7)
sample = normal_rv.rvs(size=sample_size)

print("scipy.stats")
%timeit normal_rv.pdf(sample)
print("@numba.vectorize(nopython=True, target='parallel')")
%timeit normal_pdf_numba_vectorize(sample, 0, 1)
```

```
scipy.stats
1.93 s ± 22 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
@numba.vectorize(nopython=True, target='parallel')
41.6 ms ± 1.86 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Казалось бы, `scipy` — обкатанная в боях библиотека с множеством низкоуровневых оптимизаций. Тем не менее, `@numba.vectorize` работает примерно в 20 раз быстрее за счёт распараллеливания обработки точек.

Продвинутый материал: `@numba.guvectorize`

`@numba.guvectorize` — обобщение `@numba.vectorize`, которое поддерживает вектор-функции векторного аргумента. Его синтаксис несколько контринтуитивен и напоминает `C`: функция обязательно должна возвращать `void`, возвращаемое значение при этом нужно записывать в отдельную переменную, которая подаётся на вход "по ссылке" последней в спике аргументов.

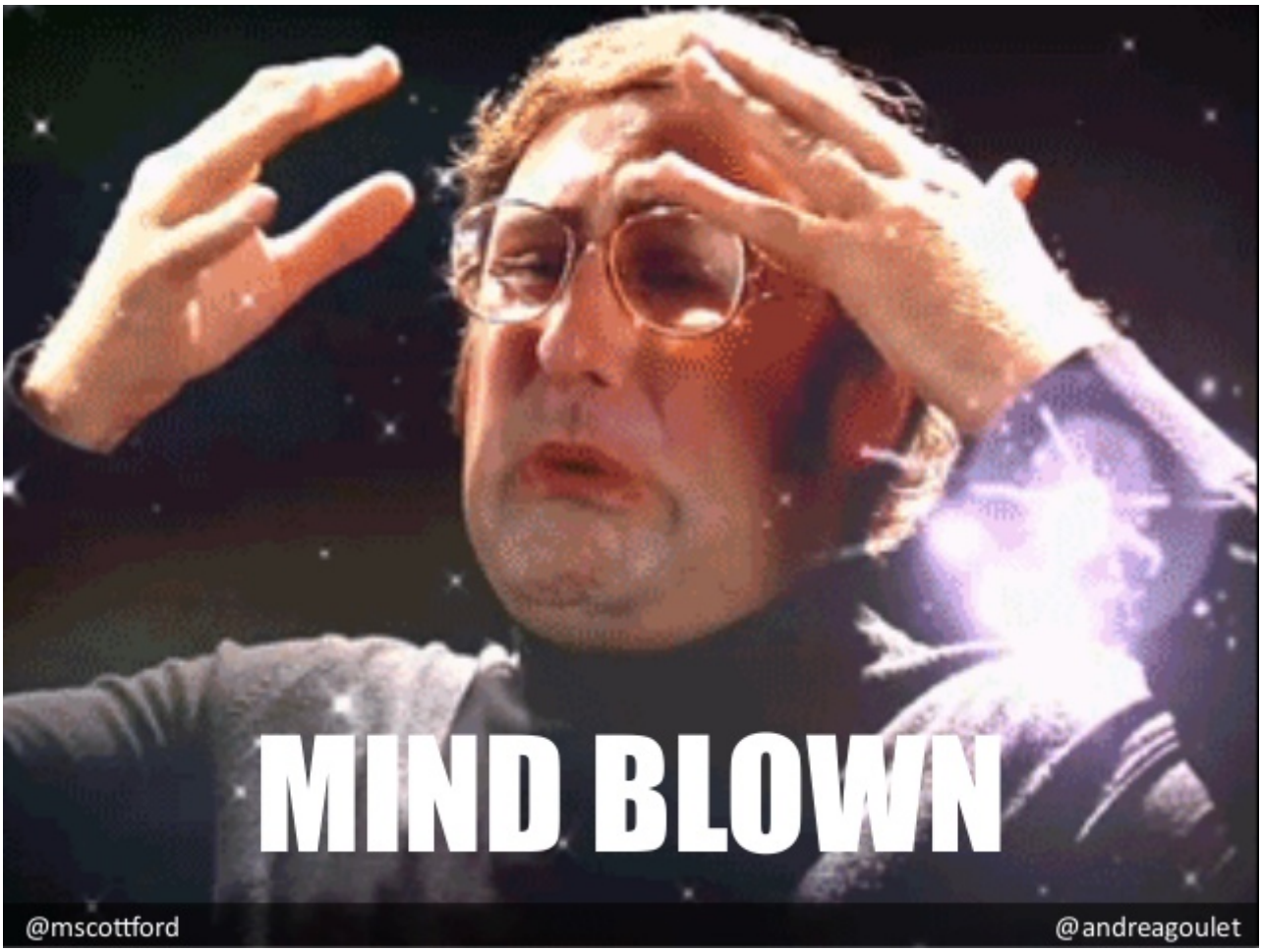
Такой дизайн продиктован поддержкой `CUDA`, т.е. необходимостью портирования кода на `GPU`. Написание нативного кода на `CUDA` — нетривиальная инженерная задача, потому проще смириться со странностями синтаксиса `numba`.

1. Векторизованная версия `numpy` -версии медианы средних Уолша

In [77]:

```
def walsh_median_np_vectorize(samples):
    return np.vectorize(walsh_median_numpy, signature="(n)->()")(samples)
```

2. Векторизация при помощи `@numba.guvectorize`



In [120]:

```python
@numba.guvectorize(
    ["void(float64[:], float64[:])"], # [:] указывает на массив
    "(n)->()", # сигнатура как в numpy.vectorize
    nopython=True, target="parallel")
def walsh_median_numba_vectorize(sample, result):
    walsh_averages = []
    n = len(sample)
    for i in range(n):
        for j in range(i + 1, n):
            walsh_averages.append((sample[i] + sample[j]) / 2)
    walsh_averages = sorted(walsh_averages)
    if n % 2 == 0:
        # [:] — особенность синтаксиса. Так нужно писать даже при том,
        # что в коде подразумевается, что result это число
        result[:] =  0.5 * (
            walsh_averages[(n - 1) // 2]
            + walsh_averages[n // 2]
        )
    else:
        result[:] = walsh_averages[n // 2]
    # функция не должна ничего возвращать, всё пишется в последний аргумент
```

Тело функции эквивалетно следующей программе на C++ :

```cpp
void walsh_median(const std::vector<double>& sample, double& result) {
    int n = static_cast<int>(sample.size());
    std::vector<double> walsh_averages ((n * (n - 1)) / 2);
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = i + 1; j < n; ++j) {
            walsh_averages[i * n + j] = (sample[i] + sample[j]) / 2.
        }
    }
    std::sort(walsh_averages.begin(), walsh_averages.end());
    if (n % 2) {
        result = walsh_averages[n / 2];
    } else {
        result = 0.5 * (walsh_averages[(n - 1) / 2] + walsh_averages[n / 2
]);
    }
    return;
}
```

Понадобилось именно такое сравнение, поскольку в Python нет чёткого эквивалента передачи объекта по ссылке. Но зачем передавать буферный массив для записи результата в качестве аргумента? Официальная документация (https://numba.pydata.org/numba-doc/latest/user/vectorize.html) не даёт внятного ответа на этот вопрос. Можно предположить, что так проще понять при компиляции, в какую переменную происходит запись ответа. Это упрощает проверку соответствия входных переменных сигнатуре функции.

Сравнение на 10000 выборок размера 100

In [79]:

```
n_samples = 10000
sample_size = 100
samples = np.random.rand(n_samples, sample_size).astype(np.float32)

print("numpy.vectorize")
%timeit walsh_median_np_vectorize(samples)
results_buffer = np.zeros(samples.shape[0], dtype=np.float32)
print("@numba.guvectorize(..., nopython=True, target='parallel')")
%timeit walsh_median_numba_vectorize(samples, results_buffer)
```

```
numpy.vectorize
4.62 s ± 156 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
@numba.guvectorize(..., nopython=True, parallel=True)
265 ms ± 5.99 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Это уже не так просто, как раньше, но почти двадцатикратный прирост в скорости оправдывает затраченные усилия. Даже при том, что наивная реализация подсчёта медианы средних работает в 2 раза медленнее, чем реализация на чистом `numpy`, выгода от распараллеливания вычислений перевешивает и итоговый результат получается лучше.

Joblib: распараллеливание и мемоизация

В тех случаях, когда использовать `numba` не получается, а распараллелить код нужно, можно воспользоваться пакетом `joblib`.
`joblib` также поддерживает эффективную мемоизацию (кэширование результатов вычисления функции).

`joblib.Parallel`, `joblib.delayed`

Распараллеливание кода с помощью `joblib` происходит за счёт вызова функций `Parallel` и `delayed`.
`Parallel` создаёт пул процессов, а `delayed` позволяет передать ему функцию. `joblib` накладывает ограничение, что и аргументы функции, и её возвращаемое значение должны быть сериализуемы. Напомним, что сериализация в случае `pickle` означает преобразование объекта в набор низкоуровневых инструкций для некоторой виртуальной машины (т.н. байт-код), которая потом по ним может восстановить объект. Подробнее о том, что можно сериализовать, а что нет, можно прочитать в официальной документации (https://docs.python.org/3.4/library/pickle.html#what-can-be-pickled-and-

unpickled). Это необходимо для того, чтобы преобразовать функцию вместе с содержимым в меньший по размерам поток байтов, который потом передаётся процессам-исполнителям. Чаще всего оно выполнено. Т.е. `joblib` может распараллелить не только всё то же, что и `numba`, но и многое другое.

Для иллюстрации использован пример векторизованной медианы средних Уолша, но в этот раз выборок мало, зато все они большого размера. Это принципиальный момент: создание процесса это тяжёлая операция, которая, по сути, создаёт копию текущего окружения. Если функция на каждом конкретном аргументе вычисляется быстро, то затраты на создание процессов и пересылку аргументов и возвращаемых значений между процессами перевесят выгоду от распараллеливания кода.

In [91]:

```
import joblib

n_samples = 100
sample_size = 10000
samples = np.random.rand(n_samples, sample_size).astype(np.float32)
```

In [92]:

```
%%time
print("joblib")
_ = joblib.Parallel(n_jobs=-1)( # использовать все доступные процессоры
        joblib.delayed(walsh_median_numpy)(sample)
        for sample in samples
    );
```

```
CPU times: user 1 s, sys: 4.12 s, total: 5.12 s
Wall time: 26.7 s
```

In [94]:

```
%%time
print("numpy.vectorize")
walsh_median_np_vectorize(samples)
```

```
numpy.vectorize
CPU times: user 4min 36s, sys: 2min 33s, total: 7min 9s
Wall time: 7min 11s
```

Out[94]:

```
array([0.500389  , 0.49510273, 0.5000276 , 0.5014577 , 0.500961  ,
       0.4937545 , 0.49917376, 0.497379  , 0.50479156, 0.49686682,
       0.49974638, 0.49999797, 0.49995032, 0.50280845, 0.5013958 ,
       0.50258434, 0.4987329 , 0.49770072, 0.50275433, 0.50121444,
       0.49644727, 0.49725574, 0.50341666, 0.49929664, 0.5045712 ,
       0.49932277, 0.50186   , 0.49669537, 0.49493605, 0.49783957,
       0.50271595, 0.49956867, 0.49926603, 0.5007514 , 0.5018302 ,
       0.49823463, 0.50099814, 0.49869508, 0.4990778 , 0.4991232 ,
       0.5041181 , 0.4969048 , 0.5013907 , 0.50074565, 0.497693  ,
       0.49503464, 0.50161374, 0.5044385 , 0.50296587, 0.49979556,
       0.50377554, 0.49805725, 0.49820244, 0.4971007 , 0.49917197,
       0.49612033, 0.50438094, 0.50135887, 0.5011893 , 0.50158656,
       0.49671024, 0.50478244, 0.5057939 , 0.50142896, 0.49185765,
       0.50307477, 0.5020547 , 0.50439763, 0.5003091 , 0.505753  ,
       0.49625278, 0.4991153 , 0.50099736, 0.5068797 , 0.50158536,
       0.5025751 , 0.500568  , 0.50391823, 0.4971952 , 0.4985718 ,
       0.50197554, 0.5005748 , 0.50316674, 0.49792293, 0.4990613 ,
       0.5019417 , 0.5015318 , 0.5000223 , 0.5014322 , 0.5021168 ,
       0.50173795, 0.5042229 , 0.49757996, 0.49556422, 0.49804616,
       0.50178355, 0.5009304 , 0.5009364 , 0.50307655, 0.50101185],
      dtype=float32)
```

`joblib` отработал примерно в 14 раз быстрее, чем последовательное исполнение.

joblib.Memory

`joblib` предоставляет механизм для кэширования результатов вычисления функций. Это очень удобно, когда запусков планируется мало, но каждый запуск занимает значительное время (скажем, несколько часов) и нет уверенности, что программа не упадёт с ошибкой, так и не досчитав.

В такой ситуации было бы очень уместно сохранить результаты промежуточных вычислений, чтобы потом просто загрузить их из памяти и начать с того же места, на котором исполнение прервалось.

Эту логику реализует класс `joblib.Memory` . Достаточно только указать папку, в которой будут хранится результаты, а остальное — проверку аргументов на соответствие, сохранение и подгрузку результатов из кэша — он сделает сам.

In [95]:

```
memory = joblib.Memory("~/.cache_dir", verbose=1)
```

Разницу можно проиллюстрировать на примере какой-нибудь медленной функции

In [96]:

```
@memory.cache # достаточно просто написать декоратор
def walsh_median_numpy_cached(sample):
    return walsh_median_numpy(sample)
```

Первый запуск

In [97]:

```
%%time
walsh_median_numpy_cached(samples[0])
```

```
_____________________________________________________________________________
__________
[Memory] Calling __main__--icgc-dkfzlsdf-analysis-B260-users-v390v-sci
entific-computing-101-03-make-python-fast-again-__ipython-input__.wals
h_median_numpy_cached...
walsh_median_numpy_cached(array([0.271054, ..., 0.217159], dtype=float
32))
_____________________________________walsh_median_numpy_cached - 4.
3s, 0.1min
CPU times: user 2.66 s, sys: 1.58 s, total: 4.24 s
Wall time: 4.3 s
```

Out[97]:

```
0.500389
```

Второй запуск с подгрузкой значения из кэша

In [98]:

```
%%time
walsh_median_numpy_cached(samples[0])
```

```
CPU times: user 4.35 ms, sys: 1.32 ms, total: 5.67 ms
Wall time: 5.33 ms
```

Out[98]:

```
0.500389
```