83.87
Рейтинг

Wunder Fund

Мы занимаемся высокочастотной торговлей на бирже

mr-pickles вчера в 14:15

Taichi и 100-кратное ускорение Python-кода

Блог компании Wunder Fund, Python*, Программирование*, Клиентская оптимизация*, Серверная оптимизация*

Перевод

Автор оригинала: Yuanming Hu

Python стал самым популярным языком во многих быстроразвивающихся областях, таких, как

производительностью. Конечно, все мы время от времени жалуемся на скорость работы программ, и Python, безусловно, не стоит винить во всех грехах. Несмотря на это, справедливым будет заявление о том, что природа Python, интерпретируемого языка, не способствует высокой производительности кода, особенно когда речь идёт о «тяжёлых» вычислениях (один из признаков таких вычислений — наличие в программе нескольких вложенных циклов).

- Выполнение внушительного цикла `for` в моём коде занимает целую вечность…
- В моей программе, в некоторых вычислительных задачах, имеются узкие места. Переписывание кода на C++ и вызов этого кода с помощью модуля ctypes может помочь решить проблему, но это не так просто и понятно, как хотелось бы. Такой подход сопряжён с риском ошибок компиляции при переносе кода на другие платформы. Гораздо лучше было бы, если бы у меня была возможность решить все свои задачи на Python.

- Я — лояльный пользователь C++/Fortran, но мне хотелось бы попробовать Python, так как этот язык становится всё популярнее и популярнее. Но после переписывания кода на Python начинается настоящий кошмар. У меня возникает такое ощущение, что производительность такого кода раз в 100 меньше, чем раньше.
- Мне надо обрабатывать много изображений, а OpenCV меня не устраивает. Поэтому мне приходится вручную писать вложенные циклы. И я от такой работы совсем не в восторге.

Если с вами случалось то, о чём говорилось выше, это значит, что вам, возможно, захочется познакомиться с Taichi. Это — предметно-ориентированный язык, встроенный в Python. У Taichi, однако, имеется собственный компилятор, ответственный за обработку кода, декорированного с помощью `@ti.kernel`. Благодаря этому достигается высокая производительность выполнения кода на самом разном аппаратном обеспечении, в том числе — на CPU и на GPU. Одна из главных сильных сторон Taichi — ускорение Python-кода. В результате Python-программистам, применяющим Taichi, больше не придётся завидовать тем, кто пользуется C++/CUDA (вот — отчёт о сравнении производительности Taichi и CUDA).

Разработчики из сообщества, сложившегося вокруг Taichi, прилагают большие усилия к улучшению совместимости Taichi и Python. На сегодняшний день все возможности Taichi способны отлично работать после импорта соответствующей библиотеки командой `import taichi as ti`. Установить Taichi можно с помощью команды `pip install`, эта библиотека способна взаимодействовать с другими Python-библиотеками, в том числе — с NumPy, Matplotlib и PyTorch.

Я ничуть не преувеличиваю, когда говорю, что Taichi может быть именно тем инструментом, который вам нужен. В этом материале я собираюсь рассказать о том, как библиотека Taichi способна ускорять Python-код как минимум в 50 раз. Свой рассказ я проиллюстрирую тремя примерами:

- Подсчёт количества простых чисел, меньших, чем N.

- Нахождение наибольшей общей подпоследовательности путём применения динамического программирования.
- Решение реакционно-диффузионных уравнений.

Код этих трёх примеров можно найти здесь.

Подсчёт количества простых чисел

Не хочу напугать читателей первым же примером, поэтому предлагаю написать простую программу, которая подсчитывает количество простых чисел, меньших, чем целевое положительное целое число `N`. Если у вас имеется некоторый опыт Python-программирования, вы можете попытаться написать собственную версию этой программы и с ней поэкспериментировать.

Вот стандартное решение нашей задачи:

```
"""Count the number of primes in range [1, n].
"""

def is_prime(n: int):
    result = True
    for k in range(2, int(n ** 0.5) + 1):
        if n % k == 0:
            result = False
            break
    return result

def count_primes(n: int) -> int:
    count = 0
```

```
    for k in range(2, n):
        if is_prime(k):
            count += 1

    return count

print(count_primes(1000000))
```

В основе этого подхода лежит логика, понятная на интуитивном уровне. Сначала мы определяем функцию `is_prime`, которая сообщает нам о том, является ли положительное целое число `N` простым. Если это так — она возвращает `1`, в противном случае она возвращает `0`. Затем мы перебираем целые числа от `2` до `sqrt(n)` и проверяем, можно ли поделить на них `N` без остатка. И наконец — мы собираем результаты. Задача решена! Сохраните этот код в файле с расширением `.py` (например — `count_primes.py`) и запустите следующей командой:

```
time python count_primes.py
```

Вот что получилось на моём компьютере:

```
78498

real        0m2.235s
user        0m2.235s
sys        0m0.000s
```

Когда `N` равно одному миллиону, для нахождения итогового результата нужно 2,235 секунды. Возможно, и для вашего компьютера это — не такая уж и сложная задача. А что если установить `N` в 10 миллионов? Готов поспорить — вам придётся ждать получения результатов как минимум секунд тридцать.

А теперь позвольте мне показать вам настоящую магию. Для того чтобы это увидеть, тело функции модифицировать не нужно. Всё в нём останется как прежде. В файл надо лишь импортировать Python-библиотеку и добавить к функциям пару декораторов.

```
"""Count the number of primes below a given bound.
"""
import taichi as ti
ti.init()

@ti.func
def is_prime(n: int):
    result = True
    for k in range(2, int(n ** 0.5) + 1):
        if n % k == 0:
            result = False
            break
    return result

@ti.kernel
def count_primes(n: int) -> int:
    count = 0
    for k in range(2, n):
        if is_prime(k):
            count += 1
```

```
    return count

print(count_primes(1000000))
```

Снова запустим код командой `time python count_primes.py`. В этот раз решение я увидел гораздо быстрее:

```
78498

real        0m0.363s
user        0m0.546s
sys         0m0.179s
```

А именно, теперь код работает примерно в шесть раз быстрее! А когда я установил `N` в 10 миллионов, код, где применяется Taichi, нашёл решение за 0,8 секунды. У обычного Python-кода ушло на это 55 секунд. Налицо 70-кратное ускорение. Но Taichi идёт и ещё дальше. Эта библиотека позволяет задавать устройство, на котором нужно производить вычисления. Это даёт больше возможностей для оптимизации производительности программ. Например, при инициализации Taichi можно воспользоваться такой конструкцией: `ti.init(arch=ti.gpu)`. Я сам провёл соответствующее испытание и хочу поделиться с вами данными из первых рук. При выполнении кода на GPU у Taichi ушло 0,45 секунд на подсчёт простых чисел, меньших, чем 10 миллионов. А это, в сравнении с обычным Python-кодом, означает 120-кратное ускорение.

Кто-то может заявить, что этот пример рассчитан на студентов или на начинающих питонистов, что он выглядит не особенно завлекательным для опытных программистов, которые сталкиваются со сложными реальными задачами. Я с этим согласен. В следующих двух примерах вам может попасться кое-что похитрее подсчёта чисел.

Динамическое программирование

Динамическое программирование — это полезная, с точки зрения оптимизации кода, алгоритмическая техника. Главная идея этого подхода заключается в том, чтобы пожертвовать некоторым объёмом памяти ради ускорения выполнения программы. То есть — чтобы хранить промежуточные результаты вычисления для сокращения времени выполнения кода. Тут я, в качестве примера, рассмотрю задачу о наибольшей общей подпоследовательности (НОП, longest common subsequence, LCS). Это — типичный пример, в котором используется динамическое программирование, взятый из книги «Алгоритмы: построение и анализ» («Introduction to Algorithms»).

Отвлечёмся ненадолго. Мне, как и многим другим, просто очень нравится эта книга. Я купил «Introduction to Algorithms» в китайском переводе много лет назад (когда мне далеко ещё было до нормального программиста). В книге было сказано, что её четыре автора были из MET. Тогда это меня озадачило, мне интересно было — как четыре художника могут так хорошо разбираться в программировании. Позже я понял, что я, должно быть, купил книгу в пиратском переводе. Через десять лет я стал аспирантом в MIT, и одним из членов комиссии на экзамене (RQE, Research Qualifying Exam) оказался профессор Лейзерсон, известный учёный из «MET», один из соавторов «Introduction to Algorithms».

Вернёмся к нашим делам.

Подпоследовательность — это часть последовательности, относительный порядок элементов в которой не меняется. Например, `[1, 2, 1]` — это подпоследовательность `[1, 2, 3, 1]`, но `[3, 2]` — нет. Основываясь на этой идее, НОП определяется как самая длинная подпоследовательность, общая для всех заданных последовательностей. Стоит отметить, что может быть несколько НОП, общих для заданных последовательностей, но длина этих НОП должна быть одной и той же.

Для последовательности `a`:

```
a = [0, 1, 0, 2, 4, 3, 1, 2, 1]
```

и последовательности `b`:

```
b = [4, 0, 1, 4, 5, 3, 1, 2]
```

НОП будет выглядеть так:

```
LCS(a, b) = [0, 1, 4, 3, 1, 2]
```

НОП широко применяется в различных задачах, где используется сравнение образцов. Например, НОП используется в Linux-команде `diff` и в аналогичной git-команде при сравнении двух текстов. Биологи, заменяя числа на буквы ACGT, могут использовать НОП для сравнения фрагментов геномов.

Для нахождения НОП двух заданных последовательностей с применением динамического программирования можно постепенно выяснять длину НОП первых `i` элементов последовательности `a` и первых `j` элементов последовательности `b`. Затем, увеличивая по очереди `i` или `j`, надо повторять этот шаг до тех пор, пока не будет получен итоговый результат.

Мы используем `f[i, j]` для представления длины подпоследовательности `LCS((prefix(a, i), prefix(b, j))`. Конструкция `prefix(a, i)` обозначает первые `i` элементов

последовательности `a`, то есть — `a[0], a[1], ..., a[i - 1]`. Это позволяет нам получить следующую рекуррентную формулу:

```
f[i, j] = max(f[i - 1, j - 1] + (a[i - 1] == b[j - 1]),
              max(f[i - 1, j], f[i, j - 1]))
```

Полное решение может быть выражено в таком коде:

```
for i in range(1, len_a + 1):
    for j in range(1, len_b + 1):
        f[i, j] = max(f[i - 1, j - 1] + (a[i - 1] == b[j - 1]),
                      max(f[i - 1, j], f[i, j - 1]))
```

Теперь ускорим вычисления с помощью Taichi:

```
import taichi as ti
import numpy as np

ti.init(arch=ti.cpu)

benchmark = True

N = 15000

f = ti.field(dtype=ti.i32, shape=(N + 1, N + 1))
```

```
if benchmark:
    a_numpy = np.random.randint(0, 100, N, dtype=np.int32)
    b_numpy = np.random.randint(0, 100, N, dtype=np.int32)
else:
    a_numpy = np.array([0, 1, 0, 2, 4, 3, 1, 2, 1], dtype=np.int32)
    b_numpy = np.array([4, 0, 1, 4, 5, 3, 1, 2], dtype=np.int32)


@ti.kernel
def compute_lcs(a: ti.types.ndarray(), b: ti.types.ndarray()) -> ti.i32:
    len_a, len_b = a.shape[0], b.shape[0]

    ti.loop_config(serialize=True) # To forbid automatic parallelism
    for i in range(1, len_a + 1):
        for j in range(1, len_b + 1):
            f[i, j] = max(f[i - 1, j - 1] + (a[i - 1] == b[j - 1]),
                          max(f[i - 1, j], f[i, j - 1]))

    return f[len_a, len_b]


print(compute_lcs(a_numpy, b_numpy))
```

Сохраните этот код в .py-файле `lcs.py` и запустите в терминале:

```
time python lcs.py
```

На моём компьютере получилось следующее:

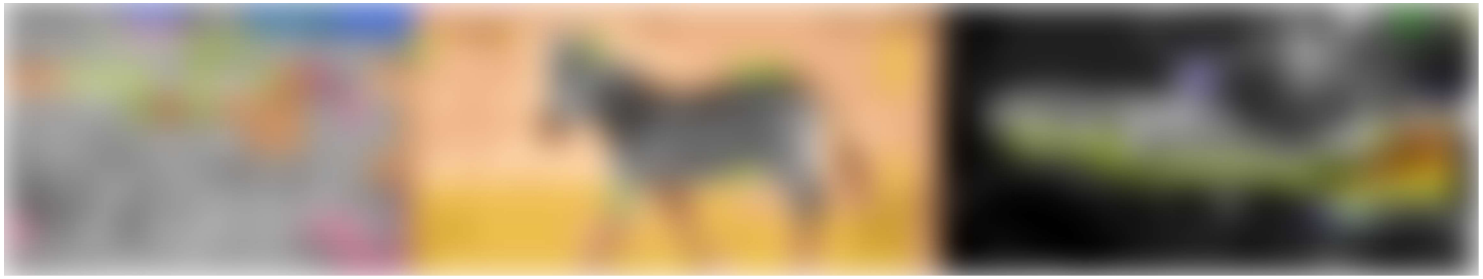
```
2721


real        0m1.409s
user        0m1.112s
sys         0m0.549s
```

Вы можете попробовать реализовать задачу поиска НОП, воспользовавшись, соответственно, Taichi и NumPy, чтобы сравнить производительность разных вариантов. При `N=15000` моя Taichi-программа нашла решение за 0,9 с, а у NumPy-версии программы это заняло 476 секунд. Как видите, тут наблюдается резкое, более чем 500-кратное отставание NumPy от Taichi. Дело в том, что библиотека NumPy специализируется на операциях, направленных на обработку массивов, а Taichi поддерживает более высокий уровень «детализации», напрямую работая с элементами контейнеров данных.

Решение реакционно-диффузионных уравнений

Пятна на шкуре гепарда, полоски зебры, точки и линии рыбы-собаки… Интересные узоры, которые делают живых существ такими заметными (или незаметными), заставляют нас задумываться над секретами матери-природы.



Элементы таких узоров распределены неравномерно, но их нельзя назвать расположенными беспорядочно. Они, с точки зрения эволюции, являются результатом естественного отбора, длившегося многие века. Но есть ли некое глубинное правило, по которому «нарисованы» эти узоры? Алан Тьюринг (изобретатель машины Тьюринга) первым разработал модель, описывающую это явление. В своей работе «Химическая основа морфогенеза» он применяет две химические субстанции (U и V) для имитации процесса создания узоров. Взаимоотношения между U и V напоминают взаимоотношения между жертвой и хищником. Они самостоятельно перемещаются и взаимодействуют друг с другом:

1. Изначально U и V случайным образом распределены по некоей области.
2. На каждом временном шаге они распространяются путём диффузии на соседние с ними участки.
3. Когда U и V встречаются, некоторая часть U поглощается V. Как следствие, концентрация V возрастает.
4. Для того чтобы вещество V не уничтожило бы U, мы, на каждом временном шаге, добавляем в систему некоторый процент (f) U и убираем из неё некоторый процент (k) V.

Вышеописанный процесс можно свести к следующему реакционно-диффузионному уравнению:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = D_u \Delta u - uv^2 + f(1-u)$$
$$\frac{\mathrm{d}v}{\mathrm{d}t} = D_v \Delta v + uv^2 - (f+k)v.$$

Тут имеется четыре ключевых параметра: `Du` (скорость диффузии U), `Dv` (скорость диффузии V), `f` (сокращение от «feed», контролирует добавление в систему U) и `k` (сокращение от «kill», контролирует удаление из системы V).

Для запуска симуляции этого процесса с помощью Taichi сначала надо создать сетку, покрывающую область, и использовать `vec2` для представления концентрации U и V в сетке. Числовые вычисления оператора Лапласа требуют обращений к соседним ячейкам сетки. Для того чтобы избежать обновления и чтения данных в одном и том же цикле, мы должны создать две сетки идентичной формы — `W × H × 2`. Каждый раз, когда мы обращаемся к данным из одной сетки, мы записываем обновлённые данные в другую сетку, после чего меняем их местами. Я разработал для этой задачи такую структуру данных:

```
W, H = 800, 600
uv = ti.Vector.field(2, float, shape=(2, W, H))
```

Сначала мы задаём концентрацию U во всех местах сетки, равную 1, а V помещаем в 50 мест, выбранных случайным образом:

```
import numpy as np

uv_grid = np.zeros((2, W, H, 2), dtype=np.float32)
uv_grid[0, :, :, 0] = 1.0
rand_rows = np.random.choice(range(W), 50)
rand_cols = np.random.choice(range(H), 50)
uv_grid[0, rand_rows, rand_cols, 1] = 1.0
uv.from_numpy(uv_grid)
```

Вычисления выполняются в следующих десяти строках кода:

```
@ti.kernel
def compute(phase: int):
    for i, j in ti.ndrange(W, H):
        cen = uv[phase, i, j]
        lapl = uv[phase, i + 1, j] + uv[phase, i, j + 1] + uv[phase, i - 1, j] + uv[phase,
        du = Du * lapl[0] - cen[0] * cen[1] * cen[1] + feed * (1 - cen[0])
        dv = Dv * lapl[1] + cen[0] * cen[1] * cen[1] - (feed + kill) * cen[1]
        val = cen + 0.5 * tm.vec2(du, dv)
        uv[1 - phase, i, j] = val
```

Мы используем целочисленную переменную `phase` (она может равняться 0 или 1) для управления тем, из какой сетки мы читаем данные. Соответственно, `1-phase` указывает на другую сетку, которая принимает обновлённые данные.

Последний шаг — окрасить субстанции в соответствии с концентрацией V. В итоге получился впечатляющий графический эффект.

> → GIF на 26 мб!

Интересно, что всегда можно прийти к похожим узорам, несмотря на то, что первоначальная концентрация V задаётся случайным образом.

У нас, кроме того, имеются две реализации этой задачи, основанные, соответственно, на Taichi и на Numba. Оба пакета используют собственные компиляторы, но Taichi-версия выполняет

вычисления на GPU, в результате скорость вывода графики с лёгкостью превышает 300 FPS. А вот Numba, где вычисления идут на CPU, достигает лишь примерно 30 FPS.

Taichi и другие Python-пакеты

До сих пор я полагался на подробный разбор трёх примеров, доказывая, что библиотека Taichi способна существенно, если не сказать больше, ускорять Python-программы. Если в двух словах, возможности Taichi, касающиеся высокопроизводительных вычислений, объясняются принципами проектирования этой системы:

1. Taichi — это компилируемый язык.
2. Taichi может автоматически выполнять код в параллельном режиме, в то время как обычный Python-код обычно выполняется в однопоточном режиме.
3. Taichi может работать и на CPU, и на GPU, а обычный Python код выполняется только на CPU.

Конечно, Python — это целая вселенная, в которой имеется множество инструментов, способных ускорить стандартный Python-код. Нет смысла нахваливать Taichi и при этом по-тихому обходить вниманием другие Python-библиотеки. Поэтому ниже я сделал сравнение Taichi с другими похожими пакетами, привёл их плюсы и минусы.

Taichi и NumPy/JAX/PyTorch/TensorFlow

Последняя группа инструментов сильно зависима от операций, основанных на массивах. Поэтому эти инструменты хорошо подходят пользователям, которых интересует анализ и обработка данных и глубокое обучение. Но эти инструменты, в сравнении с Taichi, отличаются гораздо меньшей гибкостью в случаях, когда речь идёт о научных вычислениях. В вышеприведённом примере с простыми числами выполнение вычислений с помощью массивов будет уже не таким простым, как в случае с Taichi. Если говорить о «детализации»

математических операций, то Taichi лучше сравнивать с C++ или CUDA, так как Taichi может напрямую управлять каждой итерацией циклов.

Taichi и Cython

Cython тоже часто используется для улучшения производительности Python-кода. На самом деле, многие модули в официальном коде NumPy и SciPy написаны и скомпилированы с помощью Cython. Но при таком подходе страдает читабельность кода. Cython, кроме того, не поддерживает вычисления на GPU (хотя, в определённой степени, позволяет организовывать параллельные вычисления).

Taichi и Numba

Само название компилятора, Numba, указывает на то, что он создан в расчёте на NumPy. Numba рекомендуется использовать в том случае, если в функциях применяется векторизация массивов NumPy. Taichi, в сравнении с Numba, отличается следующими преимуществами:

- Taichi поддерживает множество типов данных, в том числе — `struct`, `dataclass`, `quant` и `sparse`. Библиотека позволяет гибко управлять размещением данных в памяти. Эта возможность весьма желательна в том случае, когда программа обрабатывает огромные объёмы данных. А Numba показывает наилучшие результаты только работая с плотными массивами NumPy.
- Taichi может выполнять вычисления на различных GPU, до крайности упрощая решение крупномасштабных параллельных задач (таких, как симуляция частиц или рендеринг). Но написание подсистемы рендеринга с помощью Numba — это задача, которую даже сложно себе представить.

Taichi и PyPy

PyPy — это JIT-компилятор, который появился ещё в 2007 году. PyPy, что делает его похожим на Taichi, ускоряет Python-код посредством компиляции. PyPy — привлекательный инструмент, так как он позволяет пользователю работать с неизменным Python-кодом, не нуждающемся даже в небольших модификациях. Это, правда, ограничивает возможности по оптимизации кода. В этом смысле PyPy — тот ещё подарочек. Если вы ожидаете более серьёзного роста производительности, Taichi способен вам его дать. Но вам придётся ознакомиться с синтаксисом Taichi и с соглашениями, лежащими в его основе, которые немного отличаются от Python.

Taichi и ctypes

Библиотека ctypes позволяет программистам вызывать из Python C-функции и выполнять в Python программы, написанные на C++/CUDA. Всё это делается посредством C-совместимого API. Но ctypes отличается довольно высоким входным барьером. Для того чтобы написать нормальную программу, программисту надо воспользоваться C, Python, CMake, CUDA и даже большим количеством инструментов. Жизнь программиста становится куда спокойнее, когда всё, что ему нужно, можно написать на Python.

Итоги

В итоге можно сказать, что не существует универсального решения всех задач, требующих оптимизации производительности кода. Отчасти именно поэтому Python-разработка — это весьма увлекательное занятие. Всегда можно найти или создать простой инструмент, способный хорошо решить конкретную задачу. Если говорить о научных вычислениях, то Taichi — это идеальный компаньон Python, который способен помочь всем желающим достичь уровня производительности, сравнимого с C/C++.