# Lab2 - naive gemm 实验报告

## 实验环境

- OS 版本：`Ubuntu 22.04.3 LTS(WSL2)`
- gcc 版本：`11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)`
- CPU
  - 型号：`AMD Ryzen 7 7840H with Radeon 780M Graphics`
  - 频率：`3.80GHz`
  - 物理核数：`8`
- 内存大小：`32GB`

## 实验内容

> (3) Test_cblas_dgemm.c 修改为行主序后结果有什么不同?

主序的更改只影响矩阵运算的顺序而不影响结果，实验中行主序和列主序得到的**结果一致**，均为：

```
-4.000000 11.000000 0.000000 11.000000 -9.000000 5.000000 8.000000 5.000000 6.000000
```

> (4) time_dgemm.c
>
> time_dgemm.c 分别测试 `M=N=K=256/1024/4096/8192` 时，以下面表格的形式记录两者的 duration 和 gflops 的值。从数据中可以发现什么规律，可以尝试自己分析下。数据有波动是正常的，可以运行多次取平均。

下表展示了当 `M=N=K=256/1024/4096/8192` 时，对应的 duration 和 gflops 值：

| | | 256 | 1024 | 4096 | 8192 |
|---|---|---|---|---|---|
| 1 | cblas_dgemm duration | 0.0047135 s | 0.0120285 s | 0.301434 s | 2.654985 s |
| 2 | naive_dgemm duration | 0.074672 s | 4.35878 s | 667.005829 s | 7926.978857 s |
| 3 | cblas_dgemm gflops | 15.7936425 | 363.059582 | 911.900804 | 828.262026 |
| 4 | naive_dgemm gflops | 0.9000295 | 0.9871985 | 0.412107 | 0.277410 |

观察上表可以得出以下规律：

- 在同一矩阵规模下，cblas 实现的乘法耗时总是显著地比 naive 实现短
- 在同一矩阵规模下，cblas 实现的 gflops 值总是显著地比 naive 实现大
- 随着矩阵规模的增大，两种实现的矩阵乘法消耗的时间均增大，并且 `m` 越大，两种乘法消耗的时间增长越快
- 随着矩阵规模的增大，cblas 实现的*乘法耗时增长的速度和加速度*均比 naive 实现小
- 随着矩阵规模的增大，cblas 实现的 gflops 值大致增大，而 naive 实现大致减少，两者变化方向相反
- cblas 矩阵乘法的 gflops 值随 `m` 的增大，先快速增大，增大速度减小，最后相对稳定并小幅减小
- naive 矩阵乘法的 gflops 值随 `m` 的增大，先小幅上升，然后快速下降

注意到，与 naive 实现不同，cblas 实现在矩阵规模增大时 gflops 值反而大体上升，据此可以猜测 cblas 对大规模矩阵有特殊优化。

综上所述，不难发现，在运算*小规模和大规模*矩阵时，cblas 实现的矩阵乘法的性能均强于 naive 实现，具有**消耗时间短、大规模矩阵优化好、整体运行效率高**的优点。

## 问题与解决方案

> ϟ Openblas 编译报错
>
> 按照官方安装指南中的方法从源码编译程序，报错

> ✓ 解决方案
>
> 安装预编译版本，run:
>
> ```
> sudo apt install libopenblas-dev
> ```