并行化计算尝试

2021 年 1 月 19 日

1 引言

给定一个矩阵 $X_{n\times m}$, 设 $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$ 是它的行向量, 我们想计算出这样一个矩阵 M, 满足 $M(i,j) = \cos(\boldsymbol{x}_i, \boldsymbol{x}_j)$，具体有哪些计算方法呢?

2 方法一：循环

首先是计算 M 的第一行，然后是第二行，……，这样一行一行地计算:

```
import numpy as np
from time import time

# 要生成 1000 个随机点，每行对应一个点，每个点是一个 4 维向量.
n = 1000
data = np.random.rand(n, 4)

s = time()
cosines_method_1 = np.zeros((n, n,))
for i in range(0, n-1):
for j in range(i+1, n):
    inner_prod = data[i:(i+1), :] @ (data[j:(j+1), :].T)
    norm_prod = np.linalg.norm(data[i, :]) * np.linalg.norm(data[j, :])
    cosines_method_1[i, j] = inner_prod / norm_prod

np.fill_diagonal(cosines_method_1, 1)
e = time()
t_delta_1 = e-s
print(t_delta_1)
```

输出为

```
11.335773944854736
```

也就是说花了大概 11 秒的时间来计算这些.

3 方法二：向量化

我们注意到，对于每一组可能的 (i,j)，$M(i,j)$ 的计算都是相似的，所以我们意识到，有大量的相似的子计算过程存在. 要将同样的计算方法应用在不同的元素上，我们首先想到要利用线性代数教给我们的知识.

设 A 是一个 $n\times 1$ 的矩阵，设 B 是一个 $1\times n$ 的矩阵，那么按照线性代数的知识，我们知道 A 与 B 可以进行矩阵乘法操作，也就是说，存在矩阵 C，它的行数等于 A 的行数，它的列数等于 B 的列数，并且对任意 $1\le i<j\le n$，都有 $C(i,j)=(AB)(i,j)$．由此我们意识到，如果一个列向量作为矩阵，与自身的转置做矩阵乘法运算，那么产生的那个矩阵的第 i 行第 j 列的元素的值，就恰好等于它的第 i 个元素与第 j 个元素的乘积．

余弦值可以用范数与内积来表示，设 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 是相同维数的向量，那么

$$\cos\langle \boldsymbol{a},\boldsymbol{b}\rangle=\frac{\boldsymbol{a}\,\boldsymbol{b}^{\mathsf{T}}}{\|\boldsymbol{a}\|\|\boldsymbol{b}\|} \tag{3.1}$$

同样的，对于 $M(i,j)$ ，我们知道他是 $\cos\langle \boldsymbol{x}_i,\boldsymbol{x}_j\rangle$ ，于是有

$$M(i,j)=\frac{\boldsymbol{x}_i\,\boldsymbol{x}_j^{\mathsf{T}}}{\|\boldsymbol{x}_i\|\|\boldsymbol{x}_j\|} \tag{3.2}$$

我们设想，如果存在这样一个矩阵 A，它满足：对任意 $1\le i<j\le n$ ，都有 $A(i,j)=\boldsymbol{x}_i\,\boldsymbol{x}_j^{\mathsf{T}}$；并且如果还存在这样一个矩阵 B，它满足，对任意 $1\le i<j\le n$，都有 $B(i,j)=\|\boldsymbol{x}_i\|\|\boldsymbol{x}_j\|$，那么矩阵 M 实际上就可以表示为 A 与 B 的按元素相除，也就是说

$$M=A\,./\,B \tag{3.3}$$

现在，我们应该开始考虑具体怎么计算 A 以及 B．首先，对于 A，我们知道

$$A(i,j)=\boldsymbol{x}_i\,\boldsymbol{x}_j^{\mathsf{T}}=\sum_{k=1}^{m}X(i,k)X(j,k) \tag{3.4}$$

为了简化问题，我们假设 m 是一个不大的数，比如说 $m=4$，这时

$$A(i,j)=X(i,1)X(j,1)+X(i,2)X(j,2)+X(i,3)X(j,3)+X(i,4)X(j,4) \tag{3.5}$$

我们故技重施，将 A 写成四个矩阵的和

$$A=A_1+A_2+A_3+A_4 \tag{3.6}$$

其中 $A_k(i,j)=X(i,k)X(j,k)$，对于 $k=1,2,3,4;\ 1\le i<j\le n$．现在我们已经知道了，$A_k$ 就是 X 的第 k 列与第 k 列转置的矩阵乘积，也就是说

$$A_k=X(:,k)\,(X(:,k))^{\mathsf{T}},\quad k=1,2,3,4 \tag{3.7}$$

到了这里，我们已经知道怎样把 A 的计算向量化了．

下面来看 B，它的第 i 行第 j 列的元素的值正是 $\|\boldsymbol{x}_i\|\|\boldsymbol{x}_j\|$，由此受到启发，我们设想，存在一个 $n\times 1$ 的矩阵 N，满足 $N(i,1)=\|\boldsymbol{x}_i\|$，如果能计算出这个 N，那么 M 就可以写为

$$M=N\,N^{\mathsf{T}} \tag{3.8}$$

这个 N 也很好计算，它等于

$$N=X(:,1)\,.\!\times X(:,2)\,.\!\times X(:,3)\,.\!\times X(:,4) \tag{3.9}$$

也就是将 X 的四列看做四个矩阵，再按元素相乘．

经过这一番分析，我们已经将 M 拆解为一系列的矩阵运算与按元素运算的复合，下面就可以开始实现了：

```
s= time()

data_1 = data[:, 0:1]
data_2 = data[:, 1:2]
data_3 = data[:, 2:3]
data_4 = data[:, 3:4]

pair_inner_prods = (data_1 @ data_1.T) + \
    (data_2 @ data_2.T) + \
    (data_3 @ data_3.T) + \
    (data_4 @ data_4.T)

norms = np.sqrt(
    data_1 * data_1 + \
    data_2 * data_2 + \
    data_3 * data_3 + \
    data_4 * data_4
)

cosines_method_2 = pair_inner_prods / (norms @ norms.T)

e = time()
t_delta_2 = e-s
print(t_delta_2)
```

输出结果为

```
0.06543731689453125
```

再运行

```
abs(t_delta_1 - t_delta_2)/min(t_delta_1, t_delta_2)
```

可以看到快了约 170 倍.

4 方法三：GPU 加速

使用 NumPy 的矩阵乘法和按元素乘法功能之所以能大幅加快计算速度，是因为 NumPy 利用了专业的线性代数数值计算库如 BLAS, LAPACK 等，而这些数值计算库则充分利用了 CPU 的 SIMD[1] 指令，简而言之，NumPy 让我们在这个计算任务上，充分发挥了 CPU 的大部分的潜力.

而对于 SIMD 类型的计算任务, GPU 比 CPU 其实是更加擅长的, 再加上现在有了 CUDA[2] 和 CuPy[3], 想要验证这一点变得非常简单.

首先载入 CuPy 软件包，并且将数据迁移到显存上:

```
import cupy as cp

s_move_data_to_gpu = time()
data_1 = cp.asarray(data_1)
data_2 = cp.asarray(data_2)
```

[1] Single Instruction Multiple Data, 是指一系列可同时操作于非常多个运算数的运算指令

[2] Nvidia 公司为它家特定系列的显卡开发的一系列并行计算 API

[3] CUDA 提供给 Python 开发者的编程接口

```
data_3 = cp.asarray(data_3)
data_4 = cp.asarray(data_4)
e_move_data_to_gpu = time()
print(e_move_data_to_gpu - s_move_data_to_gpu)
```

然后就可以开始在 GPU 上进行计算了:

```
s_gpu_compute = time()

pair_inner_prods = cp.matmul(data_1, data_1.T) + \
    cp.matmul(data_2, data_2.T) + \
    cp.matmul(data_3, data_3.T) + \
    cp.matmul(data_4, data_4.T)

norms = cp.sqrt(
    data_1 * data_1 + \
    data_2 * data_2 + \
    data_3 * data_3 + \
    data_4 * data_4
)

cosines_method_2 = pair_inner_prods / cp.matmul(norms, norms.T)

e_gpu_compute = time()

print(e_gpu_compute - s_gpu_compute)
```

输出结果为:

```
0.0031576156616210938
```

事实上 CuPy 与 NumPy 是很大程度上兼容的，运行了 `cp.asarray` 之后，剩下的无非就是把代码里的 `np` 替换为 `cp`，就可以通过调用 CuPy 在 GPU 上进行计算了. 现在再运行:

```
t_delta_2 / (e_gpu_compute - s_gpu_compute)
```

可看到输出

```
20.723648444612217
```

这说明 GPU 比 CPU 还要再快上一个台阶.

5 结论

我们先后尝试了三种计算余弦矩阵 M 的方法，第一种方法是用 `for` 循环自己编程实现，耗时为 11.335773，第二种方法是把原有的计算任务用向量与矩阵之间的操作来实现，耗时为 0.065437，第三种方法是把第二种方法放到 GPU 上进行，耗时为 0.003157. 通过简单计算可得，第二种方法约比第一种快了 170 倍，第三种方法又比第一种方法快了 20 倍，可想而知，第三种方法比第一种方法快了数千倍.