几个向量化计算的小技巧

2021 年 1 月 27 日

1 求多个向量的线性变换

设 $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$ 是向量，设 A 是矩阵，我们希望找到 $\boldsymbol{y}_1, \boldsymbol{y}_2, \cdots, \boldsymbol{y}_n$ 使得满足 $A\boldsymbol{x}_1 = \boldsymbol{y}_1, A\boldsymbol{x}_2 = \boldsymbol{y}_2, \cdots, A\boldsymbol{x}_n = \boldsymbol{y}_n$，在 NumPy 中，可以有多种实现:

```
import numpy as np

mat_A = np.random.rand(3, 3)
x1 = np.random.rand(3, 1)
x2 = np.random.rand(3, 1)
x3 = np.random.rand(3, 1)
x4 = np.random.rand(3, 1)

y1 = mat_A @ x1
y2 = mat_A @ x2
y3 = mat_A @ x3
y4 = mat_A @ x4
```

还可以是:

```
x = np.concatenate((x1,x2,x3,x4,), axis=1)
y = mat_A @ x

print(np.max(np.abs(y - np.concatenate((y1,y2,y3,y4,), axis=1))))
```

依据:

$$A\boldsymbol{x}_1 = \boldsymbol{y}_1, \cdots, A\boldsymbol{x}_n = \boldsymbol{y}_n \iff A\begin{bmatrix}\boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n\end{bmatrix} = \begin{bmatrix}\boldsymbol{y}_1 & \cdots & \boldsymbol{y}_n.\end{bmatrix} \tag{1.1}$$

进一步地，假设我们知道 $(\boldsymbol{y}_1, \cdots, \boldsymbol{y}_n)^\top$，也可以批量计算出 $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n)^\top$:

```
x = np.linalg.solve(mat_A, y)
print(np.max(np.abs(x - np.concatenate((x1,x2,x3,x4,), axis=1))))
```

建议熟悉这类技巧.

2 求两组向量两两之间的余弦相似度

设 $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$ 是一组向量，设 $\boldsymbol{y}_1, \cdots, \boldsymbol{y}_m$ 也是一组向量，试计算矩阵 M 满足对任意 $1 \le i \le n$, 对任意 $1 \le j \le m$，都有 $M[i, j] = \cos\left\langle \boldsymbol{x}_i, \boldsymbol{y}_j \right\rangle$:

```
n_samples_x = 100
n_samples_y = 120
```

```
xs = np.random.rand(n_samples_x, 3)
ys = np.random.rand(n_samples_y, 3)

xs = xs / (np.linalg.norm(xs, axis=1).reshape(n_samples_x, 1))
ys = ys / (np.linalg.norm(ys, axis=1).reshape(n_samples_y, 1))
cosines1 = xs @ (ys.T)
```

以上是向量式的计算方法，下面我们来验证它的正确性:

```
from scipy.spatial.distance import cosine

cosines2 = np.zeros_like(cosines1)
for i in range(cosines2.shape[0]):
    for j in range(cosines2.shape[1]):
        cosines2[i, j] = 1 - cosine(xs[i, :], ys[j, :])

print(np.max(np.abs(cosines1 - cosines2)))
```

误差为:

```
4.440892098500626e-16
```

说明两种计算方法都正确．但是第一种由于是矩阵计算，所以容易并行化．

3 同时求多个向量对一个向量的叉乘

设 $\boldsymbol{v}, \boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$ 是向量，求叉乘：$\boldsymbol{v} \times \boldsymbol{x}_1, \cdots, \boldsymbol{v} \times \boldsymbol{x}_n$．叉乘的定义是，对于两个 3 维向量

$$\boldsymbol{a} = (a_1, a_2, a_3), \quad \boldsymbol{b} = (b_1, b_2, b_3) \tag{3.1}$$

有

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} \tag{3.2}$$

式中：$\boldsymbol{i} = (1,0,0), \boldsymbol{j} = (0,1,0), \boldsymbol{k} = (0,0,1)$．于是我们将式 3.2展开得

$$\boldsymbol{a} \times \boldsymbol{b} = (a_2 b_3 - a_3 b_2)\boldsymbol{i} - (a_1 b_3 - a_3 b_1)\boldsymbol{j} + (a_1 b_2 - a_2 b_1)\boldsymbol{k} \tag{3.3}$$

我们发现，如果要同时算 $\boldsymbol{b}$ 与多个不同的 $\boldsymbol{a}$ 的阶乘，只需将括号中的 a_1, a_2, a_3 做向量化即可:

```
a = np.random.rand(10, 3)
b = np.random.rand(1, 3)

cross_1 = np.zeros_like(a)
for i in range(cross_1.shape[0]):
    cross_1[i, :] = np.cross(a[i, :], b[0, :])

a1 = a[:, 0:1]
a2 = a[:, 1:2]
a3 = a[:, 2:3]

i = np.array([1,0,0])
j = np.array([0,1,0])
```

```python
k = np.array([0,0,1])

b1 = b[0,0]
b2 = b[0,1]
b3 = b[0,2]

cross_2 = (a2*b3 - a3*b2) * i - (a1*b3 - a3*b1) * j + (a1*b2 - a2*b1) * k
print(np.max(np.abs(cross_2 - cross_1)))
```

通过查阅 NumPy 文档，我们发现这可以用一行代码实现：

```python
np.cross(a, b, axisa=1)  # 将 a 的每一行视作向量
```

4 为何矩阵运算容易并行化

我们可以给出几个 CUDA 例子：

4.1 两组向量的按元素四则运算

以加法为例：

```python
import cupy as cp

l = 100
x = cp.random.rand(l, dtype = cp.float32)
y = cp.random.rand(l, dtype = cp.float32)

BLOCK_SIZE = 32
block_dim = (1, BLOCK_SIZE,)

from math import ceil
grid_dim = (1, ceil(l/block_dim[1]),)

add_kernel = cp.RawKernel(
    r'''
    extern "C" __global__
    void add(float *x, float *y, float *z, int l)
    {
        int tdx = blockIdx.y * blockDim.y + threadIdx.y;
        if (tdx >= l)
        {
            return;
        }

        z[tdx] = x[tdx] + y[tdx];
    }
    ''',
    'add'
)

z1 = cp.zeros_like(x, dtype=cp.float32)
```

```
add_kernel(grid_dim, block_dim, (x, y, z1, l,))

z2 = cp.add(x, y)

print(cp.max(cp.abs(z1-z2)))
```

在上列代码中，我们随机生成了两个长度为 l 的向量，然后演示了如何并行地计算两个向量的按元素相加.

4.2 矩阵乘法

对于 $n \times m$ 矩阵 A，$m \times p$ 矩阵 B，矩阵 A 与 B 的「乘法」指的是：找到矩阵 C，使得对任意 $1 \leq i \leq n$，对任意 $1 \leq j \leq p$，都有

$$C[i,j] = \sum_{k=1}^{m} A[i,k]B[k,j]. \tag{4.1}$$

矩阵 C 是 n 行 p 列的，所以我们可以创建不少于 $n \times p$ 个 CUDA 线程，保证每个 $C[i,j]$ 至少分到一个就可以了：

```
import cupy as cp

n_rows = 100
hidden_dim = 80
n_cols = 120

mat_A = cp.random.rand(n_rows, hidden_dim, dtype=cp.float32)
mat_B = cp.random.rand(hidden_dim, n_cols, dtype=cp.float32)

mul_kernel = cp.RawKernel(
    r'''
    extern "C" __global__
    void mul(float *mat_A, float *mat_B, float *mat_C, int n_rows, int hidden_dim, int n_cols)
    {
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;

        if (sizeof(float) != 4)
        {
            return;
        }

        if ((row >= n_rows) || (col >= n_cols))
        {
            return;
        }

        float temp = 0;
        for (int k = 0; k < hidden_dim; ++k)
        {
            temp = temp + mat_A[row * hidden_dim + k] * mat_B[k * n_cols + col];
        }

```

```
        mat_C[row * n_cols + col] = temp;
    }
    ''',
    'mul'
)

mat_C1 = cp.zeros((n_rows, n_cols,), dtype=cp.float32)

from math import ceil
BLOCK_SIZE = 16
block_dim = (BLOCK_SIZE, BLOCK_SIZE,)
grid_dim = (ceil(n_cols/block_dim[0]), ceil(n_rows/block_dim[1]),)

mul_kernel(grid_dim, block_dim, (mat_A, mat_B, mat_C1, n_rows, hidden_dim, n_cols,))

mat_C2 = cp.matmul(mat_A, mat_B)

print(cp.max(cp.abs(mat_C1 - mat_C2)))
```

最后一行用于验证实现是否正确.

5 为何要尽可能地并行化

以两个长度为 n 的向量相加为例，可以这样加:

```
n_length = 100
a = np.random.rand(n_length)
b = np.random.rand(n_length)
c1 = np.zeros_like(a)
for i in range(c1.shape[0]):
    c1[i] = a[i] + b[i]
```

还可以这样加:

```
c2 = a + b
print(np.max(np.abs(c1 - c2)))
```

可是在第二种方法中，两个向量的按元素加法运算，实质上被分解成了 n 个不依赖于计算次序的标量加法运算，也就是:

```
任务0: a[0] + b[0]
任务1: a[1] + b[1]
任务2: a[2] + b[2]
...
任务99: a[99] + b[99]
```

这 n 个标量加法「任务」，先做哪个，后做哪个，对最终向量 $\boldsymbol{a}$ 和向量 $\boldsymbol{b}$ 的按元素相加结果根本没有任何影响，假如说这里的 n 不是 100 而是一百万，那么我可以轻易地将这一百万个任务分发到若干台计算设备上，就如同在上文中，我们将两个 n 维向量的按元素相加任务转化为 n 个标量相加任务，再将 n 个标量相加任务分配到不少于 n 个 CUDA 线程上一样. 一句话说就是，并行化能够让多个计算设备 (无论是逻辑上的还是物理上的) 能够协同工作，从而提高总体计算能力.